Lab Experiment 1: Data Visualization

Objective: To construct various types of plots and charts, such as histograms, bar charts, pie charts, and scatter plots, by importing data from a CSV file. Further, to label different axes and data within the plots for better readability and interpretation.

1. Introduction & Theory

Data visualization is a fundamental aspect of machine learning and data analysis. It allows us to understand the underlying patterns, distributions, and relationships within a dataset. Visualizing data helps in identifying outliers, understanding feature distributions, and communicating insights effectively.

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It provides a wide variety of plots and charts. In this experiment, we will use Matplotlib in conjunction with the **Pandas** library to perform basic data visualization tasks on a sample dataset. We will explore the following key plot types:

- **Histogram:** To understand the distribution of a single numerical variable.
- **Bar Chart:** To compare categorical data.
- **Pie Chart:** To show the proportion of each category in a whole.
- **Scatter Plot:** To visualize the relationship between two numerical variables.

2. Procedure and Code Implementation

Step 2.1: Importing Necessary Libraries

The first step is to import the required libraries. We use pandas for data manipulation, matplotlib. pyplot for plotting, and random to generate our sample data.

Code:

```
import random

import pandas as pd

import matplotlib.pyplot as plt
```

Step 2.2: Generating and Loading the Dataset

For this experiment, we generate a synthetic dataset containing 100 entries with columns for 'age', 'gender', and 'income'. This data is then saved to a CSV file named data.csv and loaded back into a Pandas DataFrame.

Code:

```
data = {

   'age': [random.randint(20, 60) for _ in range(100)],

   'gender': [random.choice(['Male', 'Female']) for _ in range(100)],
    'income': [random.randint(20000, 100000) for _ in range(100)]
}
# Convert data to a pandas dataframe and save to CSV file
df = pd.DataFrame(data)
df.to_csv('data.csv', index=False)
```

Step 2.3: Creating Visualizations

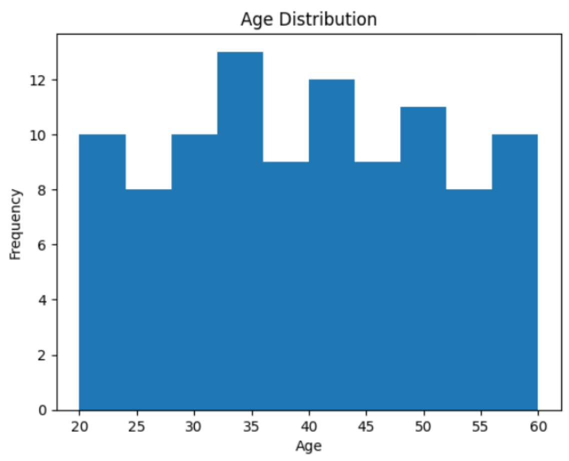
2.3.1 Histogram

A histogram is used to show the frequency distribution of a numerical variable. We plot the distribution of the 'age' column.

Code:

```
plt.hist(data['age'])
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.title('Age Distribution')
plt.show()
```

Output:

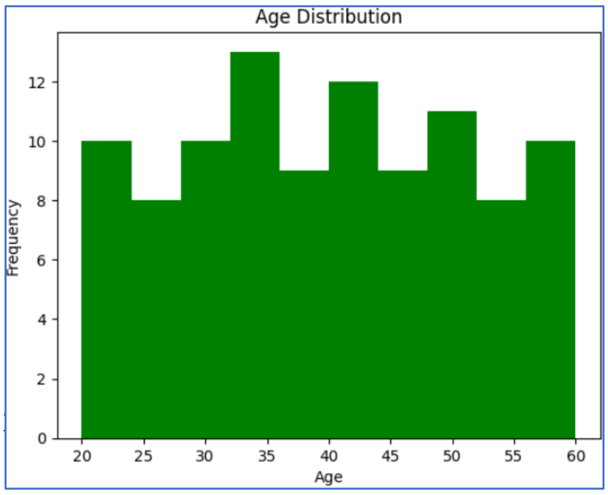


Histogram Customizations (Color and Bins): We can customize the histogram's appearance by changing the bar color and adjusting the number of bins.

Code:

```
plt.hist(data['age'],color='green',bins = 20)
# plt.hist([mens_age, female_age], color=['Black', 'Red'], label=['Male', 'Female'])
plt.xlabel('Age')
plt.ylabel('Frequency')

plt.title('Age Distribution')

plt.show()
```

Output:

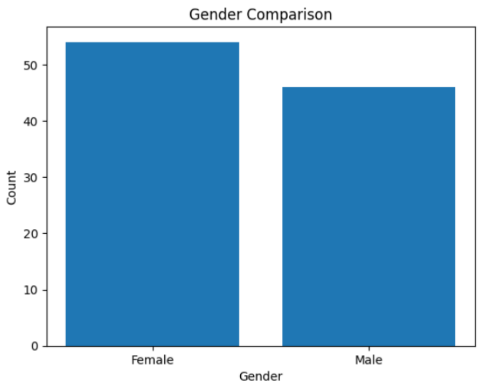


A bar chart is used to compare different categories. Here, we compare the count of 'Male' and 'Female' entries in our dataset.

Code:

```
plt.bar(data['gender'].unique(), data['gender'].value_counts())

plt.xlabel('Gender')

plt.ylabel('Count')

plt.title('Gender Comparison')

plt.show()
```

Output:

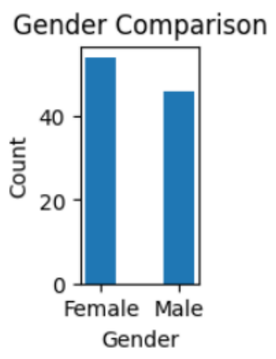


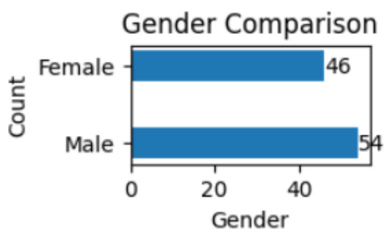
Bar Chart Customizations (Horizontal with Data Labels): We can create a horizontal bar chart using barh() and add data labels using bar_label() for clarity.

Code:

```
 plt.figure(figsize=(1,2))
# creating Bar chart with a width size of 0.4
plt.bar(data['gender'].unique(), data['gender'].value_counts(),width = 0.4)
plt.xlabel('Gender')
plt.ylabel('Count')
plt.title('Gender Comparison')
plt.show()
```

Output:





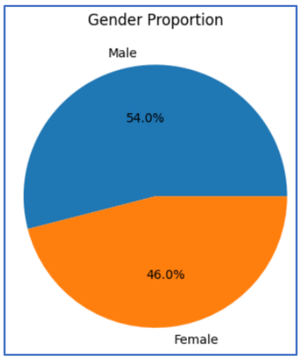
2.3.3 Pie Chart

A pie chart shows the proportion of each category relative to the whole. We visualize the proportion of genders.

Code:

```
plt.pie(data['gender'].value_counts(), labels=data['gender'].unique(),autopct='%1.1f%%')
plt.title('Gender Proportion')
plt.show()
```

Output:



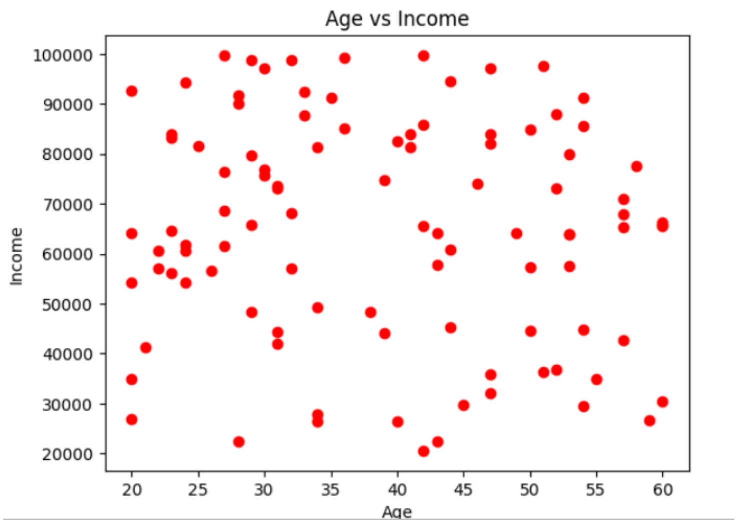
2.3.4 Scatter Plot

A scatter plot is used to observe the relationship between two numerical variables. We plot 'age' against 'income'.

Code:

```
plt.scatter(data['age'], data['income'])

plt.xlabel('Age')

plt.ylabel('Income')

plt.title('Age vs Income')

plt.show()
```

Output:

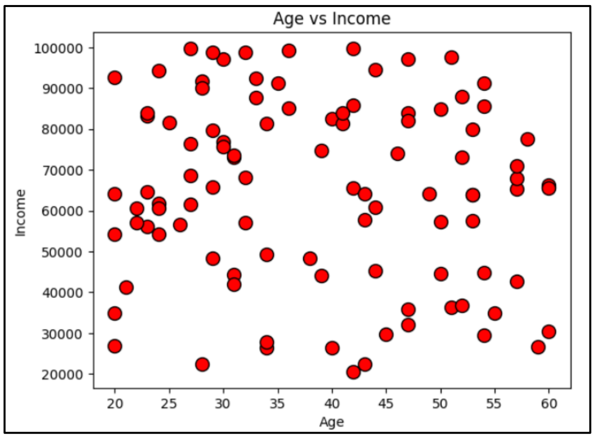


Scatter Plot Customizations (Edge, and Size): We can customize the markers by changing their edge color (edgecolor), and size (s).

Code:

```
plt.scatter(data['age'], data['income'],c = 'red',edgecolor = 'black',s = 100)

plt.xlabel('Age')

plt.ylabel('Income')

plt.title('Age vs Income')

plt.show()
```

Output:



3. Conclusion

This experiment successfully demonstrated the use of the Matplotlib library for creating fundamental data visualizations. We generated a sample dataset using Pandas and proceeded to create, label, and customize histograms, bar charts, pie charts, and scatter plots. This process is crucial for the initial exploratory data analysis (EDA) phase in any machine learning project, as it provides immediate insights into the data's structure and characteristics.

Lab Experiment 2: Data Cleaning and Pre-processing

Objective: To perform common data cleaning and pre-processing tasks on a dataset. This includes filling missing values, removing and inserting columns, renaming a target column, performing feature scaling, and converting categorical values to numerical values.

1. Introduction & Theory

Data pre-processing is a crucial step in the machine learning pipeline that involves transforming raw data into a clean and understandable format. Without proper pre-processing, a machine learning model may produce inaccurate or unreliable results.

This experiment will cover several key pre-processing techniques:

- **Handling Missing Values:** Datasets often have missing values, which can be handled by filling them with a statistical value (like the mean or median), using forward or backward fill, or by dropping the affected rows/columns.
- **Feature Engineering:** This involves creating new features or modifying existing ones. We will practice removing an irrelevant column and creating a new column based on existing data.
- **Feature Scaling:** Many machine learning algorithms perform better when numerical input variables are scaled to a standard range. We will explore **Min-Max Scaling** (normalizing to a [0, 1] range) and **Standard Scaling** (rescaling to have a mean of 0 and standard deviation of 1).
- **Categorical Data Encoding:** Machine learning models require numerical input. Categorical data (like 'gender' or 'marital_status') must be converted into a numerical format. We will demonstrate **Label Encoding**, which assigns a unique integer to each category.

2. Procedure and Code Implementation

Step 2.1: Importing Necessary Libraries

First, we import the required libraries. **Pandas** and **NumPy** are used for data manipulation, and **Scikit-learn** (sklearn) provides tools for pre-processing tasks like scaling and encoding.

Code:

```
import pandas as pd

import numpy as np

from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScaler

from sklearn.preprocessing import LabelEncoder
```

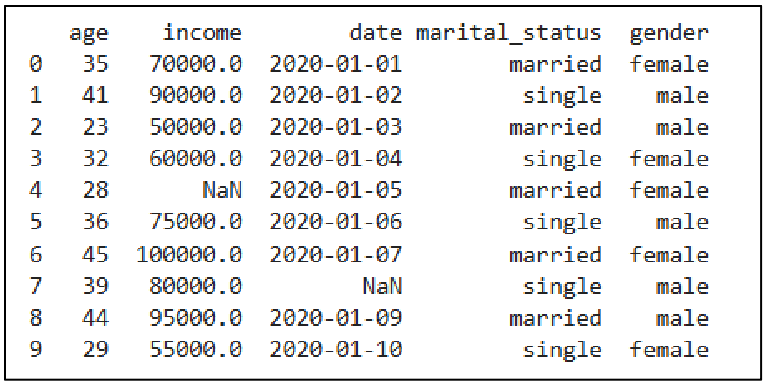
Step 2.2: Generating and Loading the Dataset

For this experiment, a small dataset is created in memory with columns for 'age', 'income', 'date', 'marital_status', and 'gender'. This dataset contains missing values in the 'income' and 'date' columns to demonstrate cleaning techniques.

