NOTRE DAME UNIVERSITY BANGLADESH

Machine Learning Lab Final Report

Course Code: CSE4214

Course Title: Machine Learning Lab

All Lab Reports have been merged

Submitted by:

Name: Istiak Alam

ID: 0692230005101005

Batch: CSE-20

Submission Date: June 7, 2026

Submitted to:

A. H. M. Saiful Islam

Chairman, Department of CSE

Notre Dame University Bangladesh

Table of Contents

Abstract

Machine Learning has become an essential component in modern data-driven systems, enabling automated decision-making and predictive analytics. This lab report presents a comprehensive overview of fundamental machine learning techniques implemented across seven laboratory sessions. The experiments cover key areas including data preprocessing, regression analysis, classification algorithms, encoding techniques, probabilistic modeling, and model evaluation.

Throughout the labs, various algorithms such as Linear Regression, K-Nearest Neighbors (KNN), Naive Bayes, Logistic Regression, Support Vector Machine (SVM), and Decision Trees were implemented and analyzed. Additionally, techniques like label encoding, one-hot encoding, and confusion matrix evaluation were explored to enhance model performance and interpretability.

The objective of this report is to demonstrate practical understanding of machine learning concepts by applying them to real-world datasets and evaluating their effectiveness through systematic experimentation.

Objective

The primary objective of this lab report is to develop a strong practical foundation in machine learning by implementing and analyzing various algorithms and techniques. The specific goals include:

- To understand and perform data preprocessing for preparing datasets.
- To implement Linear and Multiple Linear Regression for predictive modeling.
- To apply K-Nearest Neighbors (KNN) for both classification and regression tasks.
- To learn and apply encoding techniques such as label encoding and one-hot encoding.
- To implement probabilistic models using the Naive Bayes algorithm.
- To understand classification techniques using Logistic Regression and Support Vector Machines (SVM).
- To implement Decision Trees and evaluate model performance using Confusion Matrix.
- To compare different machine learning models based on accuracy and prediction performance.

Introduction

Machine Learning is a subfield of artificial intelligence that focuses on building systems capable of learning from data and improving their performance without explicit programming. It plays a vital role in various applications such as prediction, classification, recommendation systems, and pattern recognition.

In this lab course, several fundamental machine learning techniques were explored through practical implementation. The process began with data preprocessing, which involves cleaning, transforming, and preparing raw data for analysis. Proper preprocessing ensures that models perform efficiently and produce reliable results.

Following this, regression techniques such as Linear Regression and Multiple Linear Regression were implemented to understand relationships between variables and perform prediction tasks. Classification techniques including K-Nearest Neighbors, Logistic Regression, Support Vector Machine, and Decision Trees were studied to categorize data into distinct classes.

Additionally, encoding methods like label encoding and one-hot encoding were applied to handle categorical data effectively. The Naive Bayes algorithm introduced probabilistic classification concepts, while the confusion matrix provided a structured approach to evaluating model performance.

Overall, this report reflects a systematic progression from basic data handling to advanced machine learning model implementation and evaluation.

Lab 1 Data Preprocessing

Objective

The objective of this experiment is to familiarize with basic data handling and preprocessing techniques required before applying Machine Learning algorithms. This lab focuses on loading a dataset, inspecting its structure, handling categorical variables, and converting them into numerical representations suitable for model training.

Dataset Description

The dataset used in this experiment is `bank.csv`, which contains customer information related to a banking institution. The dataset includes demographic details, financial attributes, and campaign-related information. The target variable indicates whether a customer subscribed to a term deposit. The main objective is to load a CSV dataset into a Pandas DataFrame and inspect the initial records.

1.1 Importing Required Libraries

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Explanation

Machine Learning workflows rely on multiple Python libraries, each serving a specific purpose. NumPy provides efficient data structures and mathematical functions for numerical computation. Pandas enables loading, cleaning, and manipulating structured datasets. Matplotlib is used to generate basic plots and graphs, while Seaborn builds on Matplotlib to produce more informative and visually appealing statistical visualizations. Importing these libraries prepares the environment for data analysis and model development.

