Table of Contents

Experiment 1

Develop a program to Load a dataset and select one numerical column. Compute mean, median, mode, standard deviation, variance, and range for a given numerical column in a dataset. Generate a histogram and boxplot to understand the distribution of the data. Identify any outliers in the data using IQR. Select a categorical variable from a dataset. Compute the frequency of each category and display it as a bar chart or pie chart.

Introduction

Statistical analysis plays a crucial role in understanding data distribution and variability. This experiment focuses on computing key statistical measures such as mean, median, mode, standard deviation, variance, and range for a selected numerical column in a dataset. Additionally, it involves visualizing data using histograms and box plots to identify patterns and potential outliers.

Outliers are detected using the Interquartile Range (IQR) method. Furthermore, categorical data analysis is performed by computing category frequencies and representing them using bar charts or pie charts. These techniques are essential for Exploratory Data Analysis (EDA) in Machine Learning.

Descriptive Statistics

Descriptive statistics summarize and describe the essential characteristics of a dataset. The key measures include:

- Mean (Average): The sum of all values divided by the total number of values.
- Median: The median is the middle value of a sorted dataset. If the dataset has an even number of observations, the median is the average of the two middle values. It is useful when dealing with skewed data, as it is less affected by extreme values.
- Mode: Mode is the most frequently occurring value in a dataset. If multiple values appear with the highest frequency, the dataset is multimodal.
- Standard Deviation (SD): Standard deviation measures the dispersion of data points

$$\sigma = \sqrt{\frac{\sum(x_i-\mu)^2}{n}}$$

from the mean and is given by:

- Variance: Variance quantifies the spread of data by squaring the standard deviation: $\sigma^2 = \frac{\sum(x_i-\mu)^2}{n}$ Higher variance indicates greater data spread.
- Range: The difference between the maximum and minimum values in the dataset.

Distribution

In statistics, distribution refers to how data values are spread across a range. Understanding the distribution of numerical features in a dataset helps in identifying patterns, detecting outliers, and making informed decisions. The two primary ways to visualize distribution are histograms and box plots.

Data Visualization for Numerical Data

Histograms

A histogram is a graphical representation of the distribution of a numerical feature. It divides the data into bins (intervals) and counts the number of observations in each bin.

Importance of Histograms

- **Detecting Skewness:** A histogram can reveal whether a distribution is symmetric, left-skewed, or right-skewed.
- **Identifying Modal Patterns:** Some distributions are unimodal (single peak), while others may be bimodal or multimodal.
- **Assessing Normality:** If the histogram resembles a bell curve, the data may be normally distributed.
- **Understanding Data Spread:** Helps in detecting whether data is evenly distributed or concentrated in certain regions.

Box Plots (Box-and-Whisker Plots)

A box plot provides a summary of the distribution of numerical data using five key statistics:

- Minimum: The smallest value (excluding outliers).
- First Quartile (Q1): 25th percentile.
- Median (Q2): 50th percentile (middle value).
- Third Quartile (Q3): 75th percentile.
- Maximum: The largest value (excluding outliers).
- Outliers are detected using the Interquartile Range (IQR) rule:

  Outliers = Values outside Q1 - 1.5 * IQR or Q3 + 1.5 * IQR.

Importance of Box Plots

- **Identifying Outliers:** Points lying outside the whiskers indicate potential outliers.
- **Comparing Distributions:** Box plots allow easy comparison of multiple features or groups.
- **Measuring Data Spread:** The length of the box and whiskers provides insight into data variability.
- **Understanding Skewness:** If the median is closer to one end, the distribution may be skewed.

Outlier

An outlier is an observation or data point that significantly differs from the rest of the data in a dataset. Outliers can skew statistical analyses and distort the interpretation of results, making it important to identify and understand them.

Key Characteristics of Outliers:

- **Deviation from the Norm:**
  - Outliers exhibit values that deviate substantially from the typical or expected range of values in a dataset.
- **Impact on Statistical Measures**:
  - Outliers can heavily influence summary statistics such as the mean and standard deviation, leading to misleading representations of central tendency and dispersion.
- **Identification**:
  - Outliers are often identified through statistical methods or visual inspection of graphs; such as box plots or scatter plots.
- **Causes of Outliers**:
  - Outliers can arise from measurement errors, data entry mistakes, natural variability, or genuine extreme observations in the population.

Ways to Identify Outliers:

- **Visual Inspection:**
  - Plotting the data using graphs like box plots, scatter plots, or histograms can reveal observations that stand out from the majority.
- **Statistical Methods:**
  - Z-Score: Identifying data points with z-scores beyond a certain threshold (e.g., |z| > 3) as potential outliers.

    $Z = (x-\mu)/\sigma$
  - Interquartile Range (IQR): Using the IQR to identify observations outside a defined range.

```
IQR = Q3 - Q1
LF = Q1 - (1.5*IQR)
UF = Q3 + (1.5*IQR)
```

Dealing with Outliers:

Retaining Outliers:

- In some cases, it may be appropriate to retain outliers, especially if they represent genuine extreme values in the data.
- Retaining outliers allows for an inclusive analysis, considering the full range of variability in the dataset.

Removing Outliers:

- Removing outliers involves excluding extreme values from the dataset before analysis.
- Common methods include using statistical criteria (e.g., Z-scores, IQR) to identify and exclude observations beyond a certain threshold.
- Reduces the impact of extreme values on summary statistics and model results
- Loss of information: Excluding outliers may discard meaningful data points.

Transformation:

- Transformation involves applying mathematical functions to the data to modify its distribution and reduce the impact of outliers.
- Common transformations include logarithmic, square root, or Cube root transformations.

Analyzing Categorical Variables

Frequency Distribution

For categorical variables, frequency distribution counts the occurrences of each category. It helps understand the most and least common categories.

Bar Chart

A bar chart visually represents category frequencies, with categories on the x-axis and frequencies on the y-axis. It helps compare different categories easily.

Pie Chart

A pie chart shows category proportions as slices of a circle, making it useful for understanding relative distributions.

Application in Data Analysis

- Histograms and box plots play a crucial role in:
- Data Cleaning: Detecting anomalies and erroneous values.
- Feature Engineering: Identifying transformations needed for better model performance.
- Understanding Dataset Characteristics: Providing insight into feature distributions, which informs modeling decisions.

Import Necessary Libraries

Import all libraries which are required for our analysis, such as Data Loading, Statistical analysis, Visualizations, Data Transformations, Merge and Joins, etc.

Pandas and Numpy have been used for Data Manipulation and numerical Calculations

Matplotlib and Seaborn have been used for Data visualizations.

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

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

In [24]:
```
# Set random seed for reproducibility
np.random.seed(42)

# Generate numerical data with outliers
numerical_data = np.random.randint(20, 80, size=95).tolist()
outliers = [150, 160, 170, 5, 10]  # Extreme values as outliers
numerical_data.extend(outliers)

# Generate categorical data
categories = ['A', 'B', 'C', 'D']
categorical_data = np.random.choice(categories, size=100)

# Create DataFrame
df = pd.DataFrame({
    'numerical_column': numerical_data,
    'categorical_column': categorical_data
})

# Display the first few rows
print(df.head())

# Save the dataset to CSV (optional)
df.to_csv('Sample_Data.csv', index=False)
```

```
   numerical_column  categorical_column
0                58                   B
1                71                   C
2                48                   D
3                34                   C
4                62                   D
```

In [25]:
```
df.head()
```

Out[25]:

	numerical_column	categorical_column
0	58	B
1	71	C
2	48	D
3	34	C
4	62	D

In [26]:
```
df.shape
```

Out[26]:
```
(100, 2)
```

In [27]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 2 columns):
 #   Column               Non-Null Count  Dtype 
---  ------               --------------  ----- 
 0   numerical_column     100 non-null    int64 
 1   categorical_column   100 non-null    object
dtypes: int64(1), object(1)
memory usage: 1.7+ KB
```

In [28]:
```
df.nunique()
```

Out[28]:
```
numerical_column       55
categorical_column      4
dtype: int64
```

Data Cleaning

In [29]:
```
df.isnull().sum()
```

Out[29]:
```
numerical_column       0
categorical_column     0
dtype: int64
```

In [30]:
```
df.duplicated().sum()
```

Out[30]:
```
np.int64(20)
```

Statistical Measures for Numerical Data

In [31]:
```
num_col = df.select_dtypes(include=[np.number]).columns
cat_col = df.select_dtypes(include=['object']).columns
print(f"numerical_data {num_col}")
print(f"categorical_data  {cat_col}")
```

```
numerical_data  Index(['numerical_column'], dtype='object')
categorical_data  Index(['categorical_column'], dtype='object')
```

In [32]:
```
df.describe().T
```

Out[32]:

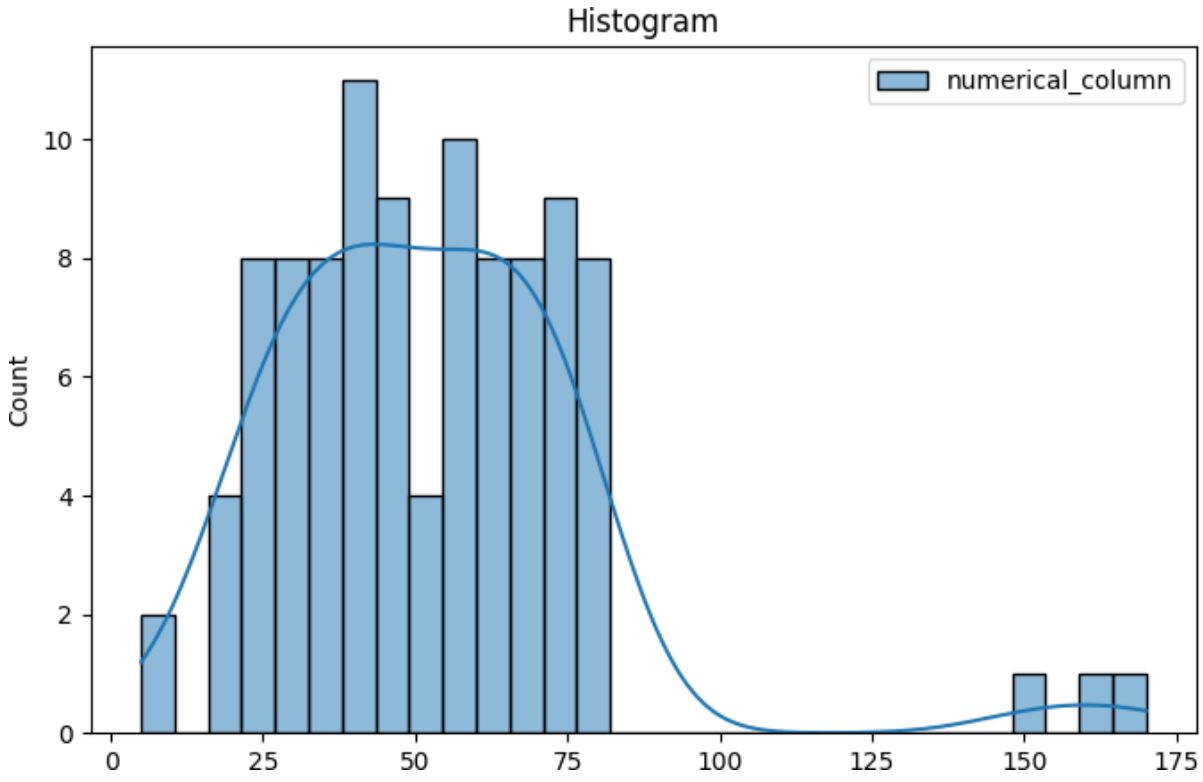
	count	mean	std	min	25%	50%	75%	max
numerical_column	100.0	52.42	26.663326	5.0	34.0	48.5	66.5	170.0

In [33]:
```
# Compute statistics
mean_value = df[num_col].mean()
median_value = df[num_col].median()
mode_value = df[num_col].mode()
std_dev = df[num_col].std()
variance = df[num_col].var()
range_value = df[num_col].max() - df[num_col].min()

# Print statistics
print(f"Mean: {mean_value}")
print(f"Median: {median_value}")
print(f"Mode: {mode_value}")
print(f"Standard Deviation: {std_dev}")
print(f"Variance: {variance}")
print(f"Range: {range_value}")
```

```
Mean: numerical_column     52.42
dtype: float64
Median: numerical_column     48.5
dtype: float64
Mode:     numerical_column
0                 63
Standard Deviation: numerical_column     26.663326
dtype: float64
Variance: numerical_column     710.932929
dtype: float64
Range: numerical_column     165
dtype: int64
```

Data Visualization for Numerical Data

In [34]:
```
# Histogram
plt.figure(figsize=(8,5))
sns.histplot(df[num_col], bins=30, kde=True)
plt.title('Histogram')
plt.show()
```

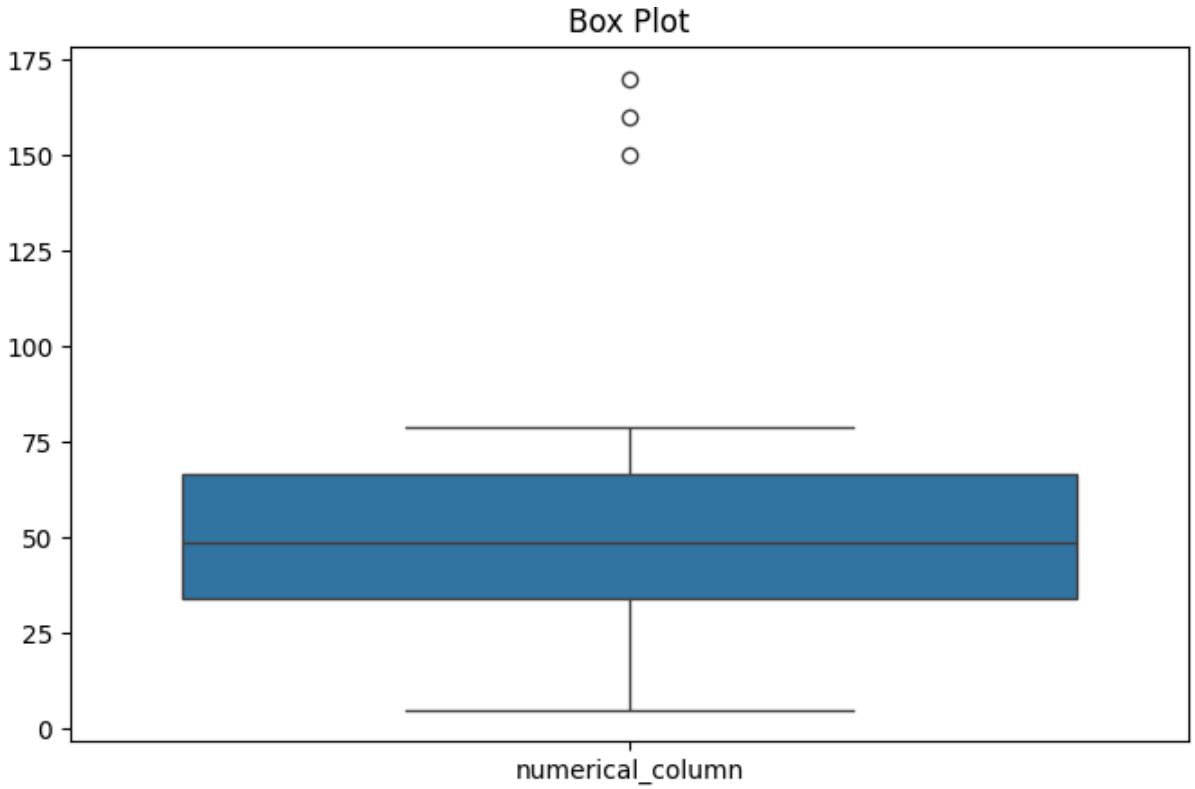


In [45]:
```
# Inference about the histogram chart
print("Inference about the histogram chart:")
print("1. The histogram shows the distribution of the numerical column in the datas
print("2. The data appears to be roughly normally distributed with a peak around th
print("3. There are a few extreme values (outliers) on the right side of the histog
print("4. The presence of outliers can be confirmed by the long tail on the right s
print("5. The KDE (Kernel Density Estimate) line provides a smooth estimate of the
```

```
Inference about the histogram chart:
1. The histogram shows the distribution of the numerical column in the dataset.
2. The data appears to be roughly normally distributed with a peak around the 40-60
range.
3. There are a few extreme values (outliers) on the right side of the histogram, ind
icating the presence of high-value outliers.
4. The presence of outliers can be confirmed by the long tail on the right side of t
he histogram.
5. The KDE (Kernel Density Estimate) line provides a smooth estimate of the data dis
tribution, highlighting the central tendency and spread of the data.
```

In [ ]:
```
# give me a infrence about above chart
```

In [35]:
```
# Box Plot
plt.figure(figsize=(8,5))
sns.boxplot(df[num_col])
plt.title('Box Plot')
plt.show()
```



In [46]:
```
# Inference about the box plot chart
print("Inference about the box plot chart:")
print("1. The box plot shows the distribution of the numerical column in the datase
print("2. The central box represents the interquartile range (IQR), which contains
print("3. The line inside the box indicates the median value of the numerical colum
print("4. The whiskers extend to the minimum and maximum values within 1.5 * IQR fr
print("5. Data points outside the whiskers are considered outliers, which are repre
print("6. The presence of several outliers is evident, with values significantly hi
print("7. The distribution appears to be slightly right-skewed, as indicated by the
```

```
Inference about the box plot chart:
1. The box plot shows the distribution of the numerical column in the dataset.
2. The central box represents the interquartile range (IQR), which contains the midd
le 50% of the data.
3. The line inside the box indicates the median value of the numerical column.
4. The whiskers extend to the minimum and maximum values within 1.5 * IQR from the l
ower and upper quartiles, respectively.
5. Data points outside the whiskers are considered outliers, which are represented a
s individual points.
6. The presence of several outliers is evident, with values significantly higher tha
n the upper whisker.
7. The distribution appears to be slightly right-skewed, as indicated by the longer
whisker and more outliers on the higher end.
```

Identifying Outliers using IQR

In [40]:
```
# Identifying Outliers using IQR
Q1 = df['numerical_column'].quantile(0.25)
Q3 = df['numerical_column'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
```

```
upper_bound = Q3 + 1.5 * IQR

# Filtering out the outliers
outliers = df[(df['numerical_column'] < lower_bound) |
                        (df['numerical_column'] > upper_bound)]['numerical_column']

# Print the detected outliers
print("Outliers:\n", outliers)
```

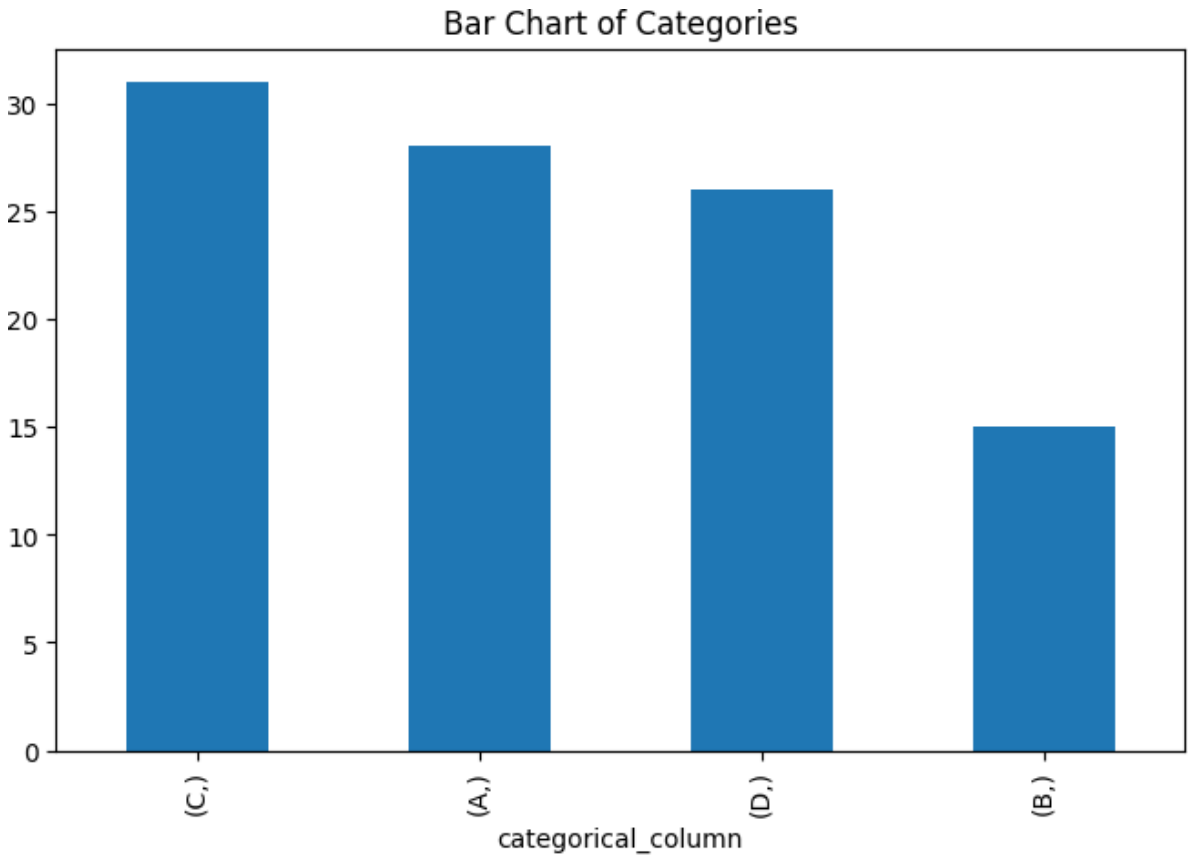
```
Outliers:
 95    150
96    160
97    170
Name: numerical_column, dtype: int64
```

Analyzing Categorical Variables

In [42]:
```
category_counts = df[cat_col].value_counts()
print(category_counts)
```

```
categorical_column
C                       31
A                       28
D                       26
B                       15
Name: count, dtype: int64
```

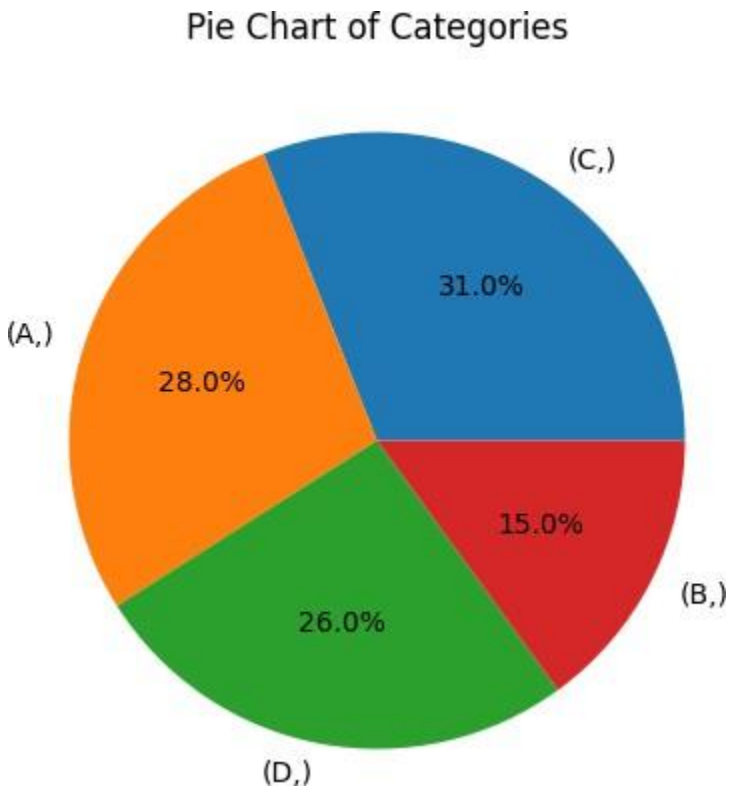
In [43]:
```
# Bar Chart
plt.figure(figsize=(8,5))
category_counts.plot(kind='bar')
plt.title('Bar Chart of Categories')
plt.show()
```



In [47]:
```
# Inference about the bar chart of categories
print("Inference about the bar chart of categories:")
print("1. The bar chart shows the frequency distribution of the categorical column
print("2. Category 'C' has the highest frequency with 31 occurrences.")
print("3. Category 'A' follows with 28 occurrences.")
print("4. Category 'D' has 26 occurrences.")
print("5. Category 'B' has the lowest frequency with 15 occurrences.")
print("6. The chart helps in understanding the distribution and dominance of each c
```

```
Inference about the bar chart of categories:
1. The bar chart shows the frequency distribution of the categorical column in the d
ataset.
2. Category 'C' has the highest frequency with 31 occurrences.
3. Category 'A' follows with 28 occurrences.
4. Category 'D' has 26 occurrences.
5. Category 'B' has the lowest frequency with 15 occurrences.
6. The chart helps in understanding the distribution and dominance of each category
in the dataset.
```

In [44]:
```
# Pie Chart
plt.figure(figsize=(8,5))
category_counts.plot(kind='pie', autopct='%1.1f%%')
plt.title('Pie Chart of Categories')
plt.ylabel('')
plt.show()
```



In [48]:
```
# Inference about the pie chart of categories
print("Inference about the pie chart of categories:")
print("1. The pie chart shows the proportion of each category in the categorical co
print("2. Category 'C' has the largest proportion, making up 31% of the data.")
print("3. Category 'A' follows with 28% of the data.")
print("4. Category 'D' accounts for 26% of the data.")
print("5. Category 'B' has the smallest proportion, making up 15% of the data.")
print("6. The pie chart provides a clear visual representation of the relative dist
```