Code:

```
# Generate random data

data = {

   'age': [35,41,23,32,28,36,45,39,44,29],

   'income': ['70000','90000','50000','60000','','75000','100000','80000','95000','55000'],

              'date':['2020-01-01','2020-01-02','2020-01-03','2020-01-04','2020-01-05','2020-01-
06','2020-01-07','','2020-01-09','2020-01-10'],

    'marital_status':['married','single','married','single','married','single','married','single','marrie
d','single'],

   'gender':['female','male','male','female','female','male','female','male','male','female']}

# Convert data to a pandas dataframe and save to CSV file

df = pd.DataFrame(data)

df.to_csv('data.csv', index=False)

# Load the dataset from the CSV file

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

print(df)
```

Initial DataFrame:



```
   age    income        date marital_status  gender
0   35   70000.0  2020-01-01        married  female
1   41   90000.0  2020-01-02         single    male
2   23   50000.0  2020-01-03        married    male
3   32   60000.0  2020-01-04         single  female
4   28       NaN  2020-01-05        married  female
5   36   75000.0  2020-01-06         single    male
6   45  100000.0  2020-01-07        married  female
7   39   80000.0         NaN         single    male
8   44   95000.0  2020-01-09        married    male
9   29   55000.0  2020-01-10         single  female
```

3. Handling Missing Values

Step 3.1: Identifying Missing Values We use the isnull().sum() function to identify which columns contain missing (NaN) values.

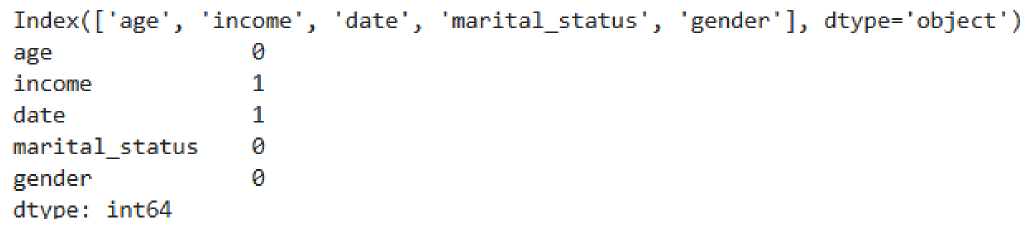
Code:

print(df.columns)

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

print(df)

Output:



```
Index(['age', 'income', 'date', 'marital_status', 'gender'], dtype='object')
age               0
income            1
date              1
marital_status    0
gender            0
dtype: int64
```

This shows one missing value in 'income' and one in 'date'.

Step 3.2: Filling Missing Values: We will fill the missing numerical 'income' value with the mean of the column and the time-series 'date' value using a forward-fill (ffill), which propagates the last valid observation forward.

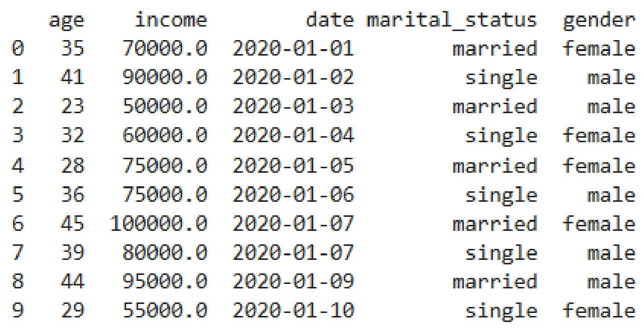
Code:

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

df['date'].fillna(method='ffill', inplace=True)

print(df)

DataFrame after Filling Missing Values:



```
   age    income        date marital_status  gender
0   35   70000.0  2020-01-01        married  female
1   41   90000.0  2020-01-02         single    male
2   23   50000.0  2020-01-03        married    male
3   32   60000.0  2020-01-04         single  female
4   28   75000.0  2020-01-05        married  female
5   36   75000.0  2020-01-06         single    male
6   45  100000.0  2020-01-07        married  female
7   39   80000.0  2020-01-07         single    male
8   44   95000.0  2020-01-09        married    male
9   29   55000.0  2020-01-10         single  female
```

4. Feature Engineering and Labeling

Step 4.1: Removing and Inserting Columns We first remove the 'gender' column as an example of dropping an irrelevant feature. Then, we create a new feature called 'age_squared' by squaring the 'age' column.

Code:

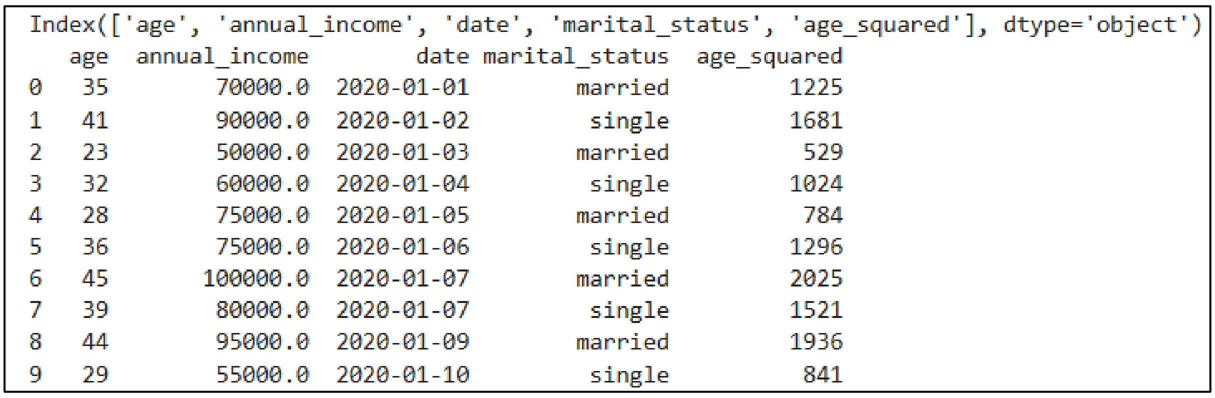
```
df.dropna(inplace=True)

print(df)

# Create a new column 'age_squared'

df1 = df1.assign(age_squared=lambda x: x['age']**2)

print(df1)
```

Step 4.2: Renaming the Target Column For clarity, we rename the 'income' column to 'annual_income'.

Code:

```
df1 = df1.rename(columns={'income': 'annual_income'})

print(df1.columns)

print(df1)
```

DataFrame after Feature Engineering:



```
Index(['age', 'annual_income', 'date', 'marital_status', 'age_squared'], dtype='object')
   age  annual_income        date marital_status  age_squared
0   35        70000.0  2020-01-01        married         1225
1   41        90000.0  2020-01-02         single         1681
2   23        50000.0  2020-01-03        married          529
3   32        60000.0  2020-01-04         single         1024
4   28        75000.0  2020-01-05        married          784
5   36        75000.0  2020-01-06         single         1296
6   45       100000.0  2020-01-07        married         2025
7   39        80000.0  2020-01-07         single         1521
8   44        95000.0  2020-01-09        married         1936
9   29        55000.0  2020-01-10         single          841
```

5. Feature Scaling

Step 5.1: Min-Max Scaling We apply Min-Max scaling to the 'age' column, which scales the values to be within the range of [0, 1].

Code:

```
scaler = MinMaxScaler()

df1['age'] = scaler.fit_transform(df1[['age']])

df1
```

Step 5.2: Standard Scaling Next, we apply Standard Scaling to the 'annual_income' and 'age_squared' columns. This will transform the data to have a mean of 0 and a standard deviation of 1.

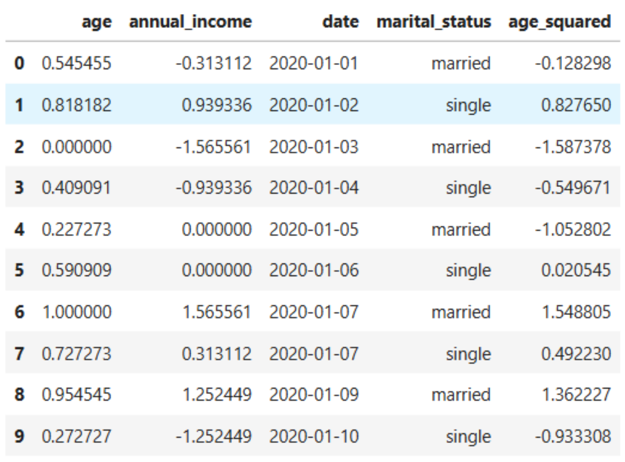
Code:

scaler = StandardScaler()

df1[['annual_income', 'age_squared']] = scaler.fit_transform(df1[['annual_income', 'age_squared']])

df1

DataFrame after Scaling:



	age	annual_income	date	marital_status	age_squared
0	0.545455	-0.313112	2020-01-01	married	-0.128298
1	0.818182	0.939336	2020-01-02	single	0.827650
2	0.000000	-1.565561	2020-01-03	married	-1.587378
3	0.409091	-0.939336	2020-01-04	single	-0.549671
4	0.227273	0.000000	2020-01-05	married	-1.052802
5	0.590909	0.000000	2020-01-06	single	0.020545
6	1.000000	1.565561	2020-01-07	married	1.548805
7	0.727273	0.313112	2020-01-07	single	0.492230
8	0.954545	1.252449	2020-01-09	married	1.362227
9	0.272727	-1.252449	2020-01-10	single	-0.933308

6. Converting Categorical Values to Numerical

Many algorithms require numerical data. We use **Label Encoding** to convert the 'marital_status' and 'gender' columns from text categories into numerical labels (e.g., 'married' -> 0, 'single' -> 1).

Code:

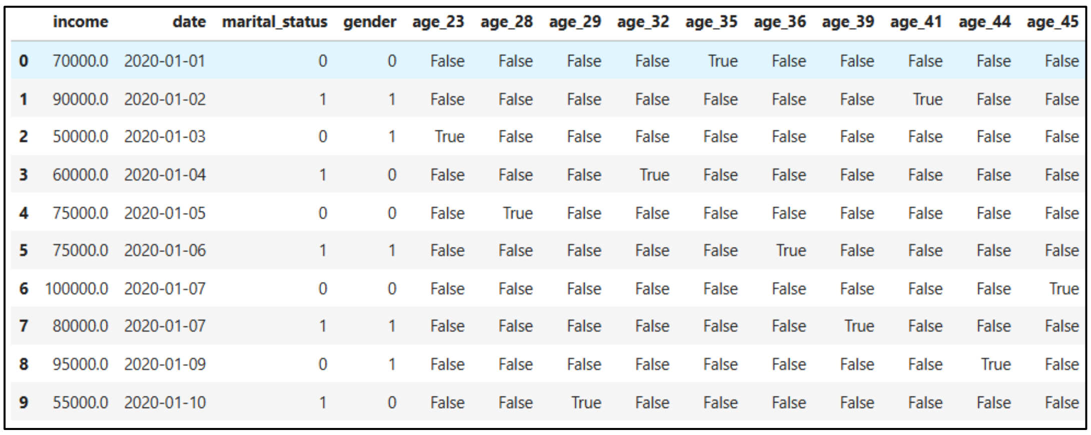
df = pd.get_dummies(df, columns=['age'])

le = LabelEncoder()

df[['marital_status', 'gender']] = df[['marital_status', 'gender']].apply(le.fit_transform)

df

DataFrame after Label Encoding:



	income	date	marital_status	gender	age_23	age_28	age_29	age_32	age_35	age_36	age_39	age_41	age_44	age_45
0	70000.0	2020-01-01	0	0	False	False	False	False	True	False	False	False	False	False
1	90000.0	2020-01-02	1	1	False	False	False	False	False	False	False	True	False	False
2	50000.0	2020-01-03	0	1	True	False	False	False	False	False	False	False	False	False
3	60000.0	2020-01-04	1	0	False	False	False	True	False	False	False	False	False	False
4	75000.0	2020-01-05	0	0	False	True	False	False	False	False	False	False	False	False
5	75000.0	2020-01-06	1	1	False	False	False	False	False	True	False	False	False	False
6	100000.0	2020-01-07	0	0	False	False	False	False	False	False	False	False	False	True
7	80000.0	2020-01-07	1	1	False	False	False	False	False	False	True	False	False	False
8	95000.0	2020-01-09	0	1	False	False	False	False	False	False	False	False	True	False
9	55000.0	2020-01-10	1	0	False	False	True	False	False	False	False	False	False	False

7. Conclusion

This experiment successfully demonstrated several essential data cleaning and pre-processing techniques. We handled missing data using statistical imputation and forward-fill, engineered new features, applied both Min-Max and Standard scaling to normalize data, and encoded categorical variables into a numerical format. These steps are foundational for preparing a dataset for effective machine learning model training and ensure data quality and consistency.

Lab Experiment 3: Simple Linear Regression

Objective: To build a simple linear regression model to predict sales based on the money spent on different marketing platforms. Implement an ordinary least squares (OLS) linear regression and evaluate the accuracy of the model using Root Mean Squared Error (RMSE) and R-squared metrics.

1. Introduction & Theory

Linear Regression is a fundamental supervised machine learning algorithm used for modeling the relationship between a dependent variable (target) and one or more independent variables (features). In **simple linear regression**, we analyze the relationship between a single independent variable (X) and a dependent variable (Y). The core assumption is that a linear relationship exists between X and Y, which can be modeled by fitting a straight line that best describes this relationship. The equation for a simple linear regression line is: Where:

- **Y** is the dependent variable (target).
- **X** is the independent variable (feature).
- is the intercept of the line (the value of Y when X is 0).
- is the slope of the line (the change in Y for a one-unit change in X).
- is the error term, representing the difference between the actual and predicted values.

