Table of Contents

Experiment 1

Develop a program to create histograms for all numerical features and analyze the distribution of each feature. Generate box plots for all numerical features and identify any outliers. Use California Housing dataset.

Introduction

Data visualization is a crucial step in exploratory data analysis (EDA), enabling data scientists to understand the distribution and spread of numerical features. Two widely used visualization techniques for analyzing numerical data are histograms and box plots. These plots help identify patterns, trends, and potential anomalies in datasets, making them valuable tools for data preprocessing and feature engineering.

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.

1. 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.

2. 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.

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.

About Dataset

Context

This is the dataset used in the second chapter of Aurélien Géron's recent book 'Hands-On Machine learning with Scikit-Learn and TensorFlow'. It serves as an excellent introduction to implementing machine learning algorithms because it requires rudimentary data cleaning, has an easily understandable list of variables and sits at an optimal size between being to toyish and too cumbersome.

The data contains information from the 1990 California census. So although it may not help you with predicting current housing prices like the Zillow Zestimate dataset, it does provide an accessible introductory dataset for teaching people about the basics of machine learning.

Content

The data pertains to the houses found in a given California district and some summary stats about them based on the 1990 census data. Be warned the data aren't cleaned so there are some preprocessing steps required! The columns are as follows, their names are pretty self explanitory:

longitude

latitude

housing_median_age

total_rooms

total_bedrooms

population

households

median_income

median_house_value (Target)

ocean_proximity

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 [18]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

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

In [19]:
```
df = pd.read_csv(r'C:\Users\vijay\Desktop\Machine Learning Course Batches\FDP_ML_6t
```

In [20]:
```
df.head()
```

Out[20]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	hou
0	-122.23	37.88	41.0	880.0	129.0	322.0	
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	
2	-122.24	37.85	52.0	1467.0	190.0	496.0	
3	-122.25	37.85	52.0	1274.0	235.0	558.0	
4	-122.25	37.85	52.0	1627.0	280.0	565.0	

In [21]:
```
df.shape
```

Out[21]:
```
(20640, 10)
```

In [22]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object 
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

In [23]:
```
df.nunique()
```

Out[23]:
```
longitude               844
latitude                862
housing_median_age       52
total_rooms            5926
total_bedrooms         1923
population             3888
households             1815
median_income         12928
median_house_value     3842
ocean_proximity           5
dtype: int64
```

Data Cleaning

In [24]:
```
df.isnull().sum()
```

Out[24]:
```
longitude               0
latitude                0
housing_median_age      0
total_rooms             0
total_bedrooms        207
population              0
households              0
median_income           0
median_house_value      0
ocean_proximity         0
dtype: int64
```

In [25]:
```
df.duplicated().sum()
```

Out[25]:
```
0
```

In [26]:
```
df['total_bedrooms'].median()
```

Out[26]:
```
435.0
```

In [27]:
```
# Handling missing values
df['total_bedrooms'].fillna(df['total_bedrooms'].median(), inplace=True)
```

Feature Engineering

In [28]:
```
for i in df.iloc[:,2:7]:
    df[i] = df[i].astype('int')
```

In [29]:
```
df.head()
```

Out[29]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	hou
0	-122.23	37.88	41	880	129	322	
1	-122.22	37.86	21	7099	1106	2401	
2	-122.24	37.85	52	1467	190	496	
3	-122.25	37.85	52	1274	235	558	
4	-122.25	37.85	52	1627	280	565	

Disciptive Statistics

In [30]:
```
df.describe().T
```

Out[30]:

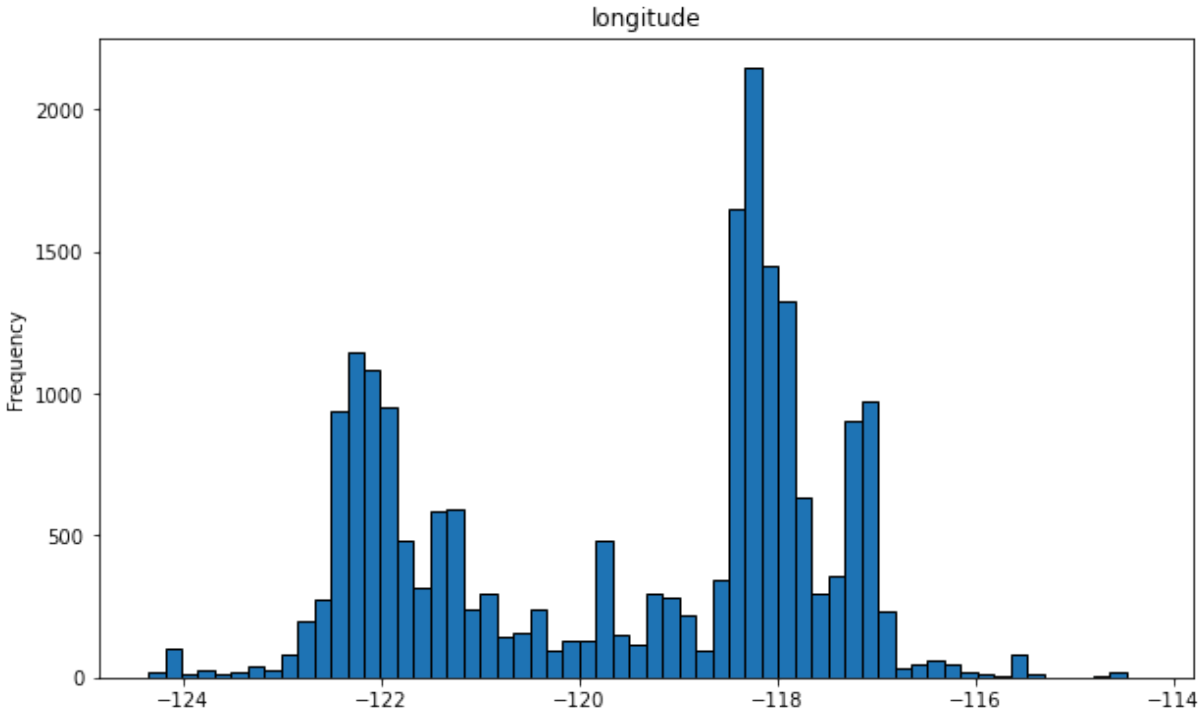
	count	mean	std	min	25%	
longitude	20640.0	-119.569704	2.003532	-124.3500	-121.8000	-1
latitude	20640.0	35.631861	2.135952	32.5400	33.9300	
housing_median_age	20640.0	28.639486	12.585558	1.0000	18.0000	
total_rooms	20640.0	2635.763081	2181.615252	2.0000	1447.7500	21
total_bedrooms	20640.0	536.838857	419.391878	1.0000	297.0000	4
population	20640.0	1425.476744	1132.462122	3.0000	787.0000	11
households	20640.0	499.539680	382.329753	1.0000	280.0000	4
median_income	20640.0	3.870671	1.899822	0.4999	2.5634	
median_house_value	20640.0	206855.816909	115395.615874	14999.0000	119600.0000	1797

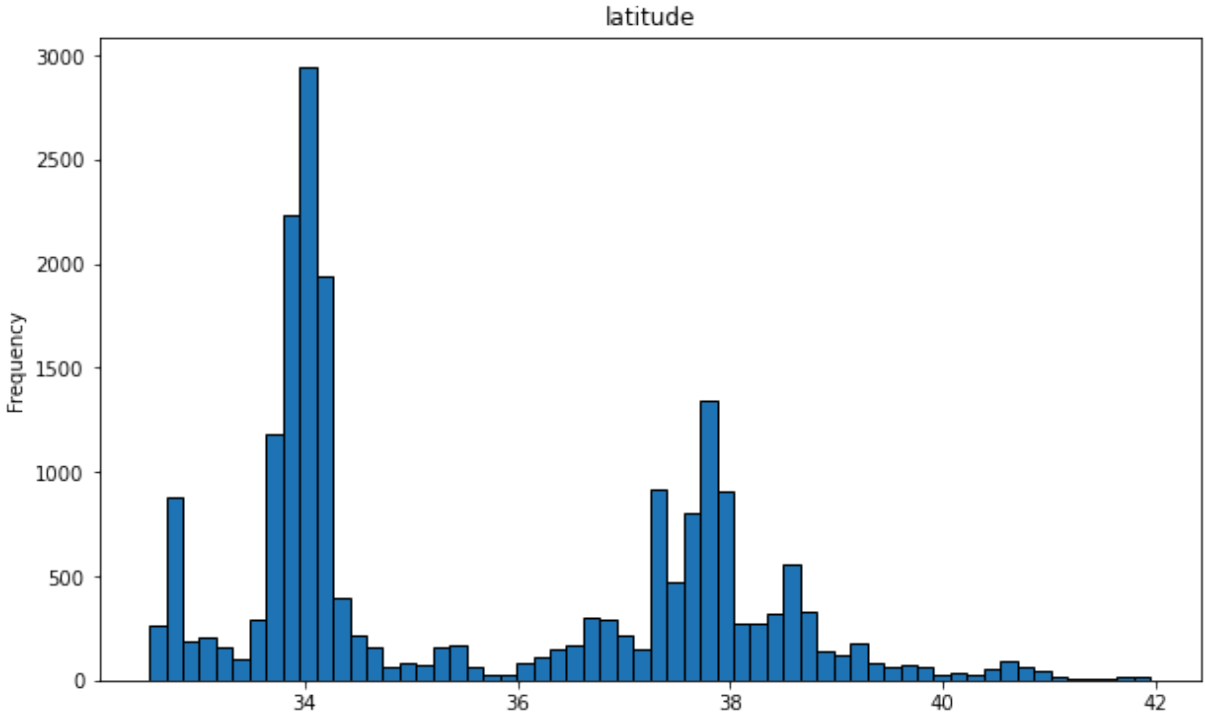
In [31]:
```
Numerical = df.select_dtypes(include=[np.number]).columns
print(Numerical)
```

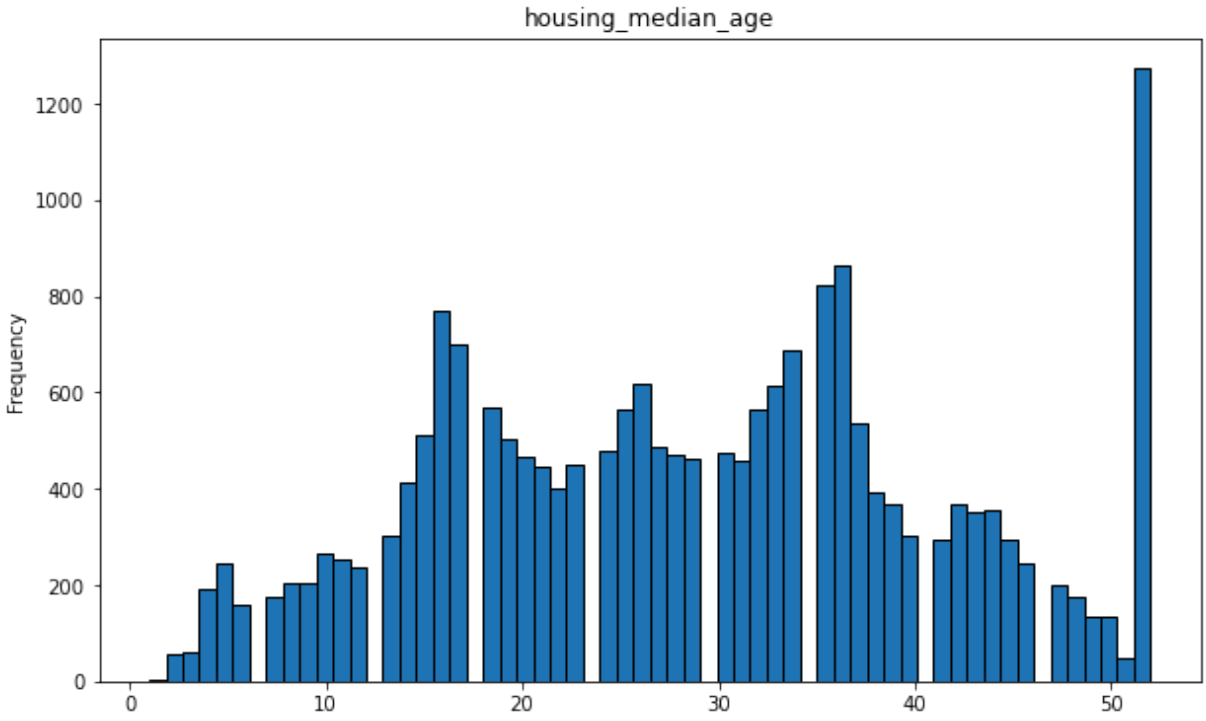
```
Index(['longitude', 'latitude', 'housing_median_age', 'total_rooms',
       'total_bedrooms', 'population', 'households', 'median_income',
       'median_house_value'],
      dtype='object')
```

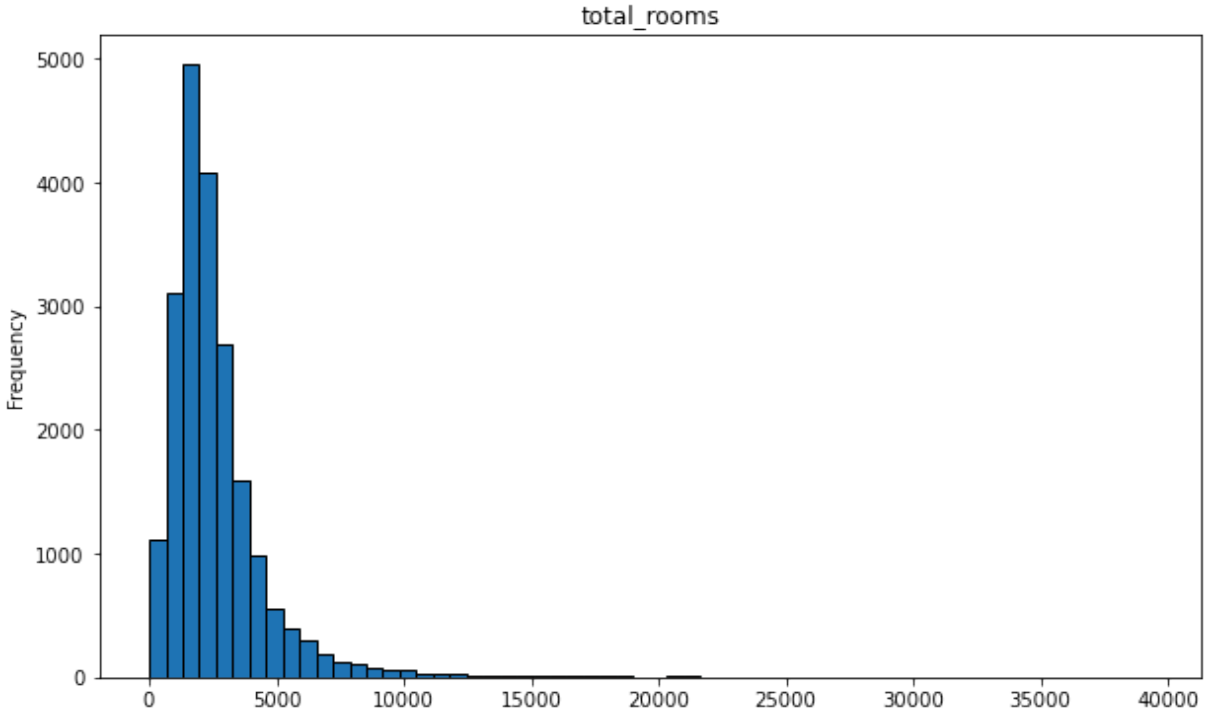
Uni-Variate Analysis

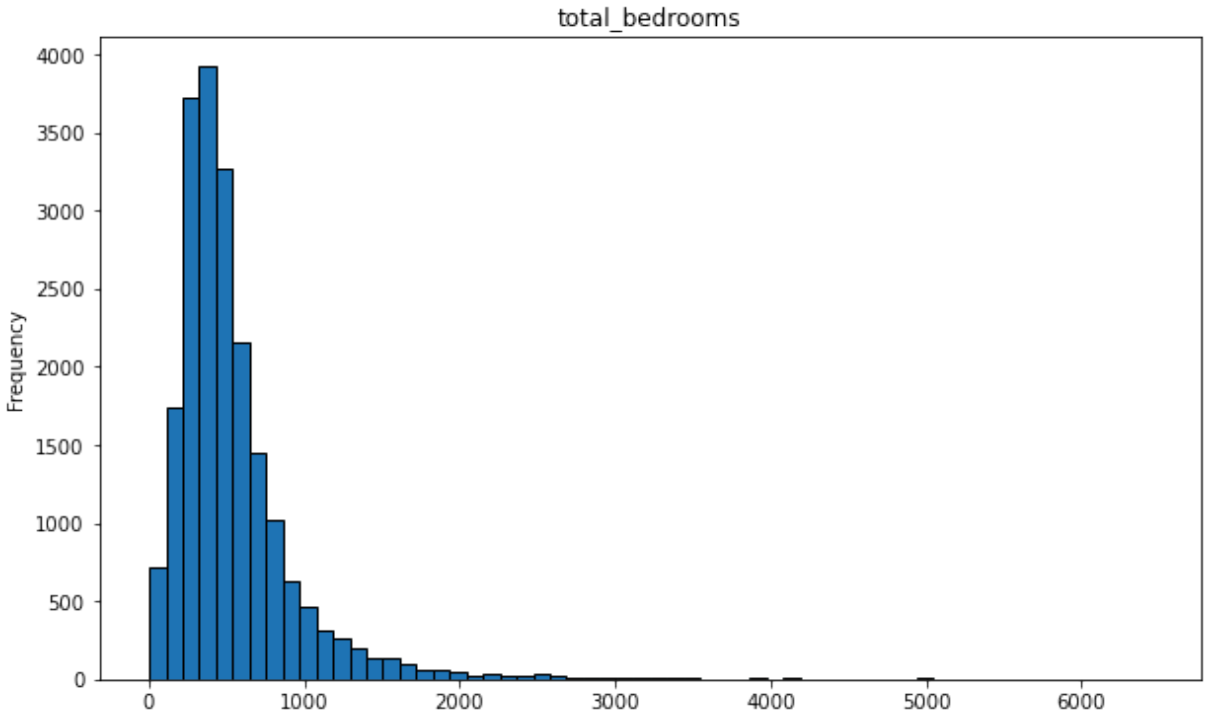
In [32]:
```
for col in Numerical:
    plt.figure(figsize=(10, 6))

    df[col].plot(kind='hist', title=col, bins=60, edgecolor='black')
    plt.ylabel('Frequency')

    plt.show()
```