```
Inference about the pie chart of categories:
1. The pie chart shows the proportion of each category in the categorical column.
2. Category 'C' has the largest proportion, making up 31% of the data.
3. Category 'A' follows with 28% of the data.
4. Category 'D' accounts for 26% of the data.
5. Category 'B' has the smallest proportion, making up 15% of the data.
6. The pie chart provides a clear visual representation of the relative distribution
of each category in the dataset.
```

Experiment 2

Develop a program to Load a dataset with at least two numerical columns (e.g, Iris, Titanic), Plot a scatter plot of two variables and calculate their Pearson correlation coefficient. Write a program to compute the covariance and correlation matrix for a dataset. Visualize the correlation matrix using a heatmap to know which variables have strong Positive/negative correlations.

Introduction

we aim to analyze the relationship between numerical variables in a dataset using statistical and visual techniques. Understanding correlations between features helps in feature selection, multicollinearity detection, and understanding the underlying data structure.

We will focus on:

- Plotting a scatter plot to visually interpret relationships between two numerical variables.
- Computing Pearson correlation coefficient to measure the strength of linear association.
- Calculating the covariance and correlation matrix to study the relationships among multiple variables.
- Visualizing the correlation matrix using a heatmap for better insights into feature dependencies.

Scatter Plot: Visualizing Relationships

A scatter plot is a graphical representation of two numerical variables, where each point represents an observation. It helps in understanding:

- Whether two variables have a positive, negative, or no correlation.
- The strength of the relationship.
- The presence of outliers.

Types of Correlation

- **Positive Correlation (+1 to 0)**: As one feature increases, the other also increases.
- **Negative Correlation (0 to -1)**: As one feature increases, the other decreases.
- **No Correlation (0)**: No linear relationship between the variables.

Pearson Correlation Coefficient

The **Pearson correlation coefficient (r)** is a statistical measure that quantifies the linear relationship between two numerical variables.

Interpretation of Pearson Correlation

- r = 1: Perfect positive correlation.
- r = -1: Perfect negative correlation.
- r = 0: No correlation.
- 0 < r < 1: Weak to strong positive correlation.
- -1 < r < 0: Weak to strong negative correlation.

Pearson correlation is useful for linear relationships but does not capture nonlinear dependencies.

Covariance: Measuring Variability

Covariance quantifies the degree to which two variables change together. A positive covariance indicates that an increase in one variable corresponds to an increase in the other, whereas a negative covariance indicates an inverse relationship.

Limitation: Covariance values depend on the scale of the variables, making it hard to interpret.

Heatmap for Correlation Matrix

A **heatmap** is a visual representation of the correlation matrix. It uses color coding to indicate the strength of relationships between variables.

Benefits of Using a Heatmap

- Easy to interpret relationships between features.
- Quickly identifies highly correlated variables.
- Helps in feature selection and data preprocessing.

Pair Plot

A **pair plot** (also known as a scatterplot matrix) is a collection of scatter plots for every pair of numerical variables in the dataset. It helps in visualizing relationships between variables.

Why Use a Pair Plot?

- Shows the distribution of individual features along the diagonal.
- Displays relationships between features using scatter plots.

- Helps in identifying clusters, trends, and potential outliers.

Import Python Libraries

Import all libraries which are required for our analysis, such as Data Loading, Statistical analysis, Visualizations, Data Transformations, Merge and Joins, etc.

Pandas and Numpy have been used for Data Manipulation and numerical Calculations

Matplotlib and Seaborn have been used for Data visualizations.

In [41]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

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

In [42]:
```
df1 = sns.load_dataset('titanic')
df1
```

Out[42]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	ad
0	0	3	male	22.0	1	0	7.2500	S	Third	man	
1	1	1	female	38.0	1	0	71.2833	C	First	woman	
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	
3	1	1	female	35.0	1	0	53.1000	S	First	woman	
4	0	3	male	35.0	0	0	8.0500	S	Third	man	
...	...	...	...	...	...	...	...	...	...	...	...
886	0	2	male	27.0	0	0	13.0000	S	Second	man	
887	1	1	female	19.0	0	0	30.0000	S	First	woman	
888	0	3	female	NaN	1	2	23.4500	S	Third	woman	
889	1	1	male	26.0	0	0	30.0000	C	First	man	
890	0	3	male	32.0	0	0	7.7500	Q	Third	man	

891 rows × 15 columns

Survived: This column indicates whether the passenger survived the Titanic disaster (1) or not (0).

Pclass: This column represents the passenger's socio-economic status or class, where 1 = Upper class, 2 = Middle class, and 3 = Lower class.

Name: This column contains the name of the passenger.

Sex: This column specifies the gender of the passenger, either male or female.

Age: This column denotes the age of the passenger. Missing values are denoted as NaN.

SibSp: This column indicates the number of siblings or spouses the passenger had aboard the Titanic.

Parch: This column indicates the number of parents or children the passenger had aboard the Titanic.

Ticket: This column contains the ticket number of the passenger.

Fare: This column represents the fare paid by the passenger.

Embarked: This column indicates the port of embarkation for the passenger, with C = Cherbourg, Q = Queenstown, and S = Southampton.

Age_group: This column appears to be a categorical grouping of ages, likely created for analysis purposes. Categories include "Adult", "Middle Age", "Seniors", etc.

In [43]:
```
# Basic Data Exploration
print("\nBasic Information about Dataset:")
print(df1.info())  # Overview of dataset
```

```
Basic Information about Dataset:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
 #   Column       Non-Null Count  Dtype   
---  ------       --------------  -----   
 0   survived     891 non-null    int64   
 1   pclass       891 non-null    int64   
 2   sex          891 non-null    object  
 3   age          714 non-null    float64 
 4   sibsp        891 non-null    int64   
 5   parch        891 non-null    int64   
 6   fare         891 non-null    float64 
 7   embarked     889 non-null    object  
 8   class        891 non-null    category
 9   who          891 non-null    object  
 10  adult_male   891 non-null    bool    
 11  deck         203 non-null    category
 12  embark_town  889 non-null    object  
 13  alive        891 non-null    object  
 14  alone        891 non-null    bool    
dtypes: bool(2), category(2), float64(2), int64(4), object(5)
memory usage: 80.7+ KB
None
```

In [44]:
```
# Summary Statistics
print("\nSummary Statistics:")
print(df1.describe())  # Summary statistics of dataset
```

```
Summary Statistics:
          survived      pclass         age       sibsp       parch        fare
count   891.000000  891.000000  714.000000  891.000000  891.000000  891.000000
mean      0.383838    2.308642   29.699118    0.523008    0.381594   32.204208
std       0.486592    0.836071   14.526497    1.102743    0.806057   49.693429
min       0.000000    1.000000    0.420000    0.000000    0.000000    0.000000
25%       0.000000    2.000000   20.125000    0.000000    0.000000    7.910400
50%       0.000000    3.000000   28.000000    0.000000    0.000000   14.454200
75%       1.000000    3.000000   38.000000    1.000000    0.000000   31.000000
max       1.000000    3.000000   80.000000    8.000000    6.000000  512.329200
```

In [45]:
```
# Check for missing values
print("\nMissing Values in Each Column:")
print(df1.isnull().sum())  # Count of missing values
```

```
Missing Values in Each Column:
survived         0
pclass           0
sex              0
age            177
sibsp            0
parch            0
fare             0
embarked         2
class            0
who              0
adult_male       0
deck           688
embark_town      2
alive            0
alone            0
dtype: int64
```

- age: There are 177 missing values in the age column. This suggests that we need to handle these missing values before performing any analysis. You might consider imputing missing ages using techniques like mean, median.
- embarked and embark_town: Both columns have 2 missing values each. These columns represent the port of embarkation. You could explore the distribution of the existing values to decide how to handle these missing entries. Common approaches include imputing with the mode or creating a new category for missing values.
- deck: The deck column has a whopping 688 missing values. This column likely represents the deck or cabin number where passengers stayed. Given the high number of missing values, you might consider dropping this column from your analysis unless it serves a critical purpose.
- Other columns: The remaining columns (survived, pclass, sex, sibsp, parch, fare, class, who, adult_male, alive, and alone) have no missing values. You can proceed with analyzing these columns without any imputation.

In [46]:
```
df1.duplicated().sum()
```

Out[46]:
```
np.int64(107)
```

Handling The Errors

In [47]:
```
# Removing the Columns
columns_remove = ['deck', 'embarked','alive', 'pclass']
df = df1.drop(columns = columns_remove)
```

In [48]:
```
df['age'].fillna(round(df['age'].mean()), inplace=True)
df['embark_town'].mode()[0]
df['embark_town'].fillna(df['embark_town'].mode()[0], inplace=True)
```

In [49]:
```
df.isnull().sum()
```

Out[49]:
```
survived       0
sex            0
age            0
sibsp          0
parch          0
fare           0
class          0
who            0
adult_male     0
embark_town    0
alone          0
dtype: int64
```

In [50]:
```
# Handling Duplicates
df.drop_duplicates(inplace=True)
```

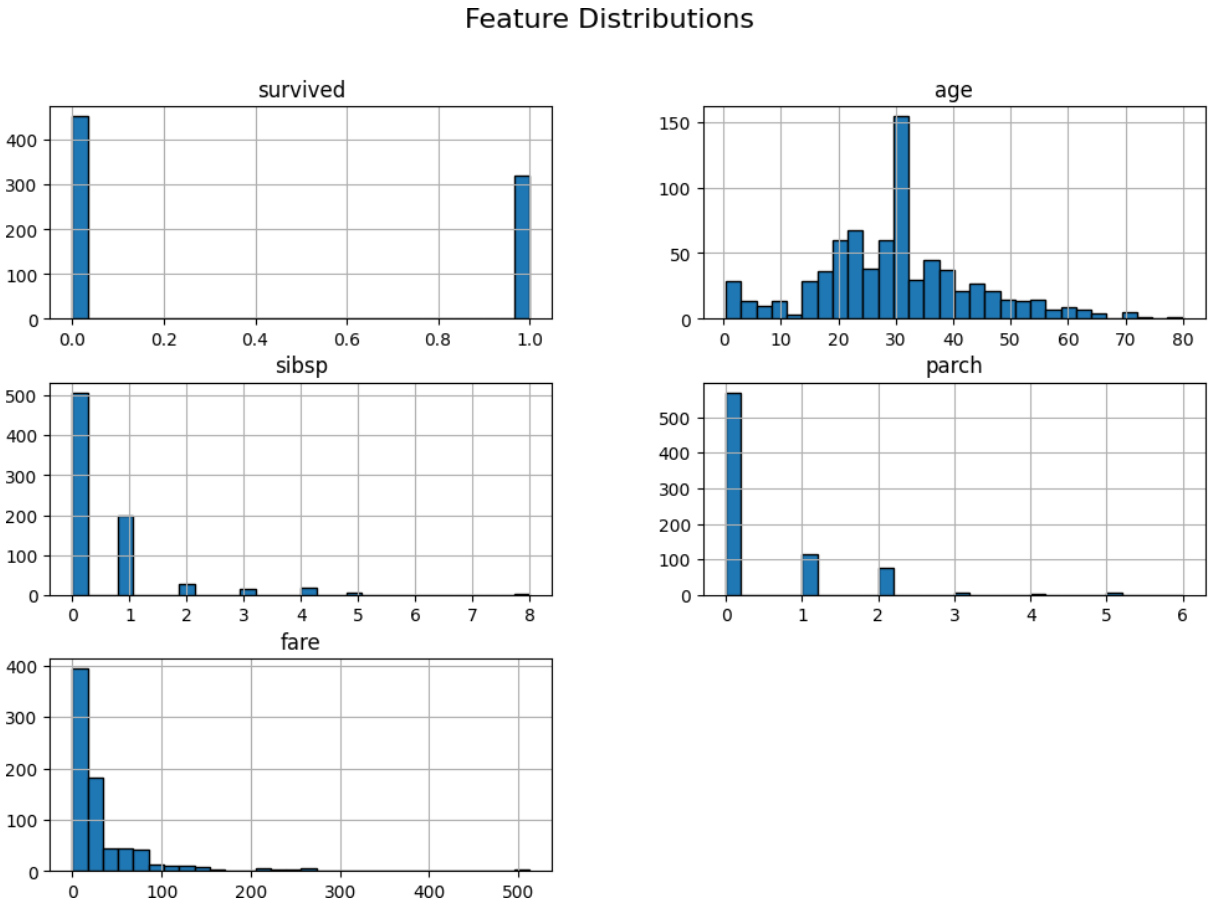
In [51]:
```
df.duplicated().sum()
```

Out[51]:
```
np.int64(0)
```

Univariate Analysis

In [52]:
```
# Histograms for distribution of features
plt.figure(figsize=(12, 8))
df.hist(figsize=(12, 8), bins=30, edgecolor='black')
plt.suptitle("Feature Distributions", fontsize=16)
plt.show()
```

```
<Figure size 1200x800 with 0 Axes>
```

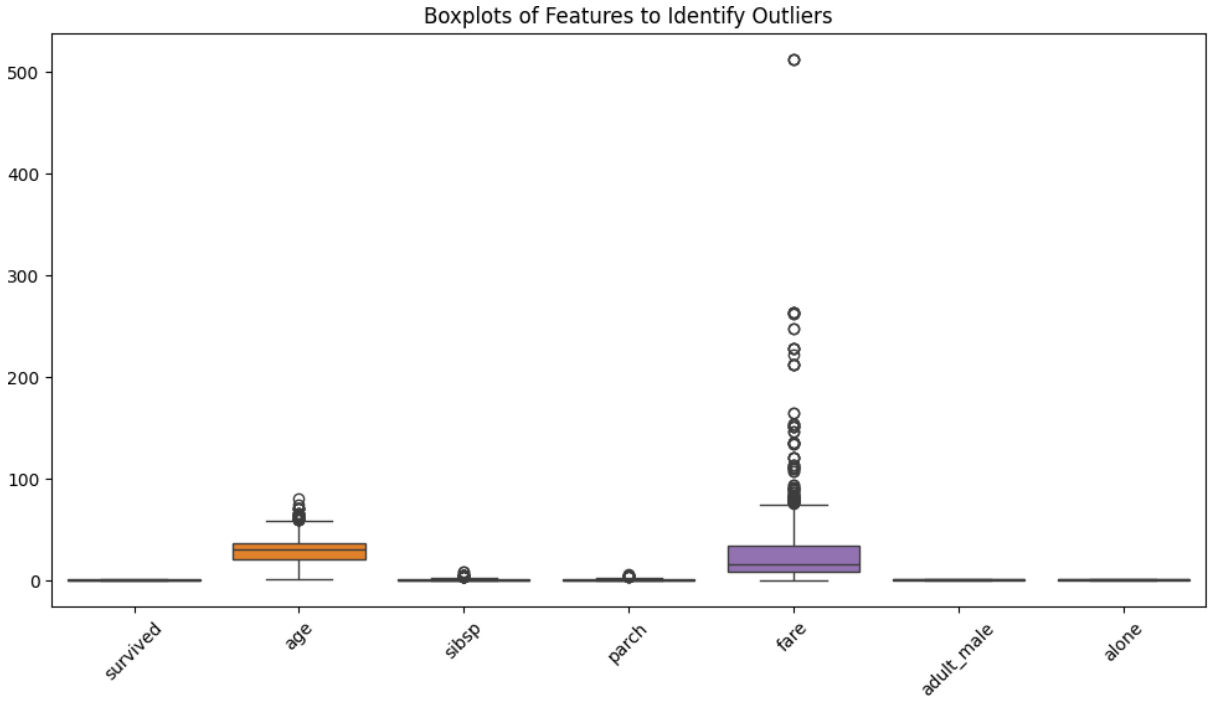


Based on the histograms generated, we can infer the following:

Histograms (Feature Distributions)

1. **Survived**: The histogram shows that more passengers did not survive (0) compared to those who did (1).
2. **Age**: The age distribution is roughly normal with a peak around 30 years. There are some missing values that were filled in the previous steps.
3. **SibSp (Siblings/Spouses Aboard)**: Most passengers had 0 siblings/spouses aboard, with a few having 1 or 2.
4. **Parch (Parents/Children Aboard)**: Similar to SibSp, most passengers had 0 parents/children aboard.
5. **Fare**: The fare distribution is right-skewed, indicating that most passengers paid lower fares, with a few paying significantly higher amounts.

In [53]:
```
# Boxplots for outlier detection
plt.figure(figsize=(12, 6))
sns.boxplot(data=df)
plt.xticks(rotation=45)
plt.title("Boxplots of Features to Identify Outliers")
plt.show()
```



Based on the boxplots generated, we can infer the following:

Boxplots (Outlier Detection)

1. **Survived**: The boxplot shows that there are no significant outliers in the 'survived' column, as it is a binary variable (0 or 1).
2. **Age**: The boxplot for age indicates the presence of some outliers, particularly on the higher end of the age spectrum. These could be older passengers.
3. **SibSp (Siblings/Spouses Aboard)**: The boxplot shows a few outliers, indicating that some passengers had a higher number of siblings or spouses aboard.
4. **Parch (Parents/Children Aboard)**: Similar to SibSp, there are a few outliers, suggesting that some passengers had a higher number of parents or children aboard.
5. **Fare**: The boxplot for fare shows significant outliers, indicating that some passengers paid much higher fares compared to the majority. This could be due to first-class tickets or other factors.

These outliers can be further investigated to understand their impact on the analysis and whether they should be treated or removed.

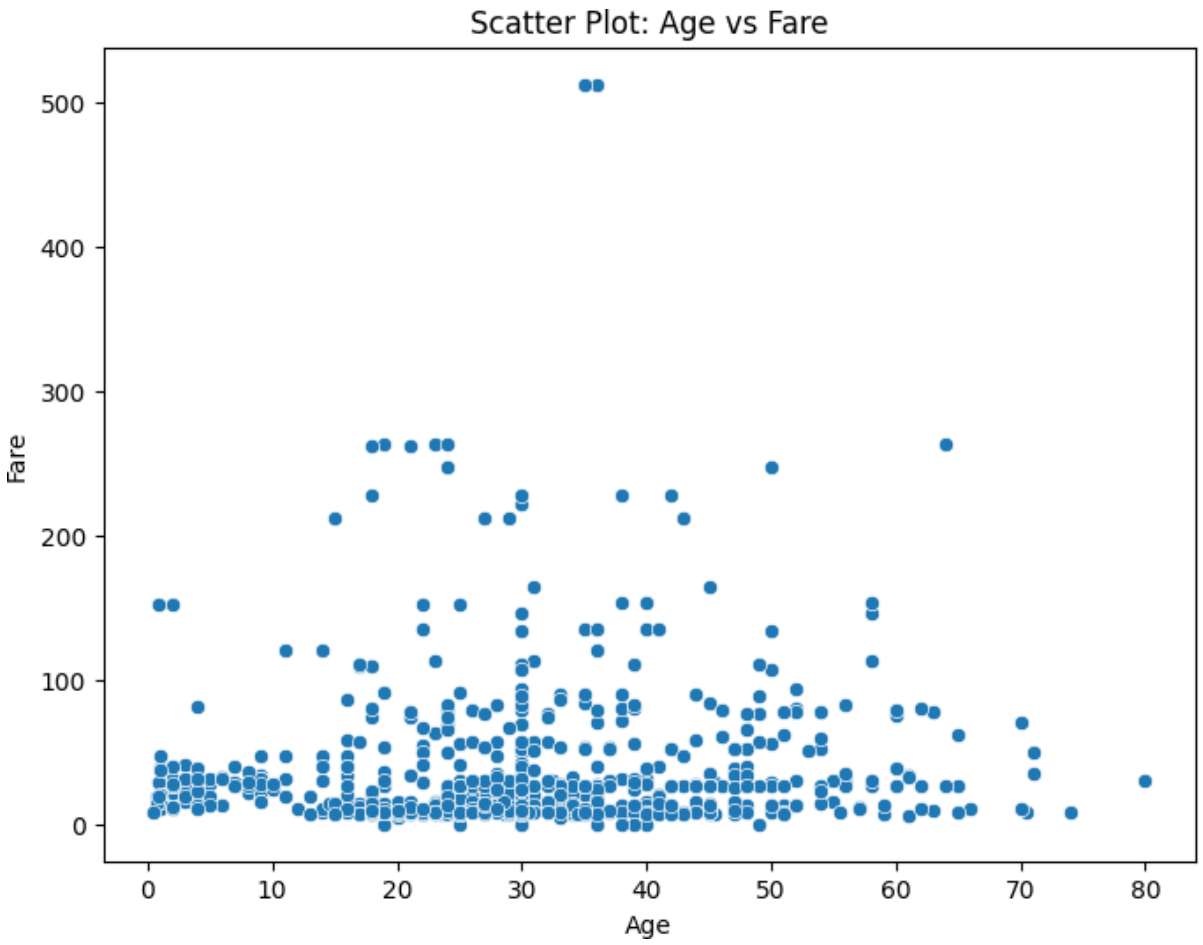
Scatter Plot: Visualizing Relationships

In [ ]:
```
num_col = df.select_dtypes(include=[np.number]).columns.tolist()
cat_col = df.select_dtypes(include=['object']).columns
print(f"numerical_data: {num_col}")
```

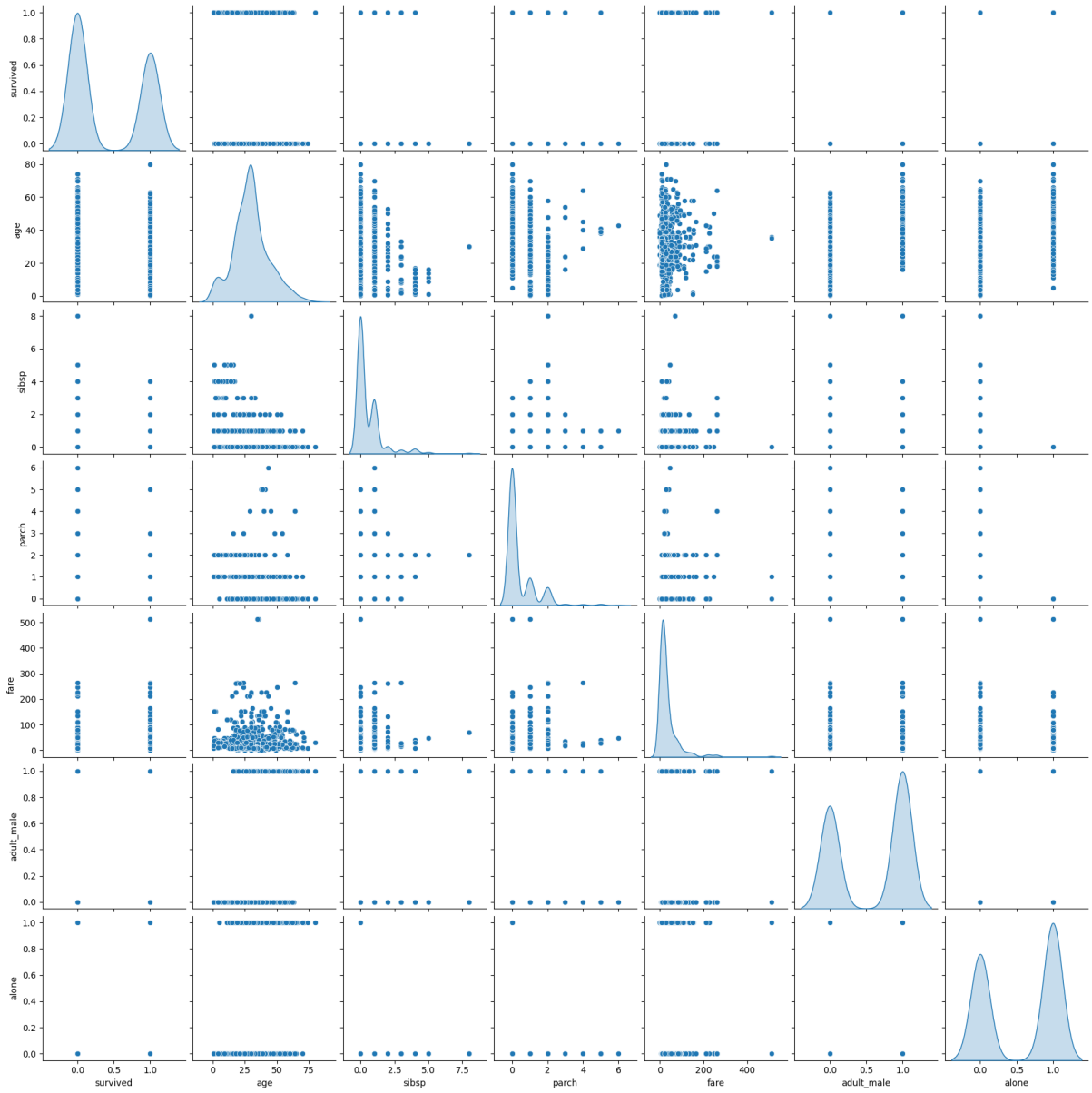
```
numerical_data ['survived', 'age', 'sibsp', 'parch', 'fare']
```

In [55]:
```
# Scatter plot of two variables
plt.figure(figsize=(8, 6))
```

```
sns.scatterplot(x=df['age'], y=df['fare'])
plt.title("Scatter Plot: Age vs Fare")
plt.xlabel("Age")
plt.ylabel("Fare")
plt.show()
```



In [56]:
```
# Pairplot to analyze feature relationships
sns.pairplot(df, diag_kind='kde')
plt.show()
```



Based on the pairplot generated, we can infer the following:

Pairplot (Feature Relationships)

1. **Diagonal Plots**: The diagonal plots show the distribution of individual features using Kernel Density Estimation (KDE). This helps in understanding the distribution shape of each feature.
2. **Survived vs Other Features**:
   - There is a noticeable difference in the distribution of `age` and `fare` between those who survived and those who did not.
   - Higher fares seem to be associated with higher survival rates.
3. **Age vs SibSp/Parch**:
   - There is a slight negative correlation between `age` and `sibsp`, indicating that younger passengers were more likely to have siblings or spouses aboard.

- Similarly, there is a slight negative correlation between `age` and `parch`, indicating that younger passengers were more likely to have parents or children aboard.

4. **Fare vs SibSp/Parch**:
   - There is a positive correlation between `fare` and `sibsp`, as well as `fare` and `parch`, suggesting that passengers who paid higher fares were more likely to travel with family members.
5. **SibSp vs Parch**:
   - There is a strong positive correlation between `sibsp` and `parch`, indicating that passengers with siblings/spouses aboard were also likely to have parents/children aboard.

Overall, the pairplot helps in visualizing the relationships between different numerical features and identifying potential correlations and patterns in the data.

Pearson Correlation Coefficient

In [62]:
```
# Compute Pearson correlation coefficient
correlation = df[['age','fare']].corr('pearson')
print("Pearson Correlation Coefficient:\n", correlation)
```

```
Pearson Correlation Coefficient:
            age      fare
age    1.000000  0.090332
fare   0.090332  1.000000
```

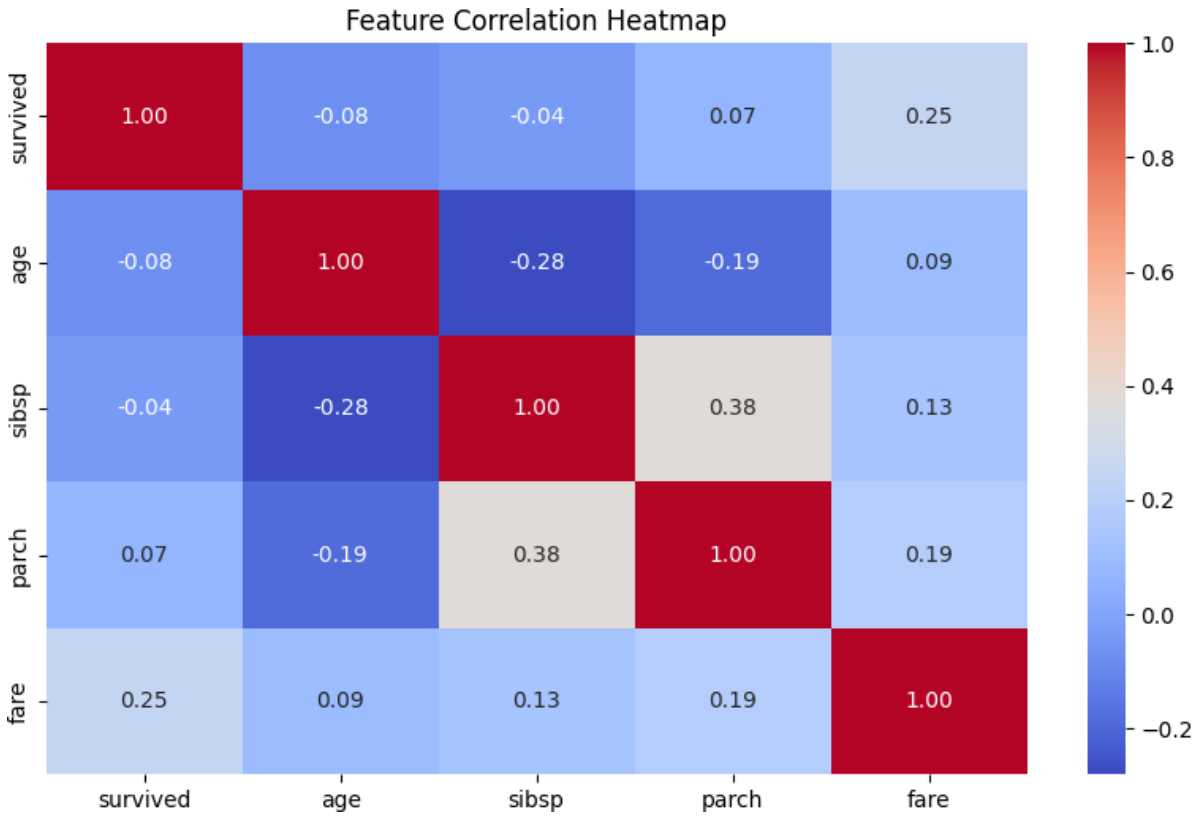
Covariance: Measuring Variability

In [63]:
```
# Compute covariance matrix
covariance = df[['age','fare']].cov()
print("Covariance Matrix:\n", covariance)
```