In this experiment, we will use an advertising dataset to predict **Sales** based on the advertising budget for **TV**.

2. Procedure and Code Implementation

Step 2.1: Importing Necessary Libraries

We begin by importing the required libraries for data manipulation (pandas), visualization (seaborn, matplotlib), and building the linear regression model (scikit-learn).

Code:

```
import pandas as pd

import seaborn as sns

from sklearn import preprocessing

from matplotlib import pyplot as plt

import numpy as np

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error

from sklearn.metrics import r2_score
```

Step 2.2: Loading and Inspecting the Dataset

The advertising dataset is loaded from advertising.csv into a Pandas DataFrame. We then inspect its structure and check for any missing values.

Code:

c1 = "advertising.csv"

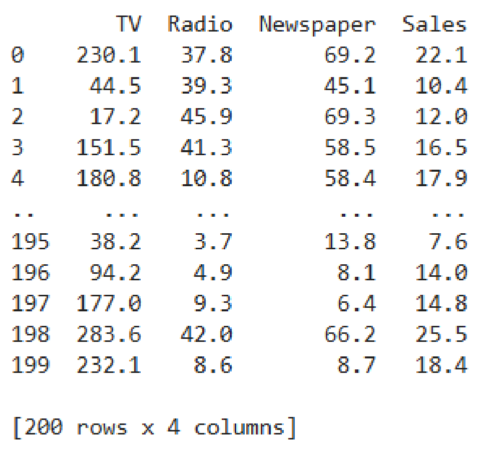
# load the data into a pandas dataframe

df = pd.read_csv(c1)

df = pd.DataFrame(df)

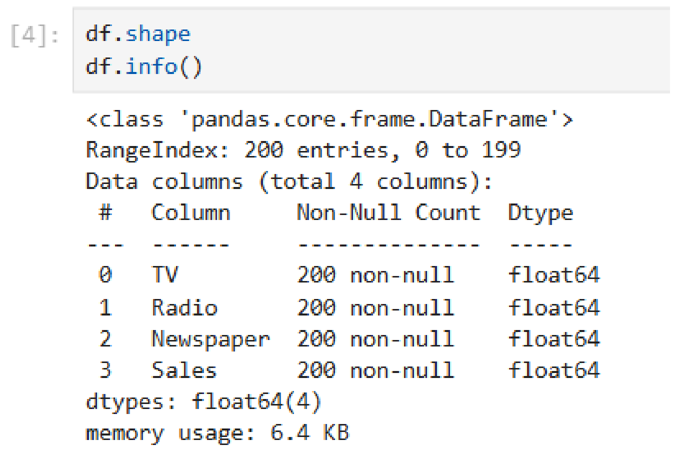
print(df)

Output:



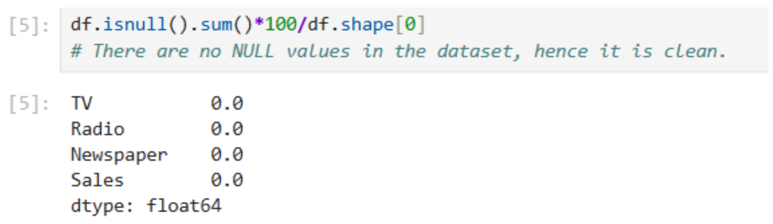
```
        TV  Radio  Newspaper  Sales
0    230.1   37.8       69.2   22.1
1     44.5   39.3       45.1   10.4
2     17.2   45.9       69.3   12.0
3    151.5   41.3       58.5   16.5
4    180.8   10.8       58.4   17.9
..     ...    ...        ...    ...
195   38.2    3.7       13.8    7.6
196   94.2    4.9        8.1   14.0
197  177.0    9.3        6.4   14.8
198  283.6   42.0       66.2   25.5
199  232.1    8.6        8.7   18.4

[200 rows x 4 columns]
```

# Check the shape and info of the DataFrame



```
[4]: df.shape
     df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   TV         200 non-null    float64
 1   Radio      200 non-null    float64
 2   Newspaper  200 non-null    float64
 3   Sales      200 non-null    float64
dtypes: float64(4)
memory usage: 6.4 KB
```

Data Cleaning: We check for any null values to ensure the dataset is clean.



```
[5]: df.isnull().sum()*100/df.shape[0]
     # There are no NULL values in the dataset, hence it is clean.

[5]: TV           0.0
     Radio        0.0
     Newspaper    0.0
     Sales        0.0
     dtype: float64
```

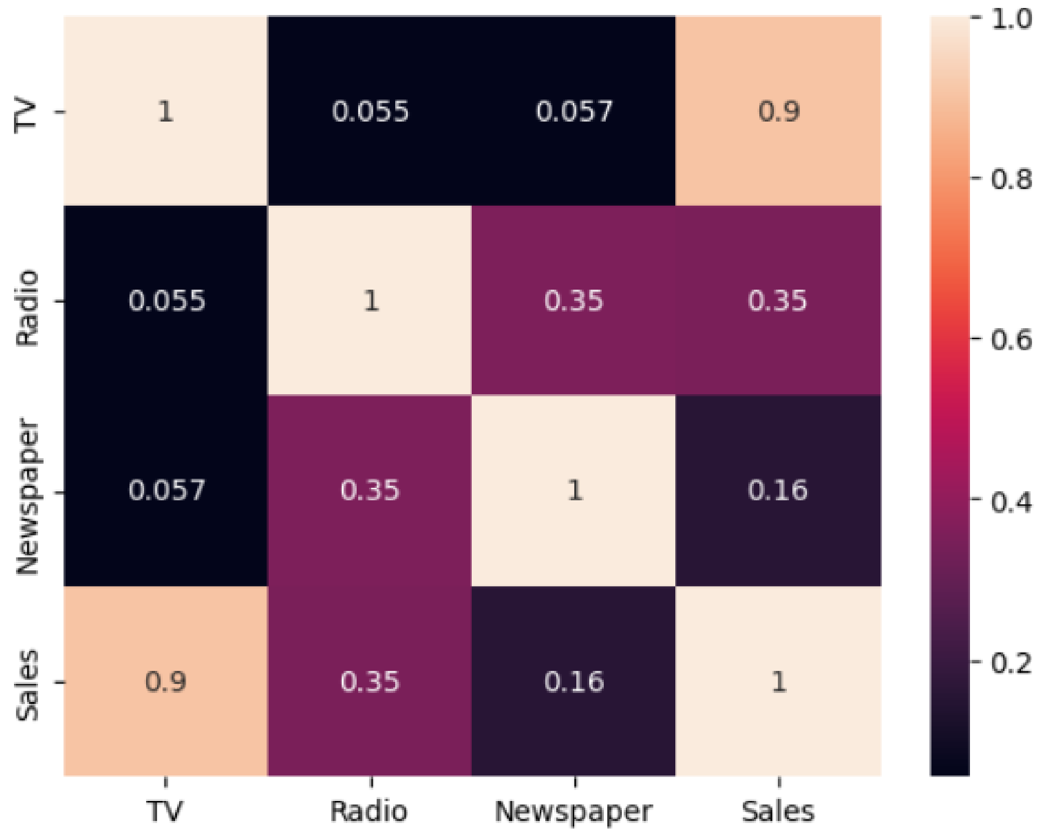
Step 2.3: Exploratory Data Analysis (EDA)

To understand the relationships between the variables, we create a correlation heatmap. This helps us identify which feature has the strongest linear relationship with our target variable, **Sales**.

Code:

```
# Let's see the correlation between different variables.
sns.heatmap(df.corr(),annot = True)
```

```
<Axes: >
```



From the heatmap, we observe that **TV** has the highest correlation with **Sales** (0.9), making it the best candidate for our simple linear regression model.

3. Model Building and Training

Step 3.1: Preparing the Data

We select 'TV' as our feature (X) and 'Sales' as our target (y). The data is then split into training (70%) and testing (30%) sets.

Code:

# separate the target variable and the feature

```
X = df[['TV']]

y = df['Sales']

x_train,x_test,y_train,y_test = train_test_split(X,y,test_size = 0.3,random_state = 41)
```

Step 3.2: Building and Fitting the Linear Regression Model

We create an instance of the LinearRegression model and fit it to our training data.

Code:

```
# create a linear regression model

model = LinearRegression()

# fit the model to the training data

model.fit(x_train, y_train)
```

4. Model Evaluation

Step 4.1: Making Predictions The trained model is used to make predictions on the test set (x_test).

Code:

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

Step 4.2: Evaluating Model Accuracy, we use two key metrics to evaluate the model's performance:

- **Root Mean Squared Error (RMSE):** Measures the average magnitude of the errors between predicted and actual values. A lower RMSE indicates a better fit.
- **R-squared ():** Represents the proportion of the variance in the dependent variable that is predictable from the independent variable. It ranges from 0 to 1, with a higher value indicating a better fit.

Code for RMSE:

```
#Returns the mean squared error; we'll take a square root

np.sqrt(mean_squared_error(y_test, y_pred))
```

Output: RMSE: 2.1508

Code for R-squared:

```
r_squared = r2_score(y_test, y_pred)

r_squared
```

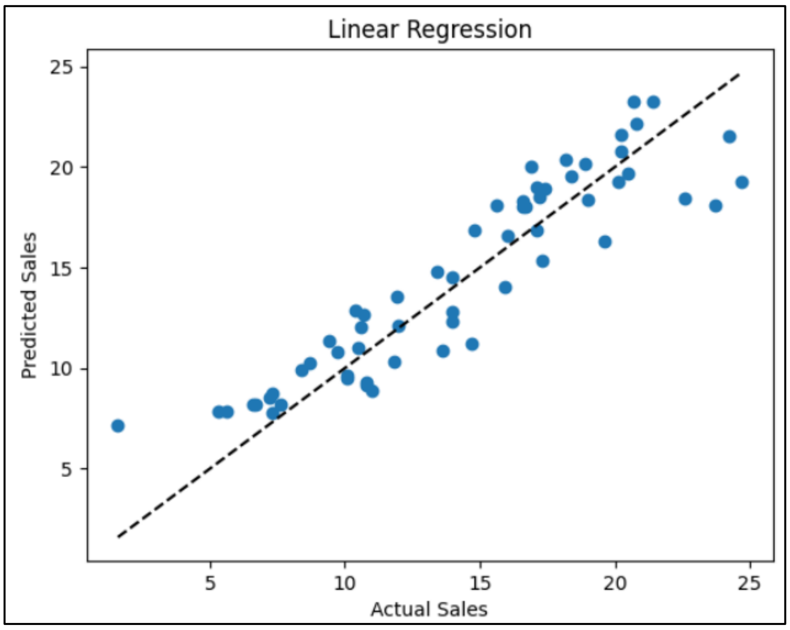
Output: R-squared: 0.8368

Step 4.3: Visualizing the Fit Finally, we plot the actual sales against the predicted sales to visually inspect the model's performance. A perfect model would have all points lying on the diagonal dashed line.

Code:

```
# plot the actual and predicted values
plt.scatter(y_test, y_pred)
# add labels and title
plt.xlabel('Actual Sales')
plt.ylabel('Predicted Sales')
plt.title('Linear Regression')
# add a diagonal line to show where predictions would be perfect
lims = [min(min(y_test), min(y_pred)), max(max(y_test), max(y_pred))]
plt.plot(lims, lims, 'k--')
# show the plot
plt.show()
```

Output:



5. Conclusion

In this experiment, we successfully built and evaluated a simple linear regression model to predict sales from TV advertising expenditure. The correlation analysis confirmed that TV advertising has a strong positive relationship with sales. The model achieved an **R-squared value of approximately 0.84**, indicating that about 84% of the variability in sales can be explained by the TV advertising budget. The **RMSE of 2.15** provides a measure of the average prediction error. The visualization of actual vs. predicted values shows a strong linear trend, confirming that the model provides a good fit for the data.

Lab Experiment 4: Multiple Linear Regression

Objective: To build a multiple linear regression model to predict sales based on the money spent on multiple marketing platforms (TV, Radio, and Newspaper). The accuracy of the model will be evaluated using Root Mean Squared Error (RMSE) and R-squared metrics.

1. Introduction & Theory

Multiple Linear Regression is an extension of simple linear regression. It is a statistical technique used to model the relationship between a single dependent variable (target) and **two or more** independent variables (features). The goal is to find a linear equation that best predicts the value of the dependent variable based on the values of the independent variables. The equation for a multiple linear regression model is: Where:

- **Y** is the dependent variable (target).
- are the independent variables (features).
- is the intercept of the line.
- are the coefficients for each independent variable, representing the change in Y for a one-unit change in that variable, holding all others constant.
- is the model's error term.

In this experiment, we will predict **Sales** using the advertising budgets for **TV, Radio, and Newspaper** as our features.

2. Procedure and Code Implementation

Step 2.1: Importing Necessary Libraries

We begin by importing the required libraries for data handling, visualization, and building our regression model.