Output

No visible output is produced if the libraries are successfully imported. If any library is missing from the environment, an error message such as `ModuleNotFoundError` is displayed.

1.2 Loading CSV Data

[2]:
```
import pandas as pd

df = pd.read_csv("bank.csv")
df.head()
```

Explanation

Pandas provides the `read_csv()` function to load structured data into a DataFrame. Initially, the dataset was loaded without specifying the delimiter, which resulted in improper column parsing. This step helps identify formatting issues in raw data.

Output

The output displays the first five records of the dataset, revealing incorrect column separation due to the delimiter mismatch.

[2]:
```
age;"job";"marital";"education";"default";"balance";"housing";"loan";
"contact";"day";"month";"duration";"campaign";"pdays";"previous";"poutcome";
"y"
0  30;"unemployed";"married";"primary";"no";1787;…
1  33;"services";"married";"secondary";"no";4789;…
2  35;"management";"single";"tertiary";"no";1350;…
3  30;"management";"married";"tertiary";"no";1476…
4  59;"blue-collar";"married";"secondary";"no";0;…
```

1.3 Loading the Dataset and Displaying Records

[3]:
```
df = pd.read_csv("bank.csv", sep = ';')
df.head()
```

Explanation

The dataset `bank.csv` uses a semicolon (`;`) as its field separator instead of the default comma. Therefore, the delimiter is explicitly specified while loading the file. The `read_csv()` function stores the data in a Pandas DataFrame, which is a tabular data structure. The `head()` function is then used to display the first five records of the dataset, allowing quick verification of data correctness and column structure.

Output

The output displays the first five rows of the dataset along with all column names, confirming that the data has been loaded correctly into the DataFrame.

[3]:

	age	job	marital	education	default	balance	housing	loan	\
0	30	unemployed	married	primary	no	1787	no	no	
1	33	services	married	secondary	no	4789	yes	yes	
2	35	management	single	tertiary	no	1350	yes	no	
3	30	management	married	tertiary	no	1476	yes	yes	
4	59	blue-collar	married	secondary	no	0	yes	no	

	contact	day	month	duration	campaign	pdays	previous	poutcome	y
0	cellular	19	oct	79	1	-1	0	unknown	no
1	cellular	11	may	220	1	339	4	failure	no
2	cellular	16	apr	185	1	330	1	failure	no
3	unknown	3	jun	199	4	-1	0	unknown	no
4	unknown	5	may	226	1	-1	0	unknown	no

1.4 Displaying the Last Records of the Dataset

[4]:
```
df.tail()
```

Explanation

The `tail()` function in Pandas is used to display the last few entries of a DataFrame. By default, it returns the final five rows. This operation is useful for verifying that the dataset has been loaded completely and for checking any anomalies or missing values at the end of the dataset.

Output

The output displays the last five rows of the dataset along with all corresponding column values.

[4]:

	age	job	marital	education	default	balance	housing	loan	\
4516	33	services	married	secondary	no	-333	yes	no	
4517	57	self-employed	married	tertiary	yes	-3313	yes	yes	
4518	57	technician	married	secondary	no	295	no	no	
4519	28	blue-collar	married	secondary	no	1137	no	no	
4520	44	entrepreneur	single	tertiary	no	1136	yes	yes	

	contact	day	month	duration	campaign	pdays	previous	poutcome	y
4516	cellular	30	jul	329	5	-1	0	unknown	no
4517	unknown	9	may	153	1	-1	0	unknown	no
4518	cellular	19	aug	151	11	-1	0	unknown	no
4519	cellular	6	feb	129	4	211	3	other	no
4520	cellular	3	apr	345	2	249	7	other	no