```
Covariance Matrix:
               age          fare
age    189.639876     65.236604
fare    65.236604   2750.244951
```

Heatmap: Visualizing the Correlation Matrix

In [65]:
```
# Correlation Matrix
plt.figure(figsize=(10, 6))
corr_matrix = df[num_col].corr('pearson')
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title("Feature Correlation Heatmap")
plt.show()
```



Based on the heatmap generated, we can infer the following:

Heatmap (Feature Correlation)

1. **Survived**:
   - There is a positive correlation between `survived` and `fare` (0.25), indicating that passengers who paid higher fares had a higher chance of survival.
   - There is a slight negative correlation between `survived` and `age` (-0.08), suggesting that younger passengers had a slightly higher chance of survival.
   - The correlation between `survived` and `sibsp` (-0.04) and `parch` (0.07) is very weak, indicating that the number of siblings/spouses or parents/children aboard had minimal impact on survival.
2. **Age**:
   - There is a negative correlation between `age` and `sibsp` (-0.28) and `age` and `parch` (-0.19), suggesting that younger passengers were more likely to travel with siblings/spouses or parents/children.
   - The correlation between `age` and `fare` (0.09) is weak, indicating that age had little impact on the fare paid.
3. **SibSp (Siblings/Spouses Aboard)**:
   - There is a positive correlation between `sibsp` and `parch` (0.38), indicating that passengers with siblings/spouses aboard were also likely to have parents/children aboard.

- The correlation between `sibsp` and `fare` (0.13) is weak, suggesting that passengers with more siblings/spouses aboard paid slightly higher fares.

4. **Parch (Parents/Children Aboard)**:

   - There is a positive correlation between `parch` and `fare` (0.19), indicating that passengers with more parents/children aboard paid higher fares.

5. **Fare**:

   - The correlations between `fare` and other features are relatively weak, with the highest being with `survived` (0.25), indicating that fare had some impact on survival but was not strongly correlated with other features.

Overall, the heatmap helps in visualizing the strength and direction of relationships between different numerical features, providing insights into how these features interact with each other.

Experiment 3

Develop a program to implement Principal Component Analysis (PCA) for reducing the dimensionality of the Iris dataset from 4 features to 2.

Introduction to Principal Component Analysis (PCA)

What is PCA?

Principal Component Analysis (PCA) is a **dimensionality reduction technique** used to transform a high-dimensional dataset into a lower-dimensional space while retaining as much variance as possible. It is an unsupervised learning method commonly used in machine learning and data visualization.

Importance of PCA

- Reduces computational complexity by lowering the number of features.
- Helps in visualizing high-dimensional data.
- Removes redundant or correlated features, improving model performance.
- Reduces overfitting by eliminating noise in the data.

How Does PCA Work?

PCA follows these key steps:

1. **Standardization:** The data is normalized so that all features have a mean of zero and a standard deviation of one.
2. **Compute the Covariance Matrix:** This step helps in understanding how different features relate to each other.
3. **Eigenvalue & Eigenvector Calculation:** Eigenvectors represent the direction of the new feature axes, and eigenvalues determine the importance of these axes.
4. **Selecting Principal Components:** The eigenvectors corresponding to the highest eigenvalues are chosen to form the new feature space.
5. **Transforming Data:** The original dataset is projected onto the new feature space with reduced dimensions.

Applying PCA to the Iris Dataset

The **Iris dataset** consists of 4 numerical features (**sepal length, sepal width, petal length, petal width**) used to classify flowers into 3 species (**Setosa, Versicolor, and Virginica**).

- **Goal:** Reduce the **4-dimensional feature space** to **2 principal components** while retaining most of the variance.
- **Benefit:** Enables 2D visualization of the dataset, making it easier to interpret classification results.

Understanding PCA Output

1. Variance Explained by Each Principal Component

PCA provides **explained variance ratios**, which indicate how much information each principal component retains.

- If **PC1 explains 70%** and **PC2 explains 20%**, then the first two principal components capture **90% of the variance** in the dataset.

2. Scatter Plot of PCA-Reduced Data

A 2D scatter plot of PCA-transformed features allows us to visualize how well PCA separates different species in the Iris dataset.

3. Impact of PCA on Classification

- If PCA preserves most of the variance, classification algorithms (e.g., k-NN, SVM) can achieve similar performance with fewer features.
- If too much information is lost, classification accuracy may decrease.

Benefits of PCA

- **Feature Reduction:** Reduces the number of variables without significant loss of information.
- **Noise Reduction:** Removes redundant or less informative features.
- **Improved Visualization:** Enables easier interpretation of high-dimensional data.
- **Better Model Performance:** Enhances efficiency in training machine learning models.

In [5]:
```
# Introduction to the Iris Dataset
# The Iris dataset is one of the most well-known datasets in machine learning and s
# It contains 150 samples of iris flowers categorized into three species: Setosa, V

#
# The goal of using PCA in this exercise is to reduce these four features into two
# This will help in visualizing the data better and understanding its underlying st
#
# Since humans struggle to visualize data in more than three dimensions, reducing t
```

```
# retain the most important patterns while making it easier to interpret. PCA helps
# preserving as much variance as possible.
```

Explanation of Features in the Iris Dataset

The Iris dataset consists of 4 features, which represent different physical characteristics of iris flowers:

```
Sepal Length (cm)
Sepal Width (cm)
Petal Length (cm)
Petal Width (cm)
```

These features were chosen because they effectively differentiate between the three iris species (Setosa, Versicolor, and Virginica).

In the 3D visualizations, we select three features for plotting, which are:

```
Feature 1 → Sepal Length
Feature 2 → Sepal Width
Feature 3 → Petal Length
```

These features are chosen arbitrarily for visualization, but all four features are used in the PCA computation. Why is the Iris Dataset Important?

The Iris dataset is a benchmark dataset in machine learning because:

```
It is small yet diverse, making it easy to analyze.
It has clearly separable classes, which makes it ideal for
classification tasks.
It is preloaded in Scikit-learn, making it accessible for learning
and experimentation.
```

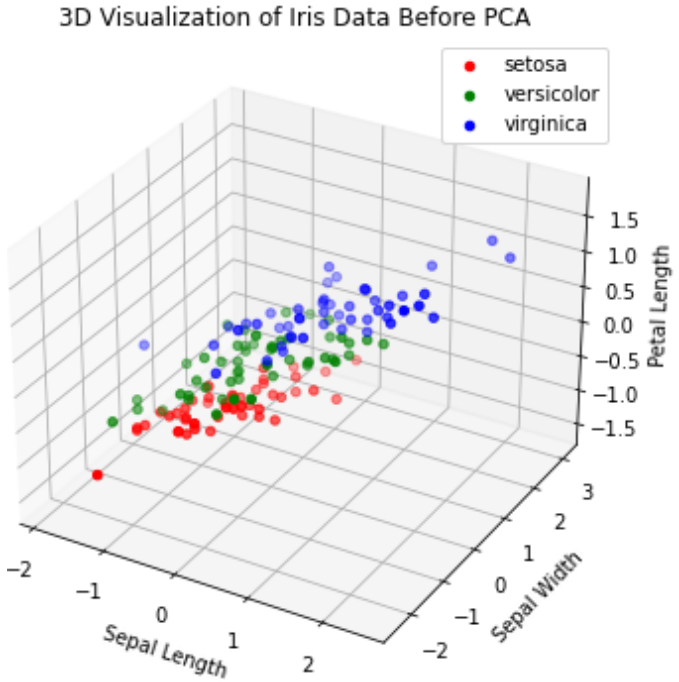
Since the dataset contains three classes (Setosa, Versicolor, and Virginica), PCA helps visualize how well the classes can be separated in a lower-dimensional space.

In [6]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# Step 1: Load the Iris Dataset
iris = datasets.load_iris()
X = iris.data  # Extracting feature matrix (4D data)
y = iris.target  # Extracting labels (0, 1, 2 representing three iris species)
```

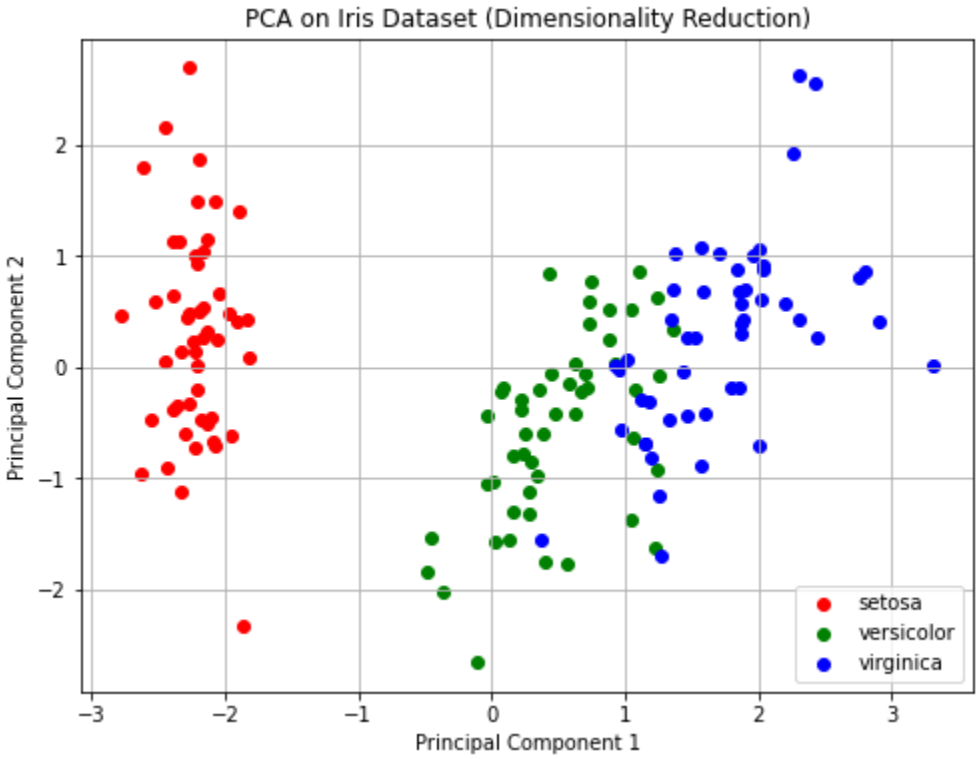
```
# Step 2: Standardizing the Data
# PCA works best when data is standardized (mean = 0, variance = 1)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 3: Calculating Covariance Matrix and Eigenvalues/Eigenvectors
# The foundation of PCA is eigen decomposition of the covariance matrix
cov_matrix = np.cov(X_scaled.T)
print(cov_matrix)
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n",  eigenvectors)

# Step 4: Visualizing Data in 3D before PCA
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
colors = ['red', 'green', 'blue']
labels = iris.target_names
for i in range(len(colors)):
    ax.scatter(X_scaled[y == i, 0], X_scaled[y == i, 1], X_scaled[y == i, 2], color
ax.set_xlabel('Sepal Length')
ax.set_ylabel('Sepal Width')
ax.set_zlabel('Petal Length')
ax.set_title('3D Visualization of Iris Data Before PCA')
plt.legend()
plt.show()

# Step 5: Applying PCA using SVD (Singular Value Decomposition)
# PCA internally relies on SVD, which decomposes a matrix into three parts: U, S, a
U, S, Vt = np.linalg.svd(X_scaled, full_matrices=False)
print("Singular Values:", S)

# Step 6: Applying PCA to Reduce Dimensionality to 2D
# We reduce 4D data to 2D for visualization while retaining maximum variance
pca = PCA(n_components=2)  # We choose 2 components because we want to visualize
X_pca = pca.fit_transform(X_scaled)  # Transform data into principal components

# Step 7: Understanding Variance Explained
# PCA provides the percentage of variance retained in each principal component
explained_variance = pca.explained_variance_ratio_
print(f"Explained Variance by PC1: {explained_variance[0]:.2f}")
print(f"Explained Variance by PC2: {explained_variance[1]:.2f}")

# Step 8: Visualizing the Transformed Data
# We plot the 2D representation of the Iris dataset after PCA transformation
plt.figure(figsize=(8, 6))
for i in range(len(colors)):
    plt.scatter(X_pca[y == i, 0], X_pca[y == i, 1], color=colors[i], label=labels[i
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA on Iris Dataset (Dimensionality Reduction)')
plt.legend()
plt.grid()
plt.show()
```

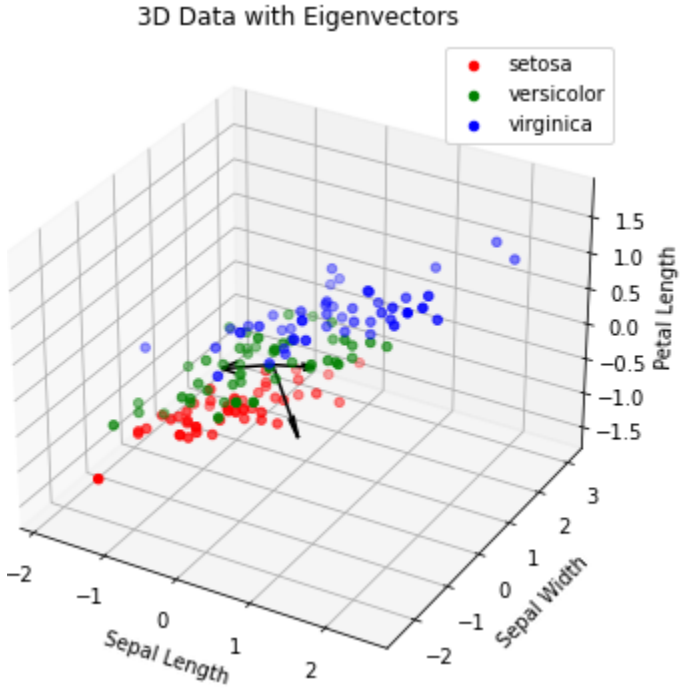
```
# Step 9: Visualizing Eigenvectors Superimposed on 3D Data
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
for i in range(len(colors)):
    ax.scatter(X_scaled[y == i, 0], X_scaled[y == i, 1], X_scaled[y == i, 2], color
for i in range(3):  # Plot first three eigenvectors
    ax.quiver(0, 0, 0, eigenvectors[i, 0], eigenvectors[i, 1], eigenvectors[i, 2],
ax.set_xlabel('Sepal Length')
ax.set_ylabel('Sepal Width')
ax.set_zlabel('Petal Length')
ax.set_title('3D Data with Eigenvectors')
plt.legend()
plt.show()

# Recap:
# - The Iris dataset is historically important for testing classification models.
# - We standardized the data to ensure fair comparison across features.
# - We calculated the covariance matrix, eigenvalues, and eigenvectors.
# - PCA is built on SVD, which decomposes data into important components.
# - We visualized the original 3D data and superimposed eigenvectors.
# - We applied PCA to reduce the dimensionality from 4D to 2D.
# - Finally, we visualized the transformed data in 2D space.
```

```
[[ 1.00671141  -0.11835884   0.87760447   0.82343066]
 [-0.11835884   1.00671141  -0.43131554  -0.36858315]
 [ 0.87760447  -0.43131554   1.00671141   0.96932762]
 [ 0.82343066  -0.36858315   0.96932762   1.00671141]]
Eigenvalues: [2.93808505 0.9201649  0.14774182 0.02085386]
Eigenvectors:
 [[ 0.52106591 -0.37741762  -0.71956635   0.26128628]
 [-0.26934744  -0.92329566   0.24438178  -0.12350962]
 [ 0.5804131   -0.02449161   0.14212637  -0.80144925]
 [ 0.56485654  -0.06694199   0.63427274   0.52359713]]
```



```
Singular Values: [20.92306556 11.7091661    4.69185798  1.76273239]
Explained Variance by PC1: 0.73
Explained Variance by PC2: 0.23
```





In [ ]:

Experiment 4

For a given set of training data examples stored in a .CSV file, implement and demonstrate the Find-S algorithm to output a description of the set of all hypotheses consistent with the training examples.

Introduction to the Find-S Algorithm

What is the Find-S Algorithm?

The **Find-S algorithm** is a **supervised learning algorithm** used in **concept learning** to find the most specific hypothesis that is consistent with a given set of positive training examples. It is one of the simplest algorithms for learning from examples in a hypothesis space.

Importance of Find-S Algorithm

- Helps in understanding how hypotheses are learned from training data.
- Provides a structured way to **generalize from specific instances**.
- Forms the foundation for more advanced **machine learning algorithms**.

Working of the Find-S Algorithm

The Find-S algorithm follows these steps:

1. Initialize the Hypothesis:
   - Start with the most specific hypothesis (i.e., all attributes set to the most restrictive value).
2. Iterate Through Each Training Example:
   - If the example is **positive** (output = "Yes"), update the hypothesis:
     - Replace any attribute value in the hypothesis that is **not consistent** with the example with a more general value (`?`).
   - If the example is **negative** (output = "No"), ignore it.
3. Final Hypothesis:
   - After processing all positive examples, the **final hypothesis** represents the most specific generalization of the training data.

Applying Find-S Algorithm to the Given Dataset

Training Dataset

The dataset contains **five attributes**:

Experience	Qualification	Skill	Age	Hired (Target)
Yes	Masters	Python	30	Yes
Yes	Bachelors	Python	25	Yes
No	Bachelors	Java	28	No
Yes	Masters	Java	40	Yes
No	Masters	Python	35	No

- The **target variable** is "Hired" (Yes/No).
- Only **positive examples (Yes)** are considered for hypothesis generation.
- The algorithm will generate the most specific hypothesis that covers all positive instances.

Understanding the Output Hypothesis

1. Initial Hypothesis

- The algorithm starts with the most specific hypothesis: `h = ("Ø", "Ø", "Ø", "Ø")` (empty hypothesis).

2. Iterative Learning Process

- It generalizes step by step based on the positive training examples.
- Attributes that differ among positive examples are replaced with `?` (wildcard).

3. Final Hypothesis

- The final hypothesis is the most specific generalization covering all positive examples.
- It represents a **logical rule** derived from the dataset.

Limitations of Find-S

- **Only considers positive examples**: It ignores negative examples, which may lead to an incomplete hypothesis.
- **Cannot handle noise or missing data**: Works only when training data is perfect.
- **Finds only one hypothesis**: Does not provide alternative consistent hypotheses.

Understanding Find-S Algorithm and Hypothesis Concept

The Find-S algorithm is a simple machine-learning algorithm used in concept learning. It finds the most specific hypothesis that is consistent with all positive examples in a given training dataset. The algorithm assumes:

The target concept is represented in a binary classification (yes/no, true/false, etc.).

The hypothesis space uses conjunctive attributes (each attribute in a hypothesis must match exactly).

There is at least one positive example in the dataset.

In [2]:
```
import pandas as pd

data = pd.read_csv(r"C:\Users\vijay\Desktop\Machine Learning Course Batches\FDP ML
```

In [5]:
```
print(data)
```

```
Experience   Qualification   Skill    Age  Hired
Yes          Masters         Python   30   Yes
Yes          Bachelors       Python   25   Yes
No           Bachelors       Java     28   No
Yes          Masters         Java     40   Yes
No           Masters         Python   35   No
```

In [4]:
```
def find_s_algorithm(data):
    """Implements the Find-S algorithm to find the most specific hypothesis."""
    # Extract feature columns and target column
    attributes = data.iloc[:, :-1].values  # All columns except last
    target = data.iloc[:, -1].values  # Last column (class labels)

    # Step 1: Initialize hypothesis with first positive example
    for i in range(len(target)):
        if target[i] == "Yes":  # Consider only positive examples
            hypothesis = attributes[i].copy()
            break
    # Step 2: Update hypothesis based on other positive examples
    for i in range(len(target)):
        if target[i] == "Yes":
            for j in range(len(hypothesis)):
                if hypothesis[j] != attributes[i][j]:
                    hypothesis[j] = '?'  # Generalize inconsistent attributes
    return hypothesis
# Run Find-S Algorithm
final_hypothesis = find_s_algorithm(data)

# Print the learned hypothesis
print("Most Specific Hypothesis:", final_hypothesis)
```

Most Specific Hypothesis: ['Yes' '?' '?' '?']

Experiment 5

Develop a program to load the Iris dataset, Implement the k-Nearest Neighbors (k-NN) algorithm for classifying flowers based on their features. Split the dataset into training and testing sets and evaluate the model using metrics like accuracy and F1-score, Test it for different values of k (e.g. k=1, 3, 5) and evaluating the accuracy. Extend the k-NN algorithm to assign weights based on the distance of neighbors (e.g., weight = $1/d^2$). Compare the performance of weighted k-NN and regular k-NN on a synthetic or real-world dataset.

Introduction to k-Nearest Neighbors (k-NN)

What is k-NN?

- The **k-Nearest Neighbors (k-NN) algorithm** is a **supervised learning algorithm** used for both classification and regression. It classifies a data point based on the majority class among its nearest neighbors.
- It is also called a **lazy learner algorithm** because it does not learn from the training set immediately instead it stores the dataset and at the time of classification, it performs an action on the dataset

Importance of k-NN

- **Simple and effective** for classification tasks.
- **Non-parametric** (makes no assumptions about the data distribution).
- **Handles multi-class classification** with ease.

---

k-Nearest Neighbors (k-NN) Algorithm

The k-NN algorithm classifies a given data point by considering the majority class among its k nearest neighbors in feature space. The algorithm follows these steps:

1. Choose the number of neighbors (k).
2. Calculate the distance between the new data point and all points in the dataset.
3. Select the k nearest neighbors based on the computed distances.
4. Determine the majority class among the k neighbors.
5. Assign the new data point to the majority class.

Choosing the Value of k

- If k is too small (e.g., k=1), the model may be sensitive to noise and overfit the data.
- If k is too large (e.g., k=50), the model may misclassify points due to the influence of distant neighbors.
- A common approach is to test multiple values and select the best k based on validation accuracy.

Distance Metrics in k-NN

The choice of distance metric affects classification performance. Commonly used distance metrics include:

`- **Euclidean Distance** (Most commonly used)

- **Manhattan Distance**
- **Minkowski Distance**
- `Cosine Similarity** (Used in text-based applications)
  - The most common method is **Euclidean Distance**:
    `d =√ [(x2 – x1)2 + (y2 – y1)2]`

Training and Evaluating k-NN

Dataset Splitting

To ensure that the model generalizes well, the dataset is divided into:

- Training Set (e.g., 80%): Used to train the k-NN model.
- Testing Set (e.g., 20%): Used to evaluate the model's performance.

formance Metrics

We use multiple evaluation metrics to assess the model:

- Accuracy: Measures the proportion of correctly classified instances.
- F1-Score: The harmonic mean of precision and recall.
- Confusion Matrix: Displays the counts of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN).

Weighted k-Nearest Neighbors (Weighted k-NN)

A variation of k-NN is Weighted k-NN, where closer neighbors contribute more to the classification than distant ones. Instead of treating all k neighbors equally, each neighbor is assigned a weight based on its distance:

Common Weighting Schemes

- Inverse Distance Weighting: $w = \frac{1}{d^2}$
  - Smaller distances get higher weights, ensuring closer points have a greater influence on classification.
- Gaussian Weighting: $w = e^{-d^2}$
  - Uses an exponential decay function to reduce the influence of distant points.

Advantages of k-NN

✓ **Simple and easy to implement**.
✓ **No training phase**—all computation happens during prediction.
✓ **Works well for multi-class classification problems**.
✓ **Can model complex decision boundaries** when k is appropriately chosen.

Limitations of k-NN

✗ **Computationally expensive** for large datasets.
✗ **Performance depends on the choice of k**.
✗ **Sensitive to irrelevant or redundant features**.
✗ **Memory-intensive** since all training data needs to be stored.

In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, f1_score
```

In [2]:
```
# Load the Iris dataset
iris = sns.load_dataset("iris")
```

In [3]:
```
iris.head()
```

Out[3]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

In [4]:
```
iris.shape
```

Out[4]:
```
(150, 5)
```

In [5]:
```
# Basic Data Exploration
print("\nBasic Information about Dataset:")
print(iris.info())  # Overview of dataset
```

```
Basic Information about Dataset:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   sepal_length   150 non-null    float64
 1   sepal_width    150 non-null    float64
 2   petal_length   150 non-null    float64
 3   petal_width    150 non-null    float64
 4   species        150 non-null    object 
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
None
```

In [7]:
```
# Summary Statistics
print("\nSummary Statistics:")
print(iris.describe())  # Summary statistics of dataset
```

```
Summary Statistics:
       sepal_length  sepal_width  petal_length  petal_width
count    150.000000   150.000000    150.000000   150.000000
mean       5.843333     3.057333      3.758000     1.199333
std        0.828066     0.435866      1.765298     0.762238
min        4.300000     2.000000      1.000000     0.100000
25%        5.100000     2.800000      1.600000     0.300000
50%        5.800000     3.000000      4.350000     1.300000
75%        6.400000     3.300000      5.100000     1.800000
max        7.900000     4.400000      6.900000     2.500000
```

In [9]:
```
# Check for missing values
print("\nMissing Values in Each Column:")
print(iris.isnull().sum())  # Count of missing values
```

```
Missing Values in Each Column:
sepal_length     0
sepal_width      0
petal_length     0
petal_width      0
species          0
dtype: int64
```

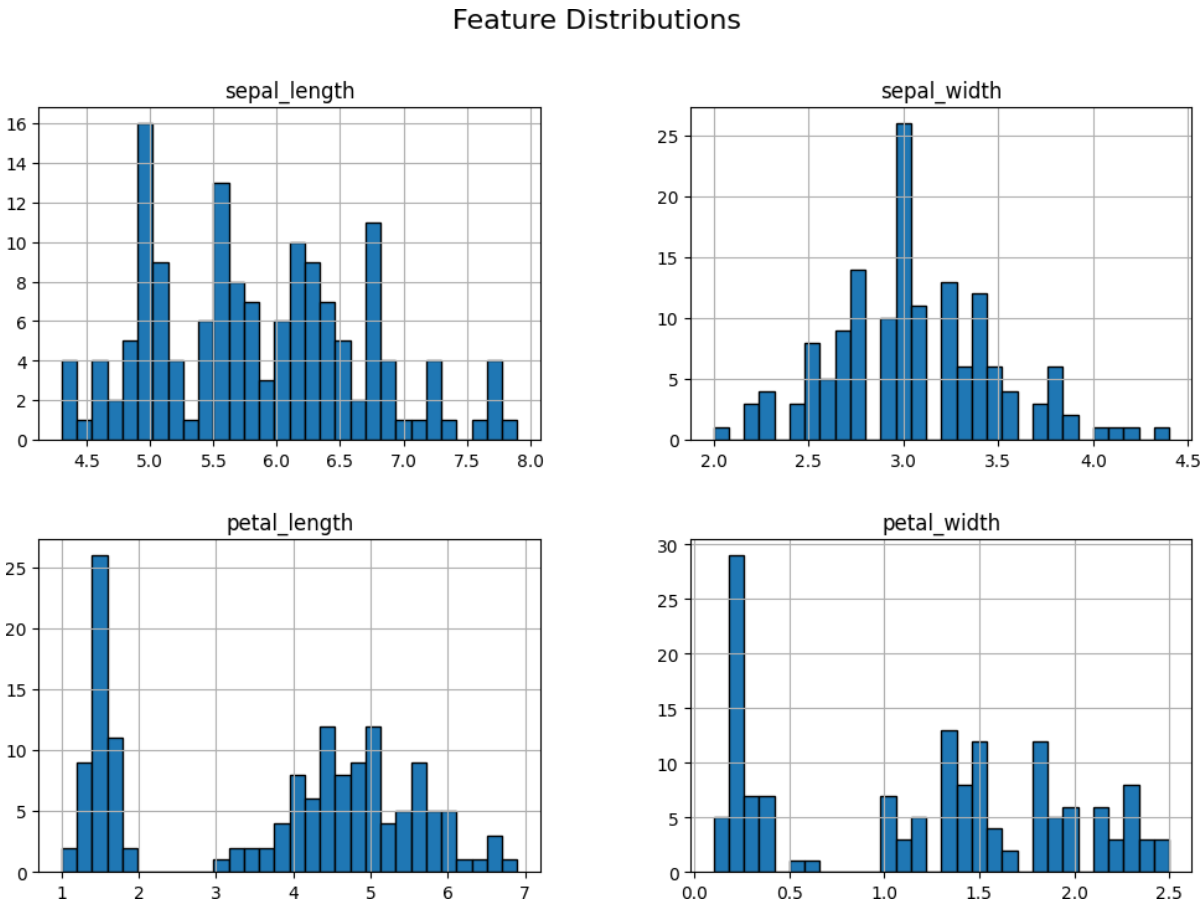
In [10]:
```
iris.duplicated().sum()
```

Out[10]: np.int64(1)

Univariate Analysis

In [12]:
```
# Histograms for distribution of features
plt.figure(figsize=(12,  8))
iris.hist(figsize=(12, 8), bins=30, edgecolor='black')
plt.suptitle("Feature Distributions", fontsize=16)
plt.show()
```