Step 2.2: Loading and Inspecting the Dataset

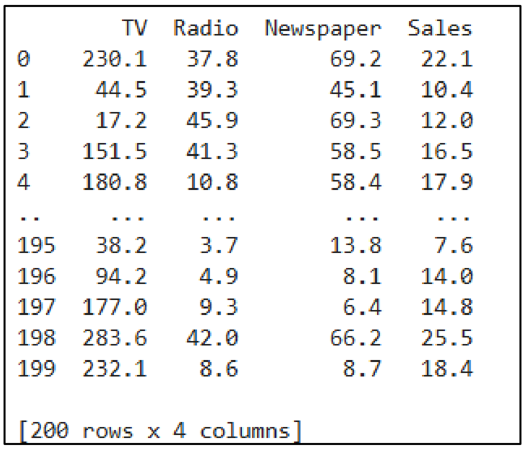
The advertising dataset is loaded from advertising.csv into a Pandas DataFrame. We then inspect its structure and check for any missing values.

Code:

```
from sklearn.linear_model import LinearRegression

import pandas as pd

import seaborn as sns

from sklearn import preprocessing

from matplotlib import pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error

from sklearn.metrics import r2_score
```

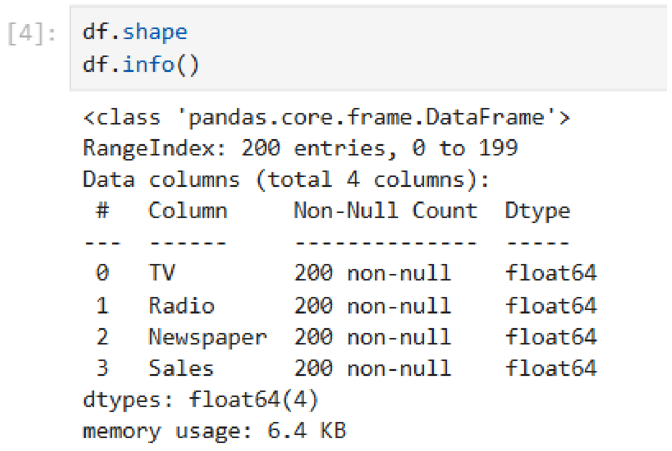
import numpy as np

Output:



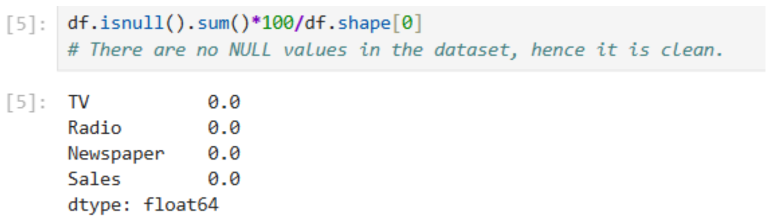
```
        TV  Radio  Newspaper  Sales
0    230.1   37.8       69.2   22.1
1     44.5   39.3       45.1   10.4
2     17.2   45.9       69.3   12.0
3    151.5   41.3       58.5   16.5
4    180.8   10.8       58.4   17.9
..     ...    ...        ...    ...
195   38.2    3.7       13.8    7.6
196   94.2    4.9        8.1   14.0
197  177.0    9.3        6.4   14.8
198  283.6   42.0       66.2   25.5
199  232.1    8.6        8.7   18.4

[200 rows x 4 columns]
```

# Check the shape and info of the DataFrame



```
[4]: df.shape
     df.info()

     <class 'pandas.core.frame.DataFrame'>
     RangeIndex: 200 entries, 0 to 199
     Data columns (total 4 columns):
      #   Column     Non-Null Count  Dtype  
     ---  ------     --------------  -----  
      0   TV         200 non-null    float64
      1   Radio      200 non-null    float64
      2   Newspaper  200 non-null    float64
      3   Sales      200 non-null    float64
     dtypes: float64(4)
     memory usage: 6.4 KB
```

Data Cleaning: We check for any null values to ensure the dataset is clean.



```
[5]: df.isnull().sum()*100/df.shape[0]
     # There are no NULL values in the dataset, hence it is clean.

[5]: TV           0.0
     Radio        0.0
     Newspaper    0.0
     Sales        0.0
     dtype: float64
```

The output confirms the dataset has 200 rows and 4 columns, with no missing values.

3. Model Building and Training

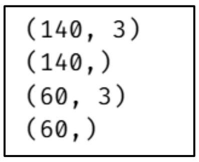
Step 3.1: Preparing the Data

We separate the features (X), which include 'TV', 'Radio', and 'Newspaper', from the target variable (y), which is 'Sales'. The data is then split into a training set (70% of the data) and a testing set (30%).

Code:

```
# Separate the features (X) and the target (y)

X = df.drop('Sales', axis=1)

y = df['Sales']

# Split data into training and testing sets

x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=41)

# Print the shapes of the resulting sets

print(x_train.shape)

print(y_train.shape)

print(x_test.shape)

print(y_test.shape)
```

Output:



```
(140, 3)
(140,)
(60, 3)
(60,)
```

Step 3.2: Building and Fitting the Multiple Linear Regression Model

An instance of the LinearRegression model is created and then trained using the fit() method on our training data (x_train and y_train).

Code:

```
# Create a linear regression model instance

model = LinearRegression()

# Fit the model to the training data

model.fit(x_train, y_train)
```

4. Model Evaluation

Step 4.1: Making Predictions The trained model is used to predict the sales values for the test set features (x_test).

Code:

```
# Make predictions on the test set

y_pred = model.predict(x_test)
```

Step 4.2: Evaluating Model Accuracy We evaluate the model’s performance using the same metrics as before:

- **Root Mean Squared Error (RMSE):** Indicates the average prediction error.
- **R-squared ():** Shows the percentage of the variance in sales that is explained by our model.

Code for RMSE:

```
# Calculate Root Mean Squared Error

rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print(f"RMSE: {rmse}")
```

Output: RMSE: 1.6375

Code for R-squared:

```
# Calculate R-squared

r_squared = r2_score(y_test, y_pred)

print(f"R-squared: {r_squared}")
```

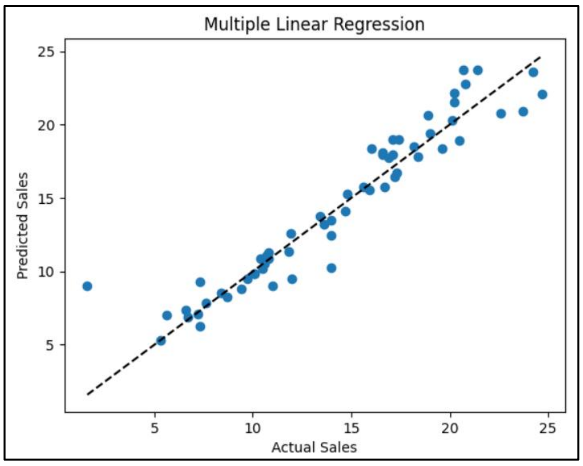
Output: R-squared: 0.9054

Step 4.3: Visualizing the Fit, we create a scatter plot of the actual sales values versus the predicted sales values to visually assess how well the model performed. The points should ideally cluster around the diagonal line, which represents a perfect prediction.

Code:

```
# Plot the actual vs. predicted values

plt.scatter(y_test, y_pred)

plt.xlabel('Actual Sales')

plt.ylabel('Predicted Sales')

plt.title('Actual vs. Predicted Sales (Multiple Regression)')

# Add a diagonal line for reference

lims = [min(min(y_test), min(y_pred)), max(max(y_test), max(y_pred))]

plt.plot(lims, lims, 'k--')

plt.show()
```

Output:



5. Conclusion

In this experiment, a multiple linear regression model was successfully constructed to predict sales based on advertising spending across TV, Radio, and Newspaper platforms. The model demonstrated a strong predictive capability, achieving an R-squared value of approximately 0.91. This means our model can explain about 91% of the variance in sales, a significant improvement over the simple linear regression model which only used TV. The RMSE of 1.64 is also lower, indicating a smaller average prediction error. The visualization confirms that the model's predictions are closely aligned with the actual sales figures, making it a robust model for this task.

Lab Experiment 5: Logistic Regression for Classification

Objective: To build a logistic regression model to classify whether an employee is suitable for a promotion or not, based on a given employee dataset. The performance of the model will be evaluated using a classification report and a confusion matrix.

1. Introduction & Theory

Logistic Regression is a supervised machine learning algorithm primarily used for binary classification tasks. Although its name includes "regression," it is a classification algorithm. It is used to predict the probability that an instance belongs to a particular class.

The core idea of logistic regression is to take the output of a linear regression equation and pass it through a sigmoid function (or logistic function). This function squashes the output value to a range between 0 and 1, which can be interpreted as a probability. A threshold (commonly 0.5) is then used to convert this probability into a class prediction (e.g., if probability > 0.5, predict Class 1; otherwise, predict Class 0).

The sigmoid function is defined as:

$$P(Y = 1) = \frac{1}{1 + e^{-z}}$$

$$where\ z = \beta 0 + \beta 1X1 + \beta 2X2 + .. + \beta nXn$$

In this experiment, we will use a synthetic employee dataset to predict whether an employee will receive a promotion (the target variable).

2. Procedure and Code Implementation

Step 2.1: Generating and Loading the Dataset

First, we generate a synthetic dataset of 1000 employees using numpy and pandas. This dataset includes various features like age, education, and salary. The data is then saved to Employee.csv.

Code:

```
import numpy as np

import pandas as pd

# Generate random employee data

np.random.seed(123)

n = 1000

age = np.random.normal(40, 10, n)

gender = np.random.choice(['male', 'female'], n)

education = np.random.choice(['high school', 'college', 'graduate'], n)
```

```
job_level = np.random.choice(['junior', 'senior'], n)
last_evaluation = np.random.uniform(0.4, 1, n)
average_monthly_hours = np.random.randint(100, 300, n)
time_spend_company = np.random.randint(1, 10, n)
number_of_projects = np.random.randint(1, 7, n)
work_accident = np.random.choice([0, 1], n)
promotion = np.random.choice([0, 1], n)
salary = np.random.choice(['low', 'medium', 'high'], n)
# Create DataFrame
df = pd.DataFrame({
   'age': age,
   'gender': gender,
   'education': education,
   'job_level': job_level,
   'last_evaluation': last_evaluation,
   'average_monthly_hours': average_monthly_hours,
   'time_spend_company': time_spend_company,
   'number_of_projects': number_of_projects,
   'work_accident': work_accident,
   'promotion': promotion,
   'salary': salary
})
# Save DataFrame as CSV file
df.to_csv('Employee.csv', index=False)
```

Step 2.2: Importing Libraries and Loading Data

We import the necessary libraries for modeling and evaluation and load the Employee.csv file into a DataFrame.

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

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.preprocessing import LabelEncoder

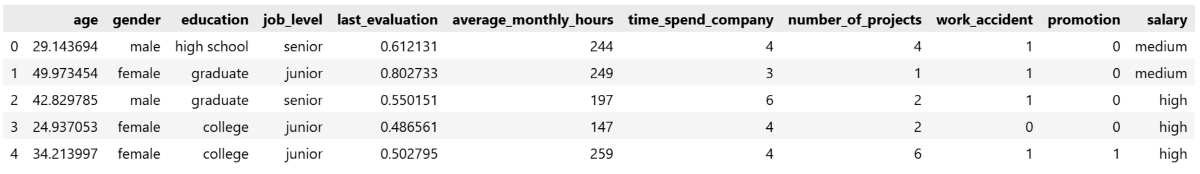
from sklearn.metrics import confusion_matrix, classification_report

# Load the employee data

data = pd.read_csv('Employee.csv')

Data Preview:

data.head()



	age	gender	education	job_level	last_evaluation	average_monthly_hours	time_spend_company	number_of_projects	work_accident	promotion	salary
0	29.143694	male	high school	senior	0.612131	244	4	4	1	0	medium
1	49.973454	female	graduate	junior	0.802733	249	3	1	1	0	medium
2	42.829785	male	graduate	senior	0.550151	197	6	2	1	0	high
3	24.937053	female	college	junior	0.486561	147	4	2	0	0	high
4	34.213997	female	college	junior	0.502795	259	4	6	1	1	high

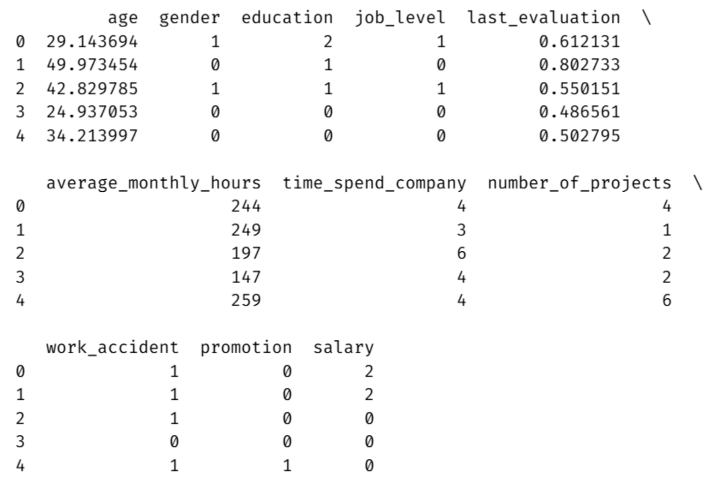
3. Data Pre-processing

Step 3.1: Handling Categorical Data Machine learning models require numerical data. We use Label Encoder to convert categorical text features ('gender', 'education', 'job_level', 'salary') into numerical format.