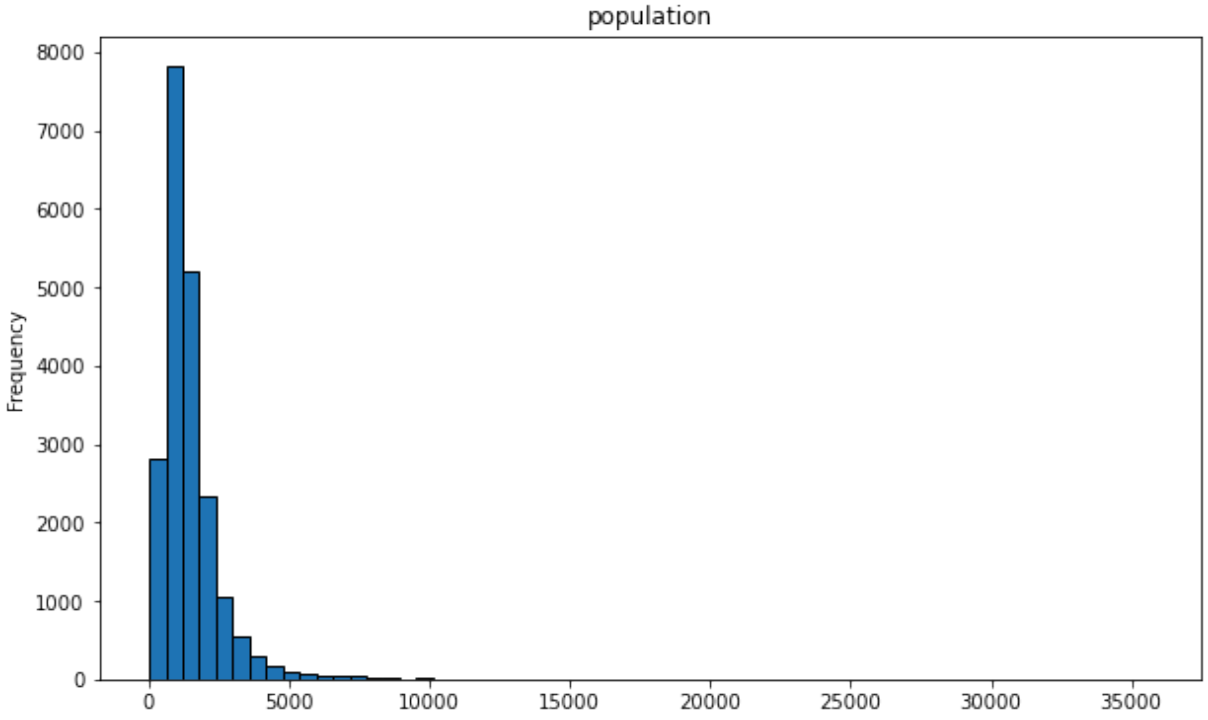

longitude
Frequency
2000
1500
1000
500
0
−124
−122
−120
−118
−116
−114

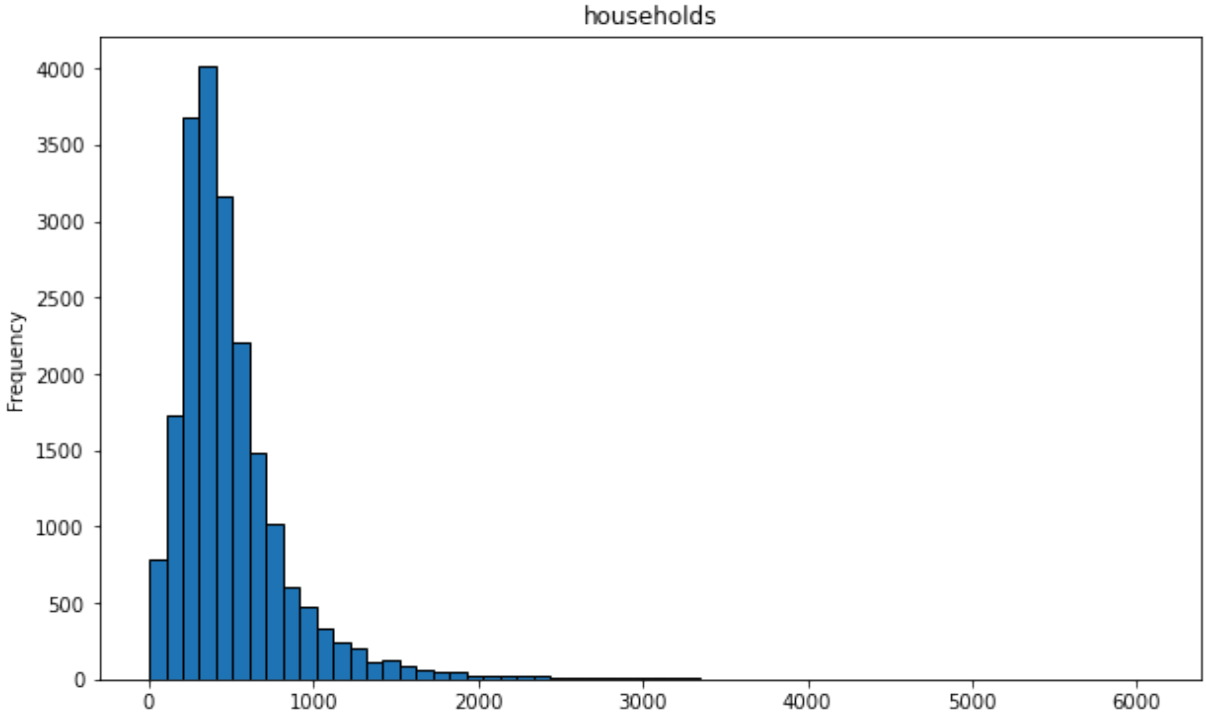

latitude
Frequency
3000
2500
2000
1500
1000
500
0
34
36
38
40
42

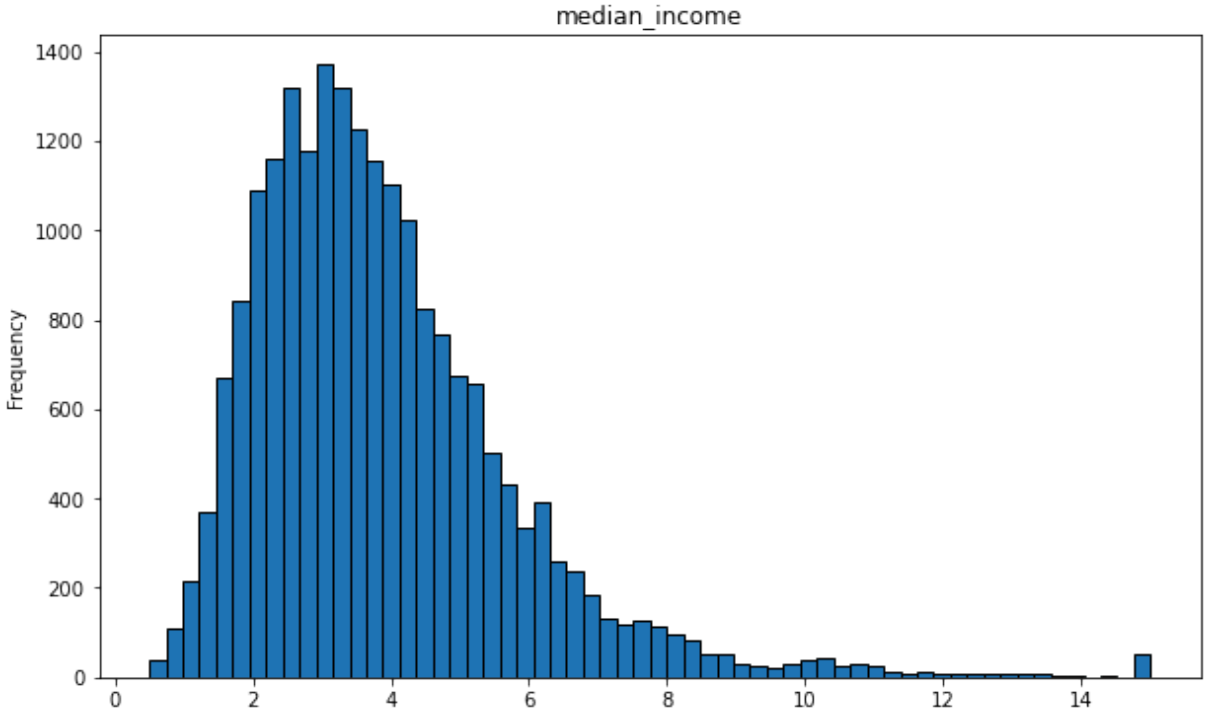

housing_median_age
Frequency
0
200
400
600
800
1000
1200
0
10
20
30
40
50

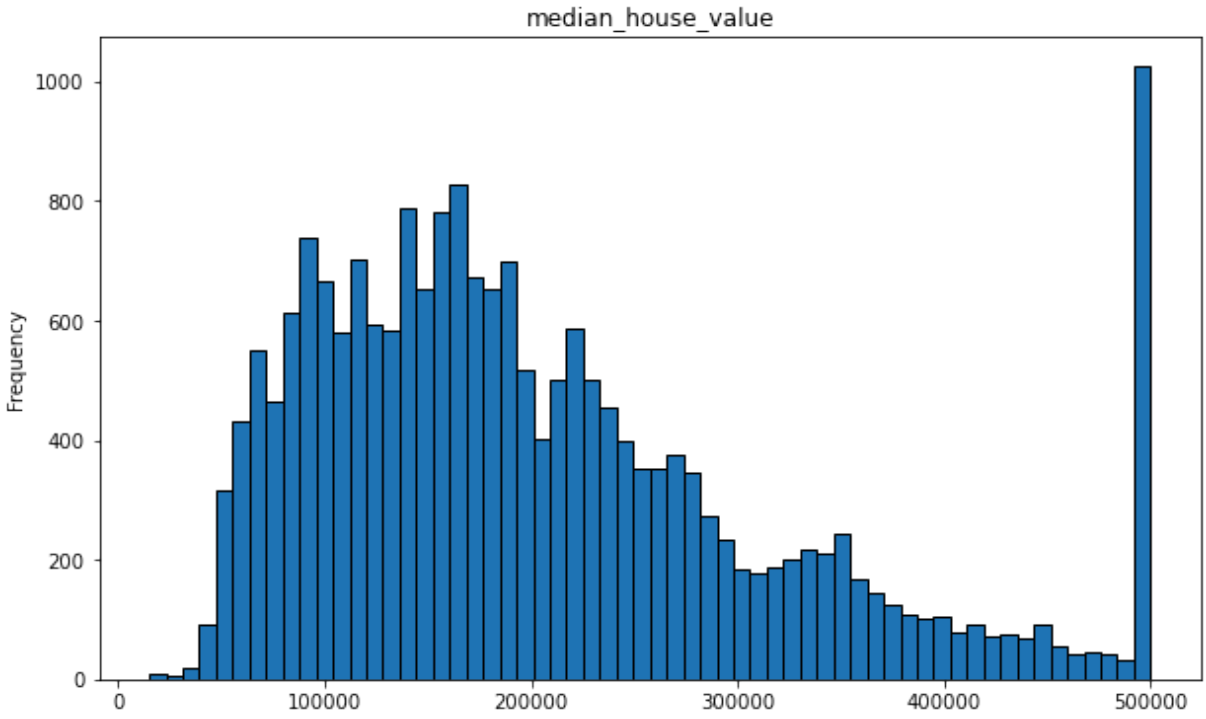

total_rooms
Frequency
0
1000
2000
3000
4000
5000
0
5000
10000
15000
20000
25000
30000
35000
40000


total_bedrooms
Frequency
4000
3500
3000
2500
2000
1500
1000
500
0
0
1000
2000
3000
4000
5000
6000


population
Frequency
8000
7000
6000
5000
4000
3000
2000
1000
0
0
5000
10000
15000
20000
25000
30000
35000


households
Frequency
4000
3500
3000
2500
2000
1500
1000
500
0
0
1000
2000
3000
4000
5000
6000


median_income
Frequency
1400
1200
1000
800
600
400
200
0
0
2
4
6
8
10
12
14



1. Longitude:

   The dataset contains houses located in specific regions (possibly coastal areas or urban zones) as indicated by the bimodal peaks. Houses are not uniformly distributed across all longitudes.

2. Latitude:

   Similar to longitude, the latitude distribution shows houses concentrated in particular zones. This suggests geographic clustering, possibly around major cities.

3. Housing Median Age:

   Most houses are relatively older, with the majority concentrated in a specific range of median ages. This might imply that housing development peaked during certain decades.

4. Total Rooms:

   The highly skewed distribution shows most houses have a lower total number of rooms. A few properties with a very high number of rooms could represent outliers (e.g., mansions or multi-unit buildings).

5. Median Income:

   Most households fall within a low-to-mid income bracket. The steep decline after the peak suggests a small proportion of high-income households in the dataset.

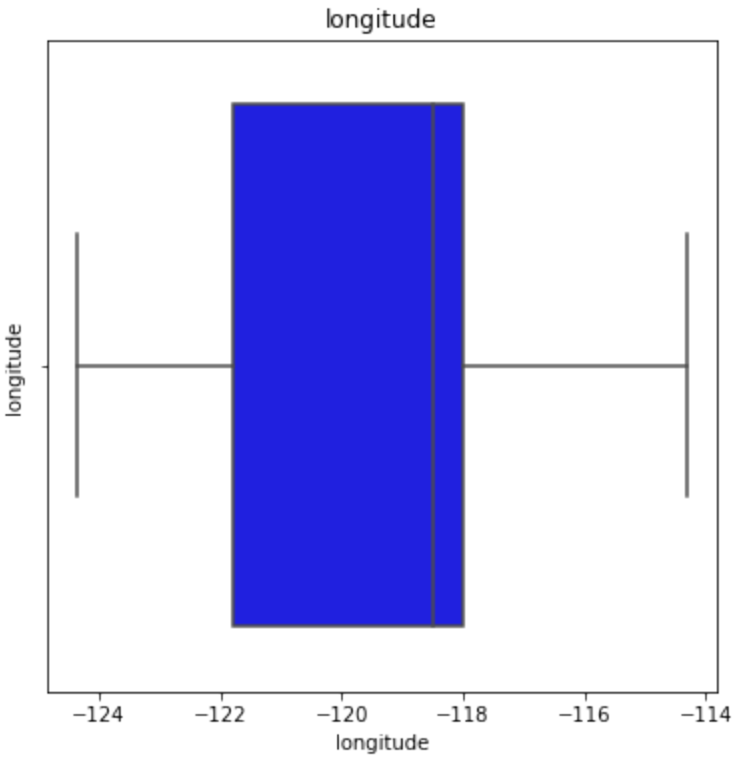
6. Population:

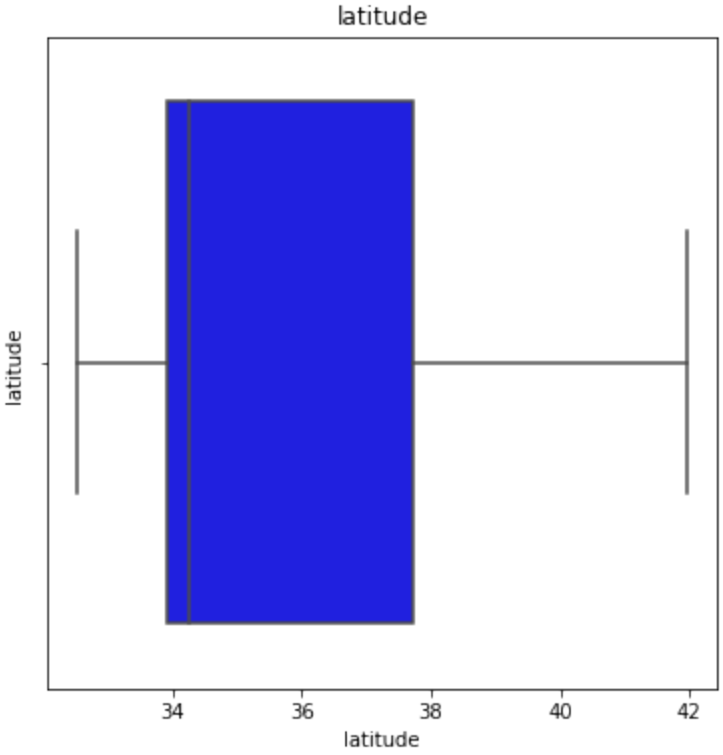
   Most areas in the dataset have a relatively low population. However, there are some highly populated areas, as evidenced by the long tail. These may represent urban centers.

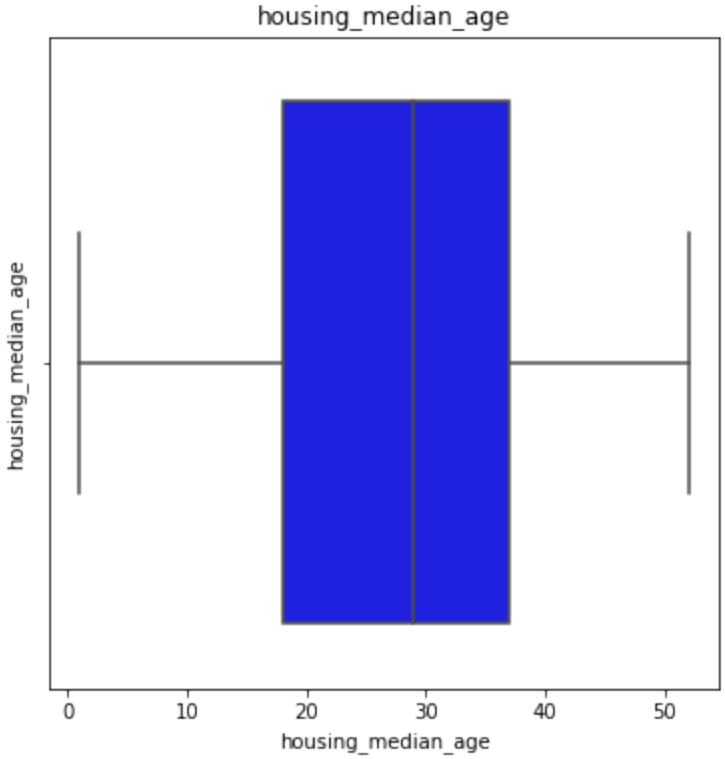
7. Median House Value:

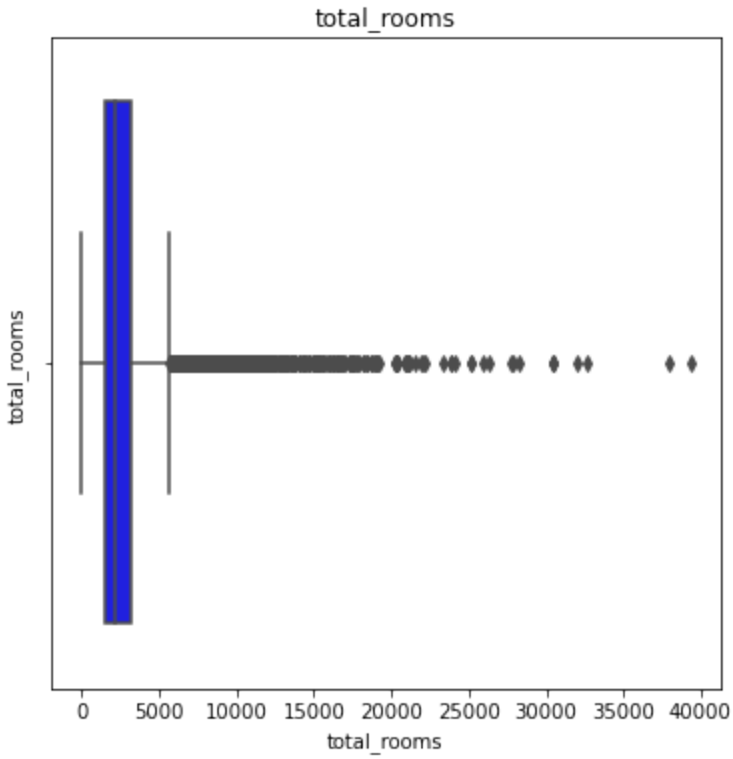
   The sharp peak at the end of the histogram suggests that house prices in the dataset are capped at a maximum value, which could limit the variability in predictions.

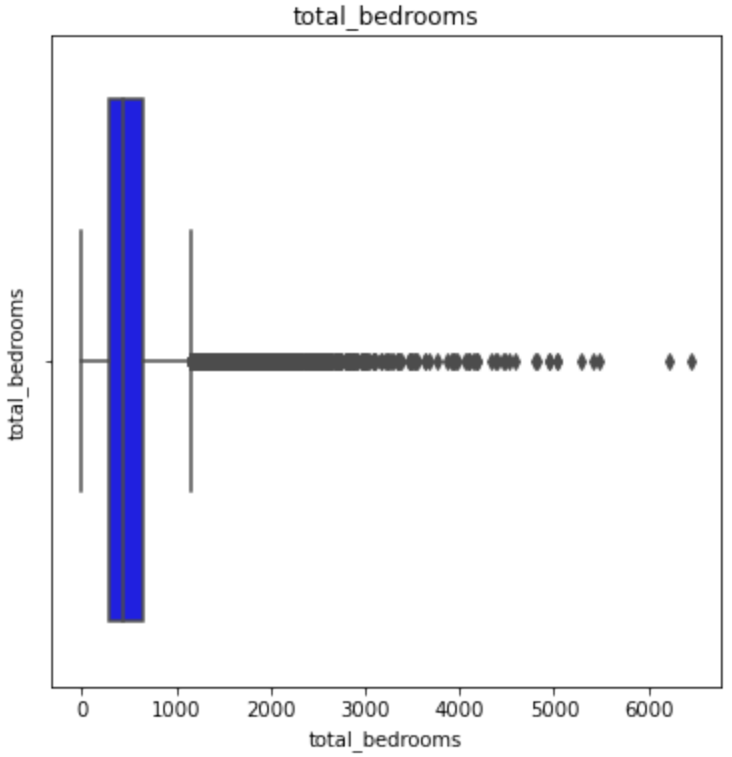
In [33]:
```
for col in Numerical:
    plt.figure(figsize=(6, 6))

    sns.boxplot(df[col], color='blue')
    plt.title(col)
    plt.ylabel(col)

    plt.show()
```

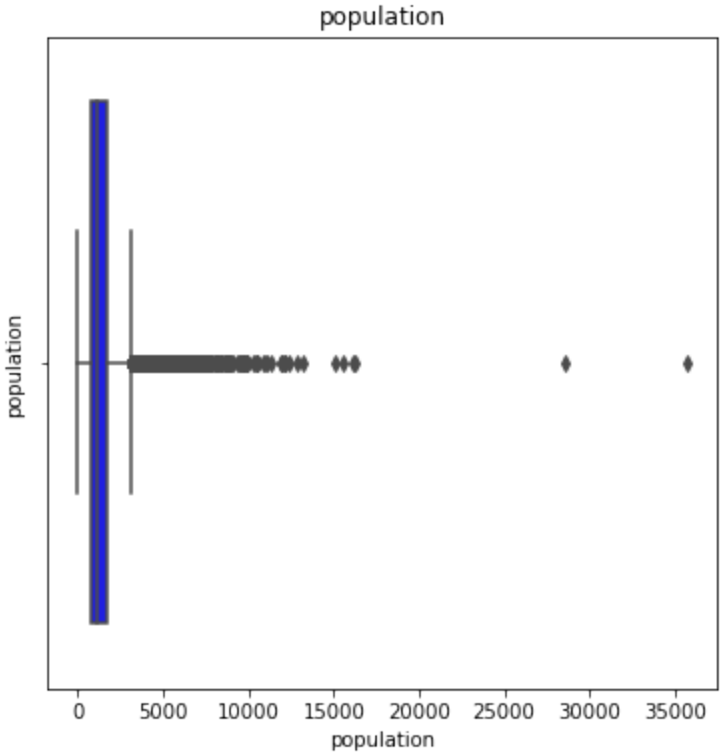


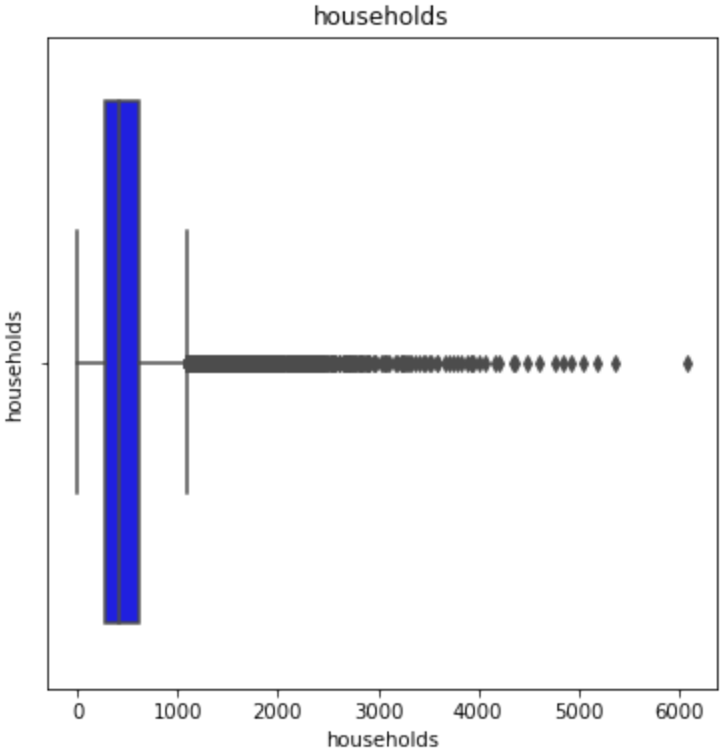

latitude
latitude
34
36
38
40
42
latitude

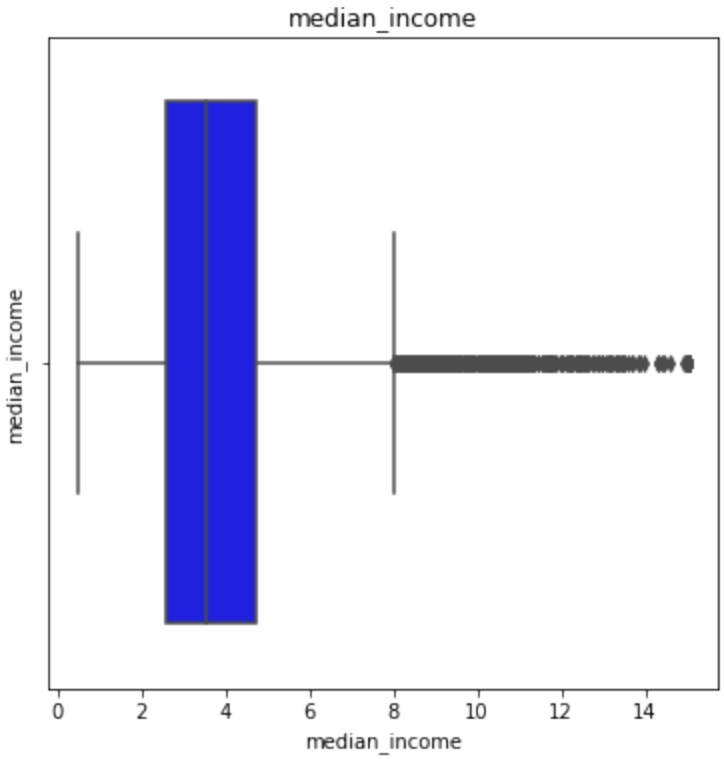

housing_median_age
housing_median_age
0
10
20
30
40
50
housing_median_age

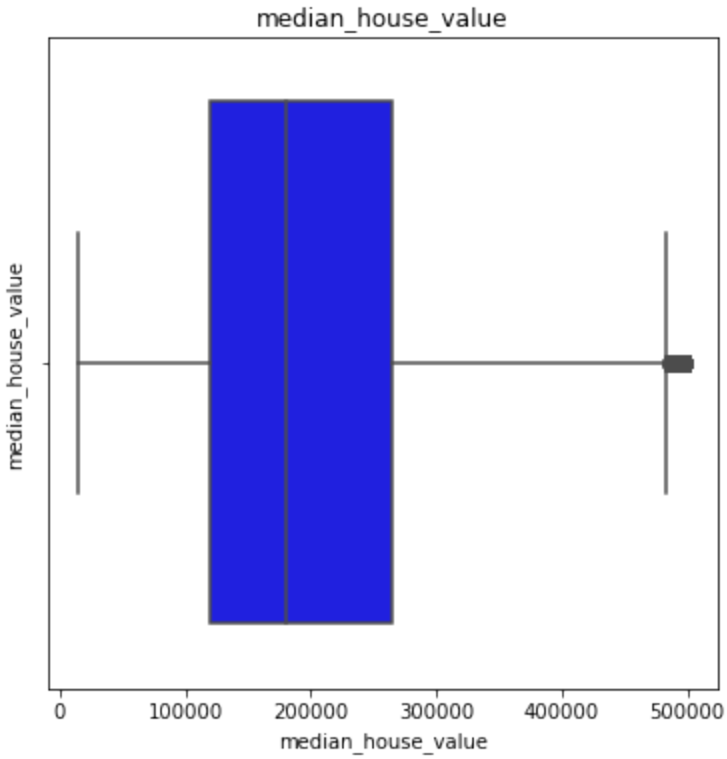

total_rooms
total_rooms
0
5000
10000
15000
20000
25000
30000
35000
40000
total_rooms


total_bedrooms
total_bedrooms
0
1000
2000
3000
4000
5000
6000
total_bedrooms


population
population
0
5000
10000
15000
20000
25000
30000
35000
population


households
households
0
1000
2000
3000
4000
5000
6000
households





Outlier Analysis for Each Feature:

1. Total Rooms: There are numerous data points above the upper whisker, indicating a significant number of outliers.
2. Total Bedrooms: Numerous data points above the upper whisker indicate a significant presence of outliers with very high total_bedrooms values.

3. Population: There are numerous outliers above the upper whisker, with extreme population values reaching beyond 35,000.

4. Households There is a significant number of outliers above the upper whisker. These values represent areas with an unusually high number of households.

5. Median Income: There are numerous data points above the upper whisker, marked as circles. These are considered potential outliers.

6. Median House Value: A small cluster of outliers is visible near the maximum value of 500,000.

General Actions for Outlier Handling:

- Transformation: Apply log or square root transformations to reduce skewness for features like total rooms, population, and median income.
- Removal: If outliers are due to data errors or are not relevant, consider removing them.

Experiment 2

Develop a program to Compute the correlation matrix to understand the relationships between pairs of features. Visualize the correlation matrix using a heatmap to know which variables have strong positive/negative correlations. Create a pair plot to visualize pairwise relationships between features. Use California Housing dataset.

Introduction

In data analysis and machine learning, understanding the relationships between features is crucial for feature selection, multicollinearity detection, and data interpretation. Correlation and pair plots are two essential techniques to analyze these relationships.

1. Correlation Matrix

A **correlation matrix** is a table showing correlation coefficients between variables. It helps in understanding how strongly features are related to each other.

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.

Why Should You Use a Correlation Matrix?

- Identifies relationships between features.
- Helps in detecting multicollinearity in machine learning models.
- Highlights redundant features that may not add value to the model.

2. 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.

3. 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.

In [13]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import fetch_california_housing

# Load California Housing dataset
data = fetch_california_housing()

# Convert to DataFrame
df = pd.DataFrame(data.data, columns=data.feature_names)
df['Target'] = data.target  # Adding the target variable (median house value)
```

In [11]:
```
# Table of Meaning of Each Variable
variable_meaning = {
    "MedInc": "Median income in block group",
    "HouseAge": "Median house age in block group",
    "AveRooms": "Average number of rooms per household",
    "AveBedrms": "Average number of bedrooms per household",
    "Population": "Population of block group",
    "AveOccup": "Average number of household members",
    "Latitude": "Latitude of block group",
    "Longitude": "Longitude of block group",
    "Target": "Median house value (in $100,000s)"
}

variable_df = pd.DataFrame(list(variable_meaning.items()), columns=["Feature", "Des
print("\nVariable Meaning Table:")
print(variable_df)
```

```
Variable Meaning Table:
      Feature                                Description
0      MedInc              Median income in block group
1    HouseAge           Median house age in block group
2    AveRooms     Average number of rooms per household
3   AveBedrms  Average number of bedrooms per household
4  Population                  Population of block group
5    AveOccup       Average number of household members
6    Latitude                    Latitude of block group
7   Longitude                   Longitude of block group
8      Target         Median house value (in $100,000s)
```

In [4]:
```
# Basic Data Exploration
print("\nBasic Information about Dataset:")
print(df.info())  # Overview of dataset
print("\nFirst Five Rows of Dataset:")
print(df.head())  # Display first few rows
```

```
Basic Information about Dataset:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 9 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   MedInc      20640 non-null  float64
 1   HouseAge    20640 non-null  float64
 2   AveRooms    20640 non-null  float64
 3   AveBedrms   20640 non-null  float64
 4   Population  20640 non-null  float64
 5   AveOccup    20640 non-null  float64
 6   Latitude    20640 non-null  float64
 7   Longitude   20640 non-null  float64
 8   Target      20640 non-null  float64
dtypes: float64(9)
memory usage: 1.4 MB
None

First Five Rows of Dataset:
   MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0  8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88   
1  8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86   
2  7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85   
3  5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85   
4  3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85   

   Longitude  Target
0    -122.23   4.526
1    -122.22   3.585
2    -122.24   3.521
3    -122.25   3.413
4    -122.25   3.422
```

In [5]:
```
# Summary Statistics
print("\nSummary Statistics:")
print(df.describe())  # Summary statistics of dataset
```

```
Summary Statistics:
             MedInc      HouseAge      AveRooms     AveBedrms    Population  \
count  20640.000000  20640.000000  20640.000000  20640.000000  20640.000000
mean       3.870671     28.639486      5.429000      1.096675   1425.476744
std        1.899822     12.585558      2.474173      0.473911   1132.462122
min        0.499900      1.000000      0.846154      0.333333      3.000000
25%        2.563400     18.000000      4.440716      1.006079    787.000000
50%        3.534800     29.000000      5.229129      1.048780   1166.000000
75%        4.743250     37.000000      6.052381      1.099526   1725.000000
max       15.000100     52.000000    141.909091     34.066667  35682.000000

            AveOccup      Latitude     Longitude        Target
count  20640.000000  20640.000000  20640.000000  20640.000000
mean       3.070655     35.631861   -119.569704      2.068558
std       10.386050      2.135952      2.003532      1.153956
min        0.692308     32.540000   -124.350000      0.149990
25%        2.429741     33.930000   -121.800000      1.196000
50%        2.818116     34.260000   -118.490000      1.797000
75%        3.282261     37.710000   -118.010000      2.647250
max     1243.333333     41.950000   -114.310000      5.000010
```

In [12]:
```
# Explanation of Summary Statistics
summary_explanation = """
The summary statistics table provides key percentiles and other descriptive metrics
- **25% (First Quartile - Q1):** This represents the value below which 25% of the d
- **50% (Median - Q2):** This is the middle value when the data is sorted. It provi
- **75% (Third Quartile - Q3):** This represents the value below which 75% of the d
- These percentiles are useful for detecting skewness, data distribution, and ident
"""
print("\nSummary Statistics Explanation:")
print(summary_explanation)
```

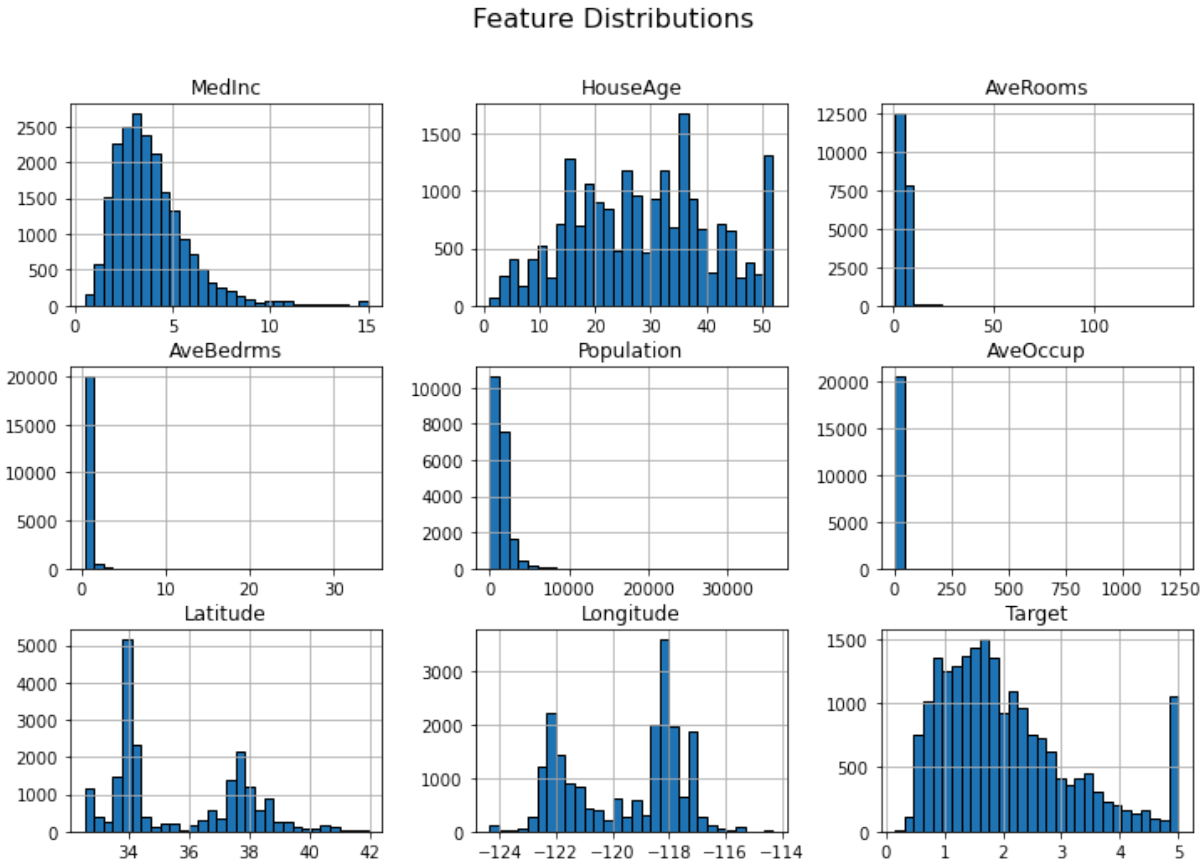
```
Summary Statistics Explanation:

The summary statistics table provides key percentiles and other descriptive metrics
for each numerical feature:
- **25% (First Quartile - Q1):** This represents the value below which 25% of the da
ta falls. It helps in understanding the lower bound of typical data values.
- **50% (Median - Q2):** This is the middle value when the data is sorted. It provid
es the central tendency of the dataset.
- **75% (Third Quartile - Q3):** This represents the value below which 75% of the da
ta falls. It helps in identifying the upper bound of typical values in the dataset.
- These percentiles are useful for detecting skewness, data distribution, and identi
fying potential outliers (values beyond Q1 - 1.5*IQR or Q3 + 1.5*IQR).
```

In [6]:
```
# Check for missing values
print("\nMissing Values in Each Column:")
print(df.isnull().sum())  # Count of missing values
```

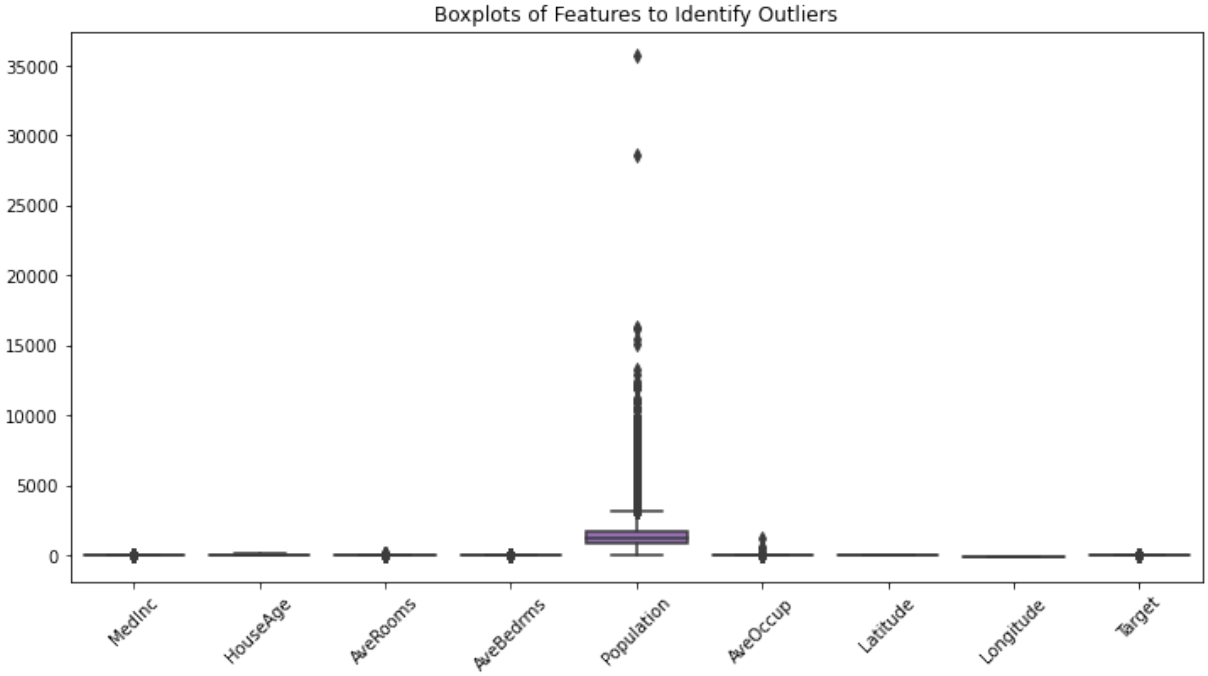
```
Missing Values in Each Column:
MedInc        0
HouseAge      0
AveRooms      0
AveBedrms     0
Population    0
AveOccup      0
Latitude      0
Longitude     0
Target        0
dtype: int64
```

In [7]:
```
# 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()
```

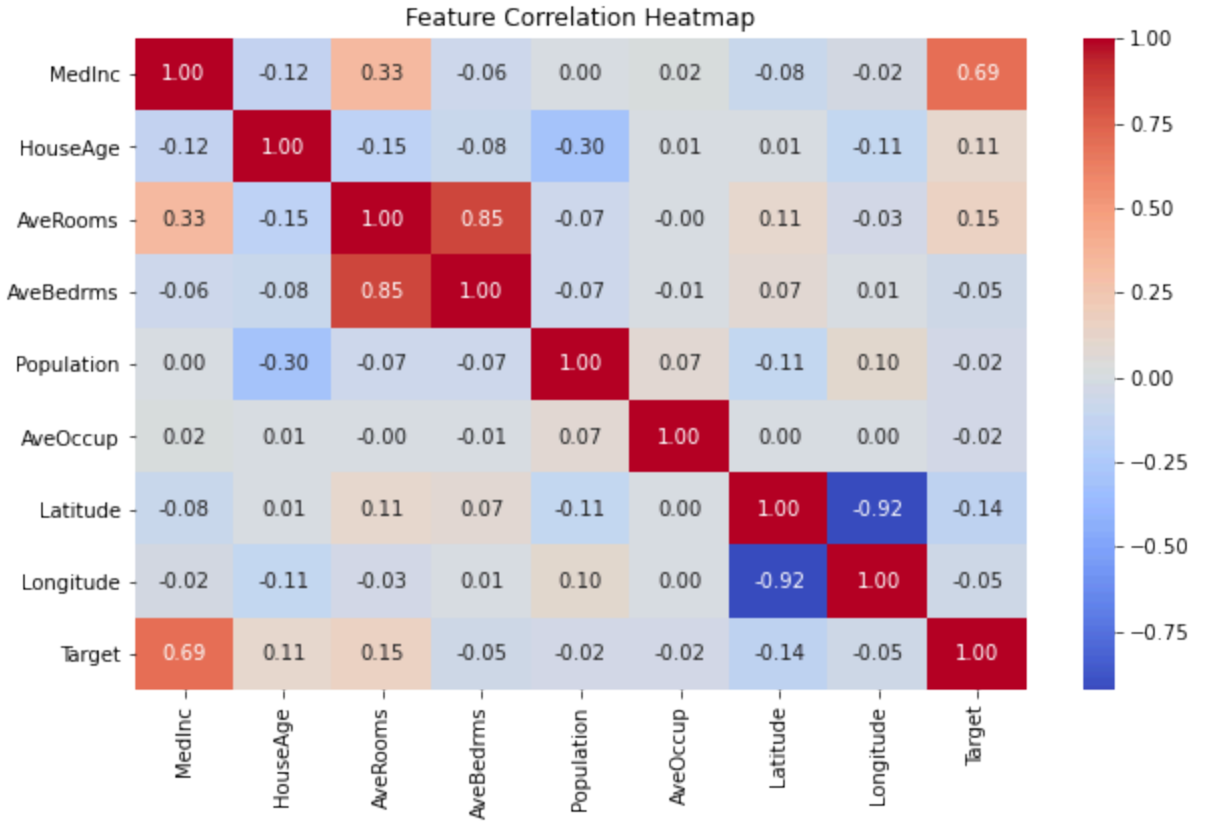
```
<Figure size 864x576 with 0 Axes>
```



In [8]:
```
# 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()
```

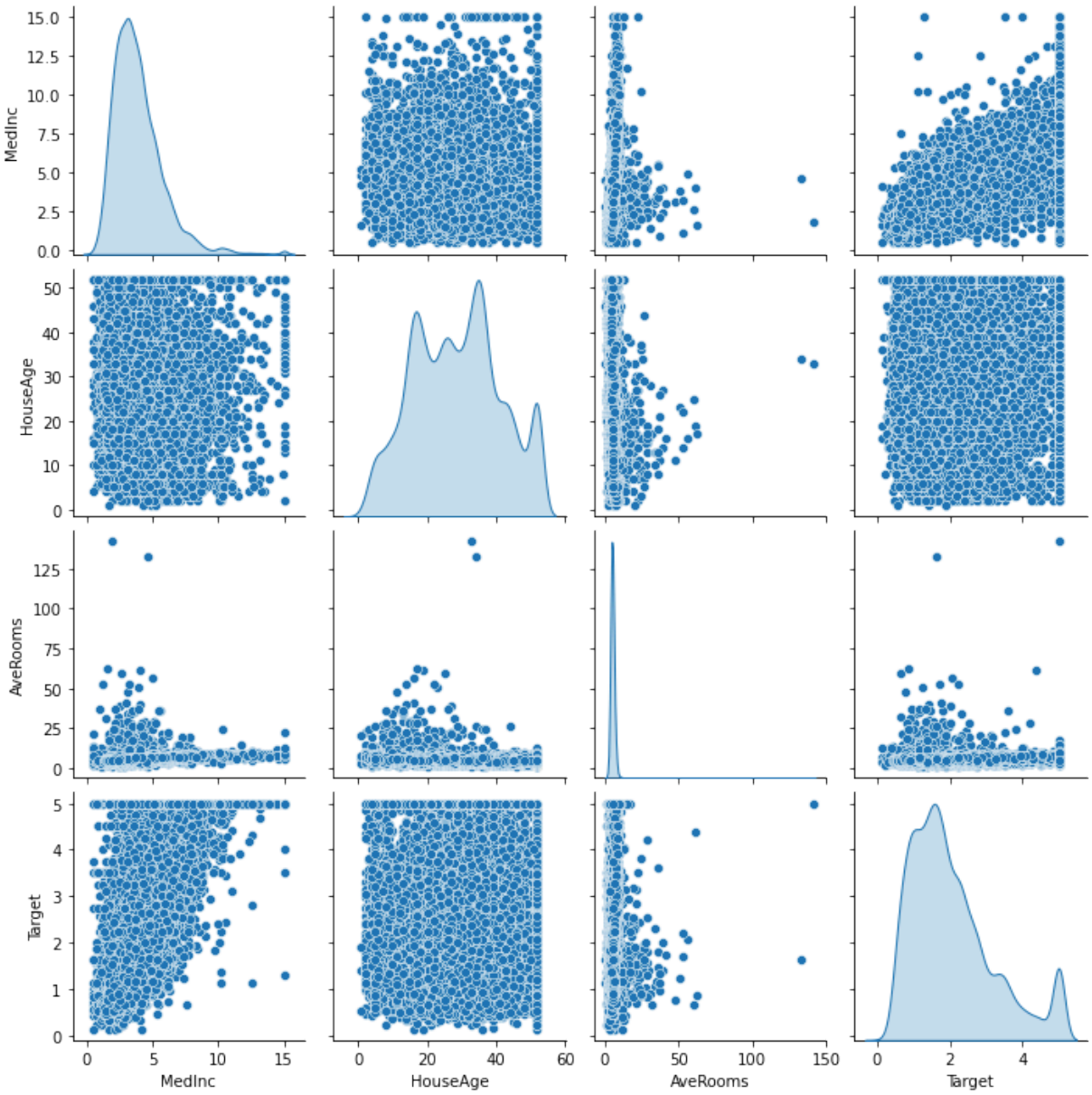


In [9]:
```
# Correlation Matrix
plt.figure(figsize=(10, 6))
corr_matrix = df.corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title("Feature Correlation Heatmap")
plt.show()
```



In [10]:
```
# Pairplot to analyze feature relationships (only a subset for clarity)
sns.pairplot(df[['MedInc', 'HouseAge', 'AveRooms', 'Target']], diag_kind='kde')
```