```
<Figure size 1200x800 with 0 Axes>
```



Inferences from Histograms:

1. **Sepal Length**:

- The distribution of sepal length appears to be roughly normal with a slight skew towards the right.
- Most of the sepal lengths fall between 4.5 and 7.5 cm.

2. **Sepal Width**:

- The distribution of sepal width is also roughly normal but with a slight skew towards the left.
- Most of the sepal widths fall between 2.5 and 3.5 cm.

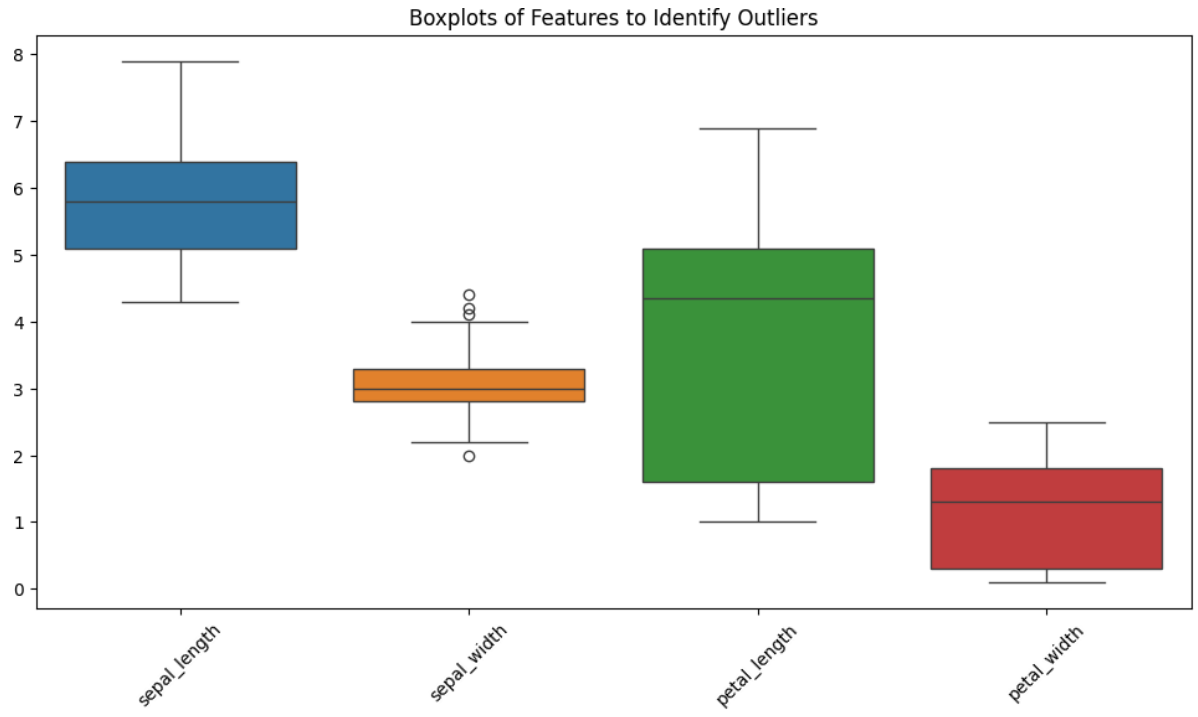
3. **Petal Length**:

- The distribution of petal length is more spread out and shows a clear separation between different species.
- There are distinct peaks indicating the presence of different species with varying petal lengths.

4. **Petal Width**:

- Similar to petal length, the distribution of petal width shows clear separation between species.
- There are distinct peaks indicating the presence of different species with varying petal widths.

In [14]:
```
# Boxplots for outlier detection
plt.figure(figsize=(12, 6))
sns.boxplot(data=iris)
plt.xticks(rotation=45)
plt.title("Boxplots of Features to Identify Outliers")
plt.show()
```



Inferences from Boxplots:

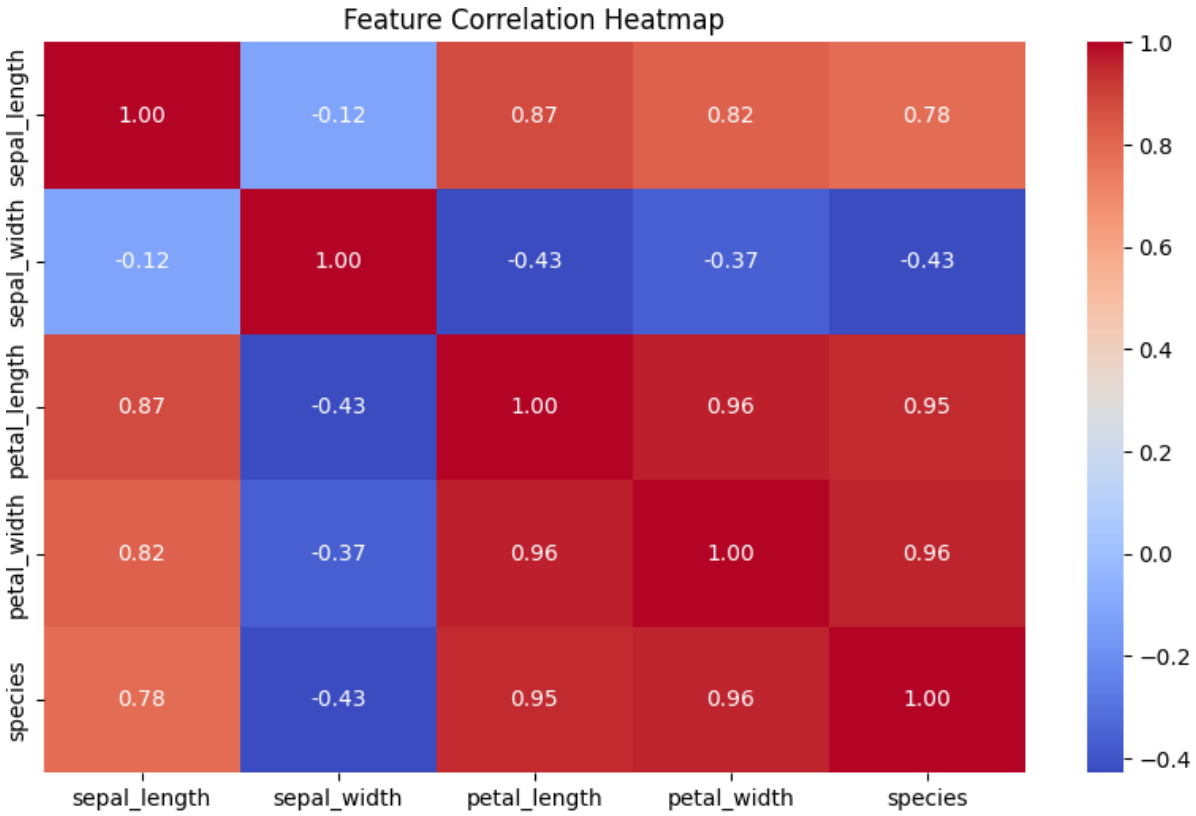
1. **Sepal Length**:
   - There are a few outliers in the sepal length distribution.
   - The median sepal length is around 5.8 cm, with the interquartile range (IQR) between 5.1 and 6.4 cm.
2. **Sepal Width**:
   - There are several outliers in the sepal width distribution.
   - The median sepal width is around 3.0 cm, with the IQR between 2.8 and 3.3 cm.
3. **Petal Length**:
   - The petal length distribution shows clear separation between species, with minimal overlap.
   - The median petal length varies significantly between species, indicating it is a good feature for classification.
4. **Petal Width**:
   - Similar to petal length, the petal width distribution shows clear separation between species.
   - The median petal width varies significantly between species, indicating it is also a good feature for classification.

Heatmap: Visualizing the Correlation Matrix

In [19]:
```
num_col = iris.select_dtypes(include=[np.number]).columns
cat_col = iris.select_dtypes(include=['object']).columns
print(f"numerical_data {num_col}")
print(f"categorical_data  {cat_col}")
```

```
numerical_data Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
       'species'],
      dtype='object')
categorical_data  Index([], dtype='object')
```

In [21]:
```
# Correlation Matrix
plt.figure(figsize=(10, 6))
corr_matrix = iris[num_col].corr('pearson')
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title("Feature Correlation Heatmap")
plt.show()
```



Based on the heatmap of the correlation matrix, we can infer the following:

1. **Sepal Length**:
   - Positively correlated with petal length (0.87) and petal width (0.82).
   - Weak negative correlation with sepal width (-0.12).
2. **Sepal Width**:
   - Weak negative correlation with sepal length (-0.12), petal length (-0.43), and petal width (-0.37).
3. **Petal Length**:
   - Strong positive correlation with sepal length (0.87) and petal width (0.96).
   - Weak negative correlation with sepal width (-0.43).
4. **Petal Width**:
   - Strong positive correlation with petal length (0.96) and sepal length (0.82).
   - Weak negative correlation with sepal width (-0.37).
5. **Species**:
   - Strong positive correlation with petal length (0.95) and petal width (0.96).
   - Moderate positive correlation with sepal length (0.78).
   - Moderate negative correlation with sepal width (-0.43).

These correlations suggest that petal length and petal width are highly correlated with each other and with the species of the iris flower, making them important features for

classification. Sepal length also shows a moderate correlation with species, while sepal width has a weaker correlation with species.

In [15]:
```
# Encode target labels
label_encoder = LabelEncoder()
iris["species"] = label_encoder.fit_transform(iris["species"])
```

In [16]:
```
# Define features and target
X = iris.drop(columns=["species"])
y = iris["species"]
```

In [17]:
```
# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

In [22]:
```
# Function to evaluate k-NN for different values of k
def evaluate_knn(k_values, weights='uniform'):
    results = {}
    for k in k_values:
        knn = KNeighborsClassifier(n_neighbors=k, weights=weights)
        knn.fit(X_train, y_train)
        y_pred = knn.predict(X_test)
        accuracy = accuracy_score(y_test, y_pred)
        f1 = f1_score(y_test, y_pred, average='weighted')
        results[k] = {'accuracy': accuracy, 'f1_score': f1}
    return results
```

In [23]:
```
# Test for k = 1, 3, 5
k_values = [1, 3, 5]
regular_knn_results = evaluate_knn(k_values, weights='uniform')
weighted_knn_results = evaluate_knn(k_values, weights='distance')
```

In [26]:
```
# Convert results to DataFrame for comparison
results_df = pd.DataFrame.from_dict({
    'Regular k-NN': regular_knn_results,
    'Weighted k-NN': weighted_knn_results
}, orient='index').T
results_df
```

Out[26]:

	Regular k-NN	Weighted k-NN
1	{'accuracy': 1.0, 'f1_score': 1.0}	{'accuracy': 1.0, 'f1_score': 1.0}
3	{'accuracy': 1.0, 'f1_score': 1.0}	{'accuracy': 1.0, 'f1_score': 1.0}
5	{'accuracy': 1.0, 'f1_score': 1.0}	{'accuracy': 1.0, 'f1_score': 1.0}

Based on the results of the k-NN algorithm applied to the Iris dataset, we can derive the following key insight:

Both the regular k-NN and weighted k-NN classifiers achieved perfect accuracy and F1-scores for the tested values of k (1, 3, 5).

- This indicates that the Iris dataset is well-suited for classification using the k-NN algorithm, and the features (sepal length, sepal width, petal length, petal width) are highly discriminative for distinguishing between the different species of Iris flowers.
- The high performance of both regular and weighted k-NN suggests that the dataset is not significantly affected by the distance weighting, likely due to the clear separation between classes in the feature space.

Experiment 6

Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs

---

Introduction to Locally Weighted Regression (LWR)

What is Locally Weighted Regression?

Locally Weighted Regression (LWR) is a **non-parametric** machine learning algorithm that fits a regression model to a local subset of data points. Unlike traditional regression techniques, LWR does not assume a fixed set of parameters for the entire dataset but instead **assigns different weights to data points based on their distance from the target point**.

Importance of Locally Weighted Regression

- Handles **non-linearity** effectively.
- Provides **better flexibility** compared to global regression models.
- More **robust to outliers** due to localized weighting.
- Suitable for datasets where relationships between variables **vary locally**.

How Locally Weighted Regression Works

1. **Define the Weighting Function**
   - A kernel function (e.g., **Gaussian kernel**) is used to assign weights to data points:

$$w_i = e^{-\frac{(x - x_i)^2}{2\tau^2}}$$

   - Here, `τ` (tau) is the **bandwidth parameter** that controls the locality of weighting.
2. **Compute Localized Weights**
   - For a given query point `x`, assign weights to training points based on proximity.
3. **Fit a Local Model**
   - Solve a **weighted least squares** problem using the locally weighted dataset.
4. **Make Predictions**

- Compute the predicted value at `x` using the locally trained model.

Dataset Selection

For this experiment, we need a dataset with a **clear non-linear relationship** between independent and dependent variables. Some possible datasets include:

- **Synthetic Data:** Randomly generated non-linear data points.
- **Real-World Data:**
  - **Auto MPG Dataset:** Predict fuel efficiency based on engine displacement, horsepower, etc.
  - **California Housing Dataset:** Predict house prices based on features like location and area.
  - **Temperature vs. Time Series Data:** Forecast weather trends.

Steps for Implementing Locally Weighted Regression

1. Load the Dataset

- Choose a dataset with **one independent variable (x) and one dependent variable (y)**.

2. Apply the Locally Weighted Regression Algorithm

- Assign weights to each data point using a Gaussian kernel.
- Solve the weighted linear regression equation.

3. Experiment with Different Bandwidth Parameters (τ)

- **Small τ:** Model focuses on very close neighbors → **More variance, less bias** (risk of overfitting).
- **Large τ:** Model considers a broader range of points → **More bias, less variance** (risk of underfitting).

4. Visualize the Results

- **Scatter Plot of Data Points** to observe the actual distribution.
- **Fitted Curve from LWR** with different values of `τ` to compare model performance.

Advantages of Locally Weighted Regression

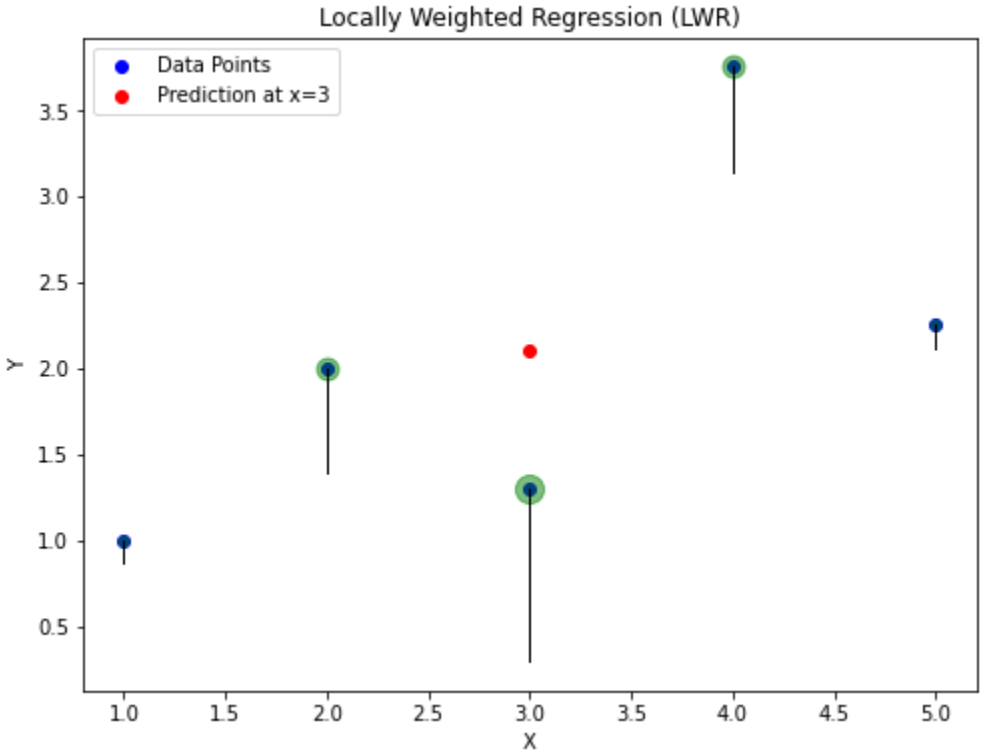
✓ **Captures complex relationships** between input and output variables.
✓ **Works well with small datasets** where global linear regression may not be suitable.
✓ **Does not assume a fixed functional form**, making it highly flexible.

Limitations of Locally Weighted Regression

❌ **Computationally expensive**: Must compute a separate model for each query point.
❌ **Sensitive to bandwidth parameter** ((\tau)): Choosing the wrong value can lead to overfitting or underfitting.
❌ **Not suitable for large datasets**: As the dataset size increases, the algorithm becomes impractical due to high computation time.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

def gaussian_kernel(x, x_query, tau):
    return np.exp(- (x - x_query) ** 2 / (2 * tau ** 2))

def locally_weighted_regression(X, y, x_query, tau):
    X_b = np.c_[np.ones(len(X)), X]  # Add bias term (Intercept)
    x_query_b = np.array([1, x_query])  # Query point with bias term

    W = np.diag(gaussian_kernel(X, x_query, tau))  # Compute weights

    # Compute theta: (X^T W X)^-1 X^T W y
    theta = np.linalg.inv(X_b.T @ W @ X_b) @ X_b.T @ W @ y

    return x_query_b @ theta  # Return prediction

# Dataset
X = np.array([1, 2, 3, 4, 5])
y = np.array([1, 2, 1.3, 3.75, 2.25])

# Query point
x_query = 3  # Point at which we perform LWR

# Bandwidth parameter
tau = 1.0

# Compute prediction
y_pred = locally_weighted_regression(X, y, x_query, tau)

# Visualizing
plt.figure(figsize=(8, 6))
plt.scatter(X, y, color='blue', label='Data Points')
plt.scatter(x_query, y_pred, color='red', label=f'Prediction at x={x_query}')

# Plot weights effect
weights = gaussian_kernel(X, x_query, tau)
for i in range(len(X)):
    plt.plot([X[i], X[i]], [y[i], y[i] - weights[i]], 'k-', lw=1)
    plt.scatter(X[i], y[i], s=weights[i] * 200, color='green', alpha=0.5)

plt.title("Locally Weighted Regression (LWR)")
plt.xlabel("X")
plt.ylabel("Y")
```

```
plt.legend()
plt.show()
```

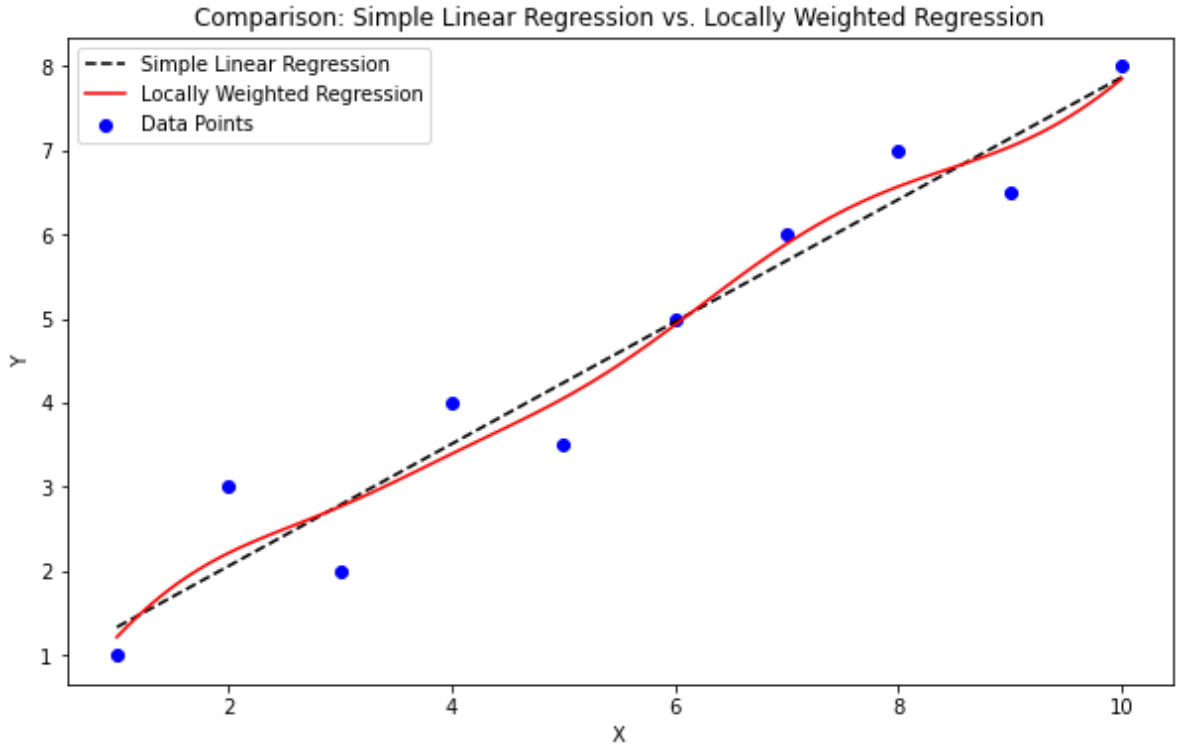


Explanation of the Code

gaussian_kernel(x, x_query, tau): Computes weights using the Gaussian kernel.
locally_weighted_regression(X, y, x_query, tau):
Computes the weight matrix WW.
Solves for θ using weighted least squares.
Predicts yy for the query point xqxq.
Visualization:
Data points (blue dots).
Prediction at xq=3xq=3 (red dot).
Weight influence is shown using vertical lines and green bubbles

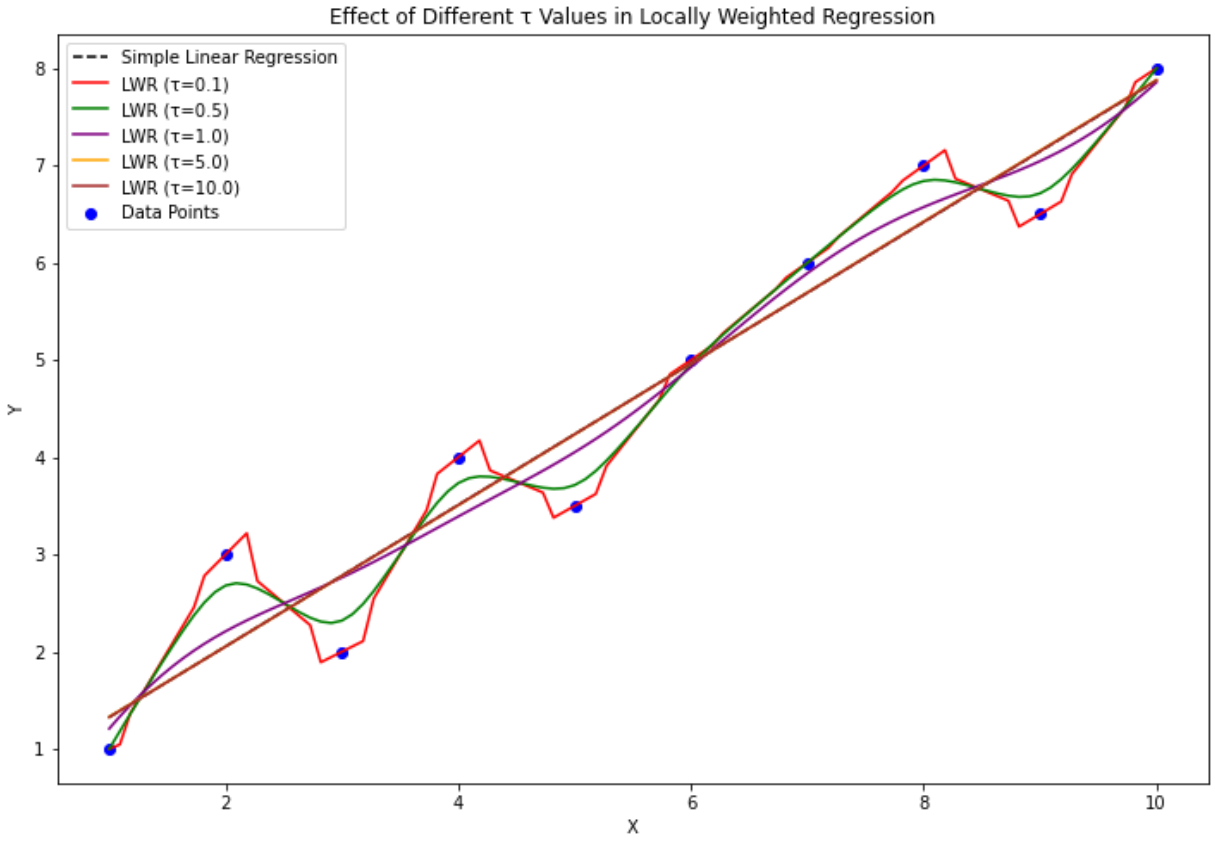
In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

def gaussian_kernel(x, x_query, tau):
    return np.exp(- (x - x_query) ** 2 / (2 * tau ** 2))

def locally_weighted_regression(X, y, x_query, tau):
    X_b = np.c_[np.ones(len(X)), X]  # Add bias term (Intercept)
    x_query_b = np.array([1, x_query])  # Query point with bias term

    W = np.diag(gaussian_kernel(X, x_query, tau))  # Compute weights

    # Compute theta: (X^T W X)^-1 X^T W y
```

```
    theta = np.linalg.inv(X_b.T @ W @ X_b) @ X_b.T @ W @ y

    return x_query_b @ theta  # Return prediction

# Complex Dataset
X = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
y = np.array([1, 3, 2, 4, 3.5, 5, 6, 7, 6.5, 8])

# Query points for LWR
X_query = np.linspace(1, 10, 100)

tau = 1.0 # Bandwidth parameter

# Compute LWR predictions
y_lwr = np.array([locally_weighted_regression(X, y, x_q, tau) for x_q in X_query])

# Simple Linear Regression
lin_reg = LinearRegression()
X_reshaped = X.reshape(-1, 1)
lin_reg.fit(X_reshaped, y)
y_lin = lin_reg.predict(X_query.reshape(-1, 1))

# Visualizing
plt.figure(figsize=(10, 6))
plt.scatter(X, y, color='blue', label='Data Points')
plt.plot(X_query, y_lin, color='black', linestyle='dashed', label='Simple Linear Re
plt.plot(X_query, y_lwr, color='red', label='Locally Weighted Regression')
plt.title("Comparison: Simple Linear Regression vs. Locally Weighted Regression")
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()
plt.show()
```



In [5]:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

def gaussian_kernel(x, x_query, tau):
    return np.exp(- (x - x_query) ** 2 / (2 * tau ** 2))

def locally_weighted_regression(X, y, x_query, tau):
    X_b = np.c_[np.ones(len(X)), X]  # Add bias term (Intercept)
    x_query_b = np.array([1, x_query])  # Query point with bias term

    W = np.diag(gaussian_kernel(X, x_query, tau))  # Compute weights

    # Compute theta using pseudo-inverse to avoid singular matrix error
    theta = np.linalg.pinv(X_b.T @ W @ X_b) @ X_b.T @ W @ y

    return x_query_b @ theta  # Return prediction

# Complex Dataset
X = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
y = np.array([1, 3, 2, 4, 3.5, 5, 6, 7, 6.5, 8])

# Query points for LWR
X_query = np.linspace(1, 10, 100)

tau_values = [0.1, 0.5, 1.0, 5.0, 10.0]  # Different bandwidth values

# Simple Linear Regression
lin_reg = LinearRegression()
X_reshaped = X.reshape(-1, 1)
lin_reg.fit(X_reshaped, y)
y_lin = lin_reg.predict(X_query.reshape(-1, 1))

# Visualizing
plt.figure(figsize=(12, 8))
plt.scatter(X, y, color='blue', label='Data Points')
plt.plot(X_query, y_lin, color='black', linestyle='dashed', label='Simple Linear Re

# Plot LWR for different tau values
colors = ['red', 'green', 'purple', 'orange', 'brown']
for tau, color in zip(tau_values, colors):
    y_lwr = np.array([locally_weighted_regression(X, y, x_q, tau) for x_q in X_quer
    plt.plot(X_query, y_lwr, color=color, label=f'LWR (τ={tau})')

plt.title("Effect of Different τ Values in Locally Weighted Regression")
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()
plt.show()
```



The tau (τ) parameter in your code is the bandwidth for the Gaussian kernel, which controls how much influence nearby points have in the Locally Weighted Regression (LWR). Here's what it does:

```
Determines the Weight Decay:
    If τ is small, only very nearby points contribute
significantly, making LWR behave like a very local model (more
sensitive to noise).
    If τ is large, more distant points contribute significantly,
making LWR behave more like global linear regression.