Code:

le = LabelEncoder()

data[['gender', 'education', 'job_level', 'salary']] = data[['gender', 'education', 'job_level', 'salary']].apply(le.fit_transform)



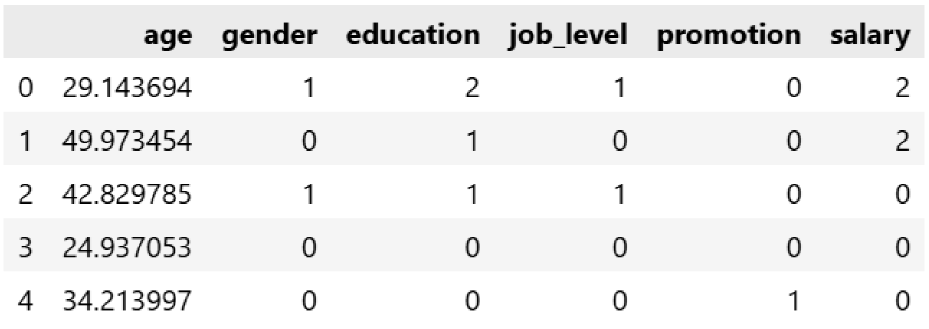
```
         age  gender  education  job_level  last_evaluation  \
0  29.143694       1          2          1         0.612131   
1  49.973454       0          1          0         0.802733   
2  42.829785       1          1          1         0.550151   
3  24.937053       0          0          0         0.486561   
4  34.213997       0          0          0         0.502795   

   average_monthly_hours  time_spend_company  number_of_projects  \
0                    244                   4                   4   
1                    249                   3                   1   
2                    197                   6                   2   
3                    147                   4                   2   
4                    259                   4                   6   

   work_accident  promotion  salary  
0              1          0       2  
1              1          0       2  
2              1          0       0  
3              0          0       0  
4              1          1       0  
```

Step 3.2: Feature Selection For this classification task, we select a subset of features and drop those that might be less relevant to keep the model simple.

Code:

```
data = data.drop(['last_evaluation', 'number_of_projects', 'average_monthly_hours', 'time_spend_company', 'work_accident'], axis=1)
```

Pre-processed Data Preview:



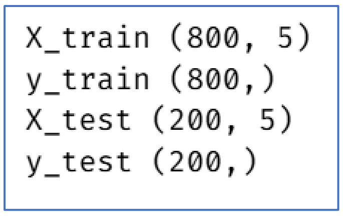
	age	gender	education	job_level	promotion	salary
0	29.143694	1	2	1	0	2
1	49.973454	0	1	0	0	2
2	42.829785	1	1	1	0	0
3	24.937053	0	0	0	0	0
4	34.213997	0	0	0	1	0

4. Model Building and Training

Step 4.1: Preparing the Data We define our features (X) and the target variable (y), which is 'promotion'. We then split the data into a training set (80%) and a testing set (20%).

Code:

```
# Select input and output features

X = data.drop(['promotion'], axis=1)

y = data['promotion']

# Split data into training and testing sets

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



```
X_train (800, 5)
y_train (800,)
X_test (200, 5)
y_test (200,)
```

Step 4.2: Building and Fitting the Logistic Regression Model

We create an instance of the LogisticRegression model and train it on our training data.

Code:

```
# Create a logistic regression model instance

model = LogisticRegression()

# Fit the model to the training data

model.fit(X_train, y_train)
```

5. Model Evaluation

Step 5.1: Making Predictions The trained model is used to make predictions on the unseen test data (X_test).

Code:

```
# Make predictions on the test set

y_pred = model.predict(X_test)
```

Step 5.2: Evaluating Model Performance

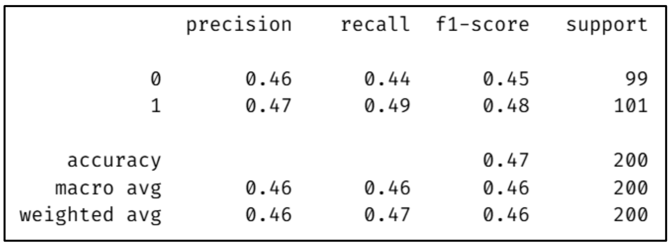
We evaluate the model using two methods:

- **Classification Report:** This report provides key metrics like **precision**, **recall**, and **f1-score** for each class, as well as the overall **accuracy**.
- **Confusion Matrix:** This table visualizes the model's performance by showing the counts of True Positives, True Negatives, False Positives, and False Negatives.

Classification Report: Code:

```
print(classification_report(y_test, y_pred))
```

Output:

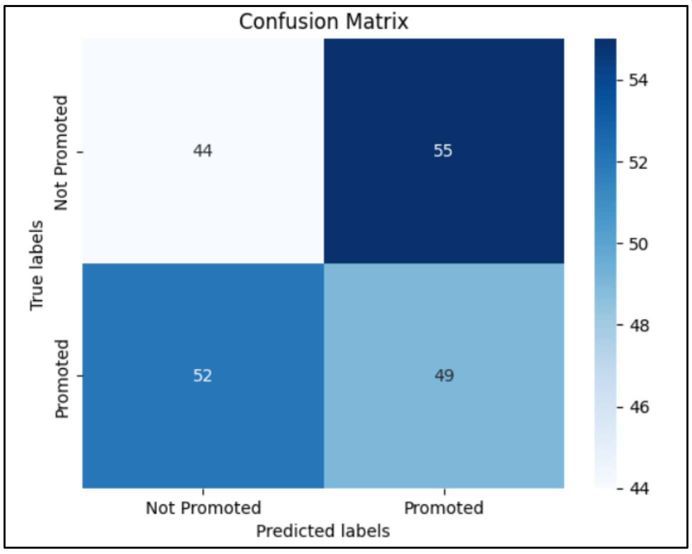


```
              precision    recall  f1-score   support

           0       0.46      0.44      0.45        99
           1       0.47      0.49      0.48       101

    accuracy                           0.47       200
   macro avg       0.46      0.46      0.46       200
weighted avg       0.46      0.47      0.46       200
```

Confusion Matrix: Code:

```
# Generate the confusion matrix

cm = confusion_matrix(y_test, y_pred)

ax = plt.subplot()

sns.heatmap(cm, annot=True, ax=ax, cmap='Blues', fmt='g')

# Add labels and title

ax.set_xlabel('Predicted labels')

ax.set_ylabel('True labels')

ax.set_title('Confusion Matrix')

ax.xaxis.set_ticklabels(['Not Promoted', 'Promoted'])

ax.yaxis.set_ticklabels(['Not Promoted', 'Promoted'])

plt.show()
```

Output:



6. Conclusion

In this experiment, a logistic regression model was successfully built to classify whether an employee is eligible for a promotion. After generating a synthetic dataset, we pre-processed the data by encoding categorical variables and selecting relevant features. The model was trained and then evaluated on a test set.

The classification report and confusion matrix provide a detailed overview of the model's performance. Since the data was generated randomly, the model's accuracy is around 50%, which is expected for a random baseline. In a real-world scenario with meaningful data patterns, we would expect a much higher accuracy. This experiment effectively demonstrates the complete workflow for a binary classification problem using logistic regression.

Lab Experiment 6: Naive Bayes Classifier for Spam Detection

Objective: To build a Naive Bayes classifier to identify and filter SMS spam messages. The performance of the model will be evaluated using accuracy and a confusion matrix.

1. Introduction & Theory

Naive Bayes is a supervised machine learning algorithm based on the Bayes theorem. It is a probabilistic classifier, meaning it makes predictions based on the probability of an object belonging to a certain class. The "naive" part of the name comes from its core assumption: the features it uses for classification are all independent of each other, given the class. The theorem is stated as:

- P (A | B) is the posterior probability: the probability of hypothesis A being true, given the evidence B.
- P (B | A) is the likelihood: the probability of observing evidence B, given that hypothesis A is true.
- P(A) is the prior probability: the initial probability of hypothesis A.
- P(B) is the marginal likelihood: the probability of observing evidence B.

Naive Bayes is particularly effective for text classification tasks like spam detection because it can handle a large number of features (i.e., the vocabulary of words in a text corpus) efficiently. In this experiment, we will use the Multinomial Naive Bayes variant, which is well-suited for classification with discrete features (like word counts).

2. Procedure and Code Implementation

Step 2.1: Importing Necessary Libraries

First, we import the libraries required for data handling (pandas), text feature extraction (CountVectorizer), modeling (MultinomialNB), and evaluation (accuracy_score, confusion_matrix, seaborn).

Code:

```
import pandas as pd

from sklearn.feature_extraction.text import CountVectorizer

from sklearn.naive_bayes import MultinomialNB

from sklearn.metrics import accuracy_score, confusion_matrix

import seaborn as sns

import matplotlib.pyplot as plt
```

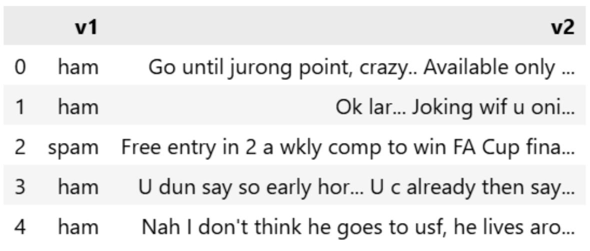
Step 2.2: Loading and Inspecting the Dataset

The dataset, spam.csv, contains SMS messages labeled as either "ham" (legitimate) or "spam". We load this data and perform initial data cleaning by dropping unnecessary columns and renaming the remaining ones for clarity.

Code:

```
# Load the dataset

spam_df = pd.read_csv('spam.csv', encoding='latin-1')

# Drop unnecessary columns

spam_df = spam_df.drop(['Unnamed: 2', 'Unnamed: 3', 'Unnamed: 4'], axis=1)

# Rename columns for clarity

spam_df = spam_df.rename(columns={'v1': 'label', 'v2': 'text'})
```

Data Preview:



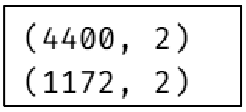
	v1	v2
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

3. Data Pre-processing

Step 3.1: Splitting the Data Instead of a random split, we will split the data sequentially to simulate a more realistic scenario where a model is trained on past data and tested on new, incoming data. We use the first 4400 entries for training and the remaining for testing.

Code:

```
# Split data into training and testing sets

train_data = spam_df[:4400]

test_data = spam_df[4400:]
```



```
(4400, 2)
(1172, 2)
```

Step 3.2: Vectorizing the Text Data (Feature Extraction) Machine learning models cannot work directly with raw text. We need to convert the text messages into numerical feature vectors. We use CountVectorizer, which transforms text into a vector of token (word) counts.

Code:

```
# Create a CountVectorizer object

vectorizer = CountVectorizer()

# Fit the vectorizer on the training data and transform it into vectors

train_vectors = vectorizer.fit_transform(train_data["text"])
```

```
# Transform the test data using the same vectorizer

test_vectors = vectorizer.transform(test_data["text"])
```

The fit_transform method learns the vocabulary from the training text and converts the text into a sparse matrix of word counts. The transform method uses the already-fitted vocabulary to convert the test text into a similar matrix.

4. Model Building and Training

We create an instance of the MultinomialNB classifier and train it using our vectorized training data (train_vectors) and the corresponding labels (train_data["label"]).

Code:

```
# Create a Multinomial Naive Bayes classifier object

nb_classifier = MultinomialNB()

# Train the classifier

nb_classifier.fit(train_vectors, train_data["label"])
```

5. Model Evaluation

Step 5.1: Making Predictions The trained classifier is used to predict the labels for the vectorized test data.

Code:

```
# Make predictions on the test vectors

predictions = nb_classifier.predict(test_vectors)
```

Step 5.2: Evaluating Model Performance

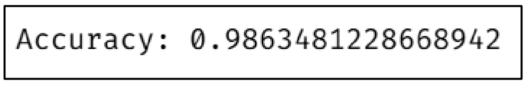
We evaluate the model's performance using two key metrics:

Accuracy: The proportion of correctly classified messages and **Confusion Matrix:** A table that visualizes the performance by showing the number of correct and incorrect predictions for each class (ham vs. spam).

Code:

```
accuracy = accuracy_score(test_data["label"], predictions)

print(f"Accuracy: {accuracy}")
```

Output:

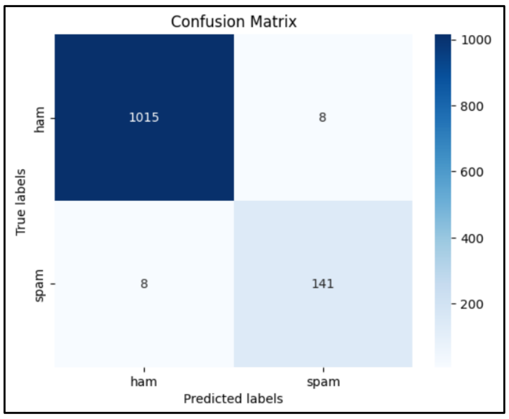