1.5 Encoding Marital Status Attribute

[5]:
```
def replace_marital(val):
  if val == 'single':
    return 0
  else:
    return 1

df['marital'] = df['marital'].apply(replace_marital)
df.head()
```

Explanation

In this step, the categorical attribute `marital` is converted into a numerical format to make it suitable for Machine Learning algorithms. A user-defined function named `replace_marital` is created, which assigns the value 0 to customers with marital status `single` and the value 1 to all other categories. The `apply()` function is then used to apply this transformation to the entire `marital` column of the dataset. This process simplifies categorical data while preserving relevant information.

Output

The output displays the first five rows of the dataset where the `marital` column is successfully transformed into numerical values. Customers with marital status `single` are represented by 0, while all other marital statuses are represented by 1.

[5]:

	age	job	marital	education	default	balance	housing	loan	\
0	30	unemployed	1	primary	no	1787	no	no	
1	33	services	1	secondary	no	4789	yes	yes	
2	35	management	0	tertiary	no	1350	yes	no	
3	30	management	1	tertiary	no	1476	yes	yes	
4	59	blue-collar	1	secondary	no	0	yes	no	

	contact	day	month	duration	campaign	pdays	previous	poutcome	y
0	cellular	19	oct	79	1	-1	0	unknown	no
1	cellular	11	may	220	1	339	4	failure	no
2	cellular	16	apr	185	1	330	1	failure	no
3	unknown	3	jun	199	4	-1	0	unknown	no
4	unknown	5	may	226	1	-1	0	unknown	no

1.6 Encoding Housing Loan Attribute

[6]:
```
df["housing"] = df["housing"].map({"yes": 1, "no": 0}.get)
df.head()
```

Explanation

In this step, the categorical attribute `housing` is converted into a numerical format using the `map()` function. The values `yes` and `no` are mapped to 1 and 0 respectively. This conversion is necessary because Machine Learning algorithms require numerical input features. The `get` method ensures safe mapping of values within the column.

Output

The output displays the first five rows of the dataset where the `housing` column has been successfully transformed into numerical values. A value of 1 indicates the presence of a housing loan, while 0 indicates the absence of a housing loan.

[6]:

	age	job	marital	education	default	balance	housing	loan	\
0	30	unemployed	1	primary	no	1787	0	no	
1	33	services	1	secondary	no	4789	1	yes	
2	35	management	0	tertiary	no	1350	1	no	
3	30	management	1	tertiary	no	1476	1	yes	

```
4   59  blue-collar        1  secondary      no        0        1   no

    contact  day month  duration  campaign  pdays  previous poutcome   y
0  cellular   19   oct        79         1     -1         0  unknown  no
1  cellular   11   may       220         1    339         4  failure  no
2  cellular   16   apr       185         1    330         1  failure  no
3   unknown    3   jun       199         4     -1         0  unknown  no
4   unknown    5   may       226         1     -1         0  unknown  no
```

1.7 Encoding Loan Attribute

[7]:
```
df["loan"] = df["loan"].replace({"yes": 1, "no": 0})
df.head()
```

Explanation

In this step, the categorical attribute `loan` is converted into numerical form to ensure compatibility with Machine Learning algorithms. The `replace()` function is used to map the value `yes` to 1 and `no` to 0. This binary encoding simplifies the data representation while retaining the original meaning of the attribute.

Output

The output displays the first five rows of the dataset, where the `loan` column is successfully transformed into numerical values. A value of 1 indicates the presence of a personal loan, while 0 indicates the absence of a loan.