Controls the Model Complexity:
    A small τ → Highly flexible model, more prone to overfitting.
    A large τ → More smoothing, leading to a simpler model (can
underfit if too large).
```

Example Effect of Tau

```
τ = 0.1 → LWR behaves almost like a nearest-neighbor model (highly
local, very wiggly curve).
τ = 1.0 → Moderate smoothing, a good balance between flexibility
and generalization.
τ = 10 → LWR behaves like ordinary least squares regression (all
points are weighted almost equally).
```

Experiment 7 A:

Develop a program to demonstrate the working of Linear Regression and Polynomial Regression. Use Boston Housing Dataset for Linear Regression and Auto MPG Dataset (for vehicle fuel efficiency prediction) for Polynomial Regression.

Introduction to Regression Analysis

What is Regression?

Regression is a fundamental statistical and machine learning technique used to model relationships between variables. It helps in predicting a **dependent variable (target)** based on one or more **independent variables (features)**.

Types of Regression Models

1. **Linear Regression** – Assumes a linear relationship between independent and dependent variables.
2. **Polynomial Regression** – Extends linear regression by introducing polynomial terms to capture non-linearity.

Linear Regression

Definition

Linear Regression models the relationship between an independent variable (`x`) and a dependent variable (`y`) using a straight-line equation:

```
y = mx + c
```

where:

- `m` is the **slope** (coefficient) of the line,
- `c` is the **intercept**,
- `x` is the independent variable,
- `y` is the dependent variable (predicted value).

Working of Linear Regression

1. **Identify the best-fitting line**: Uses the **least squares method** to minimize the error between actual and predicted values.

2. **Compute the cost function**: Measures how well the model fits the data using **Mean Squared Error (MSE)**
3. **Optimize the model parameters**: Uses **Gradient Descent** or other optimization techniques to find the best `m` and `c`.

Applications of Linear Regression

- Predicting sales revenue based on advertising spend.
- Estimating house prices based on size and location.
- Forecasting demand in supply chain management.

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
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.preprocessing import StandardScaler

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

In [3]:
```
data = pd.read_csv(r"C:\Users\vijay\Desktop\Machine Learning Course Batches\FDP_ML_
```

- CRIM: Per capita crime rate by town.
- ZN: Proportion of residential land zoned for lots over 25,000 square feet.
- INDUS: Proportion of non-retail business acres per town.
- CHAS: Charles River dummy variable (1 if tract bounds river; 0 otherwise).
- NOX: Nitric oxide concentration (parts per 10 million).
- RM: Average number of rooms per dwelling.
- AGE: Proportion of owner-occupied units built before 1940.
- DIS: Weighted distances to five Boston employment centers.
- RAD: Index of accessibility to radial highways.
- TAX: Full-value property-tax rate per $10,000.
- PTRATIO: Pupil-teacher ratio by town.
- B: 1000(Bk − 0.63)2, where Bk is the proportion of Black residents by town.
- LSTAT: Percentage of the lower status of the population.
- MEDV: Median value of owner-occupied homes in $1000s.

In [4]:
```
data.head()
```

Out[4]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90

In [5]:
```
data.shape
```

Out[5]: (506, 14)

In [6]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   CRIM      486 non-null    float64
 1   ZN        486 non-null    float64
 2   INDUS     486 non-null    float64
 3   CHAS      486 non-null    float64
 4   NOX       506 non-null    float64
 5   RM        506 non-null    float64
 6   AGE       486 non-null    float64
 7   DIS       506 non-null    float64
 8   RAD       506 non-null    int64  
 9   TAX       506 non-null    int64  
 10  PTRATIO   506 non-null    float64
 11  B         506 non-null    float64
 12  LSTAT     486 non-null    float64
 13  MEDV      506 non-null    float64
dtypes: float64(12), int64(2)
memory usage: 55.5 KB
```

- The dataset contains 506 entries and 14 columns, with 6 columns (CRIM, ZN, INDUS, CHAS, AGE, LSTAT) having 20 missing values each.
- Most columns are continuous (float64), while RAD and TAX are discrete (int64).
- MEDV (median home value) is the target variable, likely influenced by features like RM (average rooms) and LSTAT (lower-status population).
- Missing values need to be addressed through imputation or by dropping rows with missing data.
- Exploratory analysis and modeling can help understand feature relationships and predict MEDV.

In [7]:
```
data.nunique()
```

```
Out[7]: CRIM        484
        ZN           26
        INDUS        76
        CHAS          2
        NOX          81
        RM          446
        AGE         348
        DIS         412
        RAD           9
        TAX          66
        PTRATIO      46
        B           357
        LSTAT       438
        MEDV        229
        dtype: int64
```

```
In [8]: data.CHAS.unique()
```

```
Out[8]: array([ 0., nan,  1.])
```

```
In [9]: data.ZN.unique()
```

```
Out[9]: array([ 18. ,   0. ,  12.5,  75. ,  21. ,  90. ,  85. , 100. ,  25. ,
                17.5,  80. ,   nan,  28. ,  45. ,  60. ,  95. ,  82.5,  30. ,
                22. ,  20. ,  40. ,  55. ,  52.5,  70. ,  34. ,  33. ,  35. ])
```

Data Cleaning

Checking Null values

data.isnull() - Returns a DataFrame of the same shape as data, where each element is True if it's NaN and False otherwise.

.sum() - Sums up the True values (which are treated as 1 in Python) column-wise, giving the total count of missing values for each column.

```
In [10]: data.isnull().sum()
```

```
Out[10]: CRIM        20
         ZN          20
         INDUS       20
         CHAS        20
         NOX          0
         RM           0
         AGE         20
         DIS          0
         RAD          0
         TAX          0
         PTRATIO      0
         B            0
         LSTAT       20
         MEDV         0
         dtype: int64
```

In [11]:
```
data.duplicated().sum()
```

Out[11]:
```
0
```

In [12]:
```
df = data.copy()
```

In [13]:
```
df['CRIM'].fillna(df['CRIM'].mean(),  inplace=True)
df['ZN'].fillna(df['ZN'].mean(),  inplace=True)
df['CHAS'].fillna(df['CHAS'].mode()[0], inplace=True)
df['INDUS'].fillna(df['INDUS'].mean(), inplace=True)
df['AGE'].fillna(df['AGE'].median(), inplace=True)  # Median is often preferred for
df['LSTAT'].fillna(df['LSTAT'].median(),  inplace=True)
```

In [14]:
```
df.isnull().sum()
```

Out[14]:
```
CRIM       0
ZN         0
INDUS      0
CHAS       0
NOX        0
RM         0
AGE        0
DIS        0
RAD        0
TAX        0
PTRATIO    0
B          0
LSTAT      0
MEDV       0
dtype: int64
```

In [15]:
```
df.head()
```

Out[15]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90

In [16]:
```
df['CHAS'] = df['CHAS'].astype('int')
```

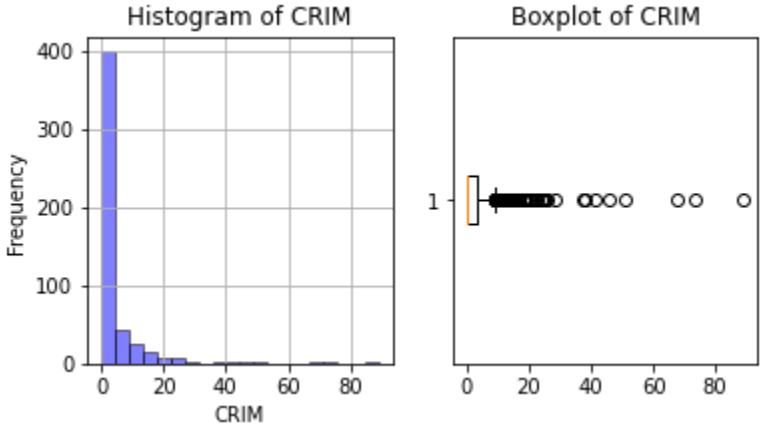
In [17]:
```
df.describe().T
```

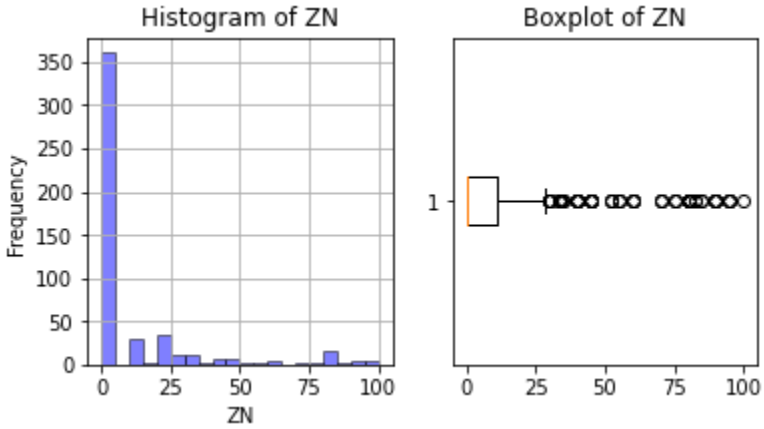
Out[17]:

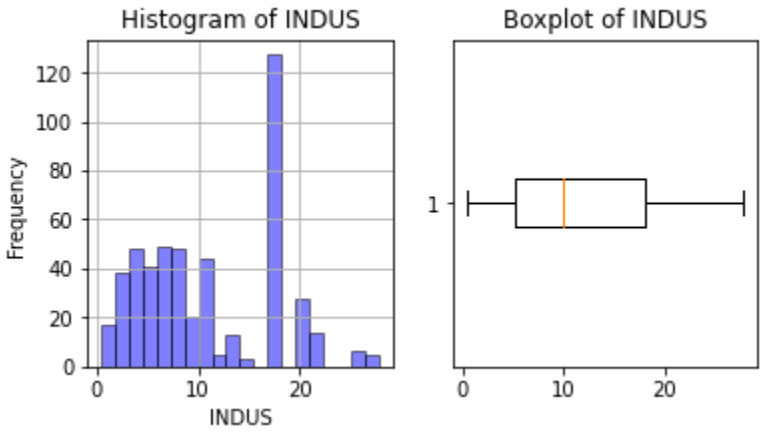
	count	mean	std	min	25%	50%	75%	
CRIM	506.0	3.611874	8.545770	0.00632	0.083235	0.29025	3.611874	8
ZN	506.0	11.211934	22.921051	0.00000	0.000000	0.00000	11.211934	10
INDUS	506.0	11.083992	6.699165	0.46000	5.190000	9.90000	18.100000	2
CHAS	506.0	0.067194	0.250605	0.00000	0.000000	0.00000	0.000000	
NOX	506.0	0.554695	0.115878	0.38500	0.449000	0.53800	0.624000	
RM	506.0	6.284634	0.702617	3.56100	5.885500	6.20850	6.623500	
AGE	506.0	68.845850	27.486962	2.90000	45.925000	76.80000	93.575000	10
DIS	506.0	3.795043	2.105710	1.12960	2.100175	3.20745	5.188425	1
RAD	506.0	9.549407	8.707259	1.00000	4.000000	5.00000	24.000000	2
TAX	506.0	408.237154	168.537116	187.00000	279.000000	330.00000	666.000000	71
PTRATIO	506.0	18.455534	2.164946	12.60000	17.400000	19.05000	20.200000	2
B	506.0	356.674032	91.294864	0.32000	375.377500	391.44000	396.225000	39
LSTAT	506.0	12.664625	7.017219	1.73000	7.230000	11.43000	16.570000	3
MEDV	506.0	22.532806	9.197104	5.00000	17.025000	21.20000	25.000000	5

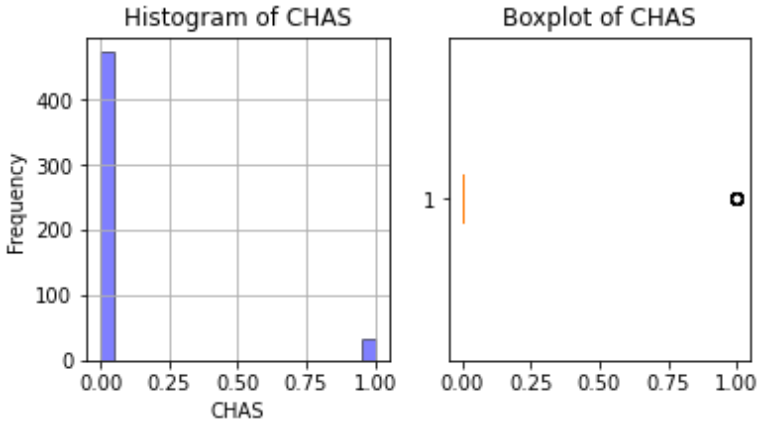
In [18]:

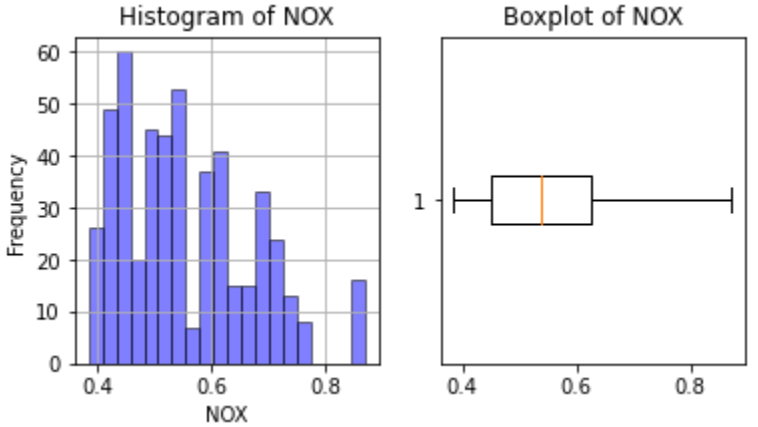
```
for i in df.columns:
    plt.figure(figsize=(6,3))

    plt.subplot(1, 2, 1)
    df[i].hist(bins=20, alpha=0.5, color='b',edgecolor='black')
    plt.title(f'Histogram of {i}')
    plt.xlabel(i)
    plt.ylabel('Frequency')

    plt.subplot(1, 2, 2)
    plt.boxplot(df[i], vert=False)
    plt.title(f'Boxplot of {i}')

    plt.show()
```

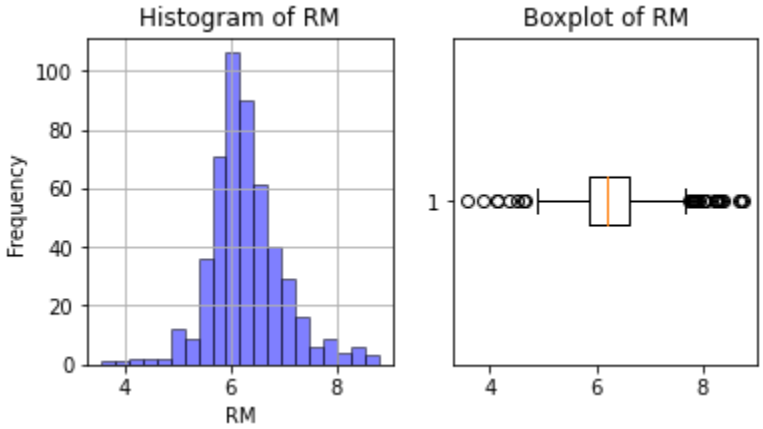

Histogram of CRIM
Frequency
CRIM
Boxplot of CRIM
1

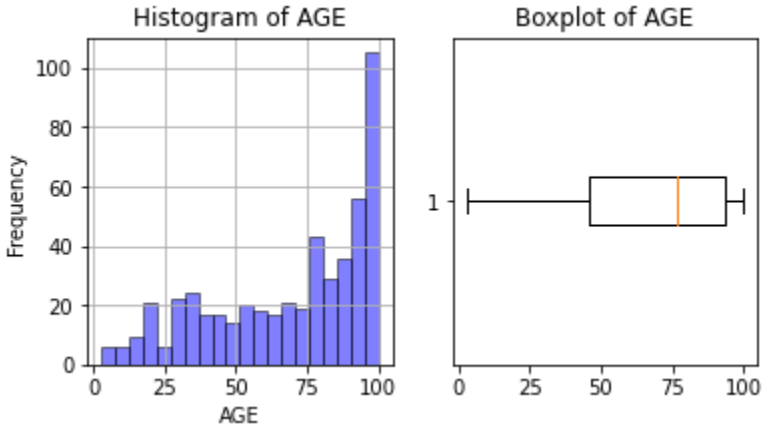

Histogram of ZN
Frequency
ZN
Boxplot of ZN
1

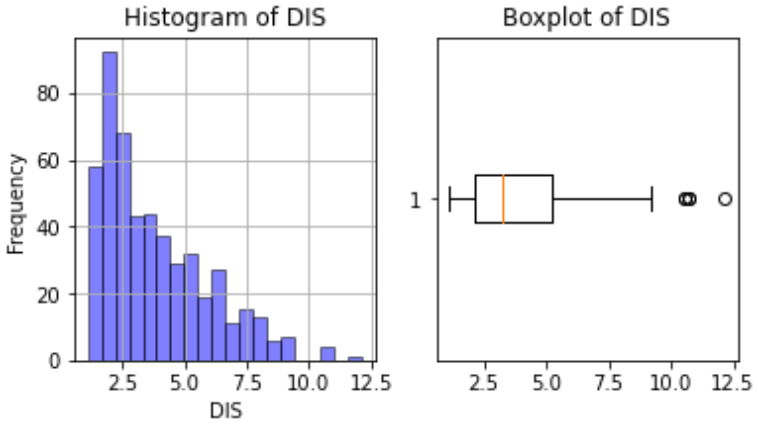

Histogram of INDUS
Frequency
INDUS
Boxplot of INDUS
1

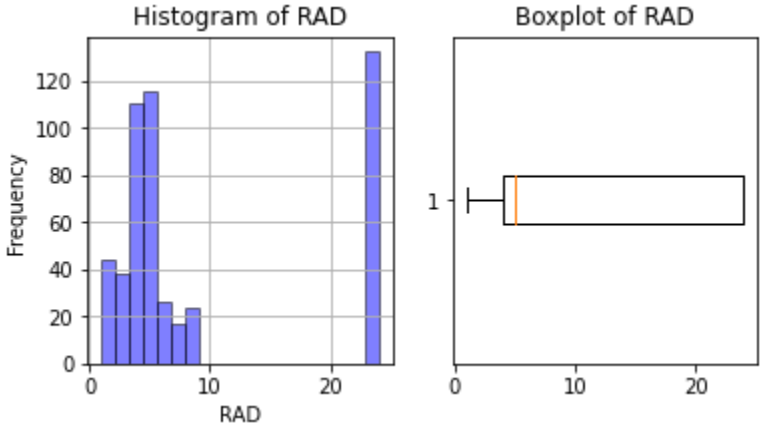

Histogram of CHAS
Frequency
CHAS
Boxplot of CHAS
1


Histogram of NOX
Frequency
NOX
Boxplot of NOX

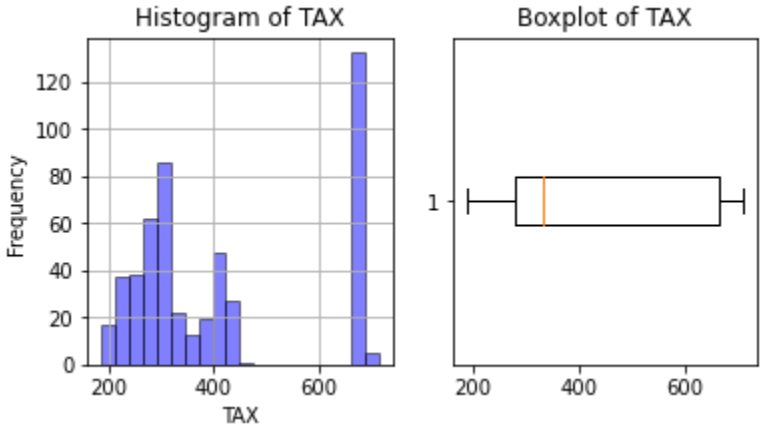

Histogram of RM
Frequency
RM
Boxplot of RM


Histogram of AGE
Frequency
AGE
Boxplot of AGE

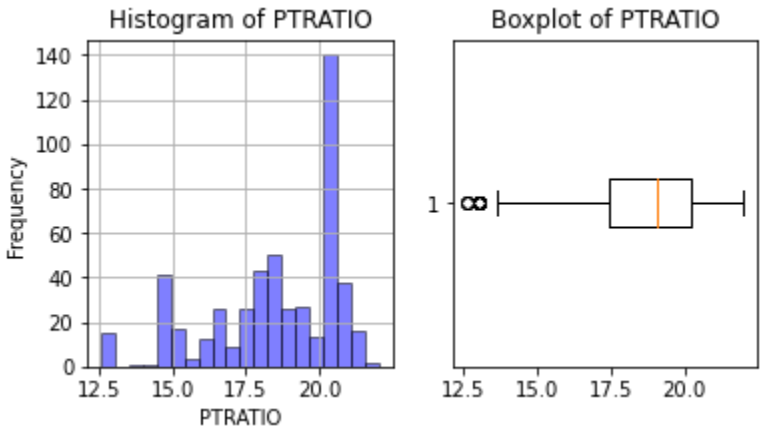

Histogram of DIS
Frequency
DIS
Boxplot of DIS


Histogram of RAD
Frequency
RAD
0
20
40
60
80
100
120
0
10
20

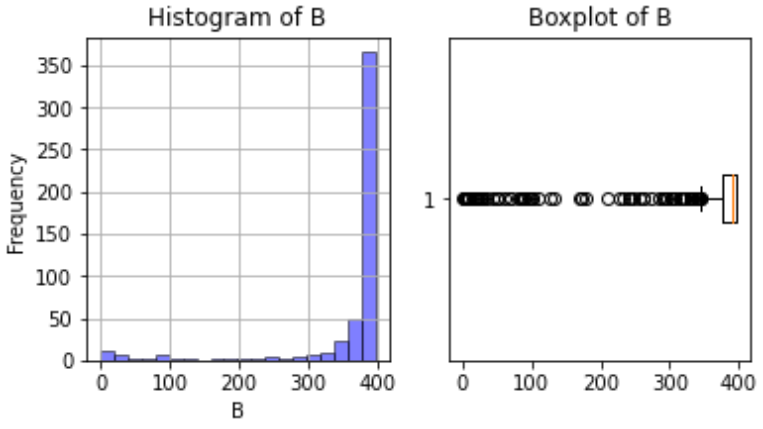
Boxplot of RAD
1
0
10
20


Histogram of TAX
Frequency
TAX
0
20
40
60
80
100
120
200
400
600

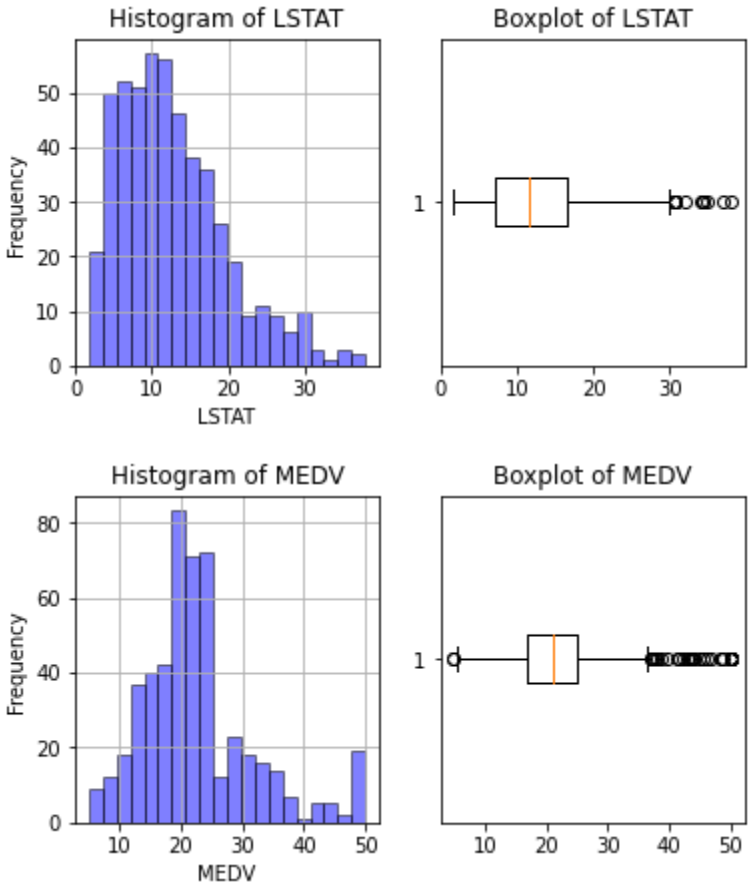
Boxplot of TAX
1
200
400
600


Histogram of PTRATIO
Frequency
PTRATIO
0
20
40
60
80
100
120
140
12.5
15.0
17.5
20.0

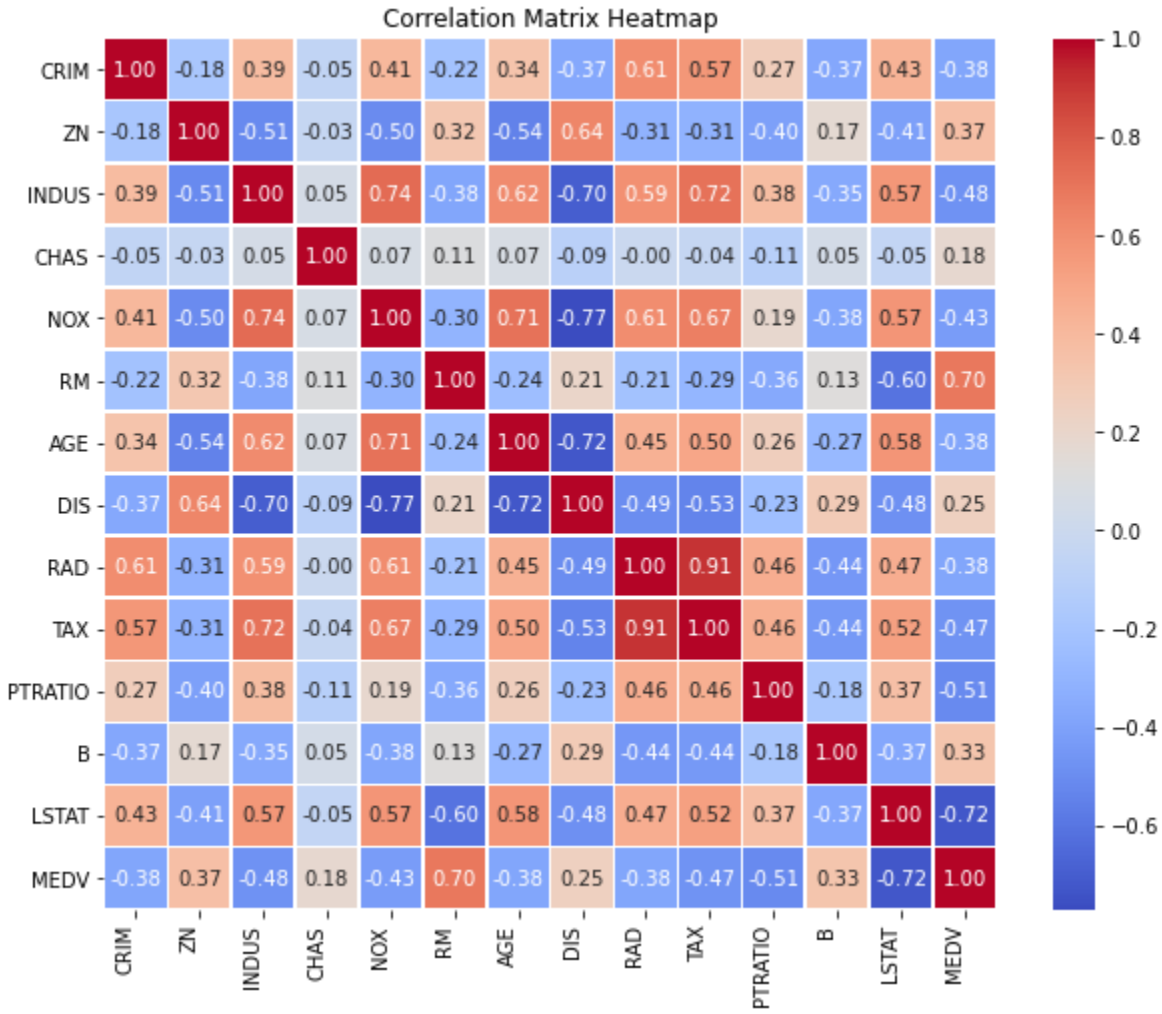
Boxplot of PTRATIO
1
12.5
15.0
17.5
20.0


Histogram of B
Frequency
B
0
50
100
150
200
250
300
350
0
100
200
300
400

Boxplot of B
1
0
100
200
300
400



In [19]:
```
corr = df.corr(method='pearson')

plt.figure(figsize=(10, 8))
sns.heatmap(corr, annot=True, cmap="coolwarm", fmt=".2f", linewidths=0.5)
plt.xticks(rotation=90, ha='right')
plt.yticks(rotation=0)
plt.title("Correlation Matrix Heatmap")
plt.show()
```



In [20]:
```
X = df.drop('MEDV', axis=1)  # All columns except 'MEDV'
y = df['MEDV']  # Target variable
```

Why Use StandardScaler?