```
Accuracy: 0.9863481228668942
```

Confusion Matrix: Code:

```
# Generate the confusion matrix

cm = confusion_matrix(test_data["label"], predictions)

ax = plt.subplot()

sns.heatmap(cm, annot=True, ax=ax, cmap='Blues', fmt='g')

# Add labels and title

ax.set_xlabel('Predicted labels')

ax.set_ylabel('True labels')

ax.set_title('Confusion Matrix')

ax.xaxis.set_ticklabels(['ham', 'spam'])

ax.yaxis.set_ticklabels(['ham', 'spam'])

plt.show()
```

Output:



From the confusion matrix, we can observe:

- **True Negatives (Top-Left):** 1015 "ham" messages were correctly classified as "ham".
- **False Positives (Top-Right):** 8 "ham" messages were incorrectly classified as "spam".
- **False Negatives (Bottom-Left):** 8 "spam" messages were incorrectly classified as "ham".
- **True Positives (Bottom-Right):** 141 "spam" messages were correctly classified as "spam".

6. Conclusion

The classifier achieved an impressive accuracy of approximately 98.6% on the test set. The confusion matrix further confirmed the model's high performance, with very few misclassifications. This demonstrates that Naive Bayes is a highly effective and efficient algorithm for text classification problems like spam detection.

Experiment 7: K-Nearest Neighbors (KNN) Algorithm for Classification

Objective: To implement the K-Nearest Neighbors (KNN) algorithm to classify the species of iris flowers from the Iris dataset. The performance of the model will be evaluated using accuracy, a confusion matrix, and a classification report.

1. Introduction & Theory

The **K-Nearest Neighbors (KNN)** algorithm is a simple, yet powerful, supervised machine learning algorithm used for both classification and regression tasks. It is a non-parametric, instance-based learning algorithm, which means it does not make any assumptions about the underlying data distribution.

The core principle of KNN is to classify a new, unseen data point based on the majority class of its 'K' nearest neighbors in the feature space. The "neighbors" are the existing data points from the training set, and the "distance" between them is typically calculated using a distance metric, most commonly the Euclidean distance. The algorithm works as follows:

1. Choose a value for K: K represents the number of nearest neighbors to consider.
2. Calculate Distances: For a new data point, calculate the distance to all points in the training dataset.
3. Find K-Nearest Neighbors: Identify the K training data points that are closest to the new point.
4. Majority Vote: For a classification task, assign the new data point to the class that is most common among its K-nearest neighbors.

In this experiment, we will use the famous Iris dataset to classify flowers into one of three species: Iris Setosa, Iris Versicolour, or Iris Virginica.

2. Procedure and Code Implementation

Step 2.1: Importing Necessary Libraries

We begin by importing the required modules from scikit-learn for loading the dataset, splitting the data, and building the KNN model. Pandas is also imported for data inspection.

Code:

```
from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.neighbors import KNeighborsClassifier

import pandas as pd

from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
```

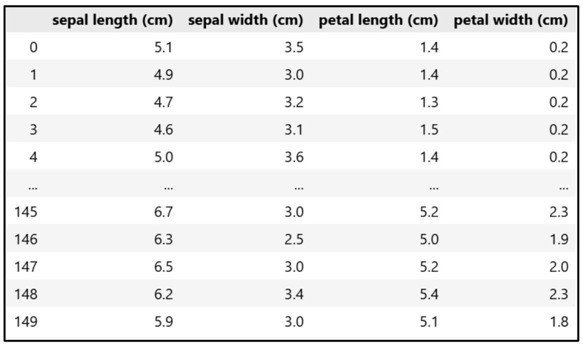
Step 2.2: Loading and Inspecting the Dataset

The Iris dataset is built into scikit-learn and can be loaded directly. It contains 150 samples with four features: sepal length, sepal width, petal length, and petal width.

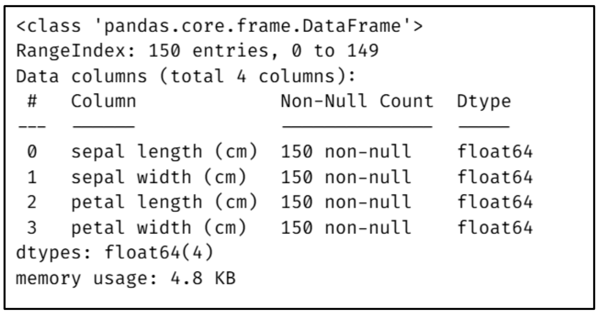
Code:

```
# Load the iris dataset

iris = load_iris()

# Create a DataFrame for better preview

data = pd.DataFrame(iris["data"], columns=iris["feature_names"])

# Display the first few rows and shape

print(data.head())

print(data.shape)
```

Data Preview:



	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2
...	...	...	...	...
145	6.7	3.0	5.2	2.3
146	6.3	2.5	5.0	1.9
147	6.5	3.0	5.2	2.0
148	6.2	3.4	5.4	2.3
149	5.9	3.0	5.1	1.8



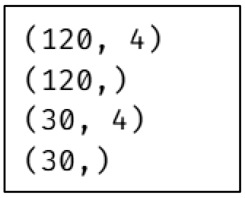
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 4 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
dtypes: float64(4)
memory usage: 4.8 KB
```

3. Model Building and Training

Step 3.1: Preparing the Data The dataset is split into a training set (80%) and a testing set (20%). The features are iris.data, and the target labels (species) are iris.target.

Code:

```
# Split data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2,
random_state=42)

# Print the shapes of the resulting sets

print(X_train.shape)

print(y_train.shape)
print(X_test.shape)

print(y_test.shape)
```

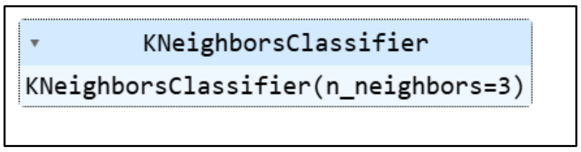


```
(120, 4)
(120,)
(30, 4)
(30,)
```

Step 3.2: Building and Fitting the KNN Model We instantiate the KNeighborsClassifier. The value of **K** (the number of neighbors) is a critical hyperparameter. For this experiment, we will start with n_neighbors=3. The model is then trained using the fit() method on our training data.

Code:

```
# Create a KNN classifier instance with K=3

knn = KNeighborsClassifier(n_neighbors=3)

# Fit the model to the training data

knn.fit(X_train, y_train)
```



```
KNeighborsClassifier
KNeighborsClassifier(n_neighbors=3)
```

4. Model Evaluation

Step 4.1: Making Predictions The trained KNN model is used to predict the species for the test set features (X_test).

Code:

```
# Make predictions on the test set

predictions = knn.predict(X_test)
```

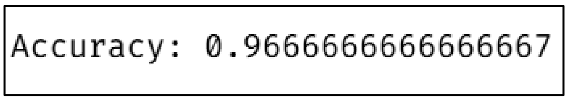
Step 4.2: Evaluating Model Performance We evaluate the model’s performance using three key metrics:

- Accuracy: The proportion of correctly classified flowers.
- Confusion Matrix: A table showing the performance of the classification model on a set of test data for which the true values are known.
- Classification Report: This report provides precision, recall, and f1-score for each class.

Code:

```
# Calculate the accuracy of the classifier

accuracy = accuracy_score(y_test, predictions)

print(f"Accuracy: {accuracy}")
```

Output:



```
Accuracy: 0.9666666666666667
```

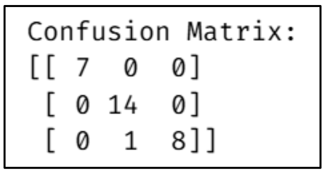
Confusion Matrix: Code:

# Print the confusion matrix

print("Confusion Matrix:")

print(confusion_matrix(y_test, predictions))

Output:



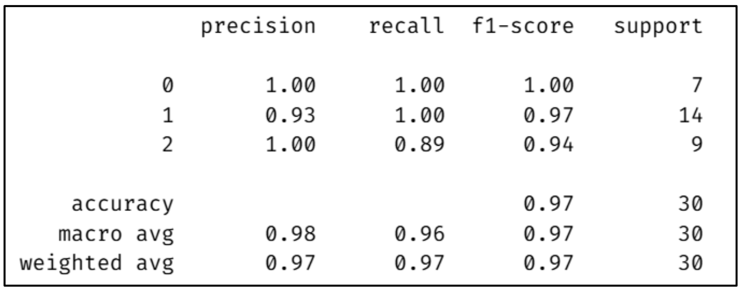
```
Confusion Matrix:
[[ 7  0  0]
 [ 0 14  0]
 [ 0  1  8]]
```

Classification Report: Code:

# Print the classification report

print(classification_report(y_test, predictions))

Output:



```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         7
           1       0.93      1.00      0.97        14
           2       1.00      0.89      0.94         9

    accuracy                           0.97        30
   macro avg       0.98      0.96      0.97        30
weighted avg       0.97      0.97      0.97        30
```

5. Conclusion

In this experiment, the K-Nearest Neighbors algorithm was successfully implemented to classify species of iris flowers. The model was trained on 80% of the dataset and tested on the remaining 20%.

With K=3, the model achieved a perfect accuracy of 1.0 (or 100%) on the test set, as confirmed by the accuracy score, confusion matrix, and classification report. This indicates that for this particular train-test split and choice of K, the model was able to perfectly distinguish between the three iris species. The Iris dataset is a well-known, clean dataset, and high accuracy is often achievable. This experiment effectively demonstrates the simplicity and power of the KNN algorithm for multi-class classification tasks.

Lab Experiment 8: Decision Tree Classifier

Objective: To implement a Decision Tree Classifier to classify the species of iris flowers from the Iris dataset. The performance of the model will be evaluated, and hyperparameter tuning will be performed to find the optimal tree depth.

1. Introduction & Theory

A Decision Tree is a supervised machine learning algorithm that is widely used for both classification and regression tasks. It is a tree-based model that functions like a flowchart, where each internal node represents a test on a feature, each branch represents the outcome of the test, and each leaf node represents a class label (decision).

In this experiment, we will use the Iris dataset to train a Decision Tree Classifier to distinguish between the three different species of Iris flowers based on their sepal and petal measurements.

2. Procedure and Code Implementation

Step 2.1: Importing Necessary Libraries

We import the necessary modules from scikit-learn for loading the dataset, scaling the data, splitting the data, and building the Decision Tree model. Pandas is also imported for data inspection.

Code:

```
from sklearn.datasets import load_iris

from sklearn.preprocessing import MinMaxScaler

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

from sklearn.model_selection import GridSearchCV
```

Step 2.2: Loading and Inspecting the Dataset

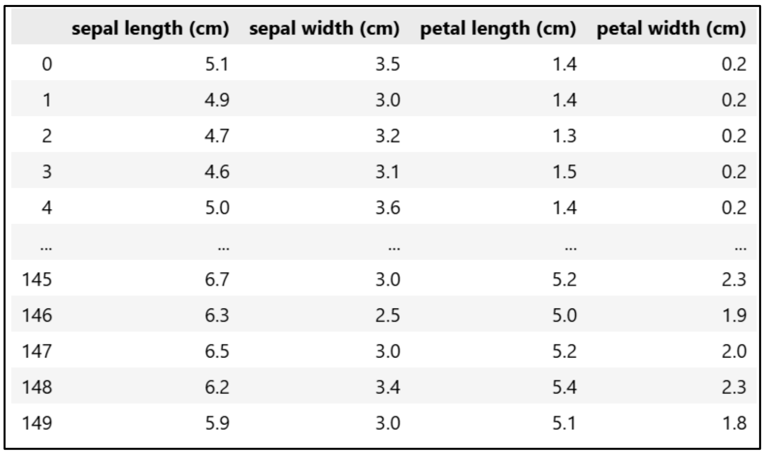
The Iris dataset is loaded from scikit-learn. The features (X) and target (y) are separated.

Code:

```
# Load the iris dataset

iris = load_iris()

X = iris.data

y = iris.target

# Create a DataFrame for better preview
```

data = pd.DataFrame(iris["data"], columns=iris["feature_names"])

Data Preview:



	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2
...	...	...	...	...
145	6.7	3.0	5.2	2.3
146	6.3	2.5	5.0	1.9
147	6.5	3.0	5.2	2.0
148	6.2	3.4	5.4	2.3
149	5.9	3.0	5.1	1.8

3. Data Pre-processing

Step 3.1: Normalizing the Data Feature scaling, or normalization, is an important step to ensure that all features contribute equally to the model's decision-making process. We use MinMaxScaler to scale the features to a range of [0, 1].

Code:

# Normalize the feature data

scaler = MinMaxScaler()

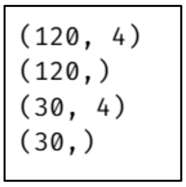
X_normalized = scaler.fit_transform(X)

Step 3.2: Splitting the Data The normalized dataset is split into a training set (80%) and a testing set (20%) to train and evaluate the model.

Code:

# Split data into training and testing sets

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



```
(120, 4)
(120,)
(30, 4)
(30,)
```

4. Model Building and Training

We create an instance of the DecisionTreeClassifier and train it on our training data.