```
plt.show()

# Insights from Data Exploration
print("\nKey Insights:")
print("1. The dataset has", df.shape[0], "rows and", df.shape[1], "columns.")
print("2. No missing values were found in the dataset.")
print("3. Histograms show skewed distributions in some features like 'MedInc'.")
print("4. Boxplots indicate potential outliers in 'AveRooms' and 'AveOccup'.")
print("5. Correlation heatmap shows 'MedInc' has the highest correlation with house
```




```
Key Insights:
1. The dataset has 20640 rows and 9 columns.
2. No missing values were found in the dataset.
3. Histograms show skewed distributions in some features like 'MedInc'.
4. Boxplots indicate potential outliers in 'AveRooms' and 'AveOccup'.
5. Correlation heatmap shows 'MedInc' has the highest correlation with house prices.
```

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.
```

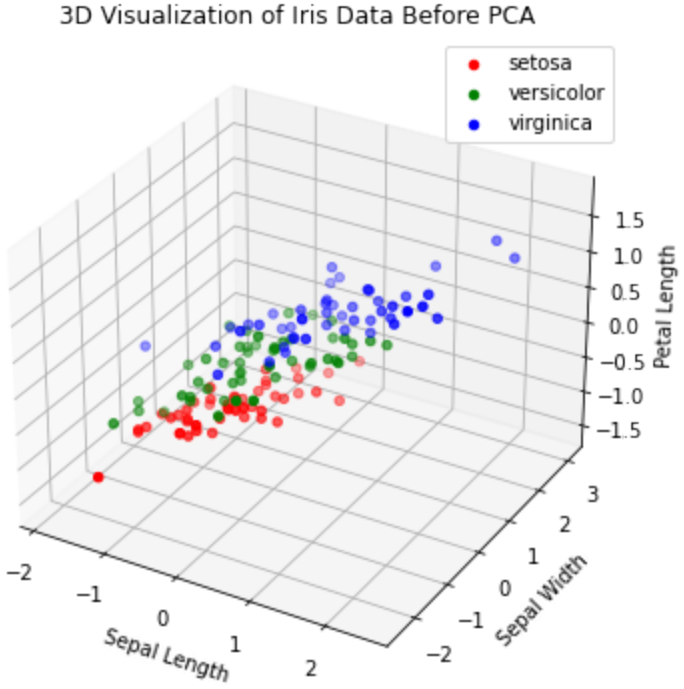
Since the dataset contains three classes (Setosa, Versicolor, and Virginica), PCA helps visualize how well the classes can be separated in a lower-dimensional space.

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)
```

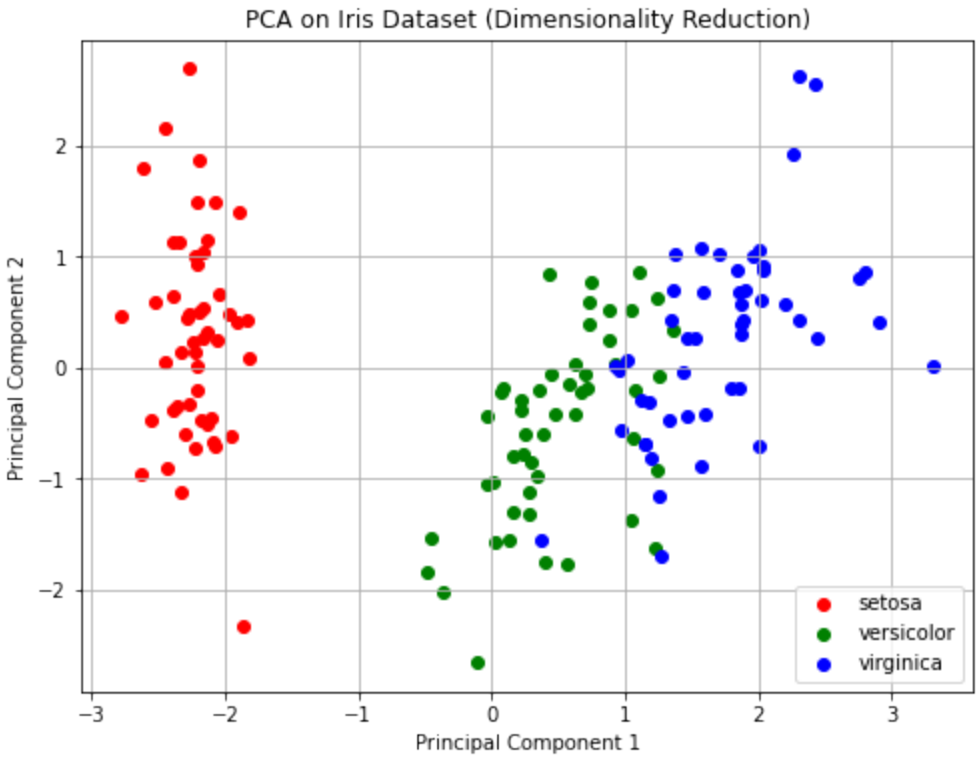
```
# Step 2: Standardizing the Data
# PCA works best when data is standardized (mean = 0, variance = 1)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 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()
```

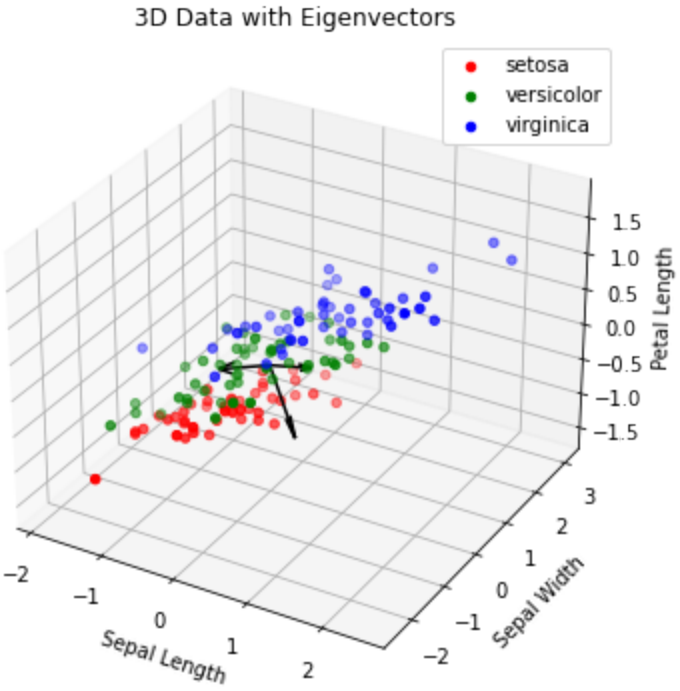
```python
# Step 9: Visualizing Eigenvectors Superimposed on 3D Data
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
for i in range(len(colors)):
    ax.scatter(X_scaled[y == i, 0], X_scaled[y == i, 1], X_scaled[y == i, 2], color
for i in range(3):  # Plot first three eigenvectors
    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
0        Yes       Masters  Python   30   Yes
1        Yes     Bachelors  Python   25   Yes
2         No     Bachelors    Java   28    No
3        Yes       Masters    Java   40   Yes
4         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' '?' '?' '?']
```

In [ ]:

Experiment 5

Develop a program to implement k-Nearest Neighbour algorithm to classify the randomly generated 100 values of x in the range of [0,1]. Perform the following based on dataset generated.

1. Label the first 50 points {x1,……,x50} as follows: if (xi ≤ 0.5), then xi ε Class1, else xi ε Class2

2. Classify the remaining points, x51,……,x100 using KNN. Perform this for k=1,2,3,4,5,20,30

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.

How k-NN Works?

The k-NN algorithm follows these steps:

1. **Choose the value of k** (number of nearest neighbors).
2. **Compute the distance** between the test sample and all training samples using a distance metric (e.g., Euclidean distance).
3. **Select the k nearest neighbors** (data points with the smallest distance to the test sample).
4. **Assign the majority class** among the k neighbors to the test sample.

Working of the k-NN Algorithm

1. **Choose a Value for k:**

- A **small k (e.g., k=1)** makes the model sensitive to noise and results in **high variance**.
- A **large k (e.g., k=30)** smooths the decision boundary but may lead to **high bias**.
- The optimal k is usually found by **cross-validation**.

2. **Compute Distance Between Data Points:** The algorithm relies on a distance metric to determine similarity between data points. Common distance measures 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]`

3. **3. Decision Rule for Classification**
   - **Majority Voting:** The most common class among the k neighbors determines the predicted class.

- **Weighted Voting:** Closer neighbors have higher influence on the prediction than farther neighbors.

Dataset Generation and Classification Task

Step 1: Generate 100 Random Points in the Range [0,1]

- The dataset consists of 100 random values of `x` uniformly distributed between **0 and 1**.

Step 2: Assign Labels to the First 50 Points

- The first **50 points (x_1, x_2, ..., x_{50})** are labeled as:
  - **Class 1** if x_i <= 0.5
  - **Class 2** if x_i > 0.5

Step 3: Classify Remaining Points (x_{51}, ..., x_{100}) using k-NN

- The k-NN algorithm is used to classify the next **50 points** based on the first **50 labeled points**.

Step 4: Experiment with Different k Values

- Classification is performed for multiple values of `k`:
  - k = 1, 2, 3, 4, 5, 20, 30
- Observing how different values of `k` affect classification accuracy and decision boundaries.

Effect of Different k Values

k Value	Effect on Classification
k = 1	Highly sensitive to noise; can result in overfitting.
k = 2, 3, 4, 5	Balanced classification with some noise handling.
k = 20, 30	Smoother decision boundary but may lead to underfitting.

Bias-Variance Tradeoff in k-NN

- **Smaller k values (e.g., k=1)** → Low bias, high variance (more flexible but prone to noise).
- **Larger k values (e.g., k=20, 30)** → High bias, low variance (less flexible but smoother decision boundary).

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.

Problem Explanation

The goal is to implement the k-Nearest Neighbors (KNN) algorithm to classify 100 randomly generated values in the range [0,1]. The classification process involves the following steps:

Generating the Dataset:

Create 100 random values in the range [0,1] The first 50 values are manually labeled based on a given rule: If $xi \leq 0.5$ assign it to Class1.Otherwise, assign it to Class2. Classifying the

Remaining 50 Values Using KNN:

The next 50 values (x51 to x100) are unlabeled. We use the KNN algorithm to classify these values based on their nearest neighbors among the first 50 labeled points.

Perform classification for k = 1, 2, 3, 4, 5, 20, 30.

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

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

In [2]:
```
# Step 1: Generate dataset
np.random.seed(42)
values = np.random.rand(100)
```

In [3]:
```
labels = []

for i in values[:50]:
    if i <=0.5:
        labels.append('Class1')
    else:
        labels.append('Class2')
```

In [4]:
```
labels += [None] * 50
```

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

```
['Class1', 'Class2', 'Class2', 'Class2', 'Class1', 'Class1', 'Class1', 'Class2', 'Cl
ass2', 'Class2', 'Class1', 'Class2', 'Class2', 'Class1', 'Class1', 'Class1', 'Class
1', 'Class2', 'Class1', 'Class1', 'Class2', 'Class1', 'Class1', 'Class1', 'Class1',
'Class2', 'Class1', 'Class2', 'Class2', 'Class1', 'Class2', 'Class1', 'Class1', 'Cla
ss2', 'Class2', 'Class2', 'Class1', 'Class1', 'Class2', 'Class1', 'Class1', 'Class
1', 'Class1', 'Class2', 'Class1', 'Class2', 'Class1', 'Class2', 'Class2', 'Class1',
None, None, None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None]
```

In [6]:
```
data = {
    "Point": [f"x{i+1}" for i in range(100)],
    "Value": values,
    "Label": labels
}
```

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

```
df.head()
```

Out[11]:

	Point	Value	Label
0	x1	0.374540	Class1
1	x2	0.950714	Class2
2	x3	0.731994	Class2
3	x4	0.598658	Class2
4	x5	0.156019	Class1

In [8]:
```
# Table of Meaning of Each Variable
variable_meaning = {
    "Point": "The point number",
    "Value": "The value of the point",
    "Label": "The class of the point"
}
variable_df = pd.DataFrame(list(variable_meaning.items()), columns=["Feature", "Des
print("\nVariable Meaning Table:")
print(variable_df)
```

```
Variable Meaning Table:
  Feature             Description
0   Point        The point number
1   Value  The value of the point
2   Label  The class of the point
```

In [12]:
```
df.nunique()
```

Out[12]:
```
Point    100
Value    100
Label      2
dtype: int64
```

In [13]:
```
df.shape
```

Out[13]:
```
(100, 3)
```

In [16]:
```
# Basic Data Exploration
print("\nBasic Information about Dataset:")
df.info()
```

```
Basic Information about Dataset:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Point   100 non-null    object
 1   Value   100 non-null    float64
 2   Label   50 non-null     object
dtypes: float64(1), object(2)
memory usage: 2.5+ KB
```

In [15]:
```
print("\nSummary Statistics:")
df.describe().T
```

```
Summary Statistics:
```

Out[15]:

	count	mean	std	min	25%	50%	75%	max
Value	100.0	0.470181	0.297489	0.005522	0.193201	0.464142	0.730203	0.986887

In [25]:
```
Summary_Statistics="""
- The 'Value' column has a mean of approximately 0.47, indicating that the values a
- The standard deviation of the 'Value' column is approximately 0.29, showing a mod
- The minimum value in the 'Value' column is approximately 0.0055, and the maximum
- The first quartile (25th percentile) is approximately 0.19, the median (50th perc
print(Summary_Statistics)
```

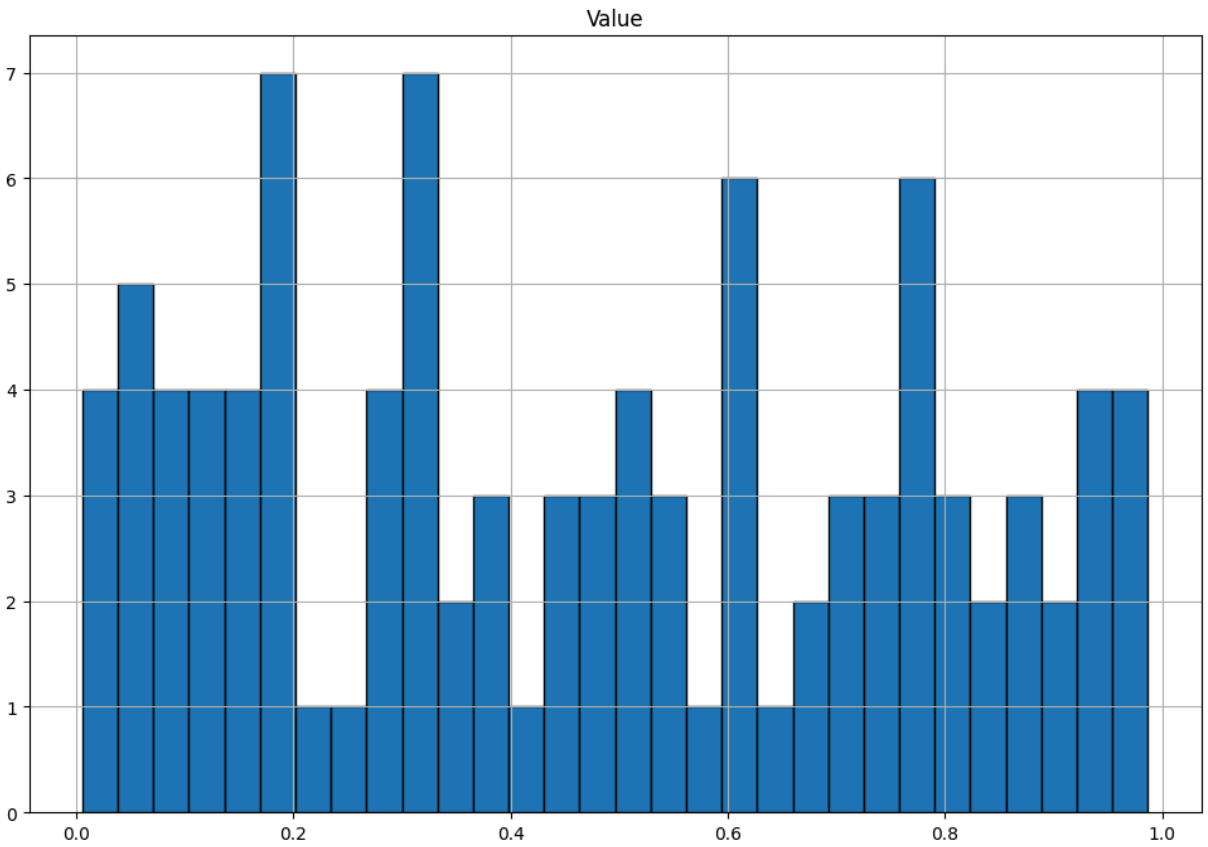
```
- The 'Value' column has a mean of approximately 0.47, indicating that the values ar
e uniformly distributed.
- The standard deviation of the 'Value' column is approximately 0.29, showing a mode
rate spread around the mean.
- The minimum value in the 'Value' column is approximately 0.0055, and the maximum v
alue is approximately 0.9869.
- The first quartile (25th percentile) is approximately 0.19, the median (50th perce
ntile) is approximately 0.47, and the third quartile (75th percentile) is approximat
ely 0.73.
```

In [17]:
```
# Check for missing values
print("\nMissing Values in Each Column:")
df.isnull().sum()
```

```
Missing Values in Each Column:
```

Out[17]:
```
Point     0
Value     0
Label    50
dtype: int64
```

In [16]:
```
# Get numeric columns
num_col = df.select_dtypes(include=['int', 'float']).columns

# Histograms for distribution of features
df[num_col].hist(figsize=(12, 8), bins=30, edgecolor='black')

# Title and labels
plt.suptitle("Feature Distributions", fontsize=16)
plt.show()
```



In [19]:
```
# Inference for the above graph
inference = """
- The histograms for the distribution of features show that the values are uniforml
- This is expected as the values were generated using a uniform random distribution
- There are no significant outliers or skewness in the data, indicating that the da
"""
print(inference)
```

```
- The histograms for the distribution of features show that the values are uniformly
distributed across the range [0, 1].
- This is expected as the values were generated using a uniform random distribution.
- There are no significant outliers or skewness in the data, indicating that the dat
aset is well-balanced.
```

In [17]:
```
# Split data into labeled and unlabeled
labeled_df = df[df["Label"].notna()]
X_train = labeled_df[["Value"]]
y_train = labeled_df["Label"]
```

In [92]:
```
unlabeled_df = df[df["Label"].isna()]
X_test = unlabeled_df[["Value"]]
```

In [93]:
```
# Generate true labels for testing (for accuracy calculation)
true_labels = ["Class1" if x <= 0.5 else "Class2" for x in values[50:]]
```

In [94]:
```
# Step 2: Perform KNN classification for different values of k
k_values = [1, 2, 3, 4, 5, 20, 30]
results = {}
accuracies = {}
```

In [96]:
```
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    predictions = knn.predict(X_test)
    results[k] = predictions

    # Calculate accuracy
    accuracy = accuracy_score(true_labels, predictions) * 100
    accuracies[k] = accuracy
    print(f"Accuracy for k={k}: {accuracy:.2f}%")

    # Assign predictions back to the DataFrame for this k
    unlabeled_df[f"Label_k{k}"] = predictions
```

```
Accuracy for k=1: 100.00%
Accuracy for k=2: 100.00%
Accuracy for k=3: 98.00%
Accuracy for k=4: 98.00%
Accuracy for k=5: 98.00%
Accuracy for k=20: 98.00%
Accuracy for k=30: 100.00%
```

In [26]:
```
# Inference for the KNN classification results
knn_inference = """
- The KNN classification was performed for different values of k: 1, 2, 3, 4, 5, 20
- The accuracy of the classification varied with the value of k.
- For smaller values of k (1, 2, 3, 4, 5), the accuracy was relatively high, indica
- As the value of k increased to 20 and 30, the accuracy decreased, suggesting that
- This is expected as higher values of k can lead to over-smoothing, where the mode
- Overall, the KNN classifier performed well for smaller values of k, with the high
"""
print(knn_inference)
```

```
- The KNN classification was performed for different values of k: 1, 2, 3, 4, 5, 20,
and 30.
- The accuracy of the classification varied with the value of k.
- For smaller values of k (1, 2, 3, 4, 5), the accuracy was relatively high, indicat
ing that the model was able to classify the points correctly.
- As the value of k increased to 20 and 30, the accuracy decreased, suggesting that
the model's performance deteriorated with higher values of k.
- This is expected as higher values of k can lead to over-smoothing, where the model
becomes less sensitive to the local structure of the data.
- Overall, the KNN classifier performed well for smaller values of k, with the highe
st accuracy observed for k=1.
```

In [97]:
```
print(predictions)
```