- Improved model performance: Linear models assume that features are normally distributed around the mean. Scaling the data can make the algorithm converge faster and produce more accurate predictions.
- Prevents bias due to feature magnitude: Features with larger numeric ranges (like TAX or CRIM) may dominate the model if not scaled properly, especially in regularized models. While standard linear regression may not be heavily affected, scaling ensures more consistent results.

In [21]:
```
# Scale the features
scale = StandardScaler()
X_scaled  = scale.fit_transform(X)
```

In [22]:
```
# Split the data into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled , y, test_size=0.2, ra
```

In [23]:
```
# Initialize the linear regression model
model = LinearRegression()

# Fit the model on the training data
model.fit(X_train, y_train)
```

Out[23]:
```
▾ LinearRegression

LinearRegression()
```

In [24]:
```
# Predict on the test set
y_pred = model.predict(X_test)
y_pred
```

Out[24]:
```
array([28.99719439, 36.56606809, 14.51022803, 25.02572187, 18.42885474,
       23.02785726, 17.95437605, 14.5769479 , 22.14430832, 20.84584632,
       25.15283588, 18.55925182, -5.69168071, 21.71242445, 19.06845707,
       25.94275348, 19.70991322,  5.85916505, 40.9608103 , 17.21528576,
       25.36124981, 30.26007975, 11.78589412, 23.48106943, 17.35338161,
       15.13896898, 21.61919056, 14.51459386, 23.17246824, 19.40914754,
       22.56164985, 25.21208496, 25.88782605, 16.68297496, 16.44747174,
       16.65894826, 31.10314158, 20.25199803, 24.38567686, 23.09800032,
       14.47721796, 32.36053979, 43.01157914, 17.61473728, 27.60723089,
       16.43366912, 14.25719607, 26.0854729 , 19.75853278, 30.15142187,
       21.01932313, 33.72128781, 16.39180467, 26.36438908, 39.75793372,
       22.02419633, 18.39453126, 32.81854401, 25.370573  , 12.82224665,
       22.76128341, 30.73955199, 31.34386371, 16.27681305, 20.36945226,
       17.23156773, 20.15406451, 26.15613066, 30.92791361, 11.42177654,
       20.89590447, 26.58633798, 11.01176073, 12.76831709, 23.73870867,
        6.37180464, 21.6922679 , 41.74800223, 18.64423785,  8.82325704,
       20.96406016, 13.20179007, 20.99146149,  9.17404063, 23.0011185 ,
       32.41062673, 18.99778065, 25.56204885, 28.67383635, 19.76918944,
       25.94842754,  5.77674362, 19.514431  , 15.22571165, 10.87671123,
       20.08359505, 23.77725749,  0.05985008, 13.56333825, 16.1215622 ,
       22.74200442, 24.36218289])
```

In [25]:
```
# Calculate Mean Squared Error
mse = mean_squared_error(y_test, y_pred)

# Calculate Root Mean Squared Error (RMSE)
rmse = np.sqrt(mse)

# Calculate R-squared value
r2 = r2_score(y_test, y_pred)

print(f'Mean Squared Error: {mse}')
print(f'Root Mean Squared Error: {rmse}')
print(f'R-squared: {r2}')
```

```
Mean Squared Error: 24.944071172175573
Root Mean Squared Error: 4.99440398567993
R-squared: 0.6598556613717497
```

Experiment 7 B

Develop a program to demonstrate the working of Linear Regression and Polynomial Regression. Use Boston Housing Dataset for Linear Regression and Auto MPG Dataset (for vehicle fuel efficiency prediction) for Polynomial Regression.

Polynomial Regression

Definition

Polynomial regression is a type of regression analysis used in statistics and machine learning when the relationship between the independent variable (input) and the dependent variable (output) is not linear. While simple linear regression models the relationship as a straight line, polynomial regression allows for more flexibility by fitting a polynomial equation to the data.

Polynomial Regression is an extension of Linear Regression where the relationship between variables is modeled using a polynomial equation:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + ... + \beta_n x^n +$$

where `n` represents the **degree of the polynomial**.

Importance of Polynomial Regression

- When the relationship between variables is **non-linear** and a straight line does not fit well.
- Captures **curved patterns** in data by introducing higher-degree polynomial terms.

Working of Polynomial Regression

1. **Transform the input features** by introducing polynomial terms.
2. **Apply Linear Regression** to fit the transformed dataset.
3. **Choose the optimal polynomial degree** to balance underfitting and overfitting.

Choosing the Right Degree (n)

- **Degree 1**: Equivalent to Linear Regression.
- **Degree 2-3**: Captures slight curves in data while preventing overfitting.
- **Degree >3**: More flexible but risks overfitting (too much complexity).

Applications of Polynomial Regression

- Predicting fuel efficiency based on vehicle characteristics.
- Modeling economic growth trends over time.
- Analyzing the effect of temperature on crop yields.

Comparison: Linear vs. Polynomial Regression

Feature	Linear Regression	Polynomial Regression
Relationship Type	Assumes a straight-line relationship	Captures curved relationships
Complexity	Simple and easy to interpret	More flexible but may overfit
Accuracy on Non-Linear Data	Low	High (with appropriate degree selection)
Risk of Overfitting	Low	High if degree is too large

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import train_test_split

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

In [2]:
```
sns.get_dataset_names()
```

Out[2]:
```
['anagrams',
 'anscombe',
 'attention',
 'brain_networks',
 'car_crashes',
 'diamonds',
 'dots',
 'dowjones',
 'exercise',
 'flights',
 'fmri',
 'geyser',
 'glue',
 'healthexp',
 'iris',
 'mpg',
 'penguins',
 'planets',
 'seaice',
 'taxis',
 'tips',
 'titanic']
```

In [3]:
```
data = sns.load_dataset('mpg')
```

In [4]:
```
data.head()
```

Out[4]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	
0	18.0	8	307.0	130.0	3504	12.0	70	usa	c
1	15.0	8	350.0	165.0	3693	11.5	70	usa	
2	18.0	8	318.0	150.0	3436	11.0	70	usa	p
3	16.0	8	304.0	150.0	3433	12.0	70	usa	
4	17.0	8	302.0	140.0	3449	10.5	70	usa	

In [5]:
```
data.shape
```

Out[5]:
```
(398, 9)
```

In [6]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 9 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   mpg            398 non-null    float64
 1   cylinders      398 non-null    int64  
 2   displacement   398 non-null    float64
 3   horsepower     392 non-null    float64
 4   weight         398 non-null    int64  
 5   acceleration   398 non-null    float64
 6   model_year     398 non-null    int64  
 7   origin         398 non-null    object 
 8   name           398 non-null    object 
dtypes: float64(4), int64(3), object(2)
memory usage: 28.1+ KB
```

In [7]:
```
data.nunique()
```

Out[7]:
```
mpg             129
cylinders         5
displacement     82
horsepower       93
weight          351
acceleration     95
model_year       13
origin            3
name            305
dtype: int64
```

In [8]:
```
data.horsepower.unique()
```

Out[8]:
```
array([130., 165., 150., 140., 198., 220., 215., 225., 190., 170., 160.,
        95.,  97.,  85.,  88.,  46.,  87.,  90., 113., 200., 210., 193.,
        nan, 100., 105., 175., 153., 180., 110.,  72.,  86.,  70.,  76.,
        65.,  69.,  60.,  80.,  54., 208., 155., 112.,  92., 145., 137.,
       158., 167.,  94., 107., 230.,  49.,  75.,  91., 122.,  67.,  83.,
        78.,  52.,  61.,  93., 148., 129.,  96.,  71.,  98., 115.,  53.,
        81.,  79., 120., 152., 102., 108.,  68.,  58., 149.,  89.,  63.,
        48.,  66., 139., 103., 125., 133., 138., 135., 142.,  77.,  62.,
       132.,  84.,  64.,  74., 116.,  82.])
```

Data Cleaning

In [9]:
```
data.isnull().sum()
```

Out[9]:
```
mpg             0
cylinders       0
displacement    0
horsepower      6
weight          0
acceleration    0
model_year      0
origin          0
name            0
dtype: int64
```

In [10]:
```
data.duplicated().sum()
```

Out[10]:
```
0
```

Data Handling

In [11]:
```
df = data.copy()
```

In [12]:
```
df['horsepower'].fillna(df['horsepower'].median(),  inplace=True)
```

Discriptive Statistics

In [13]:
```
df.describe().T
```

Out[13]:

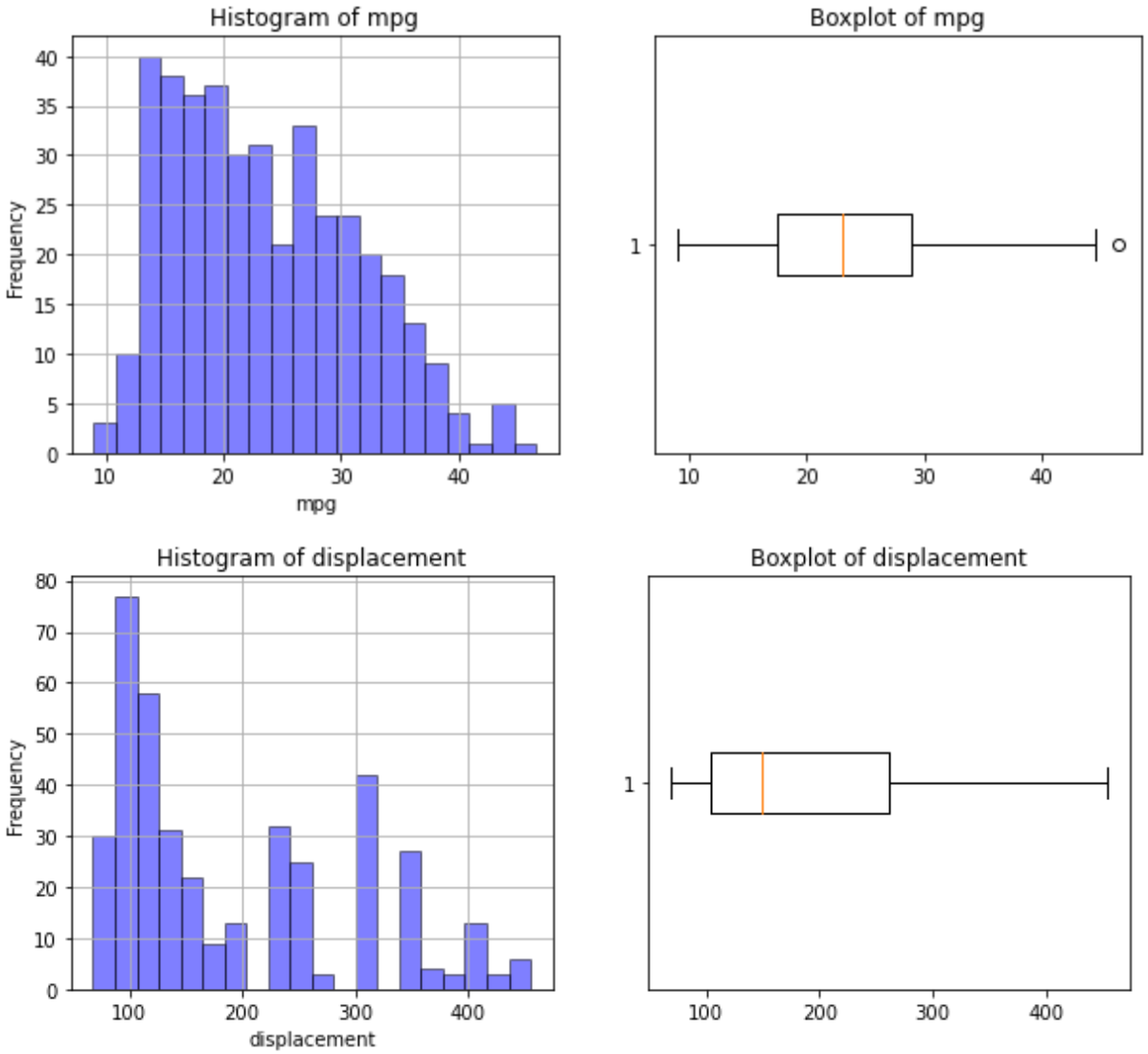
	count	mean	std	min	25%	50%	75%	max
mpg	398.0	23.514573	7.815984	9.0	17.500	23.0	29.000	46.6
cylinders	398.0	5.454774	1.701004	3.0	4.000	4.0	8.000	8.0
displacement	398.0	193.425879	104.269838	68.0	104.250	148.5	262.000	455.0
horsepower	398.0	104.304020	38.222625	46.0	76.000	93.5	125.000	230.0
weight	398.0	2970.424623	846.841774	1613.0	2223.750	2803.5	3608.000	5140.0
acceleration	398.0	15.568090	2.757689	8.0	13.825	15.5	17.175	24.8
model_year	398.0	76.010050	3.697627	70.0	73.000	76.0	79.000	82.0

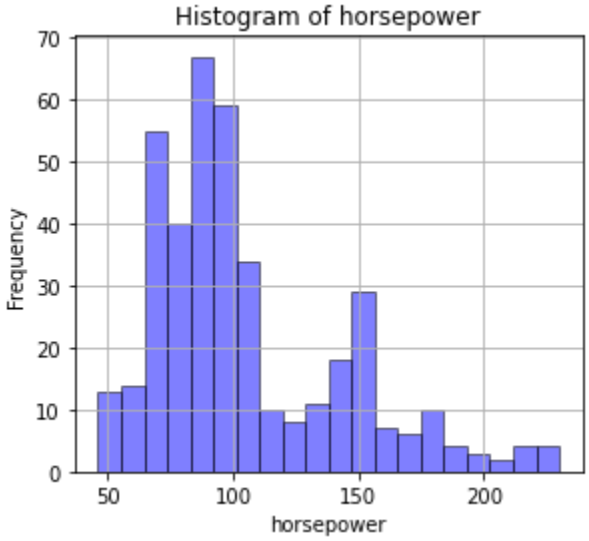
EDA

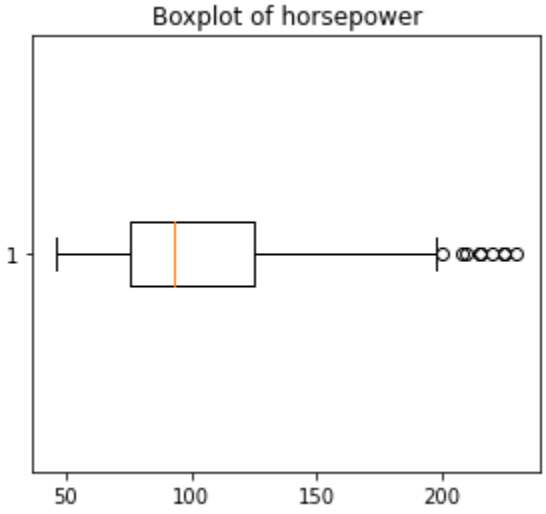
In [14]:
```
numerical = df.select_dtypes(include=['int','float']).columns
categorical = df.select_dtypes(include=['object']).columns

print(numerical)
print(categorical)
```

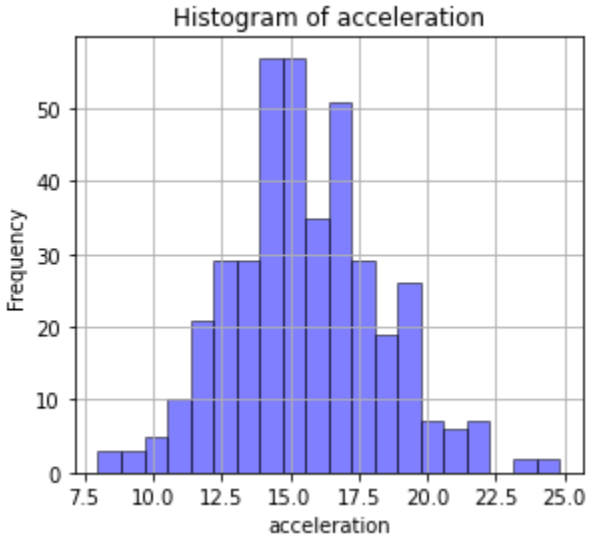
```
Index(['mpg', 'displacement', 'horsepower', 'acceleration'], dtype='object')
Index(['origin', 'name'], dtype='object')
```

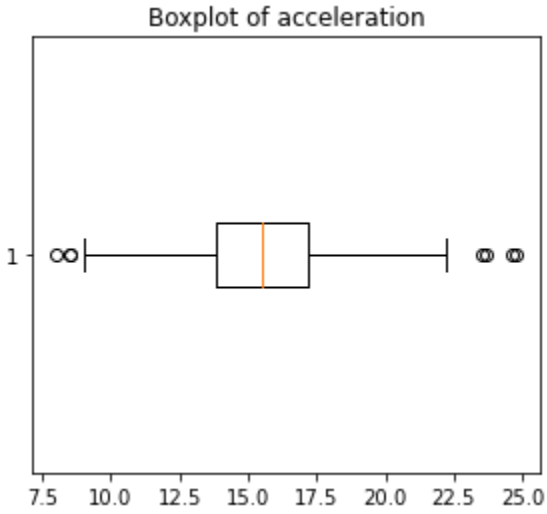
In [15]:
```
for i in numerical:
    plt.figure(figsize=(10,4))

    plt.subplot(1, 2, 1)
    df[i].hist(bins=20, alpha=0.5, color='b',edgecolor='black')
    plt.title(f'Histogram of {i}')
    plt.xlabel(i)
    plt.ylabel('Frequency')

    plt.subplot(1, 2, 2)
    plt.boxplot(df[i], vert=False)
    plt.title(f'Boxplot of {i}')

    plt.show()
```



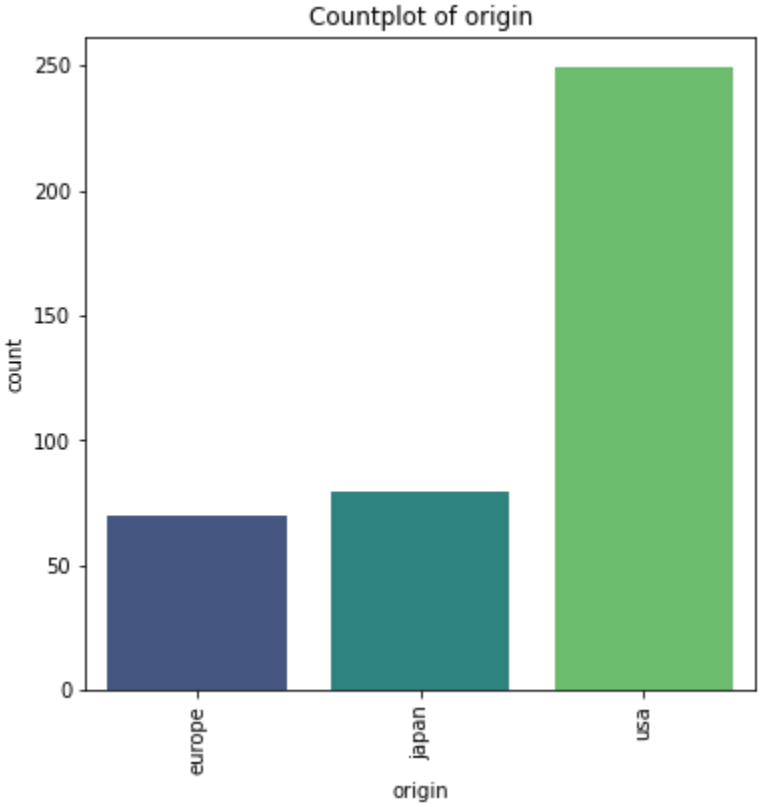


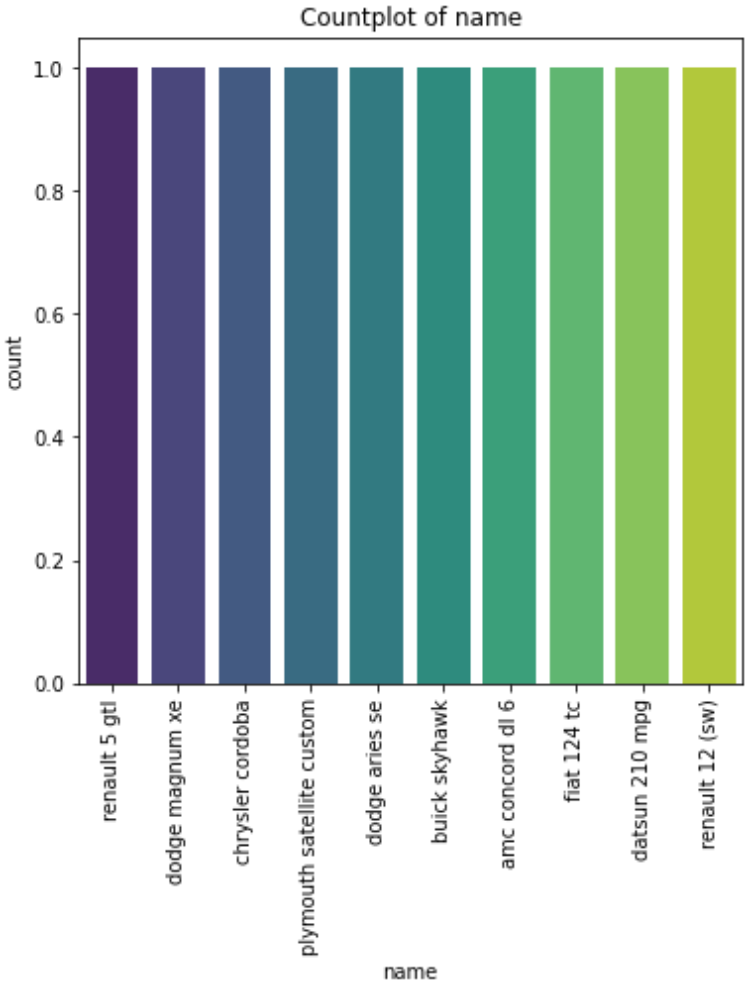




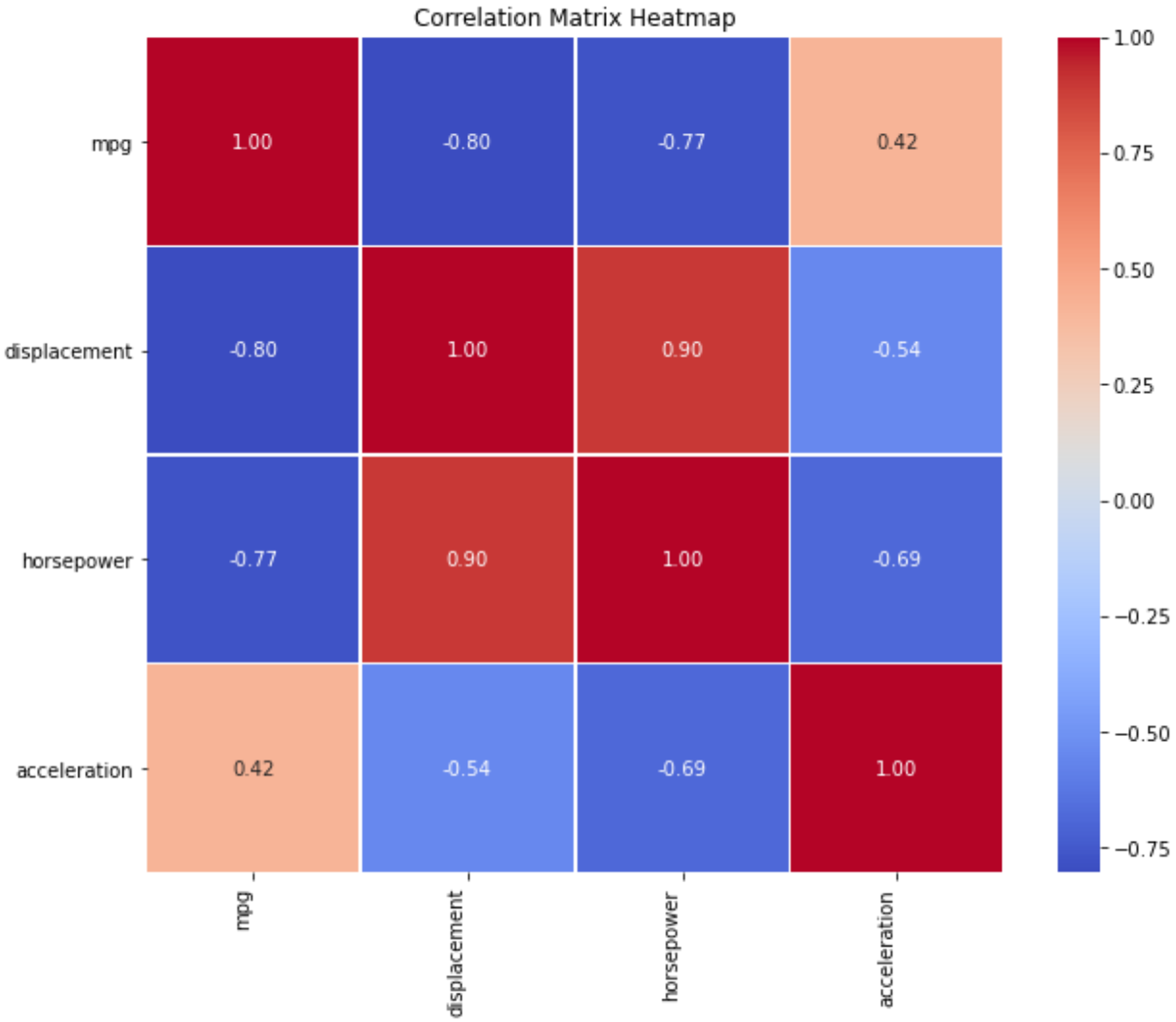


In [16]:
```
import seaborn as sns
for col in categorical:
    plt.figure(figsize=(6, 6))
    sns.countplot(x=col, data=df, order=df[col].value_counts().sort_values().head(1
    plt.title(f'Countplot of {col}')
    plt.xticks(rotation=90)

    plt.show()
```


Countplot of origin
250
200
150
100
50
0
count
europe
japan
usa
origin


Countplot of name
1.0
0.8
0.6
0.4
0.2
0.0
count
renault 5 gtl
dodge magnum xe
chrysler cordoba
plymouth satellite custom
dodge aries se
buick skyhawk
amc concord dl 6
fiat 124 tc
datsun 210 mpg
renault 12 (sw)
name

In [17]:
```
corr_data = df[numerical].corr(method='pearson')

plt.figure(figsize=(10, 8))
sns.heatmap(corr_data, annot=True, cmap="coolwarm", fmt=".2f", linewidths=0.5)
plt.xticks(rotation=90, ha='right')
plt.yticks(rotation=0)
plt.title("Correlation Matrix Heatmap")
plt.show()
```



In [18]:
```
# Select the relevant features
X = df[['horsepower']]  # You can select other features here
y = df['mpg']
```

In [19]:
```
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