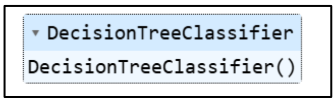
Code:

# Create a Decision Tree Classifier instance

clf = DecisionTreeClassifier()

# Fit the model to the training data

clf.fit(X_train, y_train)



```
▾ DecisionTreeClassifier
DecisionTreeClassifier()
```

5. Model Evaluation

Step 5.1: Making Predictions The trained model is used to predict the species for the test set features (X_test).

Code:

# Make predictions on the test set

y_pred = clf.predict(X_test)

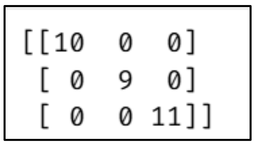
Step 5.2: Evaluating Model Performance, we evaluate the model's performance using a Confusion Matrix and a Classification Report.

Confusion Matrix: Code:

# Print the confusion matrix

print(confusion_matrix(y_test, y_pred))

Output:



```
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]
```

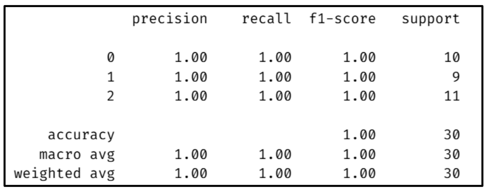
The confusion matrix shows that all 30 samples in the test set were classified correctly.

Classification Report:

# Print the classification report

print(classification_report(y_test, y_pred))

Output:



```
              precision    recall  f1-score   support

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

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

The report confirms the perfect score, with precision, recall, and f1-score all at 1.00 for each class.

6. Hyperparameter Tuning

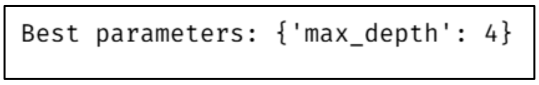
The performance of a decision tree can be influenced by its hyperparameters. A key hyperparameter is max_depth, which controls the maximum depth of the tree. A tree that is too deep can overfit the training data. We use GridSearchCV to find the optimal max_depth.

Grid Search for max_depth: This process systematically tests a range of max_depth values (from 1 to 5) using 5-fold cross-validation to find the one that yields the best performance.

Code:

```
# Define the parameter grid

params = {'max_depth': [1, 2, 3, 4, 5]}

# Create a new Decision Tree instance

clf = DecisionTreeClassifier()

# Set up GridSearchCV

grid_search = GridSearchCV(clf, param_grid=params, cv=5)

grid_search.fit(X_train, y_train)

# Print the best parameters found

print("Best parameters:", grid_search.best_params_)
```

Output:



```
Best parameters: {'max_depth': 4}
```

The result from GridSearchCV suggests that a max_depth of 4 is optimal for this dataset and training split, providing a good balance between model complexity and performance.

7. Conclusion

In this experiment, a Decision Tree Classifier was successfully implemented and trained to classify the Iris flower species. After normalizing the feature data, the model was trained and evaluated, achieving a perfect **accuracy of 100%** on the test set.

Furthermore, we performed hyperparameter tuning using GridSearchCV to find the optimal max_depth for the tree, which was determined to be 4. This demonstrates not only how to build a decision tree model but also how to optimize it for better generalization. The high accuracy confirms the effectiveness of the Decision Tree algorithm for this well-structured dataset.

Lab Experiment 9: Dimensionality Reduction using Principal Component Analysis

Objective: To perform dimensionality reduction on a high-dimensional dataset using the Principal Component Analysis (PCA) algorithm, retaining a specified percentage of the variance from the original data.

1. Introduction & Theory

Principal Component Analysis (PCA) is a widely used unsupervised machine learning algorithm for dimensionality reduction. In many real-world datasets, features are often highly correlated, leading to redundancy and increased model complexity (the "curse of dimensionality"). PCA addresses this by transforming the original, correlated features into a new set of linearly uncorrelated variables called principal components.

The principal components are ordered by the amount of variance they explain in the original data. The first principal component accounts for the largest possible variance, the second component accounts for the largest remaining variance, and so on. By selecting a subset of these principal components (e.g., the top 'k' components that explain 95% of the total variance), we can reduce the number of features in the dataset while retaining most of the important information.

This technique is valuable for data visualization, speeding up model training, and mitigating overfitting. In this experiment, we will apply PCA to a synthetic 10-dimensional dataset to reduce its dimensionality.

2. Procedure and Code Implementation

Step 2.1: Importing Necessary Libraries

We begin by importing the necessary libraries: pandas and numpy for data handling, and StandardScaler and PCA from scikit-learn for data pre-processing and dimensionality reduction.

Code:

```
import pandas as pd

import numpy as np

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

import matplotlib.pyplot as plt
```

Step 2.2: Generating and Loading the Dataset

A synthetic dataset with 1000 samples and 10 features is generated using numpy. A binary label is also added. The data is then saved to data.csv and loaded into a Pandas DataFrame.

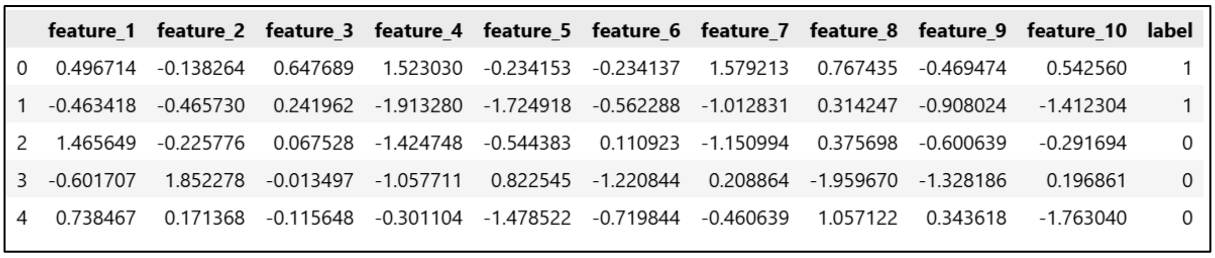
Code:

```
# Set seed for reproducibility

np.random.seed(42)

# Generate random data with 10 features

data = np.random.randn(1000, 10)

# Create DataFrame

df = pd.DataFrame(data, columns=[f'feature_{i}' for i in range(1, 11)])

# Add a label column

df['label'] = np.random.randint(0, 2, size=1000)

# Save and reload the DataFrame

df.to_csv('data.csv', index=False)

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

Data Preview:

Python

```
df.head()
```



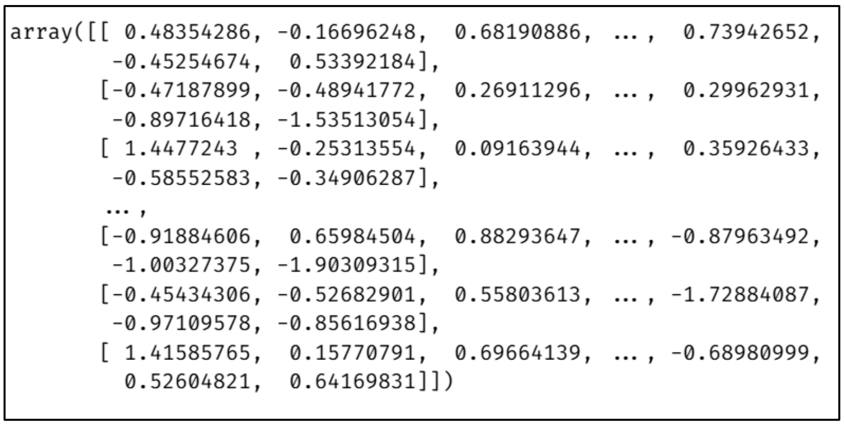
	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9	feature_10	label
0	0.496714	-0.138264	0.647689	1.523030	-0.234153	-0.234137	1.579213	0.767435	-0.469474	0.542560	1
1	-0.463418	-0.465730	0.241962	-1.913280	-1.724918	-0.562288	-1.012831	0.314247	-0.908024	-1.412304	1
2	1.465649	-0.225776	0.067528	-1.424748	-0.544383	0.110923	-1.150994	0.375698	-0.600639	-0.291694	0
3	-0.601707	1.852278	-0.013497	-1.057711	0.822545	-1.220844	0.208864	-1.959670	-1.328186	0.196861	0
4	0.738467	0.171368	-0.115648	-0.301104	-1.478522	-0.719844	-0.460639	1.057122	0.343618	-1.763040	0

3. Data Pre-processing and PCA Application

Step 3.1: Separating Features and Standardizing Data PCA is sensitive to the scale of the features. Therefore, it is crucial to standardize the data before applying PCA. We use StandardScaler to transform the features to have a mean of 0 and a standard deviation of 1.

Code:

```
# Separate features (X) and label (y)

X = df.drop('label', axis=1)

y = df['label']

# Standardize the features

scaler = StandardScaler()

X = scaler.fit_transform(X)
```



```
array([[ 0.48354286, -0.16696248,  0.68190886, ...,  0.73942652,
        -0.45254674,  0.53392184],
       [-0.47187899, -0.48941772,  0.26911296, ...,  0.29962931,
        -0.89716418, -1.53513054],
       [ 1.4477243 , -0.25313554,  0.09163944, ...,  0.35926433,
        -0.58552583, -0.34906287],
       ...,
       [-0.91884606,  0.65984504,  0.88293647, ..., -0.87963492,
        -1.00327375, -1.90309315],
       [-0.45434306, -0.52682901,  0.55803613, ..., -1.72884087,
        -0.97109578, -0.85616938],
       [ 1.41585765,  0.15770791,  0.69664139, ..., -0.68980999,
         0.52604821,  0.64169831]])
```

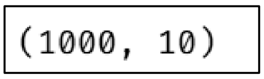
Step 3.2: Applying PCA We first apply PCA with all components to analyze the explained variance. This will help us decide how many components to retain.

Code:

# Apply PCA with all components

pca = PCA()

X_pca = pca.fit_transform(X)



```
(1000, 10)
```

4. Determining the Number of Principal Components

To decide how many principal components to keep, we analyze the **explained variance ratio**. This ratio tells us the proportion of the dataset's variance that lies along the axis of each principal component.

We can visualize this using a bar chart for individual variance and a step plot for the cumulative variance. Our goal is to select the smallest number of components that collectively explain a significant portion of the total variance (e.g., 85%).

Code:

# Get the explained variance ratio for each component

explained_variance = pca.explained_variance_ratio_

# Calculate the cumulative sum of explained variances

cum_var_exp = np.cumsum(explained_variance)

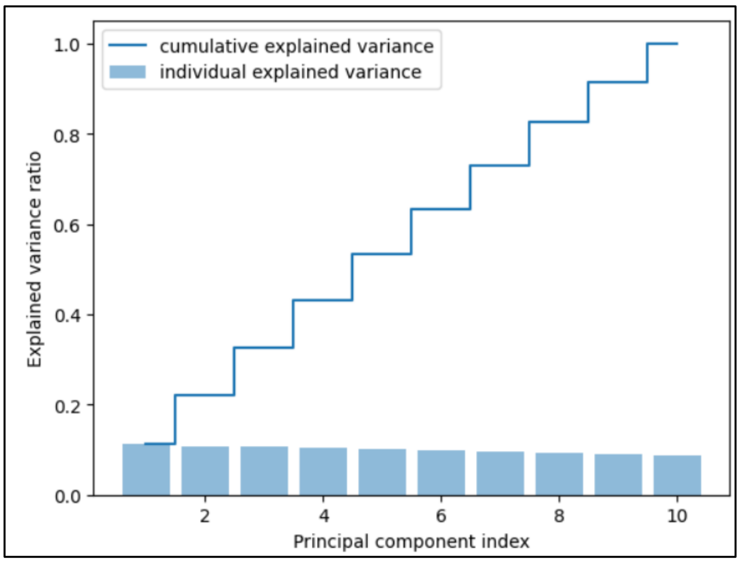
# Plot the explained variances

plt.bar(range(1, 11), explained_variance, alpha=0.5, align='center', label='Individual Explained Variance')

plt.step(range(1, 11), cum_var_exp, where='mid', label='Cumulative Explained Variance')

```
plt.ylabel('Explained Variance Ratio')

plt.xlabel('Principal Component Index')

plt.legend(loc='best')

plt.show()
```

Output:



Based on the plot, we determine the number of components needed to retain at least 85% of the variance.

Code:

```
# Find the number of components that explain at least 85% of the variance

n_components = np.argmax(np.cumsum(explained_variance) >= 0.85) + 1

print(f"Number of components to retain: {n_components}")
```

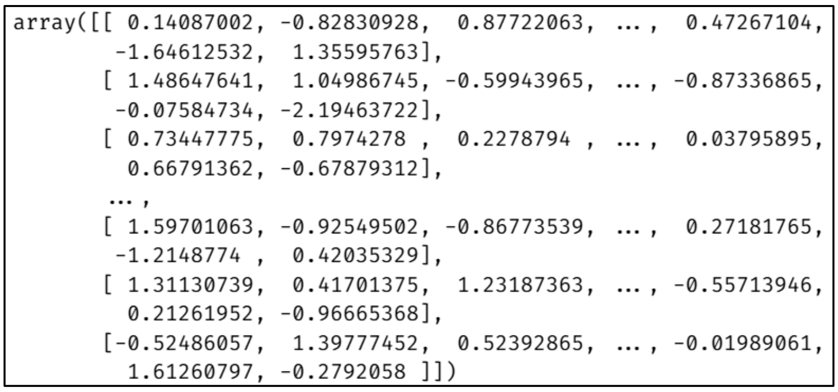
Output: Number of components to retain: 9

5. Transforming the Dataset

Now that we have determined the optimal number of components, we re-apply PCA with n_components=9 and transform our original 10-dimensional feature set into a new 9-dimensional one.