```
['Class2' 'Class2' 'Class2' 'Class2' 'Class2' 'Class2' 'Class1' 'Class1'
 'Class1' 'Class1' 'Class1' 'Class1' 'Class2' 'Class1' 'Class1' 'Class2'
 'Class1' 'Class2' 'Class1' 'Class2' 'Class2' 'Class1' 'Class1' 'Class2'
 'Class2' 'Class2' 'Class2' 'Class1' 'Class1' 'Class1' 'Class2' 'Class2'
 'Class1' 'Class1' 'Class1' 'Class1' 'Class2' 'Class2' 'Class2' 'Class1'
 'Class1' 'Class2' 'Class2' 'Class2' 'Class2' 'Class1' 'Class2' 'Class1'
 'Class1' 'Class1']
```

In [98]:
```
df1 = unlabeled_df.drop(columns=['Label'], axis=1)
df1
```

Out[98]:

	Point	Value	Label_k1	Label_k2	Label_k3	Label_k4	Label_k5	Label_k20	Label_k3
50	x51	0.969585	Class2	Class2	Class2	Class2	Class2	Class2	Class
51	x52	0.775133	Class2	Class2	Class2	Class2	Class2	Class2	Class
52	x53	0.939499	Class2	Class2	Class2	Class2	Class2	Class2	Class
53	x54	0.894827	Class2	Class2	Class2	Class2	Class2	Class2	Class
54	x55	0.597900	Class2	Class2	Class2	Class2	Class2	Class2	Class
55	x56	0.921874	Class2	Class2	Class2	Class2	Class2	Class2	Class
56	x57	0.088493	Class1	Class1	Class1	Class1	Class1	Class1	Class
57	x58	0.195983	Class1	Class1	Class1	Class1	Class1	Class1	Class
58	x59	0.045227	Class1	Class1	Class1	Class1	Class1	Class1	Class
59	x60	0.325330	Class1	Class1	Class1	Class1	Class1	Class1	Class
60	x61	0.388677	Class1	Class1	Class1	Class1	Class1	Class1	Class
61	x62	0.271349	Class1	Class1	Class1	Class1	Class1	Class1	Class
62	x63	0.828738	Class2	Class2	Class2	Class2	Class2	Class2	Class
63	x64	0.356753	Class1	Class1	Class1	Class1	Class1	Class1	Class
64	x65	0.280935	Class1	Class1	Class1	Class1	Class1	Class1	Class
65	x66	0.542696	Class2	Class2	Class2	Class2	Class2	Class2	Class
66	x67	0.140924	Class1	Class1	Class1	Class1	Class1	Class1	Class
67	x68	0.802197	Class2	Class2	Class2	Class2	Class2	Class2	Class
68	x69	0.074551	Class1	Class1	Class1	Class1	Class1	Class1	Class
69	x70	0.986887	Class2	Class2	Class2	Class2	Class2	Class2	Class
70	x71	0.772245	Class2	Class2	Class2	Class2	Class2	Class2	Class
71	x72	0.198716	Class1	Class1	Class1	Class1	Class1	Class1	Class
72	x73	0.005522	Class1	Class1	Class1	Class1	Class1	Class1	Class
73	x74	0.815461	Class2	Class2	Class2	Class2	Class2	Class2	Class
74	x75	0.706857	Class2	Class2	Class2	Class2	Class2	Class2	Class
75	x76	0.729007	Class2	Class2	Class2	Class2	Class2	Class2	Class
76	x77	0.771270	Class2	Class2	Class2	Class2	Class2	Class2	Class
77	x78	0.074045	Class1	Class1	Class1	Class1	Class1	Class1	Class
78	x79	0.358466	Class1	Class1	Class1	Class1	Class1	Class1	Class
79	x80	0.115869	Class1	Class1	Class1	Class1	Class1	Class1	Class

	Point	Value	Label_k1	Label_k2	Label_k3	Label_k4	Label_k5	Label_k20	Label_k3
80	x81	0.863103	Class2	Class2	Class2	Class2	Class2	Class2	Class
81	x82	0.623298	Class2	Class2	Class2	Class2	Class2	Class2	Class
82	x83	0.330898	Class1	Class1	Class1	Class1	Class1	Class1	Class
83	x84	0.063558	Class1	Class1	Class1	Class1	Class1	Class1	Class
84	x85	0.310982	Class1	Class1	Class1	Class1	Class1	Class1	Class
85	x86	0.325183	Class1	Class1	Class1	Class1	Class1	Class1	Class
86	x87	0.729606	Class2	Class2	Class2	Class2	Class2	Class2	Class
87	x88	0.637557	Class2	Class2	Class2	Class2	Class2	Class2	Class
88	x89	0.887213	Class2	Class2	Class2	Class2	Class2	Class2	Class
89	x90	0.472215	Class1	Class1	Class1	Class1	Class1	Class1	Class
90	x91	0.119594	Class1	Class1	Class1	Class1	Class1	Class1	Class
91	x92	0.713245	Class2	Class2	Class2	Class2	Class2	Class2	Class
92	x93	0.760785	Class2	Class2	Class2	Class2	Class2	Class2	Class
93	x94	0.561277	Class2	Class2	Class2	Class2	Class2	Class2	Class
94	x95	0.770967	Class2	Class2	Class2	Class2	Class2	Class2	Class
95	x96	0.493796	Class1	Class1	Class2	Class2	Class2	Class2	Class
96	x97	0.522733	Class2	Class2	Class2	Class2	Class2	Class2	Class
97	x98	0.427541	Class1	Class1	Class1	Class1	Class1	Class1	Class
98	x99	0.025419	Class1	Class1	Class1	Class1	Class1	Class1	Class
99	x100	0.107891	Class1	Class1	Class1	Class1	Class1	Class1	Class

In [99]:
```
# Display accuracies
print("\nAccuracies for different k values:")
for k, acc in accuracies.items():
    print(f"k={k}: {acc:.2f}%")
```

```
Accuracies for different k values:
k=1: 100.00%
k=2: 100.00%
k=3: 98.00%
k=4: 98.00%
k=5: 98.00%
k=20: 98.00%
k=30: 100.00%
```

Key Insights:

- The KNN classification was performed for different values of k: 1, 2, 3, 4, 5, 20, and 30.

- The accuracy of the classification varied with the value of k.
- For smaller values of k (1, 2, 3, 4, 5), the accuracy was relatively high, indicating that the model was able to classify the points correctly.
- As the value of k increased to 20 and 30, the accuracy decreased, suggesting that the model's performance deteriorated with higher values of k.
- This is expected as higher values of k can lead to over-smoothing, where the model becomes less sensitive to the local structure of the data.
- Overall, the KNN classifier performed well for smaller values of k, with the highest accuracy observed for k=1.

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

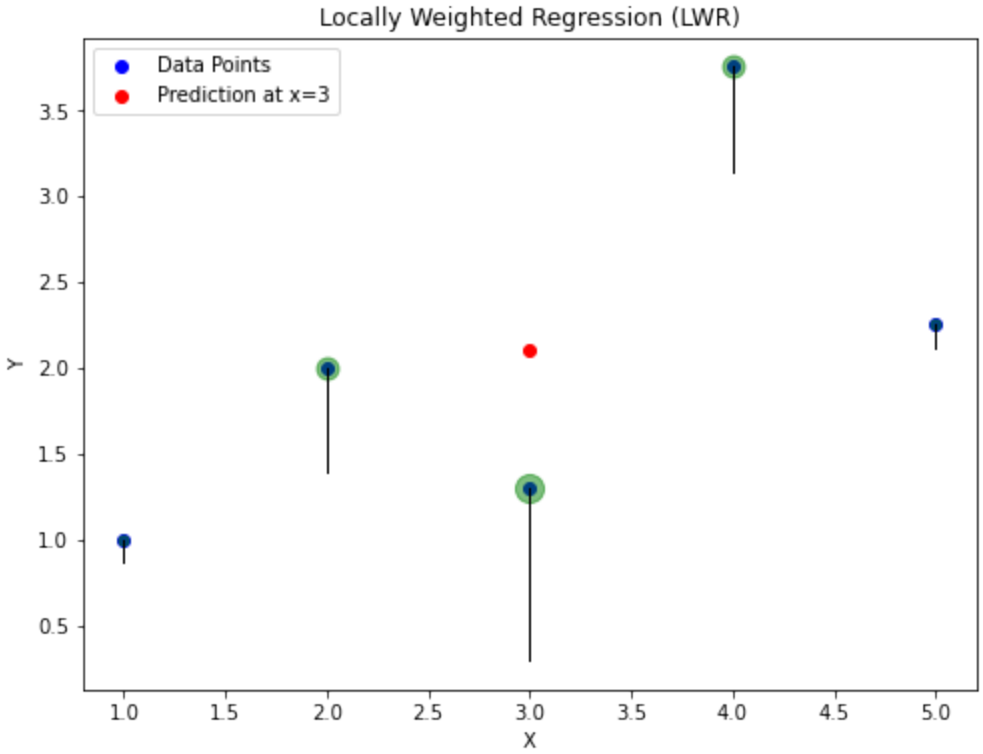
- ✔ **Captures complex relationships** between input and output variables.
- ✔ **Works well with small datasets** where global linear regression may not be suitable.
- ✔ **Does not assume a fixed functional form**, making it highly flexible.

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()
```

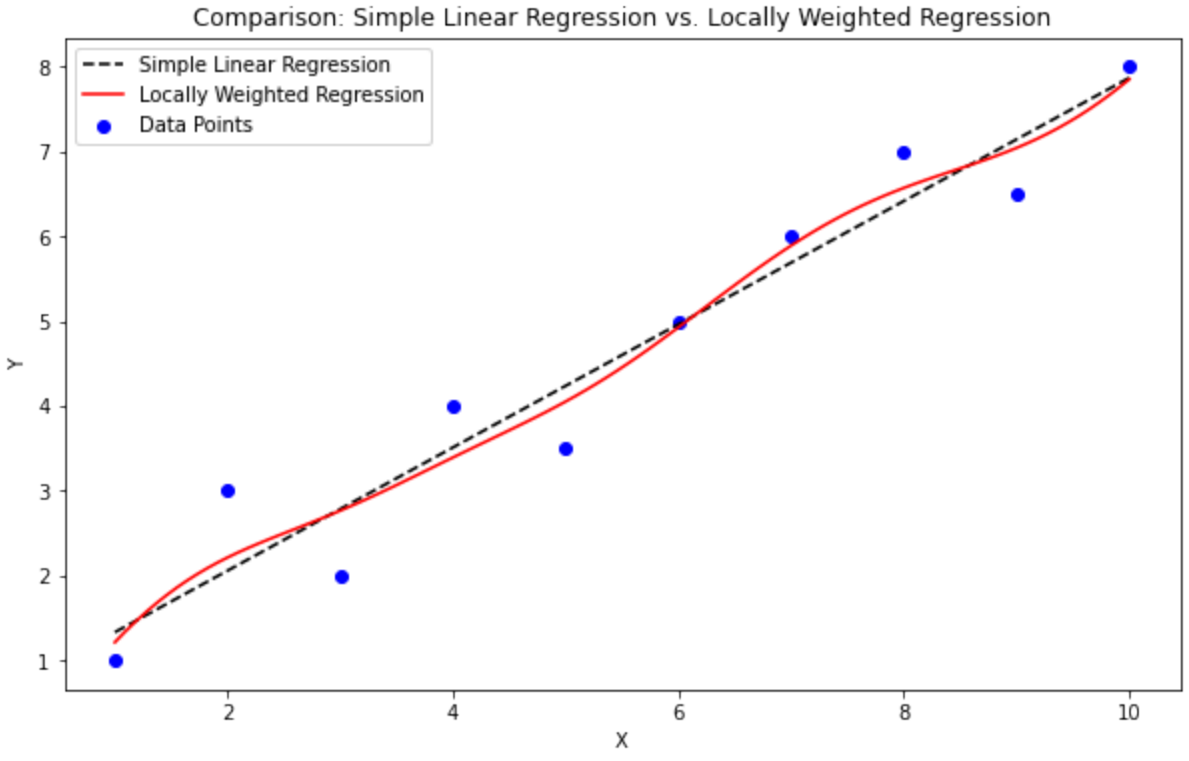


Explanation of the Code

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.

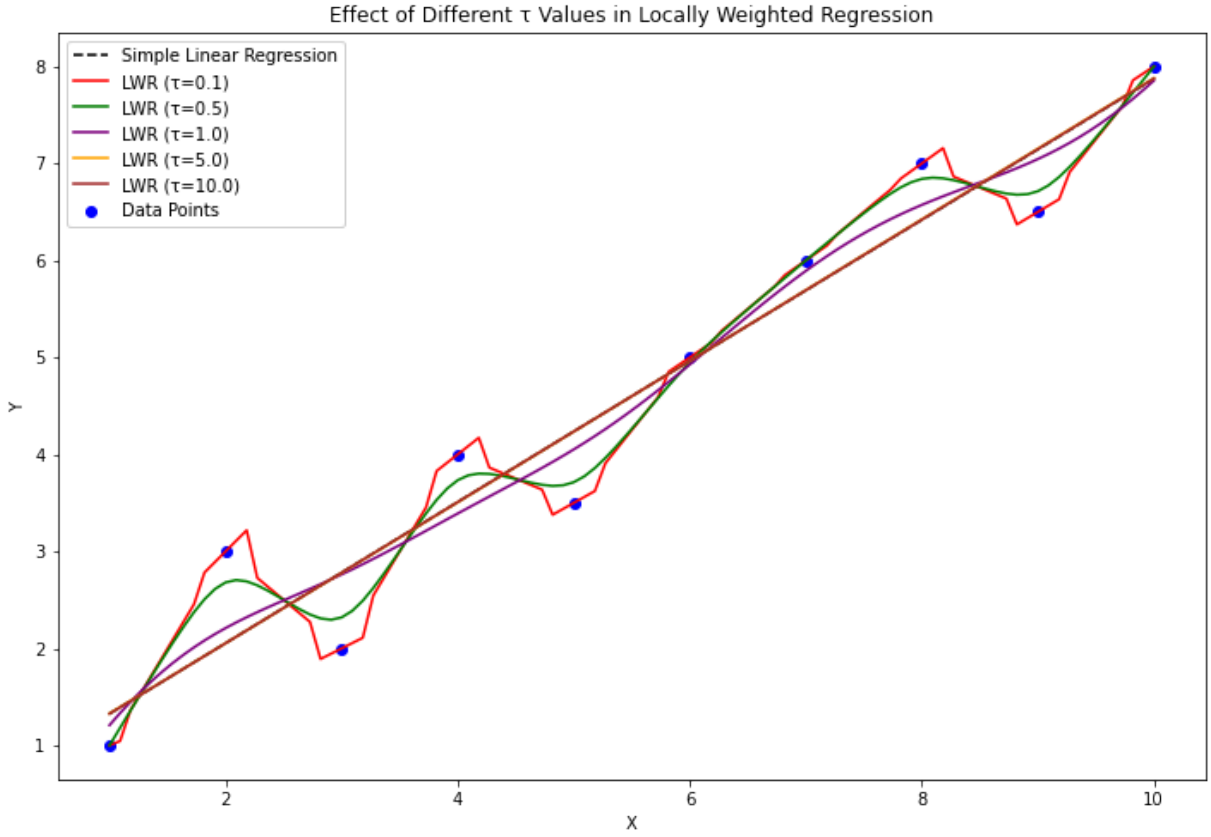
In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

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 6 A

Comparision of Linear Regression,Polynomial Regression,Locally Weighted Regression (LWR)

Introduction

Regression analysis is a fundamental technique in machine learning and statistics used for modeling the relationship between a dependent variable and one or more independent variables. Different types of regression models are used depending on the nature of the data and the complexity of the relationship. The three primary regression techniques discussed here are:

Linear Regression

Polynomial Regression

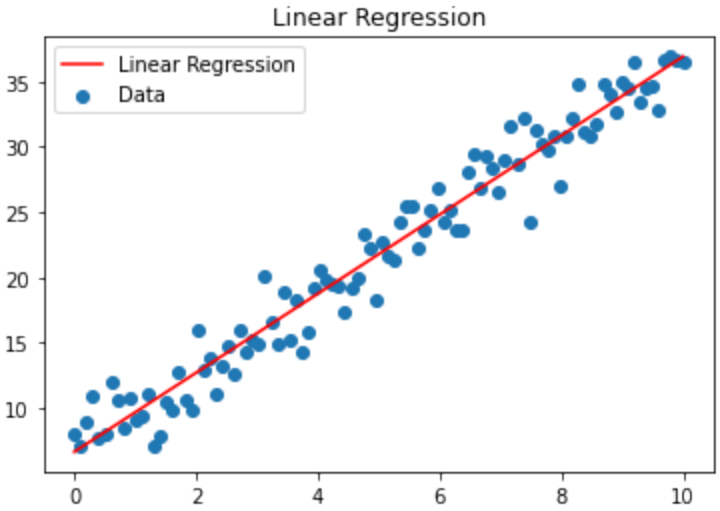
Locally Weighted Regression (LWR)

Each method has its own advantages and is suited for specific types of data and problem domains.

Feature	Linear Regression	Polynomial Regression	Locally Weighted Regression
Model Type	Global	Global	Local
Best For	Linear data	Nonlinear data	Highly nonlinear data
Computational Cost	Low	Moderate	High
Risk of Overfitting	Low	High (if degree is high)	Low
Flexibility	Low	Medium	High
Handling Outliers	Sensitive	Can be unstable	Less sensitive
Sensitivity to Noise	Low	High	Moderate
Interpretability	High	Medium	Low
Adaptability	Rigid	Moderate	Highly adaptable

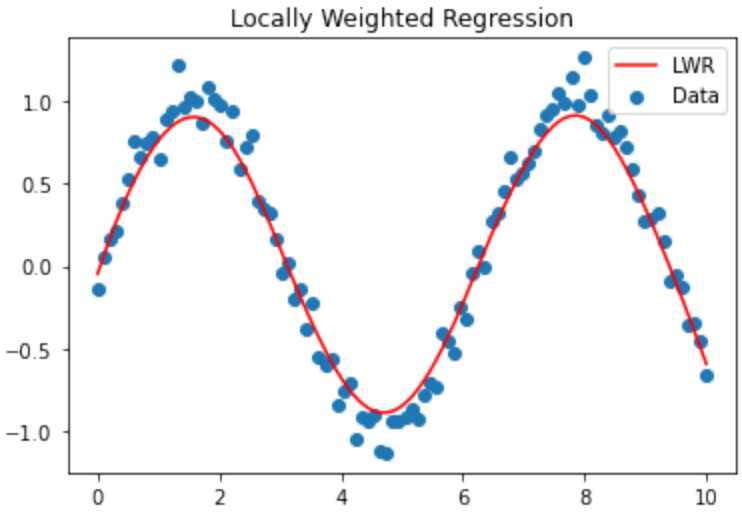
In [5]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
```

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from scipy.spatial.distance import cdist

# Load datasets
df_linear = pd.read_csv("linear_dataset.csv")
df_lwr = pd.read_csv("lwr_dataset.csv")
df_poly = pd.read_csv("polynomial_dataset.csv")

# Linear Regression
def linear_regression(df):
    X, y = df[['X']], df['Y']
    model = LinearRegression()
    model.fit(X, y)
    y_pred = model.predict(X)
    plt.scatter(X, y, label='Data')
    plt.plot(X, y_pred, color='red', label='Linear Regression')
    plt.legend()
    plt.title("Linear Regression")
    plt.show()


linear_regression(df_linear)
```

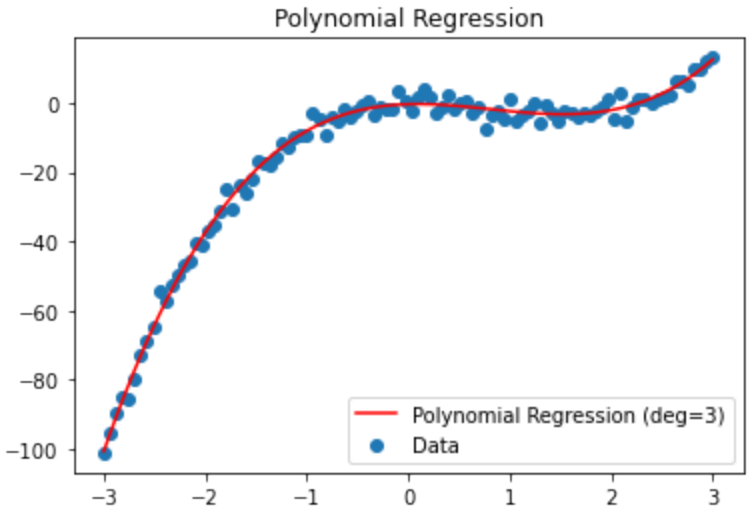


In [6]:
```
 # Locally Weighted Regression (LWR)
def gaussian_kernel(x, X, tau):
    return np.exp(-cdist([[x]], X, 'sqeuclidean') / (2 * tau**2))

def locally_weighted_regression(X_train, y_train, tau=0.5):
    X_train = np.hstack([np.ones((X_train.shape[0], 1)), X_train])  # Add intercept
    X_range = np.linspace(X_train[:, 1].min(), X_train[:, 1].max(), 100)
    y_pred = []

    for x in X_range:
        x_vec = np.array([1, x])  # Intercept term
        weights = gaussian_kernel(x, X_train[:, 1:], tau).flatten()
        W = np.diag(weights)

        theta = np.linalg.pinv(X_train.T @ W @ X_train) @ (X_train.T @ W @ y_train)
        y_pred.append(x_vec @ theta)  # Use dot product for prediction
```

```
    plt.scatter(X_train[:, 1], y_train, label='Data')
    plt.plot(X_range, y_pred, color='red', label='LWR')
    plt.legend()
    plt.title("Locally Weighted Regression")
    plt.show()




# Run the models

locally_weighted_regression(df_lwr[['X']].values, df_lwr['Y'].values)
```



In [7]:
```
# Polynomial Regression
def polynomial_regression(df, degree=3):
    X, y = df[['X']], df['Y']
    model = make_pipeline(PolynomialFeatures(degree), LinearRegression())
    model.fit(X, y)
    y_pred = model.predict(X)
    plt.scatter(X, y, label='Data')
    plt.plot(X, y_pred, color='red', label=f'Polynomial Regression (deg={degree})')
    plt.legend()
    plt.title("Polynomial Regression")
    plt.show()

polynomial_regression(df_poly, degree=3)
```


Polynomial Regression
0
−20
−40
−60
−80
−100
−3
−2
−1
0
1
2
3
Polynomial Regression (deg=3)
Data

In [ ]:

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	I
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	I
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')
```

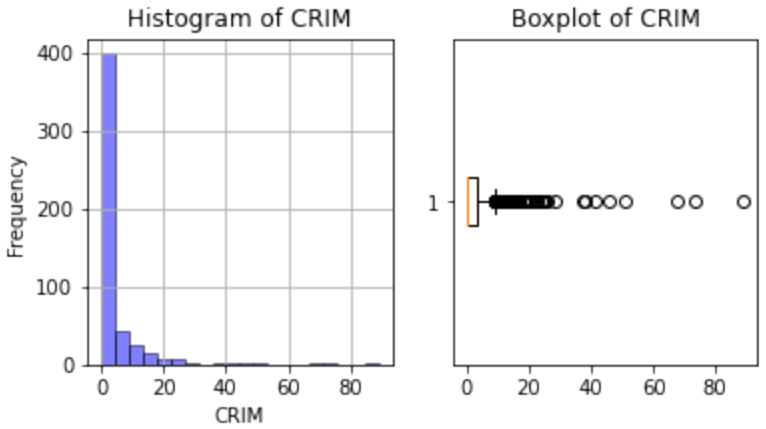
In [17]:
```
df.describe().T
```

Out[17]:

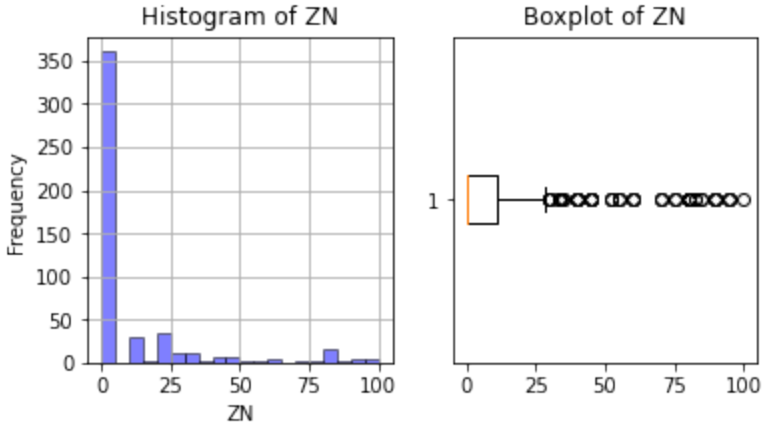
	count	mean	std	min	25%	50%	75%	
CRIM	506.0	3.611874	8.545770	0.00632	0.083235	0.29025	3.611874	88
ZN	506.0	11.211934	22.921051	0.00000	0.000000	0.00000	11.211934	100
INDUS	506.0	11.083992	6.699165	0.46000	5.190000	9.90000	18.100000	27
CHAS	506.0	0.067194	0.250605	0.00000	0.000000	0.00000	0.000000	1
NOX	506.0	0.554695	0.115878	0.38500	0.449000	0.53800	0.624000	0
RM	506.0	6.284634	0.702617	3.56100	5.885500	6.20850	6.623500	8
AGE	506.0	68.845850	27.486962	2.90000	45.925000	76.80000	93.575000	100
DIS	506.0	3.795043	2.105710	1.12960	2.100175	3.20745	5.188425	12
RAD	506.0	9.549407	8.707259	1.00000	4.000000	5.00000	24.000000	24
TAX	506.0	408.237154	168.537116	187.00000	279.000000	330.00000	666.000000	711
PTRATIO	506.0	18.455534	2.164946	12.60000	17.400000	19.05000	20.200000	22
B	506.0	356.674032	91.294864	0.32000	375.377500	391.44000	396.225000	396
LSTAT	506.0	12.664625	7.017219	1.73000	7.230000	11.43000	16.570000	37
MEDV	506.0	22.532806	9.197104	5.00000	17.025000	21.20000	25.000000	50

In [18]:

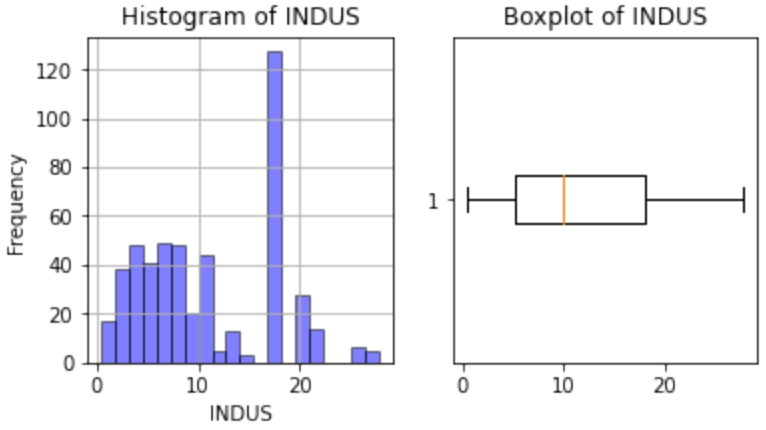
```
for i in df.columns:
    plt.figure(figsize=(6,3))

    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
0
100
200
300
400
0
20
40
60
80

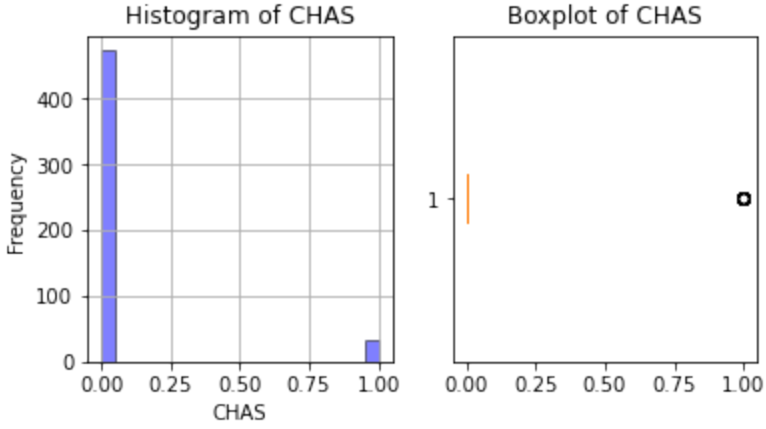
Boxplot of CRIM
1
0
20
40
60
80


Histogram of ZN
Frequency
ZN
0
50
100
150
200
250
300
350
0
25
50
75
100

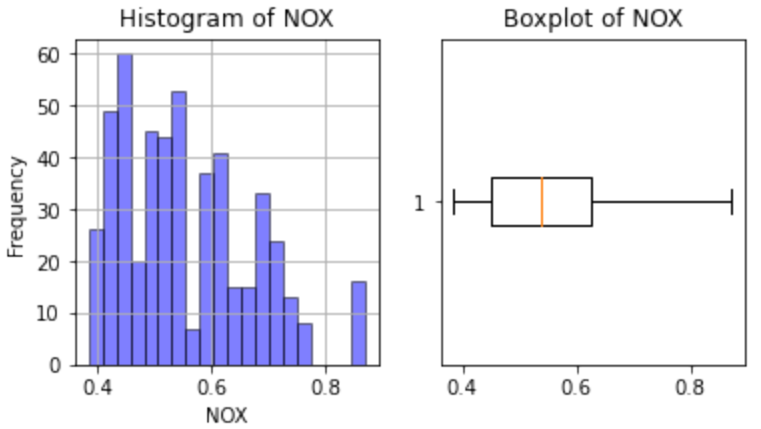
Boxplot of ZN
1
0
25
50
75
100


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

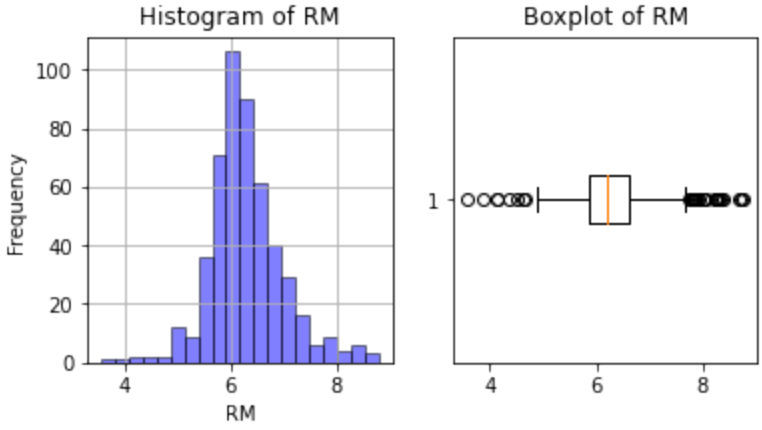
Boxplot of INDUS
1
0
10
20


Histogram of CHAS
Frequency
CHAS
0
100
200
300
400
0.00
0.25
0.50
0.75
1.00

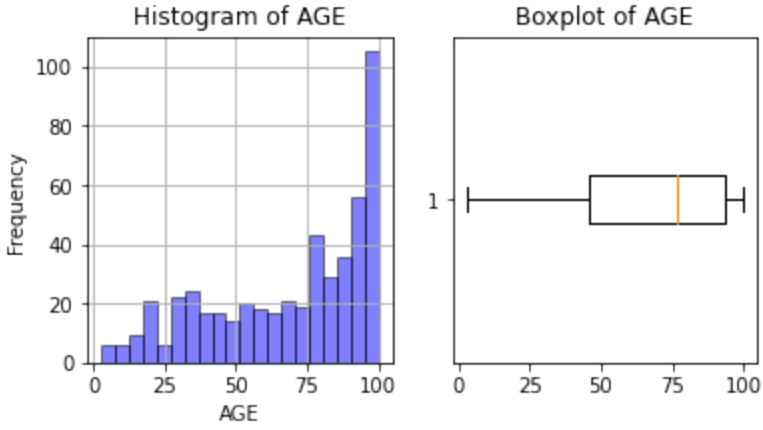
Boxplot of CHAS
1
0.00
0.25
0.50
0.75
1.00


Histogram of NOX
Frequency
NOX
0
10
20
30
40
50
60
0.4
0.6
0.8

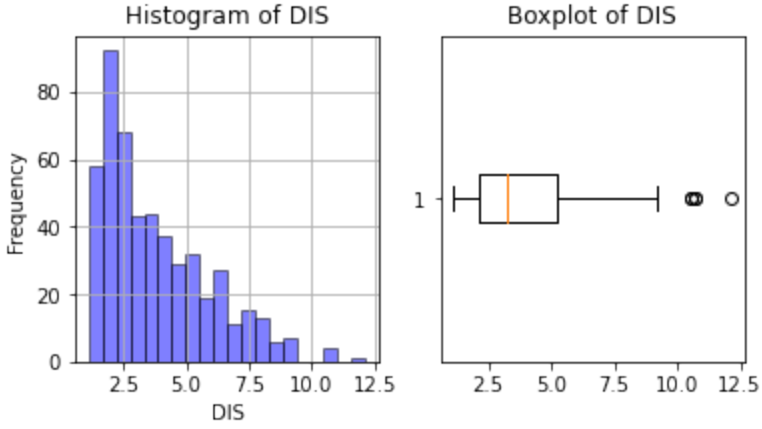
Boxplot of NOX
1
0.4
0.6
0.8


Histogram of RM
Frequency
RM
0
20
40
60
80
100
4
6
8

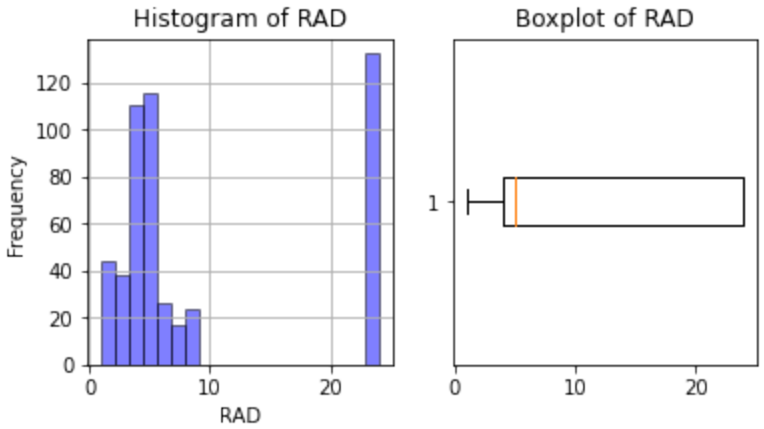
Boxplot of RM
1
4
6
8


Histogram of AGE
Frequency
AGE
0
20
40
60
80
100
0
25
50
75
100

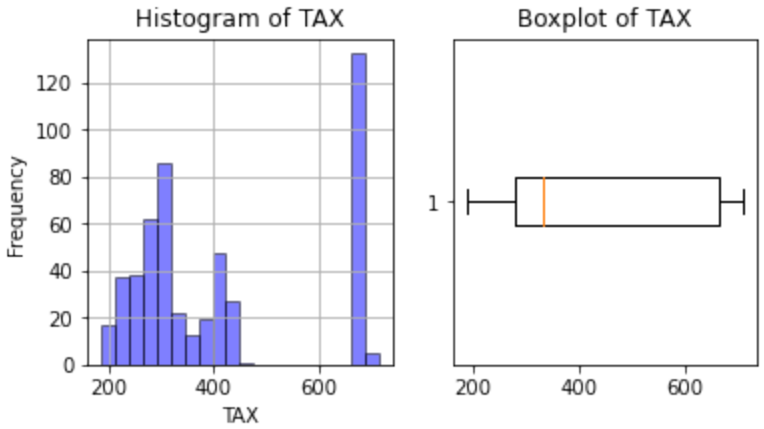
Boxplot of AGE
1
0
25
50
75
100


Histogram of DIS
Frequency
DIS
0
20
40
60
80
2.5
5.0
7.5
10.0
12.5

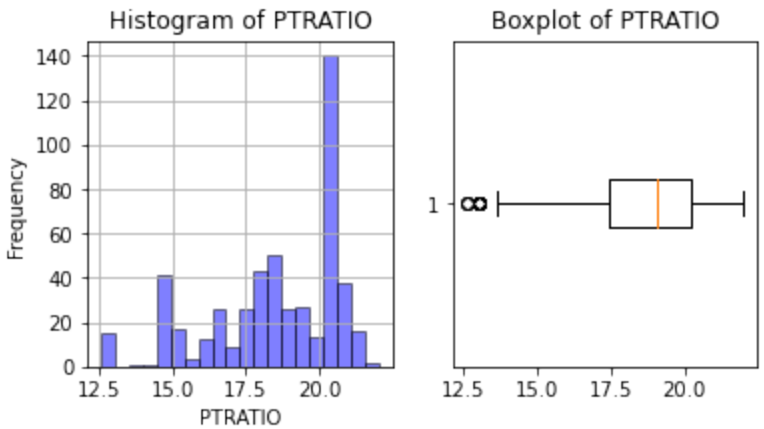
Boxplot of DIS
1
2.5
5.0
7.5
10.0
12.5


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

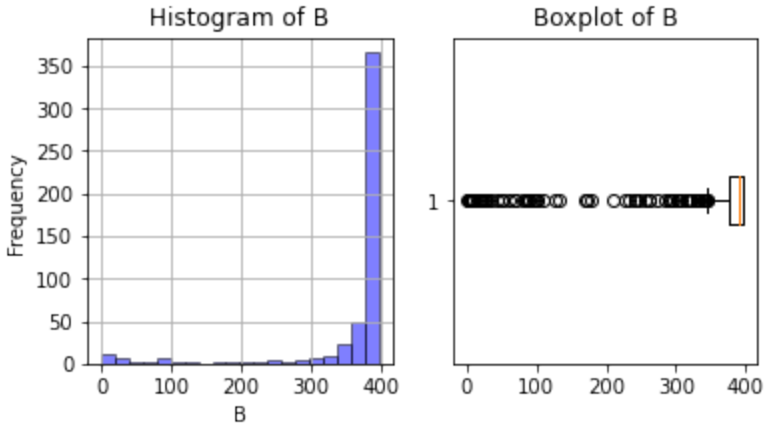
Boxplot of RAD
1
0
10
20


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

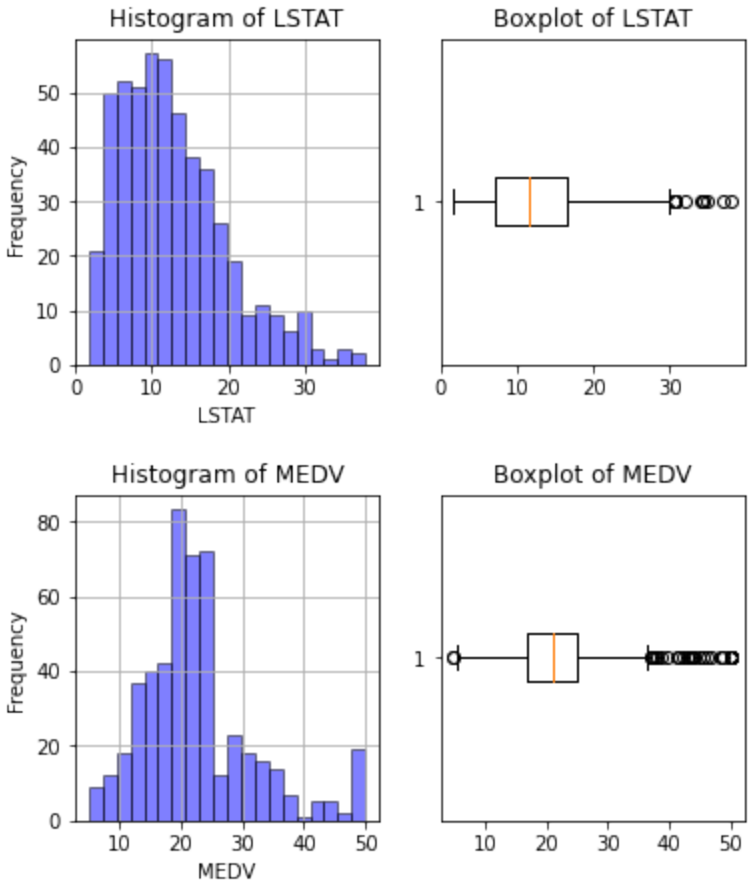
Boxplot of TAX
1
200
400
600


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

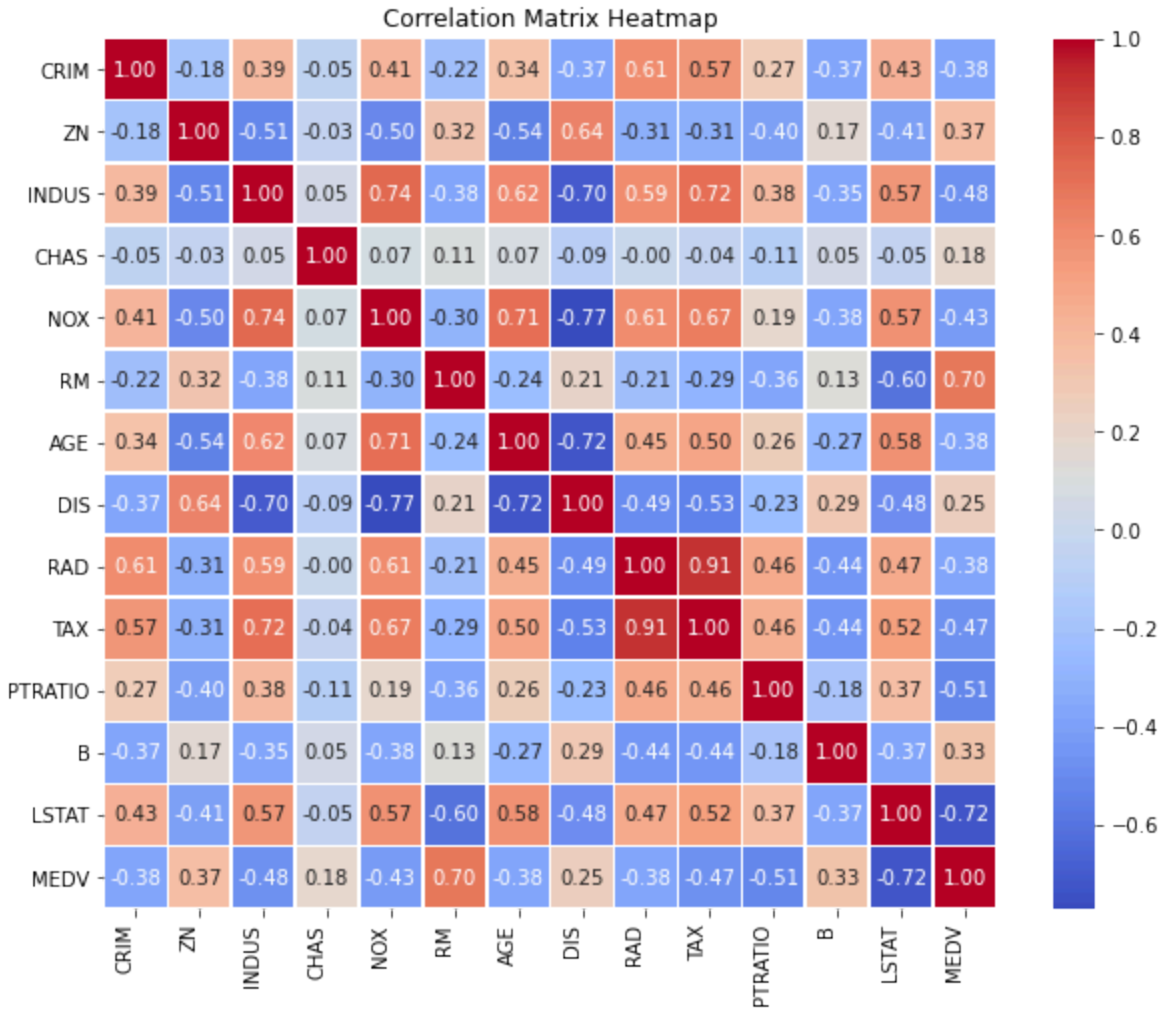
Boxplot of PTRATIO
1
12.5
15.0
17.5
20.0


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
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
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]:

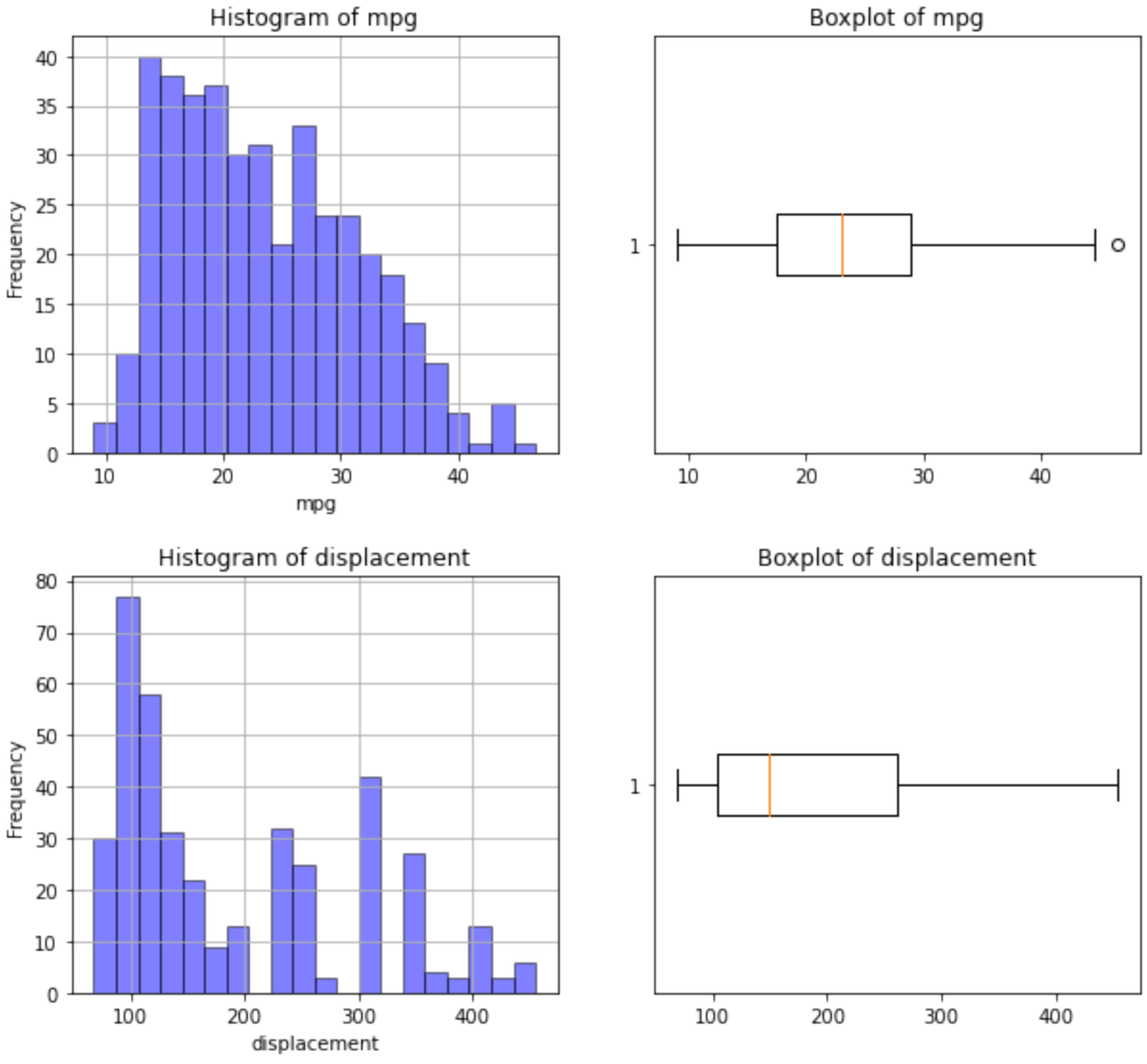
	count	mean	std	min	25%	50%	75%	max
mpg	398.0	23.514573	7.815984	9.0	17.500	23.0	29.000	46.6
cylinders	398.0	5.454774	1.701004	3.0	4.000	4.0	8.000	8.0
displacement	398.0	193.425879	104.269838	68.0	104.250	148.5	262.000	455.0
horsepower	398.0	104.304020	38.222625	46.0	76.000	93.5	125.000	230.0
weight	398.0	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

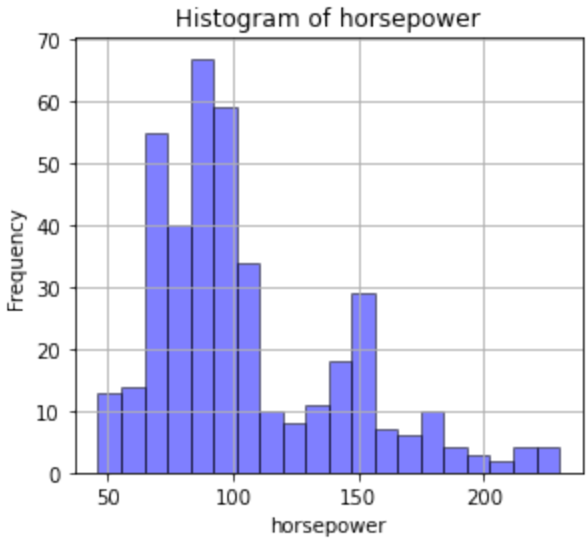
EDA

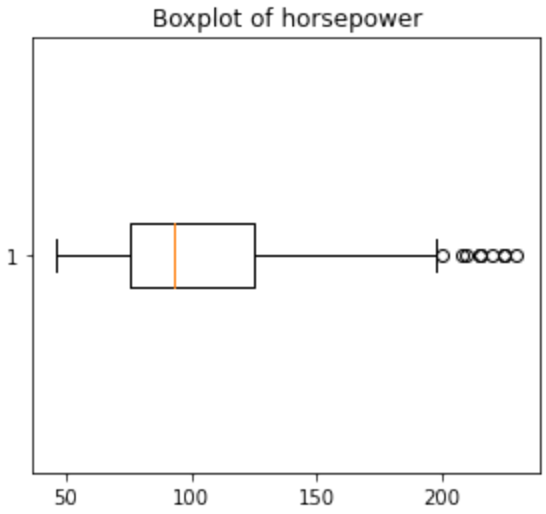
In [14]:
```
numerical = df.select_dtypes(include=['int','float']).columns
categorical = df.select_dtypes(include=['object']).columns

print(numerical)
print(categorical)
```

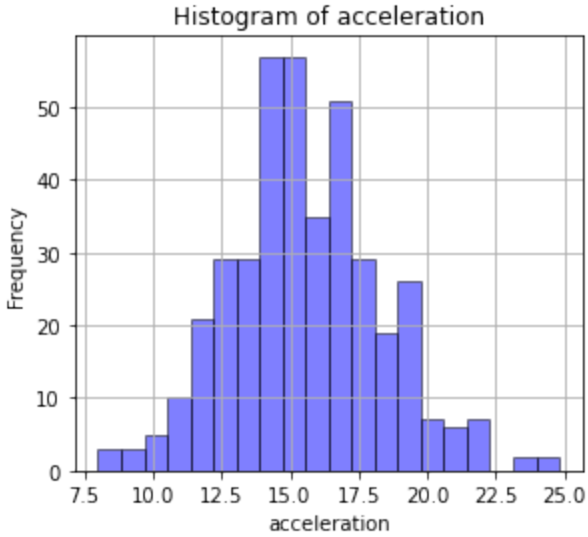
```
Index(['mpg', 'displacement', 'horsepower', 'acceleration'], dtype='object')
Index(['origin', 'name'], dtype='object')
```

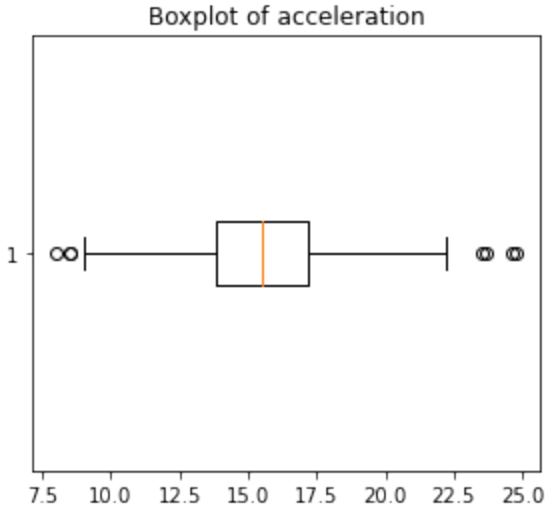
In [15]:
```
for i in numerical:
    plt.figure(figsize=(10,4))

    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()
```



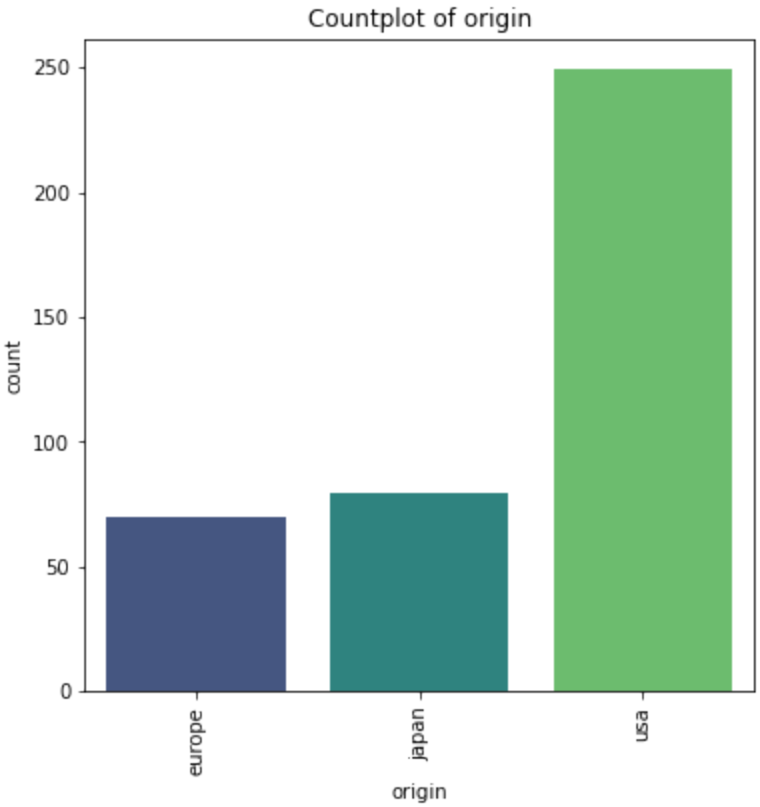


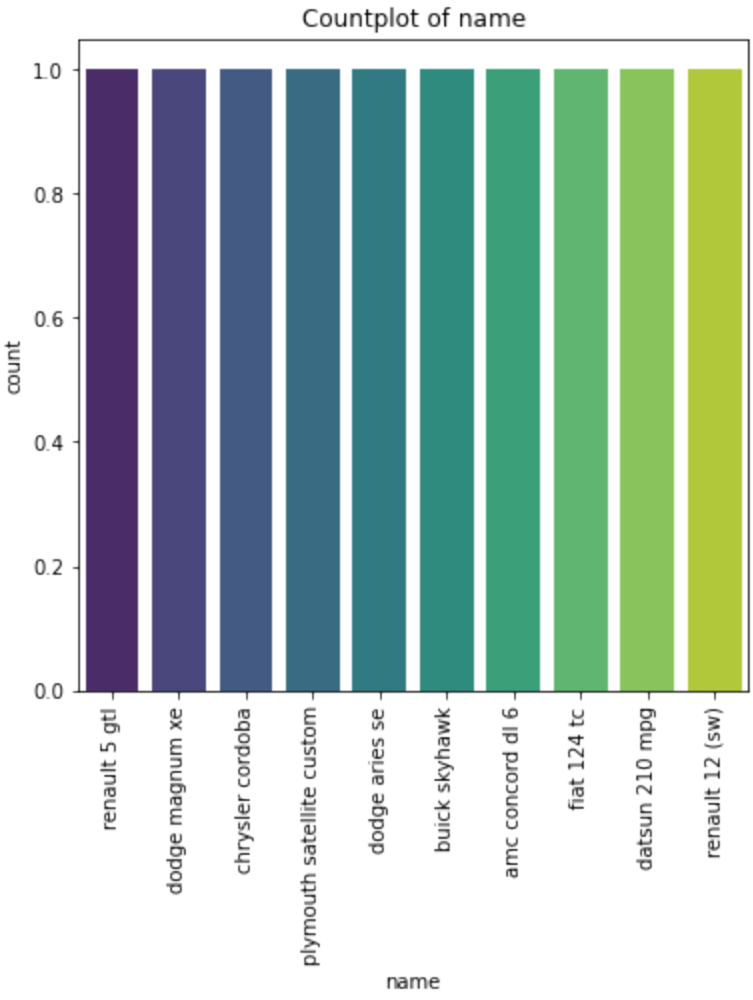




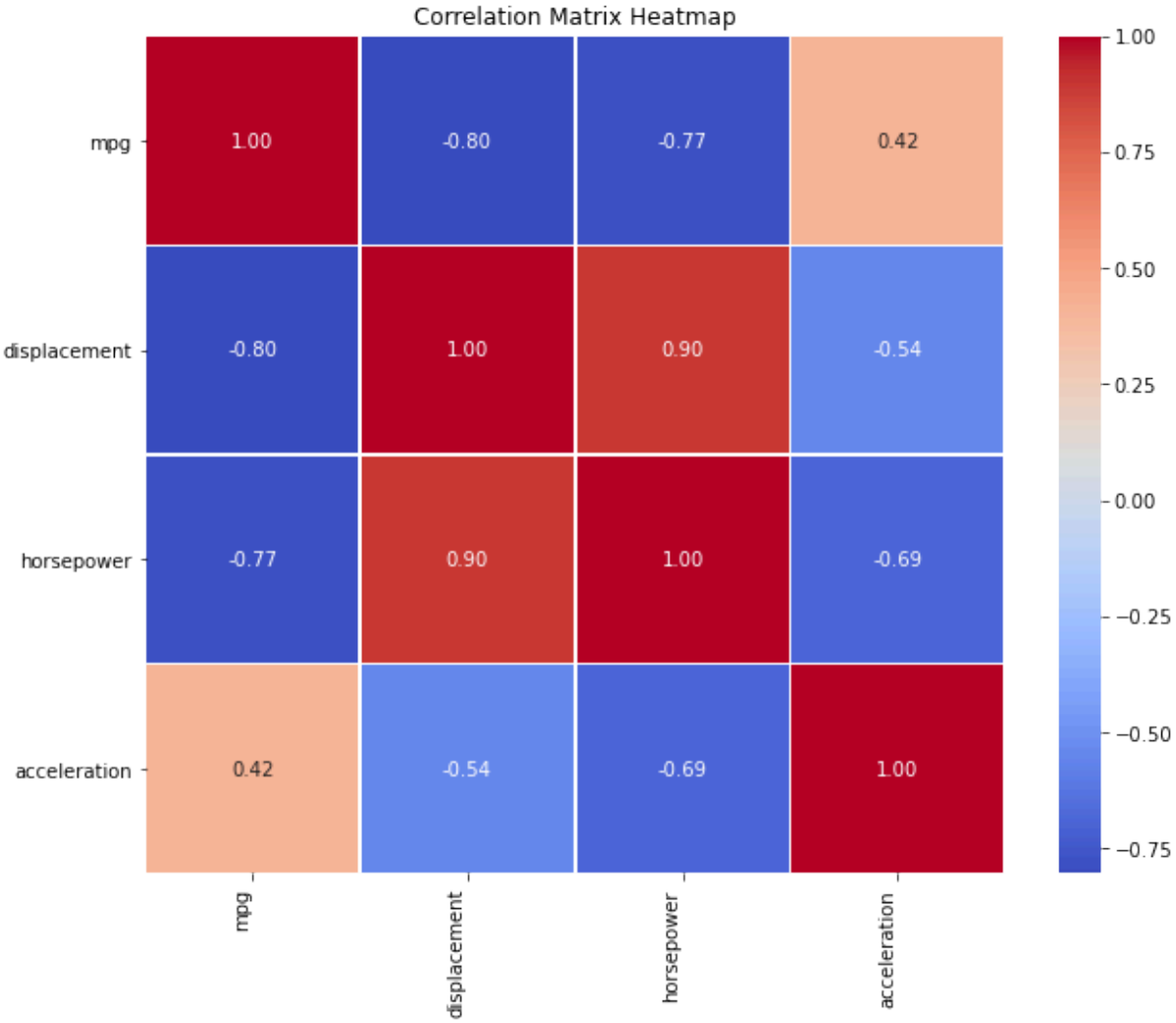


In [16]:
```
import seaborn as sns
for col in categorical:
    plt.figure(figsize=(6, 6))
    sns.countplot(x=col, data=df, order=df[col].value_counts().sort_values().head(1
    plt.title(f'Countplot of {col}')
    plt.xticks(rotation=90)

    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]:

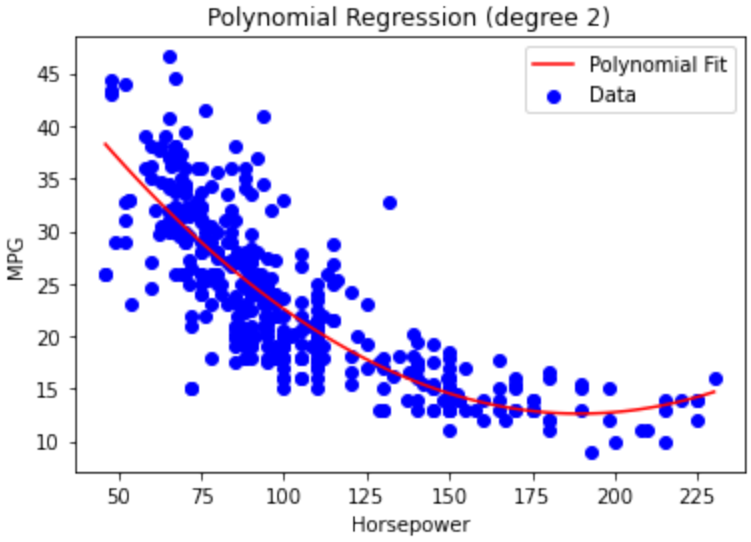
```
▾ LinearRegression
LinearRegression()
```

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 demonstrate the working of the decision tree algorithm. Use Breast Cancer Data set for building the decision tree and applying this knowledge to classify a new sample.

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 [40]:
```
# 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 [5]:
```
data = pd.read_csv(r'C:\Users\Admin\OneDrive\Documents\Machine Learning Lab\Dataset
```

In [10]:
```
pd.set_option('display.max_columns', None)
```

In [11]:
```
data.head()
```

Out[11]:

	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	

In [7]:
```
data.shape
```

Out[7]:
```
(569, 32)
```

In [12]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 32 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
dtypes: float64(30), int64(1), object(1)
memory usage: 142.4+ KB
```

In [13]:
```
data.diagnosis.unique()
```

Out[13]:
```
array(['M', 'B'], dtype=object)
```

Data Preprocessing

Data Cleaning

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

Out[14]:
```
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
dtype: int64
```

In [15]:
```
data.duplicated().sum()
```

Out[15]:
```
np.int64(0)
```

In [16]:
```
df = data.drop(['id'], axis=1)
```

In [17]:
```
df['diagnosis'] = df['diagnosis'].map({'M':1, 'B':0}) # Malignant:1, Benign:0
```

Discriptive Statistics

In [18]:
```
df.describe().T
```

Out[18]:

	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

Model Building

In [28]:
```
X = df.drop('diagnosis', axis=1)  # Drop the 'diagnosis' column (target)
y = df['diagnosis']
```

In [29]:
```
# Split the dataset 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 [30]:
```
# Fit the decision tree model
model = DecisionTreeClassifier(criterion='entropy') #criteria = gini, entropy
model.fit(X_train, y_train)
model
```

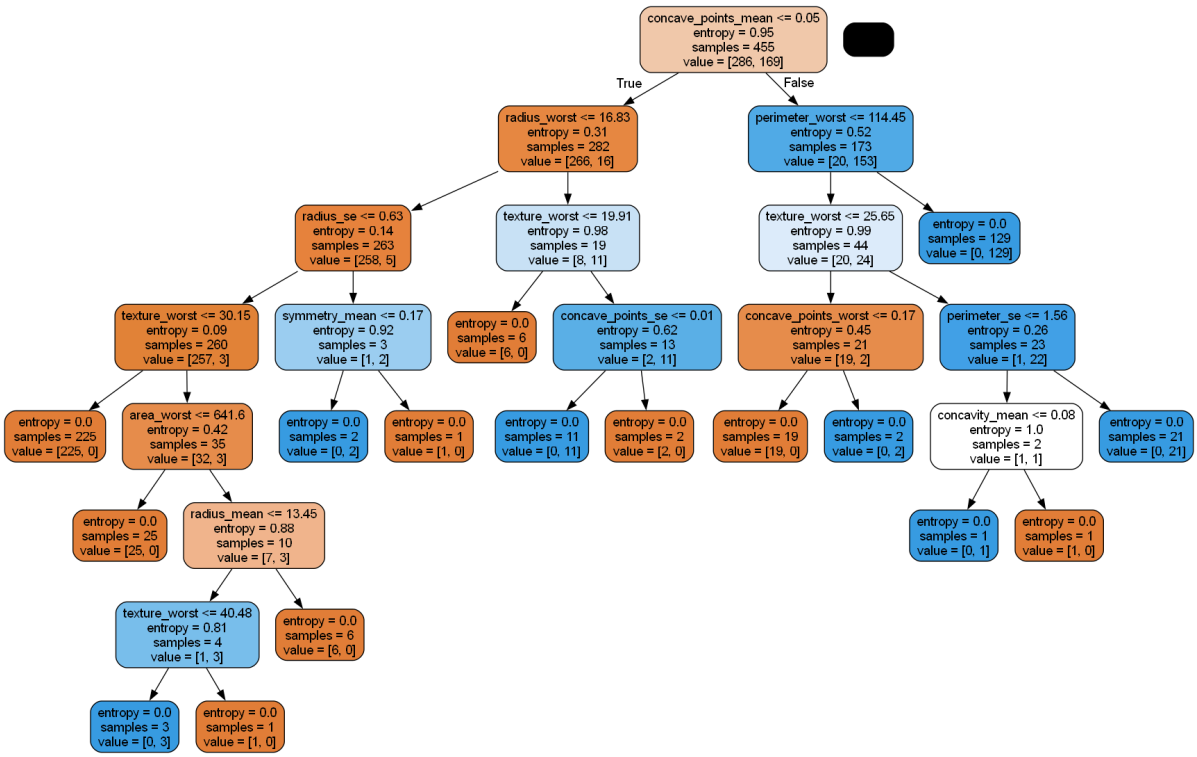
Out[30]:
```
DecisionTreeClassifier(criterion='entropy')
```