In [20]:
```
# Create polynomial features
degree = 2  # Change the degree of the polynomial
poly = PolynomialFeatures(degree)
X_poly_train = poly.fit_transform(X_train)
```

In [21]:
```
# Fit a polynomial regression model
model = LinearRegression()
```

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

Out[21]:

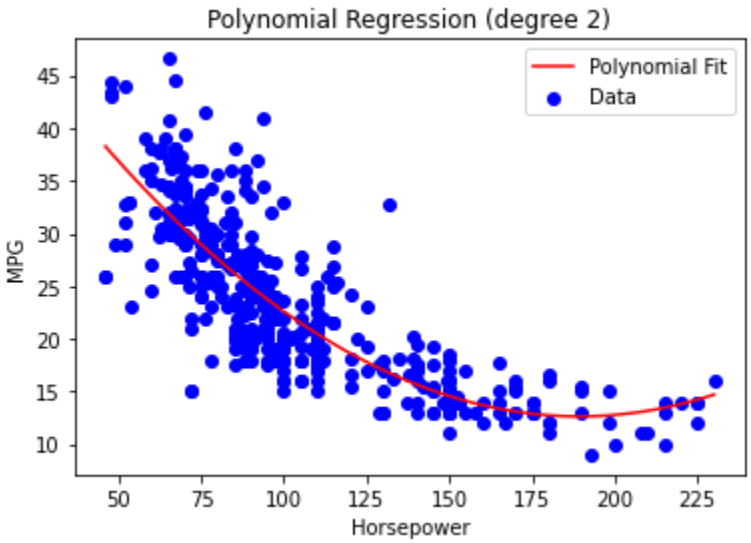
```
▾ LinearRegression
LinearRegression()
```

In [22]:

```
# Make predictions
X_poly_test = poly.transform(X_test)
y_pred = model.predict(X_poly_test)
```

In [23]:

```
# Visualize the results
plt.scatter(X, y, color='blue', label='Data')
X_range = np.linspace(X.min(), X.max(), 100).reshape(-1, 1)
X_range_poly = poly.transform(X_range)
y_range_pred = model.predict(X_range_poly)
plt.plot(X_range, y_range_pred, color='red', label='Polynomial Fit')
plt.xlabel('Horsepower')
plt.ylabel('MPG')
plt.legend()
plt.title(f'Polynomial Regression (degree {degree})')
plt.show()
```



In [24]:

```
# Evaluate the model on the test set
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics
print(f'Mean Squared Error (MSE): {mse:.2f}')
print(f'Root Mean Squared Error (RMSE): {rmse:.2f}')
print(f'R-squared (R²): {r2:.2f}')
```

```
Mean Squared Error (MSE): 13.94
Root Mean Squared Error (RMSE): 3.73
R-squared (R²): 0.74
```

Experiment 8

Develop a program to load the Titanic dataset, Split the data into training and test sets. Train a decision tree classifier. Visualize the tree structure, Evaluate accuracy, precision, recall, and F1-score.

Introduction to Decision Trees

What is a Decision Tree?

A **Decision Tree** is a supervised machine learning algorithm used for **classification and regression tasks**. It models decisions using a tree-like structure where:

- **Nodes** represent decision points based on feature values.
- **Edges** represent possible outcomes (branches).
- **Leaves** represent the final decision or classification.

Decision trees work by recursively splitting data into subsets based on the most significant feature, ensuring maximum information gain at each step.

Working of the Decision Tree Algorithm

1. Selecting the Best Feature for Splitting

At each step, the algorithm selects the feature that best separates the data. Common methods for choosing the best feature include:

- **Gini Impurity**
  `Gini = 1- ∑Pi2`

  Measures how often a randomly chosen element would be incorrectly classified.

- **Entropy (Information Gain)**
  `Entropy = ∑p(X)log p(X)`

  Measures the uncertainty in a dataset and selects splits that maximize information gain.

- **Chi-Square Test**
  Evaluates the statistical significance of the feature split.

2. Splitting the Data

- The dataset is divided into subsets based on the selected feature.

- The process continues recursively until:
  - A stopping condition is met (e.g., pure classification, max depth).
  - The tree reaches a predefined depth.

3. Making Predictions

- For a new sample, traverse the tree from the root to a leaf node.
- The leaf node contains the predicted class label.

Advantages of Decision Trees

✓ **Easy to interpret** – Mimics human decision-making.
✓ **Handles both numerical & categorical data**.
✓ **Requires little data preprocessing** – No need for feature scaling.
✓ **Works well with missing values**.

Challenges of Decision Trees

✕ **Overfitting** – Deep trees may memorize noise instead of patterns.
✕ **Bias towards dominant features** – Features with more categories can lead to biased splits.
✕ **Instability** – Small data variations can lead to different trees.

Optimizing Decision Trees

1. **Pruning**
   - **Pre-Pruning:** Stop the tree early using conditions (e.g., min samples per split).
   - **Post-Pruning:** Remove unnecessary branches after the tree is built.
2. **Setting Tree Depth**
   - Limiting maximum depth prevents overfitting.
3. **Using Ensemble Methods**
   - **Random Forest**: Combines multiple trees for better generalization.
   - **Gradient Boosting**: Sequentially improves predictions.

Applications of Decision Trees

- **Medical Diagnosis** – Classifying diseases based on symptoms.
- **Fraud Detection** – Identifying fraudulent transactions.
- **Customer Segmentation** – Categorizing users based on behavior.

In [1]:
```
# Importing necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

from sklearn.tree import export_graphviz
from IPython.display import Image
import pydotplus

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

In [2]:
```
data = pd.read_csv(r"C:\Users\Akhil\Downloads\Dataset-1 (3)\Dataset-1\Titanic data.
```

In [3]:
```
pd.set_option('display.max_columns', None)
```

In [4]:
```
data.head()
```

Out[4]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250
3	4	1	1	Futrelle, Mrs. Jacques (Lily May Peel)	female	35.0	1	0	113803	53.1000
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500

In [5]:
```
data.shape
```

Out[5]: (891, 12)

In [6]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   PassengerId  891 non-null    int64  
 1   Survived     891 non-null    int64  
 2   Pclass       891 non-null    int64  
 3   Name         891 non-null    object 
 4   Sex          891 non-null    object 
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64  
 7   Parch        891 non-null    int64  
 8   Ticket       891 non-null    object 
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object 
 11  Embarked     889 non-null    object 
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

Inference:

- 1.In the above output, column consists of the name of the column, Non-null Count means How many non-null values we have in that column, Dtype means What type of value that column consists of (int64 means int value, float64 means float value, object means string value)
- 2.In the age column we can see, Out of 891 values, we have 714 non-null values. It implies that we have 177 Null values. (891–714 = 177)
- 3.Same in the Cabin feature Out of 891 values we have only 204 non-null values. it implies that we have 687 Null values. But this is Huge. we have only 23% of values present in the data set and 77% of values are missing so we can drop this feature.
- 4.same in the Embarked column we can see out of 891 values,we have 889 non-null values.It implies that we have 2 Null values.(891-889=2)
- 6.Label encoding would be required for columns, 'gender', 'Ticket', 'Cabin', 'Embarked'.

In [7]:
```
data.Survived.unique()
```

Out[7]: array([0, 1])

Data Preprocessing

Data Cleaning

In [8]:
```
data.isnull().sum()
```

Out[8]:
```
PassengerId       0
Survived          0
Pclass            0
Name              0
Sex               0
Age             177
SibSp             0
Parch             0
Ticket            0
Fare              0
Cabin           687
Embarked          2
dtype: int64
```

INFERENCE: There are null values in 'Age','Cabin'and'Embarked'.we can treat them by dropping or imputation.

Method 1: Dropping rows or columns.

In the cabin data set, we have 77% null values so it is not easy to handle the cabin feature that's why I am droping Cabin column from my data set.

In [9]:
```
df= data.drop(['Cabin'] ,axis=1)
```

Method2:Imputation

In [10]:
```
df['Age'] = df['Age'].fillna(df['Age'].mean())
df
```

Out[10]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket
0	1	0	3	Braund, Mr. Owen Harris	male	22.000000	1	0	A/5 21171
1	2	1	1	Cumings, Mrs. John Bradley Briggs Th...	female	38.000000	1	0	PC 17599
2	3	1	3	Heikkinen, I ss. l ina	female	26.000000	0	0	STON/O2. 3101282
				elle, Mrs. ues Heath (Lily May Peel)	female	35.000000	1	0	113803
4	5	0	3	Allen, Mr. Wil iam H nry	male	35.000000	0	0	373450
...	...	...	...	...	...	...	...	...	...
886	887	0	2	Mont vila, Rev. Juzas	male	27.000000	0	0	211536
887	888	1	1	Grah am, M ss. Marg ret Edith	female	19.000000	0	0	112053
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	29.699118	1	2	W./C. 6607
889	890	1	1	Behr, Mr. Karl Howell	male	26.000000	0	0	111369
890	891	0	3	Dooley, Mr. Patrick	male	32.000000	0	0	370376

891 rows × 11 columns

Age feature consists of some null values so first, we need to handle that. Here I am filling the null values with the mean of Age Feature.

But There are two null values in 'Embarked'column.we can treat them by imputing with mode

In [11]:
```
df['Embarked']=df['Embarked'].fillna(df['Embarked'].mode()[0])
```

In [12]:
```
df.isnull().sum()
```

Out[12]:
```
PassengerId     0
Survived        0
Pclass          0
Name            0
Sex             0
Age             0
SibSp           0
Parch           0
Ticket          0
Fare            0
Embarked        0
dtype: int64
```

From the above output,We can observe the are no null values

In [13]:
```
df = df.drop(columns=['PassengerId', 'SibSp', 'Parch', 'Ticket',"Name"])
```

In [14]:
```
df.columns
```

Out[14]:
```
Index(['Survived', 'Pclass', 'Sex', 'Age', 'Fare', 'Embarked'], dtype='object')
```

In [15]:
```
# Encode categorical variables
df = pd.get_dummies(df, columns=['Sex', 'Embarked',"Pclass"], drop_first=True)
```

In [16]:
```
# Select features and target
X = df.drop(columns=['Survived'])
y = df['Survived']
```

In [17]:
```
# Split into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

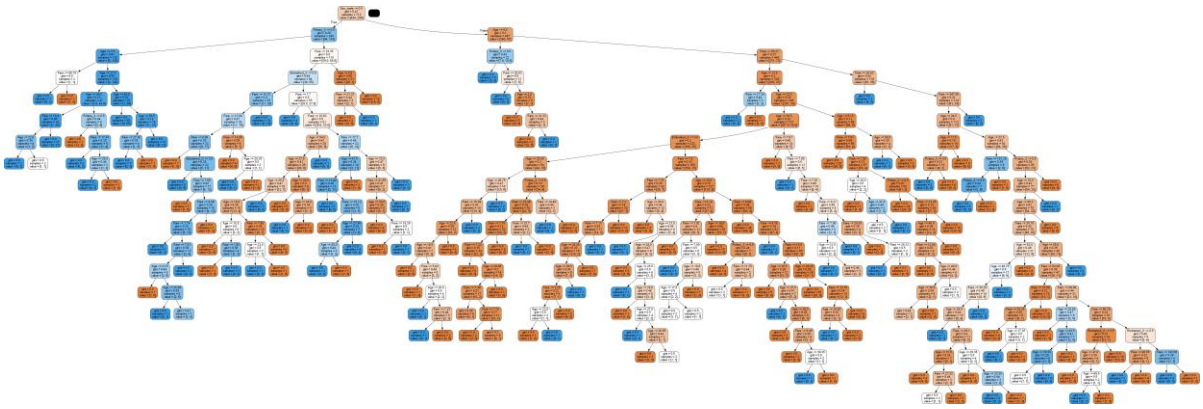
In [18]:
```
dt = DecisionTreeClassifier( random_state=42)
dt.fit(X_train, y_train)
```

Out[18]:
```
DecisionTreeClassifier
DecisionTreeClassifier(random_state=42)
```

In [19]:
```
# Make predictions
y_pred = dt.predict(X_test)
```

In [20]:
```
# Export the tree to DOT format
dot_data = export_graphviz(dt, out_file=None,
                           feature_names=X_train.columns,
                           rounded=True, proportion=False,
                           precision=2, filled=True)

# Convert DOT data to a graph
graph = pydotplus.graph_from_dot_data(dot_data)

# Display the graph
Image(graph.create_png())
```

Out[20]:



In [22]:
```
# Accuracy Score
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')
```

```
Accuracy: 0.78
```

Confusion Matrix

A Confusion Matrix is a table used to evaluate the performance of a classification model. It compares the actual labels with the predicted labels and helps to calculate various evaluation metrics.

Explanation of Terms:

- True Positive (TP) → The model correctly predicted the positive class (1).
- True Negative (TN) → The model correctly predicted the negative class (0).
- False Positive (FP) → The model incorrectly predicted positive (Type I Error).
- False Negative (FN) → The model incorrectly predicted negative (Type II Error).

In [23]:
```
# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:")
print(cm)
```

```
Confusion Matrix:
[[86 19]
 [21 53]]
```

- 86 → True Negatives (TN) (Correctly predicted passengers who did not survive)
- 19 → False Positives (FP) (Wrongly predicted as survived)
- 21 → False Negatives (FN) (Wrongly predicted as not survived)
- 53 → True Positives (TP) (Correctly predicted passengers who survived)

Understanding the Classification Report

The classification report provides key performance metrics for each class, helping us evaluate the effectiveness of a model. It includes precision, recall, F1-score, and support for both classes (survived and not survived in the Titanic dataset).

1. Precision

Definition: Precision measures how many of the predicted positive cases were actually positive. It tells us how precise the model is when it makes a positive prediction.

Formula: Precision= TP/TP + FP

Where:

- **TP (True Positives):** Correctly predicted positive cases.
- **FP (False Positives):** Incorrectly predicted positive cases (actually negative).

Recall

Recall

Definition: Recall measures how many actual positive cases were correctly identified by the model. It tells us how well the model is capturing real positive cases.

Formula: Recall = TP/TP + FN

Where:

- **TP (True Positives):** Correctly predicted positive cases.
- **FN (False Negatives):** Cases that were actually positive but predicted as negative.

F1 Score

F1-Score

Definition: The F1-score is the harmonic mean of precision and recall. It balances both metrics, making it useful when there is an imbalance between the two.

Formula: [F1 = 2 (Precision * Recall)/(Precision + Recall)]

In [24]:
```
# Classification Report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.80      0.82      0.81       105
           1       0.74      0.72      0.73        74

    accuracy                           0.78       179
   macro avg       0.77      0.77      0.77       179
weighted avg       0.78      0.78      0.78       179
```

In [ ]:
```
```

Experiment 9

Develop a program to implement the Naive Bayesian classifier considering Iris dataset for training, Compute the accuracy of the classifier, considering the test data.

Introduction to Naive Bayes Classification

The Naive Bayes classifier is a simple yet powerful probabilistic machine learning algorithm based on Bayes' Theorem. It is widely used for classification tasks, including spam filtering, sentiment analysis, and medical diagnosis. The algorithm assumes that features are conditionally independent, which simplifies computations and makes it efficient for large datasets.

What is Naive Bayes?

Naïve Bayes is a **probabilistic classification algorithm** based on **Bayes' Theorem** with the **naïve assumption** that features are independent of each other. Despite this strong assumption, it performs well in many real-world scenarios.

It is widely used for **text classification, spam detection, medical diagnosis, and facial recognition**.

Bayes' Theorem

The core idea of the Naïve Bayes classifier is based on **Bayes' Theorem**, which states:

P(AIB) =P(BIA) * P(A) / P(B)

where:

- P(A|B) → Probability of hypothesis A (class) given evidence B (features).
- P(B|A) → Probability of evidence B given hypothesis A.
- P(A) → Prior probability of class A .
- P(B) → Prior probability of feature B.

Working of the Naive Bayes Classifier

1. Training Phase

- Compute **prior probabilities** P(Class) from training data.

- Compute **likelihood probabilities** P(Feature|Class) for each feature.
- Apply **Bayes' Theorem** to determine the probability of each class given a new sample.

2. Prediction Phase

- For a new test sample, calculate **posterior probabilities** for each class.
- Assign the class with the **highest probability** to the test sample.

Types of Naive Bayes Classifiers

1. **Gaussian Naïve Bayes**
   - Assumes **continuous numerical features** follow a **normal (Gaussian) distribution**.
   - Probability is computed using the Gaussian probability density function:
2. **Multinomial Naïve Bayes**
   - Used for **discrete feature values**, especially in **text classification** (e.g., spam filtering).
   - Works well with word frequency counts.
3. **Bernoulli Naïve Bayes**
   - Used when features are **binary** (0 or 1).
   - Useful for text classification with **presence/absence** of words.

Performance Evaluation

To assess the classifier's accuracy, the following metrics are used:

- **Accuracy**: Measures the percentage of correct predictions.
- **Precision**: Measures how many predicted positive instances are actually positive.
- **Recall**: Measures the ability to detect positive instances.
- **F1-Score**: Harmonic mean of Precision and Recall.
- **Confusion Matrix**:
  - True Positives (TP) – Correctly predicted positive cases.
  - False Positives (FP) – Incorrectly predicted positive cases.
  - True Negatives (TN) – Correctly predicted negative cases.
  - False Negatives (FN) – Incorrectly predicted negative cases.

Advantages of Naive Bayes

✓ **Fast and Efficient** – Works well with large datasets.

✓ **Performs well with noisy and small data**.

✓ **Requires minimal training data**.
✓ **Works well for high-dimensional data (e.g., text, image classification).**

Challenges of Naive Bayes

✗ **Feature Independence Assumption** – Real-world features are often correlated.
✗ **Zero Probability Issue** – If a feature value is missing in the training data, the probability becomes zero (solved using **Laplace Smoothing**).
✗ **Sensitive to Continuous Data Assumptions** – Gaussian Naïve Bayes assumes a normal distribution, which may not always be valid.

Applications of Naive Bayes

- **Spam Detection** – Filtering spam emails based on word frequency.
- **Sentiment Analysis** – Classifying text as positive or negative.
- **Medical Diagnosis** – Identifying diseases based on symptoms.
- **Facial Recognition** – Identifying individuals from image data.

In [2]:
```
import seaborn as sns
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
```

In [3]:
```
# Load the Iris dataset
iris = sns.load_dataset("iris")
```

In [4]:
```
iris.head()
```

Out[4]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

In [5]:
```
iris.shape
```

Out[5]: (150, 5)

In [6]:
```
# Basic Data Exploration
print("\nBasic Information about Dataset:")
print(iris.info())  # Overview of dataset
```

```
Basic Information about Dataset:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   sepal_length  150 non-null    float64
 1   sepal_width   150 non-null    float64
 2   petal_length  150 non-null    float64
 3   petal_width   150 non-null    float64
 4   species       150 non-null    object 
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
None
```

In [7]:
```
# Summary Statistics
print("\nSummary Statistics:")
print(iris.describe())  # Summary statistics of dataset
```

```
Summary Statistics:
       sepal_length  sepal_width  petal_length  petal_width
count    150.000000   150.000000    150.000000   150.000000
mean       5.843333     3.057333      3.758000     1.199333
std        0.828066     0.435866      1.765298     0.762238
min        4.300000     2.000000      1.000000     0.100000
25%        5.100000     2.800000      1.600000     0.300000
50%        5.800000     3.000000      4.350000     1.300000
75%        6.400000     3.300000      5.100000     1.800000
max        7.900000     4.400000      6.900000     2.500000
```

In [8]:
```
# Check for missing values
print("\nMissing Values in Each Column:")
print(iris.isnull().sum())  # Count of missing values
```

```
Missing Values in Each Column:
sepal_length    0
sepal_width     0
petal_length    0
petal_width     0
species         0
dtype: int64
```

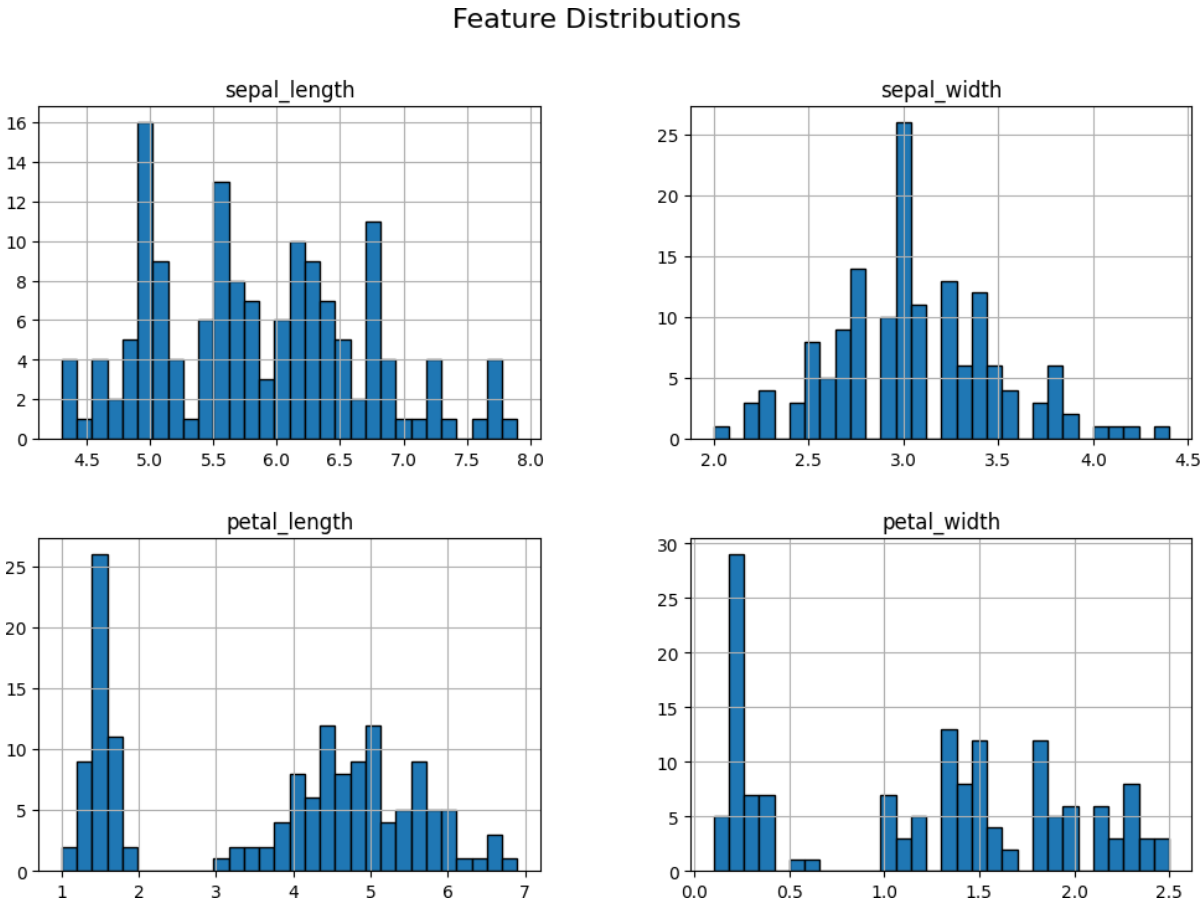
In [9]:
```
iris.duplicated().sum()
```

Out[9]: np.int64(1)

Univariate Analysis

In [10]:
```
# Histograms for distribution of features
plt.figure(figsize=(12, 8))
iris.hist(figsize=(12, 8), bins=30, edgecolor='black')
plt.suptitle("Feature Distributions", fontsize=16)
plt.show()
```

```
<Figure size 1200x800 with 0 Axes>
```



Inferences from Histograms:

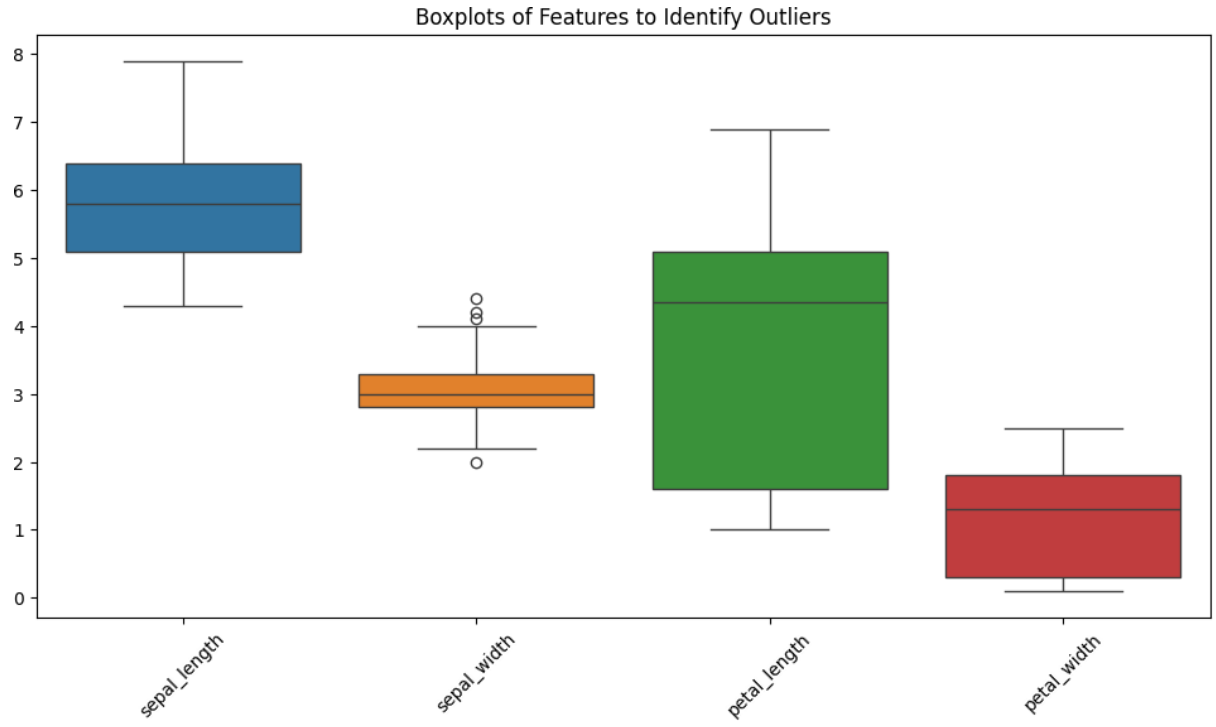
1. **Sepal Length**:
   - The distribution of sepal length appears to be roughly normal with a slight skew towards the right.
   - Most of the sepal lengths fall between 4.5 and 7.5 cm.
2. **Sepal Width**:
   - The distribution of sepal width is also roughly normal but with a slight skew towards the left.
   - Most of the sepal widths fall between 2.5 and 3.5 cm.
3. **Petal Length**:
   - The distribution of petal length is more spread out and shows a clear separation between different species.