[7]:
```
   age          job  marital  education default  balance  housing  loan  \
0   30   unemployed        1    primary      no     1787        0     0
1   33     services        1  secondary      no     4789        1     1
2   35   management        0   tertiary      no     1350        1     0
3   30   management        1   tertiary      no     1476        1     1
4   59  blue-collar        1  secondary      no        0        1     0

    contact  day month  duration  campaign  pdays  previous poutcome   y
0  cellular   19   oct        79         1     -1         0  unknown  no
1  cellular   11   may       220         1    339         4  failure  no
2  cellular   16   apr       185         1    330         1  failure  no
3   unknown    3   jun       199         4     -1         0  unknown  no
4   unknown    5   may       226         1     -1         0  unknown  no
```

1.8 Checking Unique Job Categories

[8]:
```
df["job"].unique()
```

Explanation

The `unique()` function is used to identify all distinct values present in the `job` column of the dataset. This helps in understanding the different job categories of the customers and is useful for preprocessing and encoding categorical variables before applying Machine Learning algorithms.

Output

The output displays an array of unique job categories present in the dataset, such as `unemployed`, `services`, `management`, `blue-collar`, `self-employed`, `technician`, `entrepreneur`, `admin.`, `student`, `housemaid`, `retired`, and `unknown`.

[8]:
```
array(['unemployed', 'services', 'management', 'blue-collar',
       'self-employed', 'technician', 'entrepreneur', 'admin.', 'student',
       'housemaid', 'retired', 'unknown'], dtype=object)
```

1.9 Encoding Job Attribute

[9]:
```
df["job"] = df["job"].replace({'unemployed': 0,
                    'services': 0,
                    'management': 1,
                    'blue-collar': 0,
                    'self-employed': 0,
                    'technician': 1,
                    'entrepreneur': 1,
                    'admin.': 0,
                    'student': 1,
                    'housemaid': 0,
                    'retired': 0,
                    'unknown': np.nan})
df.head()
```

Explanation

The categorical attribute `job` is transformed into a numerical format to make it suitable for Machine Learning algorithms. The `replace()` function is used to map specific job categories to binary values: jobs considered as professional or managerial (e.g., `management`, `technician`, `entrepreneur`, `student`) are assigned 1, while other jobs (e.g., `unemployed`, `services`, `blue-collar`, `self-employed`, `admin.`, `housemaid`, `retired`) are assigned 0. Any unknown job entries are replaced with `NaN`. This encoding simplifies categorical data for model training.

Output

The output displays the first five rows of the dataset with the `job` column converted to numerical values. Job categories are represented as 0 or 1, and any unknown values are represented as `NaN`.

[9]:

	age	job	marital	education	default	balance	housing	loan	contact	\
0	30	0.0	1	1.0	no	1787	0	0	cellular	
1	33	0.0	1	2.0	no	4789	1	1	cellular	
2	35	1.0	0	3.0	no	1350	1	0	cellular	
3	30	1.0	1	3.0	no	1476	1	1	unknown	
4	59	0.0	1	2.0	no	0	1	0	unknown	

	day	month	duration	campaign	pdays	previous	poutcome	y
0	19	10	79	1	-1	0	unknown	no
1	11	5	220	1	339	4	failure	no

```
2   16      4       185         1    330         1  failure  no
3    3      6       199         4     -1         0  unknown  no
4    5      5       226         1     -1         0  unknown  no
```

1.10 Checking Unique Values of Month Attribute

[10]:
```
df["month"].unique()
```

Explanation

The `unique()` function is used to identify all distinct values present in the `month` column of the dataset. This operation helps to understand the range of months represented in the data and is useful for preprocessing, such as converting categorical month names into numerical values for Machine Learning algorithms.