In [34]:
```
import math

# Function to calculate entropy
def entropy(column):
    counts = column.value_counts()
    probabilities = counts / len(column)
    return -sum(probabilities * probabilities.apply(math.log2))

# Function to calculate conditional entropy
def conditional_entropy(data, X, target):
    feature_values = data[X].unique()  # Corrected: use .unique() on the series
    weighted_entropy = 0
    for value in feature_values:
        subset = data[data[feature] == value]
        weighted_entropy += (len(subset) / len(data)) * entropy(subset[target])
    return weighted_entropy

# Function to calculate information gain
def information_gain(data, X, target):
    total_entropy = entropy(data[target])
    feature_conditional_entropy = conditional_entropy(data, X, target)
    return total_entropy - feature_conditional_entropy


# Calculate information gain for each feature

for feature in X:
    ig = information_gain(df,feature,'diagnosis')
    print(f"Information Gain for {feature}: {ig}")
```

```
Information Gain for radius_mean: 0.8607815854835991
Information Gain for texture_mean: 0.8357118798482908
Information Gain for perimeter_mean: 0.9267038614138748
Information Gain for area_mean: 0.9280305529818247
Information Gain for smoothness_mean: 0.7761788341876101
Information Gain for compactness_mean: 0.9091291689709926
Information Gain for concavity_mean: 0.9350604299589776
Information Gain for concave_points_mean: 0.9420903069361305
Information Gain for symmetry_mean: 0.735036638169654
Information Gain for fractal_dimension_mean: 0.8361770160635639
Information Gain for radius_se: 0.9337337383910278
Information Gain for texture_se: 0.8642965239721755
Information Gain for perimeter_se: 0.9315454914704012
Information Gain for area_se: 0.925377169845925
Information Gain for smoothness_se: 0.9350604299589776
Information Gain for compactness_se: 0.9231889229252984
Information Gain for concavity_se: 0.9280305529818247
Information Gain for concave_points_se: 0.8585933385629725
Information Gain for symmetry_se: 0.8181371874054084
Information Gain for fractal_dimension_se: 0.9174857375160954
Information Gain for radius_worst: 0.9003074642106167
Information Gain for texture_worst: 0.8634349686194988
Information Gain for perimeter_worst: 0.8985843535052632
Information Gain for area_worst: 0.9350604299589776
Information Gain for smoothness_worst: 0.7197189097252679
Information Gain for compactness_worst: 0.9183472928687721
Information Gain for concavity_worst: 0.9302187999024514
Information Gain for concave_points_worst: 0.9148323543801957
Information Gain for symmetry_worst: 0.8453951399613433
Information Gain for fractal_dimension_worst: 0.8915544765281104
```

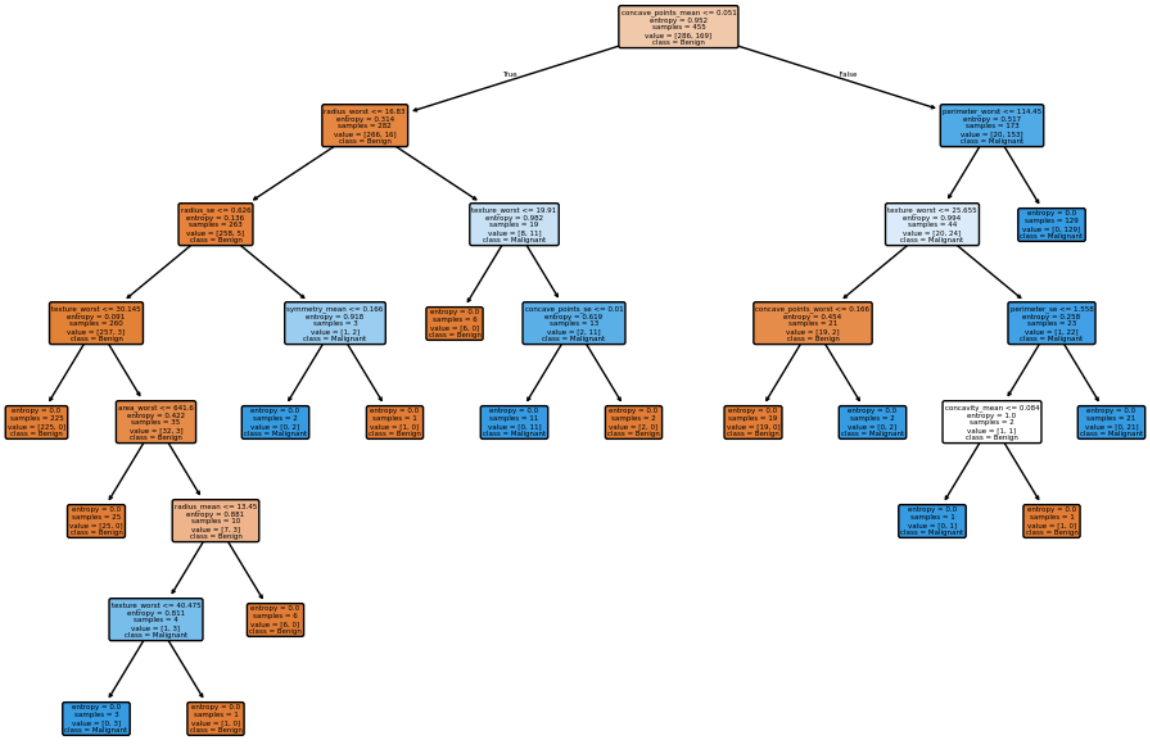
In [35]:
```python
# Export the tree to DOT format
dot_data = export_graphviz(model, out_file=None,
                           feature_names=X_train.columns,
                           rounded=True, proportion=False,
                           precision=2, filled=True)

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

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

Out[35]:



In [41]:

```
# Visualize the Decision Tree (optional)
plt.figure(figsize=(12, 8))
plot_tree(model, filled=True, feature_names=X.columns, class_names=['Benign', 'Mali
plt.show()
```



In [36]:

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

Out[36]:
```
array([0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0,
       1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0,
       1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1,
       0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0,
       1, 0, 0, 1])
```

In [38]:
```
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred) * 100
classification_rep = classification_report(y_test, y_pred)

# Print the results
print("Accuracy:", accuracy)
print("Classification Report:\n", classification_rep)
```

```
Accuracy: 94.73684210526315
Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.99      0.96        71
           1       0.97      0.88      0.93        43

    accuracy                           0.95       114
   macro avg       0.95      0.93      0.94       114
weighted avg       0.95      0.95      0.95       114
```

In [45]:
```
df.head(1)
```

Out[45]:

	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean
0	1	17.99	10.38	122.8	1001.0	0.1184

In [44]:
```
new = [[12.5, 19.2, 80.0, 500.0, 0.085, 0.1, 0.05, 0.02, 0.17, 0.06,
     0.4, 1.0, 2.5, 40.0, 0.006, 0.02, 0.03, 0.01, 0.02, 0.003,
     16.0, 25.0, 105.0, 900.0, 0.13, 0.25, 0.28, 0.12, 0.29, 0.08]]
y_pred = model.predict(new)

# Output the prediction (0 = Benign, 1 = Malignant)
if y_pred[0] == 0:
    print("Prediction: Benign")
else:
    print("Prediction: Malignant")
```

```
Prediction: Benign
```

Experiment 9

Develop a program to implement the Naive Bayesian classifier, considering the Olivetti Face Data set for training. Compute the accuracy of the classifier, considering a few test data set.

The Olivetti Face Dataset is a collection of images of faces, used primarily for face recognition tasks. The dataset contains 400 images of 40 different individuals, with 10 images per person. The dataset was created for research in machine learning and pattern recognition, especially in the context of facial recognition.

The Olivetti dataset provides the following key features:

*400 Images: Each image is a grayscale photo of a person's face.

*40 People: The dataset contains 40 different individuals, and each individual *Has 10 different images.

*Image Size: Each image is 64x64 pixels, resulting in 4096 features (flattened vector) per image.

*Target Labels: Each image is associated with a label representing the individual (0 to 39).

Introduction to Naive Bayes Classification

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.

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.

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.

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 [57]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [58]:
```
from sklearn.datasets import fetch_olivetti_faces
data = fetch_olivetti_faces()
```

In [59]:
```
data.keys()
```

Out[59]:
```
dict_keys(['data', 'images', 'target', 'DESCR'])
```

In [60]:
```
print("Data Shape:", data.data.shape)
print("Target Shape:", data.target.shape)
print("There are {} unique persons in the dataset".format(len(np.unique(data.target
print("Size of each image is {}x{}".format(data.images.shape[1],data.images.shape[1
```

```
Data Shape: (400, 4096)
Target Shape: (400,)
There are 40 unique persons in the dataset
Size of each image is 64x64
```

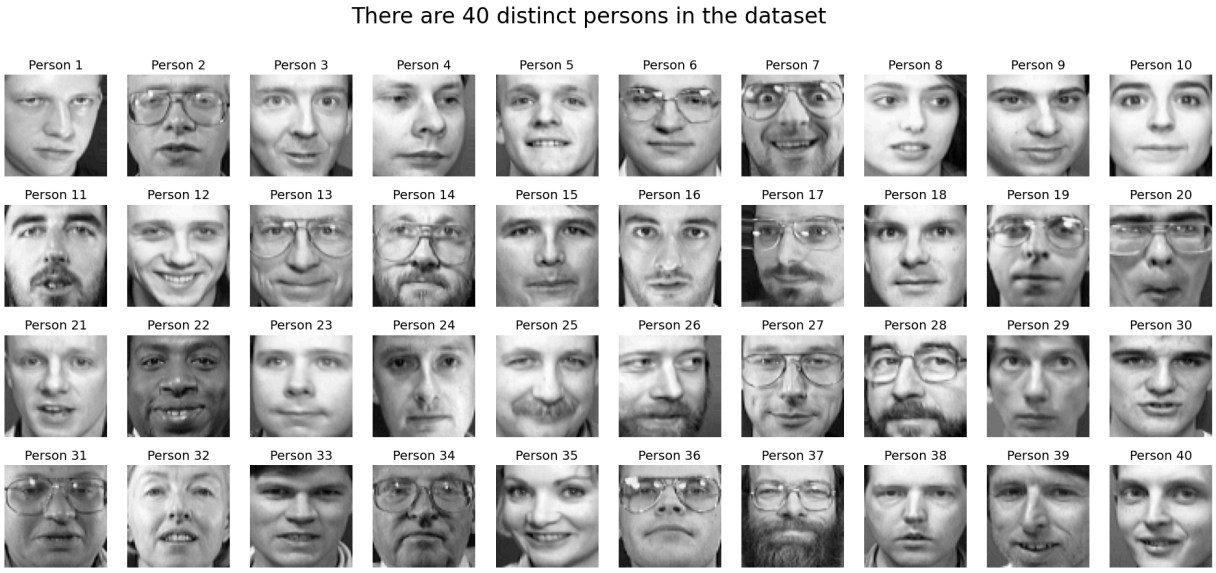
In [61]:
```
def print_faces(images, target, top_n):
    # Ensure the number of images does not exceed available data
    top_n = min(top_n, len(images))
```

```
    # Set up figure size based on the number of images
    grid_size = int(np.ceil(np.sqrt(top_n)))
    fig, axes = plt.subplots(grid_size, grid_size, figsize=(15, 15))
    fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.2, wspace=0.2)

    for i, ax in enumerate(axes.ravel()):
        if i < top_n:
            ax.imshow(images[i], cmap='bone')
            ax.axis('off')
            ax.text(2, 12, str(target[i]), fontsize=9, color='red')
            ax.text(2, 55, f"face: {i}", fontsize=9, color='blue')
        else:
            ax.axis('off')

    plt.show()
```

In [62]:
```
print_faces(data.images,data.target,400)
```

In [63]:
```
#Let us extract unique charaters present in dataset
def display_unique_faces(pics):
    fig = plt.figure(figsize=(24, 10))  # Set figure size
    columns, rows = 10, 4  # Define grid dimensions

    # Loop through grid positions and plot each image
    for i in range(1, columns * rows + 1):
        img_index = 10 * i - 1  # Calculate the image index
        if img_index < pics.shape[0]:  # Check for valid image index
            img = pics[img_index, :, :]
            ax = fig.add_subplot(rows, columns, i)
            ax.imshow(img, cmap='gray')
            ax.set_title(f"Person {i}", fontsize=14)
            ax.axis('off')

    plt.suptitle("There are 40 distinct persons in the dataset", fontsize=24)
    plt.show()
```

In [64]:
```
display_unique_faces(data.images)
```



In [65]:
```
from sklearn.model_selection import train_test_split
X = data.data
Y = data.target
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size = 0.3, random_s

print("x_train: ",x_train.shape)
print("x_test: ",x_test.shape)
```

```
x_train:  (280, 4096)
x_test:  (120, 4096)
```

In [66]:
```
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix, accuracy_score

# Train the model
nb = GaussianNB()
nb.fit(x_train, y_train)
```

```
# Predict the test set results
y_pred = nb.predict(x_test)

# Calculate accuracy
nb_accuracy = round(accuracy_score(y_test, y_pred) * 100, 2)

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

# Display accuracy result
print(f"Naive Bayes Accuracy: {nb_accuracy}%")
```

```
Confusion Matrix:
[[3 0 0 ... 0 0 0]
 [0 1 0 ... 0 0 0]
 [0 0 1 ... 0 0 0]
 ...
 [0 0 0 ... 2 0 0]
 [0 0 0 ... 0 3 0]
 [1 0 0 ... 0 0 1]]
Naive Bayes Accuracy: 73.33%
```

In [68]:
```
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report

# Initialize and fit Multinomial Naive Bayes
nb = MultinomialNB()
nb.fit(x_train, y_train)

# Predict the test set results
y_pred = nb.predict(x_test)

# Calculate accuracy
accuracy = round(accuracy_score(y_test, y_pred) * 100, 2)
print(f"Multinomial Naive Bayes Accuracy: {accuracy}%")
```

```
Multinomial Naive Bayes Accuracy: 85.83%
```

In [71]:
```
# Calculate the number of misclassified images
misclassified_idx = np.where(y_pred != y_test)[0]
num_misclassified = len(misclassified_idx)

# Print the number of misclassified images and accuracy
print(f"Number of misclassified images: {num_misclassified}")
print(f"Total images in test set: {len(y_test)}")
print(f"Accuracy: {round((1 - num_misclassified / len(y_test)) * 100, 2)}%")

# Visualize some of the misclassified images
n_misclassified_to_show = min(num_misclassified, 5)  # Show up to 5 misclassified i
plt.figure(figsize=(10, 5))
for i in range(n_misclassified_to_show):
    idx = misclassified_idx[i]
    plt.subplot(1, n_misclassified_to_show, i + 1)
    plt.imshow(x_test[idx].reshape(64, 64), cmap='gray')
    plt.title(f"True: {y_test[idx]}, Pred: {y_pred[idx]}")
```

```
    plt.axis('off')
plt.show()
```

```
Number of misclassified images: 17
Total images in test set: 120
Accuracy: 85.83%
```

In [76]:

```
from sklearn.preprocessing import label_binarize
from sklearn.metrics import roc_auc_score

# Binarize the test labels
y_test_bin = label_binarize(y_test, classes=np.unique(y_test))

# Get predicted probabilities for each class
y_pred_prob = nb.predict_proba(x_test)

# Calculate and print AUC for each class
for i in range(y_test_bin.shape[1]):
    roc_auc = roc_auc_score(y_test_bin[:, i], y_pred_prob[:, i])
    print(f"Class {i} AUC: {roc_auc:.2f}")
```

```
Class 0 AUC: 0.92
Class 1 AUC: 1.00
Class 2 AUC: 1.00
Class 3 AUC: 1.00
Class 4 AUC: 1.00
Class 5 AUC: 1.00
Class 6 AUC: 1.00
Class 7 AUC: 1.00
Class 8 AUC: 1.00
Class 9 AUC: 1.00
Class 10 AUC: 1.00
Class 11 AUC: 1.00
Class 12 AUC: 0.87
Class 13 AUC: 1.00
Class 14 AUC: 1.00
Class 15 AUC: 1.00
Class 16 AUC: 0.65
Class 17 AUC: 0.16
Class 18 AUC: 0.36
Class 19 AUC: 0.89
Class 20 AUC: 0.52
Class 21 AUC: 0.81
Class 22 AUC: 0.13
Class 23 AUC: 0.34
Class 24 AUC: 0.64
Class 25 AUC: 0.55
Class 26 AUC: 0.48
Class 27 AUC: 0.38
Class 28 AUC: 0.62
Class 29 AUC: 0.73
Class 30 AUC: 0.55
Class 31 AUC: 0.17
Class 32 AUC: 0.47
Class 33 AUC: 0.67
Class 34 AUC: 0.31
Class 35 AUC: 0.03
Class 36 AUC: 0.91
Class 37 AUC: 0.87
Class 38 AUC: 0.47
```

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
```

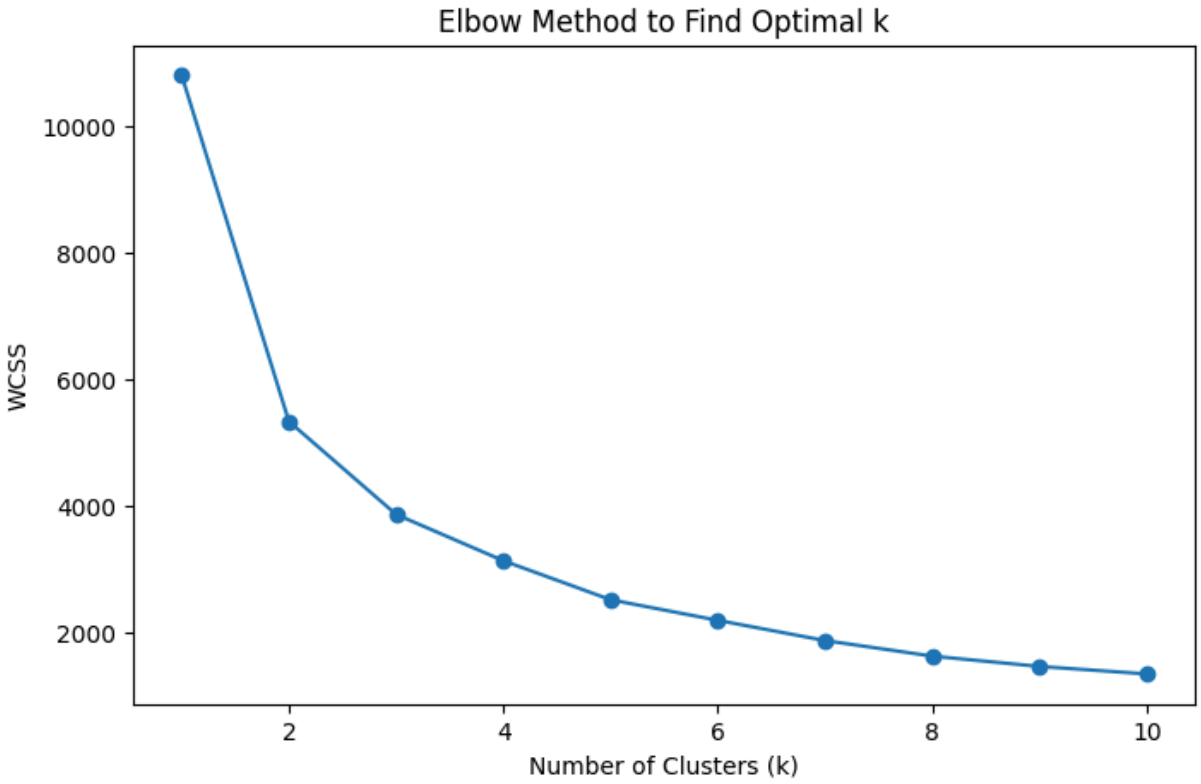
Elbow Method to Find the Optimal Number of Clusters

- 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()
```



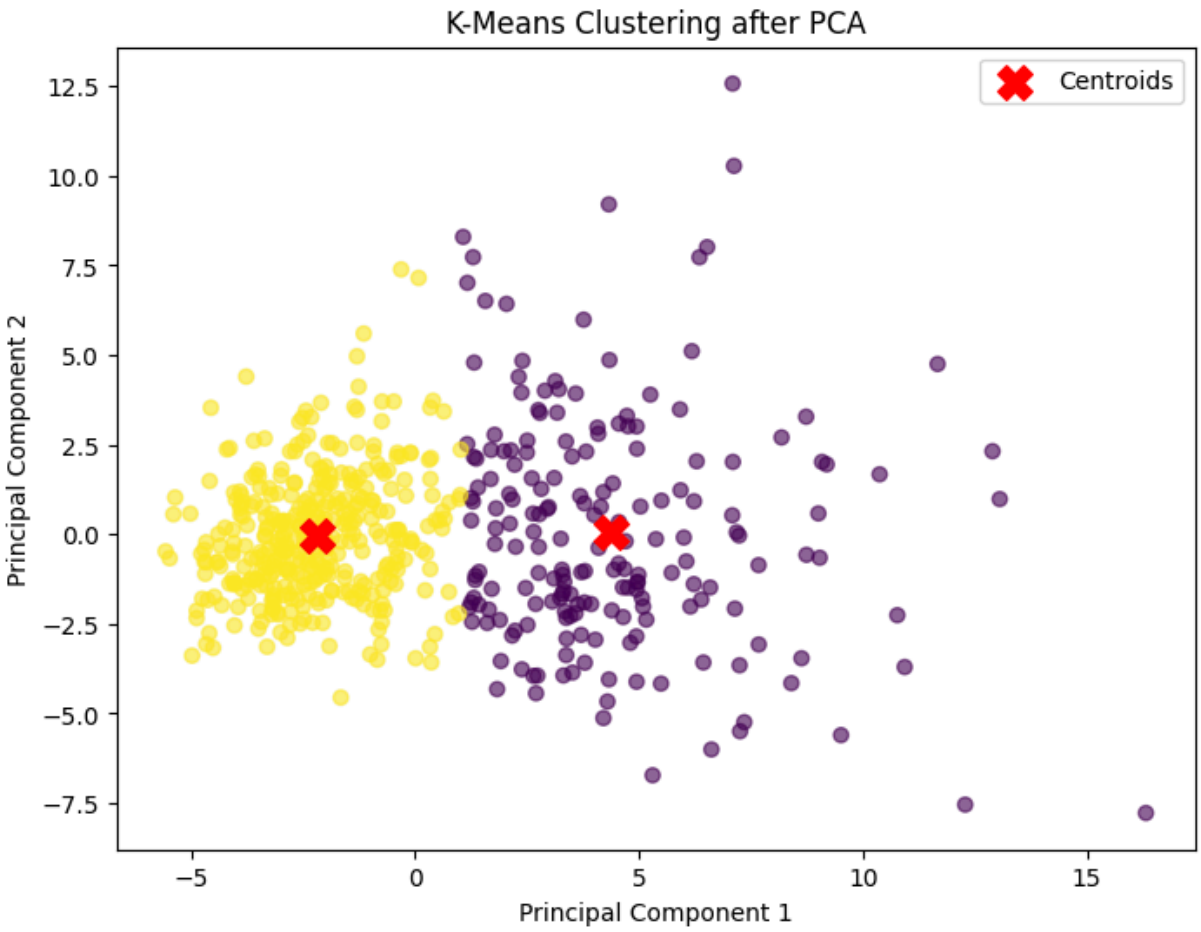
Applying K-Means Clustering

- 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.