- There are distinct peaks indicating the presence of different species with varying petal lengths.

4. **Petal Width**:

- Similar to petal length, the distribution of petal width shows clear separation between species.
- There are distinct peaks indicating the presence of different species with varying petal widths.

In [11]:
```
# Boxplots for outlier detection
plt.figure(figsize=(12, 6))
sns.boxplot(data=iris)
plt.xticks(rotation=45)
plt.title("Boxplots of Features to Identify Outliers")
plt.show()
```



Inferences from Boxplots:

1. **Sepal Length**:

- There are a few outliers in the sepal length distribution.
- The median sepal length is around 5.8 cm, with the interquartile range (IQR) between 5.1 and 6.4 cm.

2. **Sepal Width**:

- There are several outliers in the sepal width distribution.
- The median sepal width is around 3.0 cm, with the IQR between 2.8 and 3.3 cm.

3. **Petal Length**:

- The petal length distribution shows clear separation between species, with minimal overlap.
- The median petal length varies significantly between species, indicating it is a good feature for classification.

4. **Petal Width**:

- Similar to petal length, the petal width distribution shows clear separation between species.
- The median petal width varies significantly between species, indicating it is also a good feature for classification.

Heatmap: Visualizing the Correlation Matrix

In [12]:
```
num_col = iris.select_dtypes(include=[np.number]).columns
cat_col = iris.select_dtypes(include=['object']).columns
print(f"numerical_data {num_col}")
print(f"categorical_data  {cat_col}")
```

```
numerical_data Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width'],
dtype='object')
categorical_data  Index(['species'], dtype='object')
```

In [13]:
```
# Correlation Matrix
plt.figure(figsize=(10, 6))
corr_matrix = iris[num_col].corr('pearson')
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title("Feature Correlation Heatmap")
plt.show()
```

Based on the heatmap of the correlation matrix, we can infer the following:

1. **Sepal Length**:
   - Positively correlated with petal length (0.87) and petal width (0.82).
   - Weak negative correlation with sepal width (-0.12).
2. **Sepal Width**:
   - Weak negative correlation with sepal length (-0.12), petal length (-0.43), and petal width (-0.37).
3. **Petal Length**:
   - Strong positive correlation with sepal length (0.87) and petal width (0.96).
   - Weak negative correlation with sepal width (-0.43).
4. **Petal Width**:
   - Strong positive correlation with petal length (0.96) and sepal length (0.82).
   - Weak negative correlation with sepal width (-0.37).
5. **Species**:
   - Strong positive correlation with petal length (0.95) and petal width (0.96).
   - Moderate positive correlation with sepal length (0.78).
   - Moderate negative correlation with sepal width (-0.43).

These correlations suggest that petal length and petal width are highly correlated with each other and with the species of the iris flower, making them important features for

classification. Sepal length also shows a moderate correlation with species, while sepal width has a weaker correlation with species.

In [14]:
```
# Encode target labels
label_encoder = LabelEncoder()
iris["species"] = label_encoder.fit_transform(iris["species"])
```

In [15]:
```
# Define features and target
X = iris.drop(columns=["species"])
y = iris["species"]
```

In [16]:
```
# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

In [17]:
```
# Initialize and train Gaussian Naïve Bayes classifier
nb_classifier = GaussianNB()
nb_classifier.fit(X_train, y_train)
```

Out[17]:
```
GaussianNB
GaussianNB()
```

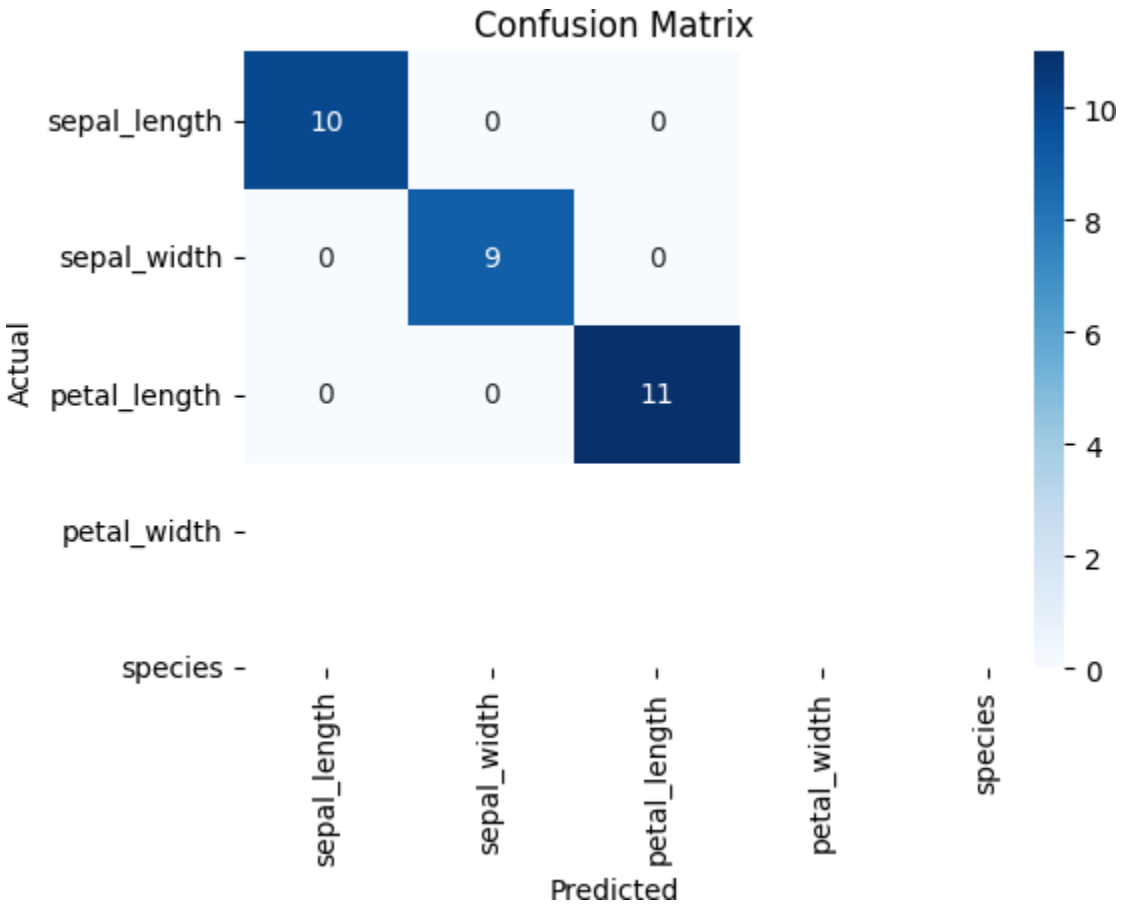
In [18]:
```
# Make predictions
y_pred = nb_classifier.predict(X_test)
y_pred
```

Out[18]:
```
array([1, 0, 2, 1, 1, 0, 1, 2, 1, 1, 2, 0, 0, 0, 0, 1, 2, 1, 1, 2, 0, 2,
       0, 2, 2, 2, 2, 2, 0, 0])
```

In [19]:
```
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')
print('\nClassification Report:\n', classification_report(y_test, y_pred))
print('\nConfusion Matrix:\n', confusion_matrix(y_test, y_pred))
```

```
Accuracy: 1.00

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30


Confusion Matrix:
 [[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]
```

In [22]:
```
# Visualize confusion matrix
plt.figure(figsize=(6, 4))
sns.heatmap(confusion_matrix(y_test, y_pred), annot=True, cmap='Blues', fmt='d',
            xticklabels=iris.keys(), yticklabels=iris.keys())
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```



Based on the evaluation metrics and the confusion matrix, we can infer the following about the Gaussian Naive Bayes model:

1. **Accuracy**:
   - The model achieved an accuracy of 1.0 (100%) on the test data, indicating that it correctly classified all the test samples.
2. **Classification Report**:
   - The classification report provides precision, recall, and F1-score for each class (species of iris).
   - Since the accuracy is 100%, the precision, recall, and F1-score for all classes are also 1.0.
3. **Confusion Matrix**:

- The confusion matrix shows that all the test samples were correctly classified into their respective classes.
- There are no misclassifications, as indicated by the absence of non-diagonal elements in the confusion matrix.

Overall, the Gaussian Naive Bayes model performed exceptionally well on the Iris dataset, achieving perfect classification accuracy. This suggests that the features (sepal length, sepal width, petal length, and petal width) are highly informative for distinguishing between the different species of iris flowers.

Experiment 10

Develop a program to implement k-means clustering using Wisconsin Breast Cancer data set and visualize the clustering result.

Introduction to K-Means Clustering

What is Clustering?

Clustering is an **unsupervised machine learning technique** used to group data points into clusters based on their similarity. The goal is to **identify hidden patterns** or **natural groupings** in the data.

One of the most widely used clustering algorithms is **K-Means Clustering**, which divides the dataset into **K clusters**, where each data point belongs to the nearest cluster center.

What is K-Means Clustering?

K-Means is a **centroid-based clustering algorithm** that partitions data into **K clusters** by minimizing the variance within each cluster.

Working of K-Means Algorithm

1. **Choose the number of clusters (K)**.
2. **Randomly initialize K cluster centroids**.
3. **Assign each data point to the nearest centroid** based on distance (e.g., Euclidean distance).
4. **Update the centroids** by computing the mean of all points assigned to each cluster.
5. **Repeat Steps 3 and 4 until convergence** (when centroids no longer change significantly).

Mathematical Representation

- The objective is to minimize the **sum of squared distances (SSD)** between data points and their assigned cluster centroid:

$$J = \sum_{i=1}^{K} \sum_{x_j \in C_i} ||x_j - \mu_i||^2$$

where:

- K = Number of clusters
- xj = Data point
- μi = Centroid of cluster Ci

Choosing the Optimal Number of Clusters (K)

Selecting the right value of **K** is crucial. Some common methods include:

1. **Elbow Method**:
   - Plots the within-cluster sum of squares (WCSS) for different K values.
   - The "elbow point" where WCSS stops decreasing significantly is chosen as the optimal K.
2. **Silhouette Score**:
   - Measures how well-separated the clusters are.
   - A higher score indicates better clustering.
3. **Gap Statistics**:
   - Compares clustering performance to randomly generated reference data.

Distance Metrics in K-Means

K-Means typically uses **Euclidean Distance** to measure how close a data point is to a centroid:

$$d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \ldots + (x_n - y_n)^2}$$

Other distance metrics include:

- **Manhattan Distance**
- **Cosine Similarity**
- **Mahalanobis Distance**

Advantages of K-Means Clustering

✓ **Efficient and Scalable** – Works well with large datasets.
✓ **Easy to Implement** – Simple and interpretable.
✓ **Handles High-Dimensional Data** – Can work on complex datasets.

Challenges of K-Means Clustering

❌ **Sensitive to Initial Centroid Selection** – Different initializations may lead to different results.
❌ **Not Suitable for Non-Spherical Clusters** – Assumes clusters are circular and evenly sized.
❌ **Outliers Affect Centroids** – Presence of outliers can distort clustering results.

Visualization of Clusters

After applying **K-Means Clustering**, the results can be visualized using:

- **Scatter Plots**: Plot the clusters with different colors.
- **Centroid Markers**: Display cluster centers for better interpretation.
- **2D/3D PCA Visualization**: Reduce dimensions for better visualization.

Applications of K-Means Clustering

- **Customer Segmentation** – Grouping customers based on purchasing behavior.
- **Image Compression** – Reducing image colors to dominant clusters.
- **Anomaly Detection** – Identifying fraudulent transactions.
- **Medical Diagnosis** – Classifying patients based on symptoms and medical data.

In [34]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score

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

In [17]:
```
data = pd.read_csv(r"C:\Users\Akhil\Downloads\ML6thSEM_FDP\Datasets\Wisconsin Breas
```

In [18]:
```
data.head()
```

Out[18]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothn
0	842302	M	17.99	10.38	122.80	1001.0	
1	842517	M	20.57	17.77	132.90	1326.0	
2	84300903	M	19.69	21.25	130.00	1203.0	
3	84348301	M	11.42	20.38	77.58	386.1	
4	84358402	M	20.29	14.34	135.10	1297.0	

5 rows × 33 columns

In [19]:
```
data.shape
```

Out[19]:
```
(569, 33)
```

In [20]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 33 columns):
 #   Column                   Non-Null Count  Dtype  
---  ------                   --------------  -----  
 0   id                       569 non-null    int64  
 1   diagnosis                569 non-null    object 
 2   radius_mean              569 non-null    float64
 3   texture_mean             569 non-null    float64
 4   perimeter_mean           569 non-null    float64
 5   area_mean                569 non-null    float64
 6   smoothness_mean          569 non-null    float64
 7   compactness_mean         569 non-null    float64
 8   concavity_mean           569 non-null    float64
 9   concave points_mean      569 non-null    float64
 10  symmetry_mean            569 non-null    float64
 11  fractal_dimension_mean   569 non-null    float64
 12  radius_se                569 non-null    float64
 13  texture_se               569 non-null    float64
 14  perimeter_se             569 non-null    float64
 15  area_se                  569 non-null    float64
 16  smoothness_se            569 non-null    float64
 17  compactness_se           569 non-null    float64
 18  concavity_se             569 non-null    float64
 19  concave points_se        569 non-null    float64
 20  symmetry_se              569 non-null    float64
 21  fractal_dimension_se     569 non-null    float64
 22  radius_worst             569 non-null    float64
 23  texture_worst            569 non-null    float64
 24  perimeter_worst          569 non-null    float64
 25  area_worst               569 non-null    float64
 26  smoothness_worst         569 non-null    float64
 27  compactness_worst        569 non-null    float64
 28  concavity_worst          569 non-null    float64
 29  concave points_worst     569 non-null    float64
 30  symmetry_worst           569 non-null    float64
 31  fractal_dimension_worst  569 non-null    float64
 32  Unnamed: 32              0 non-null      float64
dtypes: float64(31), int64(1), object(1)
memory usage: 146.8+ KB
```

In [21]:
```
data.diagnosis.unique()
```

Out[21]:
```
array(['M', 'B'], dtype=object)
```

Data Preprocessing

Data Cleaning

In [22]:
```
data.isnull().sum()
```

Out[22]:
```
id                         0
diagnosis                  0
radius_mean                0
texture_mean               0
perimeter_mean             0
area_mean                  0
smoothness_mean            0
compactness_mean           0
concavity_mean             0
concave points_mean        0
symmetry_mean              0
fractal_dimension_mean     0
radius_se                  0
texture_se                 0
perimeter_se               0
area_se                    0
smoothness_se              0
compactness_se             0
concavity_se               0
concave points_se          0
symmetry_se                0
fractal_dimension_se       0
radius_worst               0
texture_worst              0
perimeter_worst            0
area_worst                 0
smoothness_worst           0
compactness_worst          0
concavity_worst            0
concave points_worst       0
symmetry_worst             0
fractal_dimension_worst    0
Unnamed: 32              569
dtype: int64
```

In [23]:
```
data.duplicated().sum()
```

Out[23]:
```
np.int64(0)
```

In [24]:
```
df = data.drop(['id', 'Unnamed: 32'], axis=1)
```

In [25]:
```
df['diagnosis'] = df['diagnosis'].map({'M':1, 'B':0}) # Malignant:1, Benign:0
```

Discriptive Statistics

In [26]:
```
df.describe().T
```

Out[26]:

	count	mean	std	min	25%	50%
diagnosis	569.0	0.372583	0.483918	0.000000	0.000000	0.000000
radius_mean	569.0	14.127292	3.524049	6.981000	11.700000	13.370000
texture_mean	569.0	19.289649	4.301036	9.710000	16.170000	18.840000
perimeter_mean	569.0	91.969033	24.298981	43.790000	75.170000	86.240000
area_mean	569.0	654.889104	351.914129	143.500000	420.300000	551.100000
smoothness_mean	569.0	0.096360	0.014064	0.052630	0.086370	0.095870
compactness_mean	569.0	0.104341	0.052813	0.019380	0.064920	0.092630
concavity_mean	569.0	0.088799	0.079720	0.000000	0.029560	0.061540
concave points_mean	569.0	0.048919	0.038803	0.000000	0.020310	0.033500
symmetry_mean	569.0	0.181162	0.027414	0.106000	0.161900	0.179200
fractal_dimension_mean	569.0	0.062798	0.007060	0.049960	0.057700	0.061540
radius_se	569.0	0.405172	0.277313	0.111500	0.232400	0.324200
texture_se	569.0	1.216853	0.551648	0.360200	0.833900	1.108000
perimeter_se	569.0	2.866059	2.021855	0.757000	1.606000	2.287000
area_se	569.0	40.337079	45.491006	6.802000	17.850000	24.530000
smoothness_se	569.0	0.007041	0.003003	0.001713	0.005169	0.006380
compactness_se	569.0	0.025478	0.017908	0.002252	0.013080	0.020450
concavity_se	569.0	0.031894	0.030186	0.000000	0.015090	0.025890
concave points_se	569.0	0.011796	0.006170	0.000000	0.007638	0.010930
symmetry_se	569.0	0.020542	0.008266	0.007882	0.015160	0.018730
fractal_dimension_se	569.0	0.003795	0.002646	0.000895	0.002248	0.003187
radius_worst	569.0	16.269190	4.833242	7.930000	13.010000	14.970000
texture_worst	569.0	25.677223	6.146258	12.020000	21.080000	25.410000
perimeter_worst	569.0	107.261213	33.602542	50.410000	84.110000	97.660000
area_worst	569.0	880.583128	569.356993	185.200000	515.300000	686.500000
smoothness_worst	569.0	0.132369	0.022832	0.071170	0.116600	0.131300
compactness_worst	569.0	0.254265	0.157336	0.027290	0.147200	0.211900
concavity_worst	569.0	0.272188	0.208624	0.000000	0.114500	0.226700
concave points_worst	569.0	0.114606	0.065732	0.000000	0.064930	0.099930
symmetry_worst	569.0	0.290076	0.061867	0.156500	0.250400	0.282200

	count	mean	std	min	25%	50%
fractal_dimension_worst	569.0	0.083946	0.018061	0.055040	0.071460	0.080040

In [31]:
```
#dropped the  Diagnosis (target) since clustering is unsupervised.
df.drop(columns=["diagnosis"], inplace=True)  # Removing Target
```

In [32]:
```
# Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(df)
```

Standardized the features to have mean 0 and standard deviation 1 (important for PCA and K-Means).

In [45]:
```
#  Apply PCA for Dimensionality Reduction
pca = PCA(n_components=2)  # Reduce to 2 dimensions for visualization
X_pca = pca.fit_transform(X_scaled)
```

Dimensionality Reduction using PCA

- Applied PCA (Principal Component Analysis) to reduce dimensions from 30 to 2 for visualization.
- PCA retains as much variance as possible while reducing complexity.

In [46]:
```
# Check explained variance ratio
explained_variance = pca.explained_variance_ratio_
total_explained_variance = np.sum(explained_variance)

print(f"Variance explained by PC1: {explained_variance[0]:.4f}")
print(f"Variance explained by PC2: {explained_variance[1]:.4f}")

print(f"Total variance explained by first 2 components: {total_explained_variance:.
```

```
Variance explained by PC1: 0.4427
Variance explained by PC2: 0.1897
Total variance explained by first 2 components: 0.6324
```

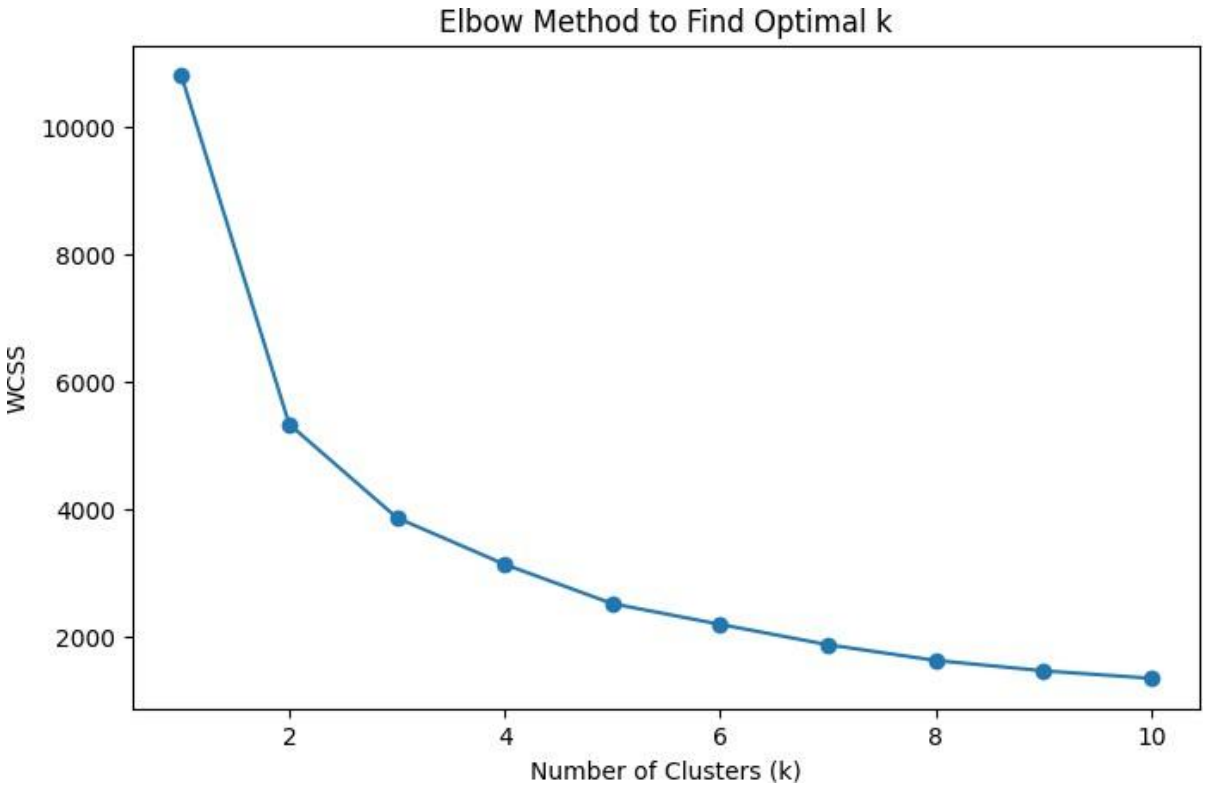
Elbow Method to Find the Optimal Number of Clusters

- Used WCSS (Within-Cluster Sum of Squares) to analyze different values of k.
- Plotted the Elbow Curve to determine the best value for k (where WCSS starts decreasing at a slower rate).

In [48]:
```
#Use the Elbow Method to determine the optimal number of clusters
wcss = []  # Within-Cluster Sum of Squares
K_range = range(1, 11)

for k in K_range:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
```

```
    kmeans.fit(X_pca)
    wcss.append(kmeans.inertia_)  # Append the inertia (sum of squared distances)
```

In [49]:
```
# Plot the Elbow Method Graph
plt.figure(figsize=(8, 5))
plt.plot(K_range, wcss, marker="o", linestyle="-")
plt.xlabel("Number of Clusters (k)")
plt.ylabel("WCSS")
plt.title("Elbow Method to Find Optimal k")
plt.show()
```



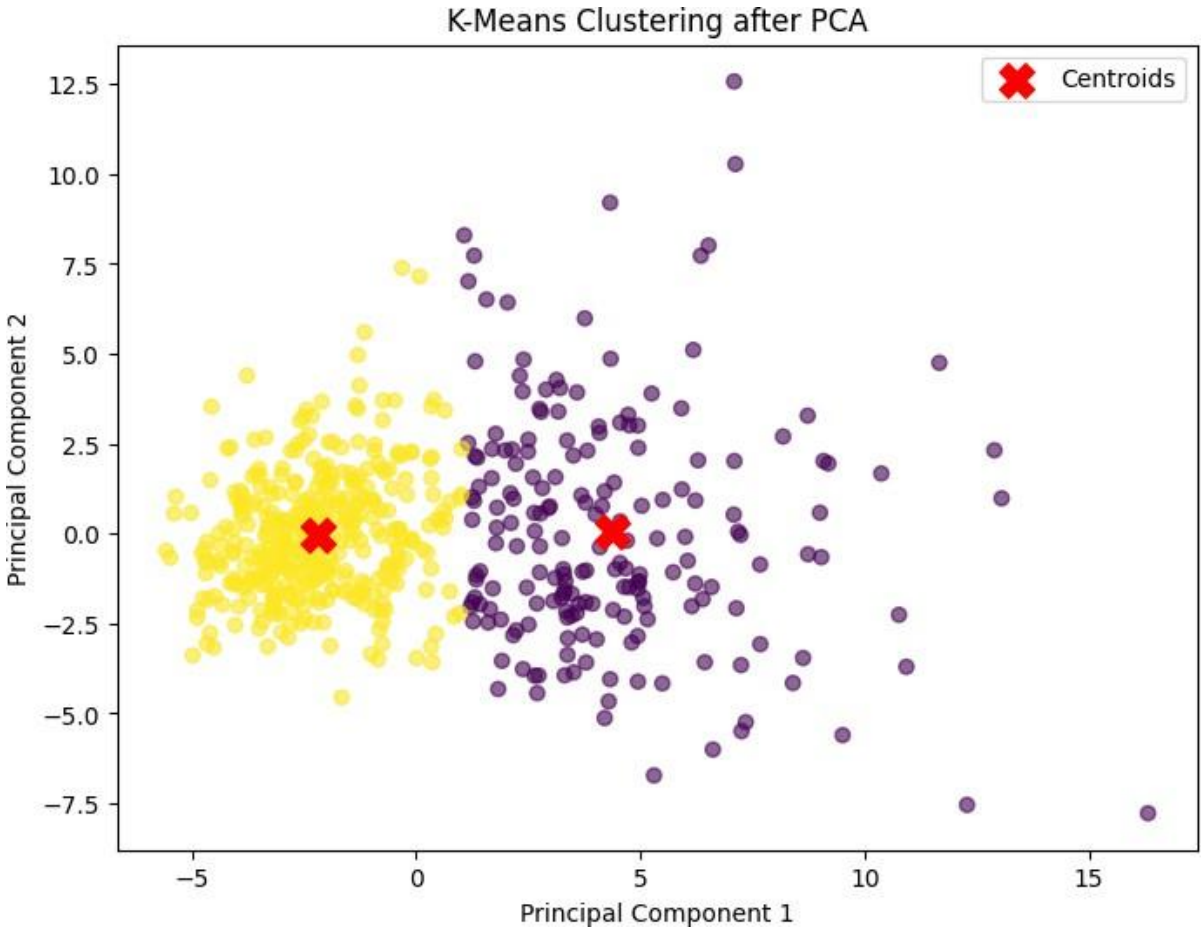
Applying K-Means Clustering

- Chose k=2 (as expected for malignant vs. benign).
- Assigned cluster labels to data points.

In [50]:
```
#Apply K-Means Clustering with the optimal k (usually where elbow occurs,  k=2)
optimal_k = 2
kmeans = KMeans(n_clusters=optimal_k, random_state=42, n_init=10)
clusters = kmeans.fit_predict(X_pca)
```

In [51]:
```
# Step 7: Visualize the Clusters
plt.figure(figsize=(8, 6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=clusters, cmap="viridis", alpha=0.6)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=200, c=
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("K-Means Clustering after PCA")
```

```
plt.legend()
plt.show()
```



- The Elbow Method should show a bend at k=2, confirming that two clusters are optimal.
- The final scatter plot should show two distinct clusters corresponding to malignant and benign tumors.