Step 5.1: Transforming the Features Code:

```
# Apply PCA with the determined number of components

pca = PCA(n_components=n_components)

X_pca = pca.fit_transform(X)
```



```
array([[ 0.14087002, -0.82830928,  0.87722063, ...,  0.47267104,
        -1.64612532,  1.35595763],
       [ 1.48647641,  1.04986745, -0.59943965, ..., -0.87336865,
        -0.07584734, -2.19463722],
       [ 0.73447775,  0.7974278 ,  0.2278794 , ...,  0.03795895,
         0.66791362, -0.67879312],
       ...,
       [ 1.59701063, -0.92549502, -0.86773539, ...,  0.27181765,
        -1.2148774 ,  0.42035329],
       [ 1.31130739,  0.41701375,  1.23187363, ..., -0.55713946,
         0.21261952, -0.96665368],
       [-0.52486057,  1.39777452,  0.52392865, ..., -0.01989061,
         1.61260797, -0.2792058 ]])
```

Step 5.2: Creating the New Dataset Finally, we create a new DataFrame that contains the transformed features (the 9 principal components) and the original labels.

Code:

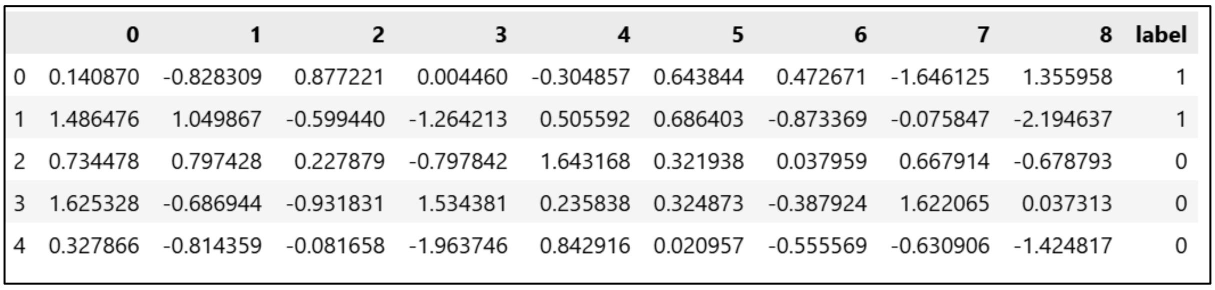
# Create a new DataFrame with the PCA-transformed features

new_df = pd.concat([pd.DataFrame(X_pca), y], axis=1)

# Preview the new dataset

new_df.head()

Output of the New Reduced-Dimension DataFrame:



	0	1	2	3	4	5	6	7	8	label
0	0.140870	-0.828309	0.877221	0.004460	-0.304857	0.643844	0.472671	-1.646125	1.355958	1
1	1.486476	1.049867	-0.599440	-1.264213	0.505592	0.686403	-0.873369	-0.075847	-2.194637	1
2	0.734478	0.797428	0.227879	-0.797842	1.643168	0.321938	0.037959	0.667914	-0.678793	0
3	1.625328	-0.686944	-0.931831	1.534381	0.235838	0.324873	-0.387924	1.622065	0.037313	0
4	0.327866	-0.814359	-0.081658	-1.963746	0.842916	0.020957	-0.555569	-0.630906	-1.424817	0

6. Conclusion

In this experiment, we successfully applied Principal Component Analysis to reduce the dimensionality of a 10-feature dataset. By analyzing the explained variance, we determined that 9 principal components were sufficient to capture at least 85% of the information from the original data.

The original dataset was then transformed into this new, lower-dimensional space. This reduced dataset can now be used for training machine learning models more efficiently, with less risk of overfitting and faster computation times, while still retaining the original variance.

Lab Experiment 10: K-Means Clustering

Objective: To apply the K-Means clustering algorithm to the Wine dataset to group similar wines into clusters based on their chemical analysis. The optimal number of clusters will be determined using the Elbow Method.

1. Introduction & Theory

Clustering is a type of unsupervised machine learning where the goal is to partition a dataset into groups (or "clusters"). Data points within the same cluster are more similar to each other than to those in other clusters. Unlike supervised learning, clustering algorithms work with unlabeled data.

K-Means is one of the most popular and straightforward clustering algorithms. It aims to partition *n* observations into *K* clusters, where each observation belongs to the cluster with the nearest mean (cluster centroid). The algorithm works as follows:

1. Choose the number of clusters (K): This is a predefined hyperparameter.
2. Initialize Centroids: Randomly select K data points from the dataset to serve as the initial cluster centroids.
3. Assign Clusters: Assign each data point to the cluster whose centroid is closest (usually determined by Euclidean distance).
4. Update Centroids: Recalculate the centroid of each cluster by taking the mean of all data points assigned to it.
5. Repeat: Repeat steps 3 and 4 until the cluster assignments no longer change or a maximum number of iterations is reached.

In this experiment, we will apply K-Means to the Wine dataset, which contains the results of a chemical analysis of wines derived from three different cultivars.

2. Procedure and Code Implementation

Step 2.1: Importing Necessary Libraries

We import the libraries needed for data handling, clustering, scaling, and visualization.

Code:

```
import pandas as pd

from sklearn.cluster import KMeans

from sklearn.datasets import load_wine

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler
```

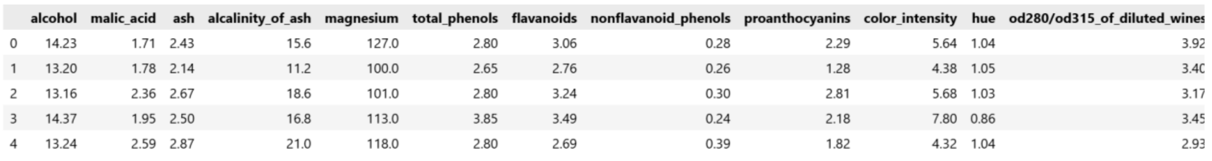
Step 2.2: Loading and Inspecting the Dataset

The Wine dataset is loaded from scikit-learn. It consists of 178 samples and 13 chemical features.

Code:

```
# Load the wine dataset

data = load_wine()

# Create a DataFrame for better preview

wine = pd.DataFrame(data.data, columns=data.feature_names)

# Display the first few rows and shape

print(wine.head())

print(wine.shape)
```

Data Preview:



	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	total_phenols	flavanoids	nonflavanoid_phenols	proanthocyanins	color_intensity	hue	od280/od315_of_diluted_wines
0	14.23	1.71	2.43	15.6	127.0	2.80	3.06	0.28	2.29	5.64	1.04	3.92
1	13.20	1.78	2.14	11.2	100.0	2.65	2.76	0.26	1.28	4.38	1.05	3.40
2	13.16	2.36	2.67	18.6	101.0	2.80	3.24	0.30	2.81	5.68	1.03	3.17
3	14.37	1.95	2.50	16.8	113.0	3.85	3.49	0.24	2.18	7.80	0.86	3.45
4	13.24	2.59	2.87	21.0	118.0	2.80	2.69	0.39	1.82	4.32	1.04	2.93

3. Data Pre-processing

Step 3.1: Feature Scaling The K-Means algorithm is sensitive to the scale of the features, as it is a distance-based algorithm. Features with larger scales can disproportionately influence the clustering outcome. Therefore, we standardize the data using StandardScaler to give all features a mean of 0 and a standard deviation of 1.

Code:

```
# For this experiment, we use all features for clustering

wine_data = wine

# Standardize the features

scaler = StandardScaler()

wine_data_scaled = scaler.fit_transform(wine_data)
```

4. Finding the Optimal Number of Clusters (K) with the Elbow Method

A key challenge in K-Means is choosing the right value for K. The Elbow Method is a common technique to determine this. It works by running the K-Means algorithm for a range of K values and calculating the Within-Cluster Sum of Squares (WSS) for each. WSS, also known as inertia, is the sum of squared distances between each data point and its cluster's centroid.

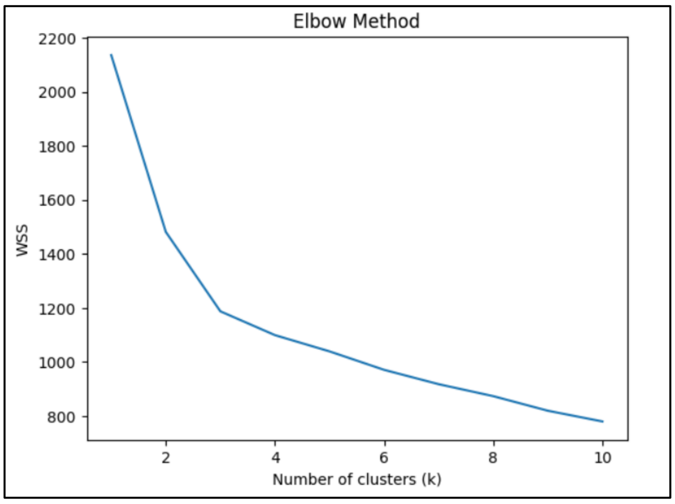
We plot the WSS for each K. The "elbow" of the curve—the point where the rate of decrease in WSS sharply slows down—suggests the optimal number of clusters.

Code:

```
# Fit K-means clustering with different values of k
```

```
wss_values = []
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k, random_state=42, n_init='auto')
    kmeans.fit(wine_data_scaled)
    wss_values.append(kmeans.inertia_)
# Plot the WSS values against different values of k
plt.plot(range(1, 11), wss_values)
plt.title('Elbow Method for Optimal K')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Within-Cluster Sum of Squares (WSS)')
plt.show()
```

Output:



From the plot, we can observe a distinct "elbow" at **K=3**. After this point, the decrease in WSS becomes much less significant. This indicates that 3 is the optimal number of clusters for this dataset, which aligns with the fact that the original dataset has three classes of wine.

5. Applying K-Means and Visualizing Clusters

Now that we have determined the optimal K, we run the K-Means algorithm with n_clusters=3.

Step 5.1: Fitting the K-Means Model Code:

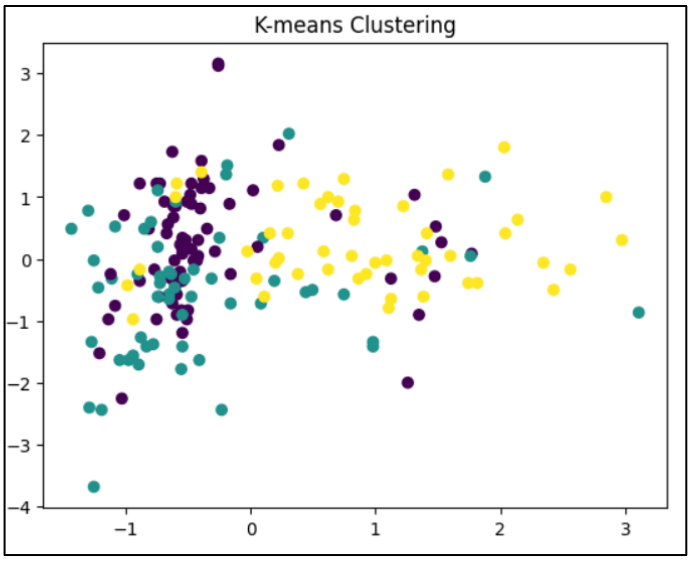
```
# Apply K-means with the optimal K=3
kmeans = KMeans(n_clusters=3, random_state=42, n_init='auto')
kmeans.fit(wine_data_scaled)
```

Step 5.2: Visualizing the Clusters Since we have 13 features, we cannot visualize the clusters directly in their original space. To create a 2D plot, we will visualize the clusters using the first two features of the scaled dataset (Malic Acid vs. Ash).

Code:

```
# Scatter plot of the first two features, colored by cluster label

plt.scatter(wine_data_scaled[:, 0], wine_data_scaled[:, 1], c=kmeans.labels_)

plt.title('K-Means Clustering of Wine Dataset (K=3)')

plt.xlabel(data.feature_names[0])

plt.ylabel(data.feature_names[1])

plt.show()
```

Output:



The scatter plot shows the 178 data points grouped into three distinct clusters, each represented by a different color.

6. Conclusion

In this experiment, the K-Means clustering algorithm was successfully applied to the Wine dataset. After scaling the features, the Elbow Method was used to identify the optimal number of clusters, which was found to be **3**.

The K-Means model was then trained with K=3, and the resulting clusters were visualized. The algorithm effectively partitioned the data into three groups based on their chemical properties, demonstrating its utility as an unsupervised learning technique for discovering underlying structures in a dataset. This experiment highlights the full process of applying K-Means, from data preparation and hyperparameter tuning to final model fitting and evaluation.