Output

The output displays an array of unique month names in the dataset. For example:

[10]:
```
array(['oct', 'may', 'apr', 'jun', 'feb', 'aug', 'jan', 'jul', 'nov',
       'sep', 'mar', 'dec'], dtype=object)
```

1.11 Encoding Month Attribute

[11]:
```
df.month = df.month.map({
    'oct': 10,
    'may': 5,
    'apr': 4,
    'jun': 6,
    'feb': 2,
    'aug': 8,
    'jan': 1,
    'jul': 7,
    'nov': 11,
    'sep': 9,
    'mar': 3,
    'dec': 12
})
df.head(10)
```

Explanation

The `month` column contains the names of months as categorical string values. To convert this categorical data into a numerical format suitable for Machine Learning models, a mapping is applied where each month name is replaced with its corresponding integer value (e.g., `'jan'` = 1, `'feb'` = 2, ..., `'dec'` = 12). The `map()` function is used to transform the entire `month` column according to this mapping. This preprocessing step ensures that the month attribute can be interpreted quantitatively by algorithms.

Output

The output displays the first ten rows of the dataset after transformation. The `month` column now contains integer values representing the months, while all other columns remain unchanged.

[11]:

	age	job	marital	education	default	balance	housing	loan	contact	\
0	30	0.0	1	1.0	no	1787	0	0	cellular	
1	33	0.0	1	2.0	no	4789	1	1	cellular	
2	35	1.0	0	3.0	no	1350	1	0	cellular	
3	30	1.0	1	3.0	no	1476	1	1	unknown	
4	59	0.0	1	2.0	no	0	1	0	unknown	
5	35	1.0	0	3.0	no	747	0	0	cellular	
6	36	0.0	1	3.0	no	307	1	0	cellular	
7	39	1.0	1	2.0	no	147	1	0	cellular	
8	41	1.0	1	3.0	no	221	1	0	unknown	
9	43	0.0	1	1.0	no	-88	1	1	cellular	

	day	month	duration	campaign	pdays	previous	poutcome	y
0	19	NaN	79	1	-1	0	unknown	no
1	11	NaN	220	1	339	4	failure	no
2	16	NaN	185	1	330	1	failure	no
3	3	NaN	199	4	-1	0	unknown	no
4	5	NaN	226	1	-1	0	unknown	no
5	23	NaN	141	2	176	3	failure	no
6	14	NaN	341	1	330	2	other	no
7	6	NaN	151	2	-1	0	unknown	no
8	14	NaN	57	2	-1	0	unknown	no
9	17	NaN	313	1	147	2	failure	no

1.12 Inspecting Unique Values of Education Attribute

[12]:
```
df["education"].unique()
```

Explanation

The `unique()` function is used on the `education` column to identify all distinct categories present in the dataset. This step helps in understanding the range of values for the attribute and is useful before encoding categorical variables into numerical format for Machine Learning models.

Output

The output is an array of unique values in the `education` column. For example: `['primary', 'secondary', 'tertiary', 'unknown']`, which shows all education levels present in the dataset.

[12]:
```
array(['primary', 'secondary', 'tertiary', 'unknown'], dtype=object)
```

1.13 Encoding Education Attribute

[13]:
```
df.education = df.education.map({
    'primary': 1,
    'secondary': 2,
    'tertiary': 3,
    'unknown': np.nan
})
df.head(10)
```

Explanation

The `education` column contains categorical values representing the education level of customers. To prepare the data for Machine Learning models, these categorical values are converted to numerical values using the `map()` function. The mapping is as follows: `'primary'` = 1, `'secondary'` = 2, `'tertiary'` = 3, and `'unknown'` is replaced with `NaN` to indicate missing data. This transformation allows algorithms to process the education levels quantitatively.

Output

The output displays the first ten rows of the dataset after the transformation. The `education` column now contains numerical values (1, 2, 3) corresponding to education levels, with `NaN` for unknown entries.

[13]:

	age	job	marital	education	default	balance	housing	loan	contact	\
0	30	0.0	1	1.0	no	1787	0	0	cellular	
1	33	0.0	1	2.0	no	4789	1	1	cellular	
2	35	1.0	0	3.0	no	1350	1	0	cellular	
3	30	1.0	1	3.0	no	1476	1	1	unknown	
4	59	0.0	1	2.0	no	0	1	0	unknown	
5	35	1.0	0	3.0	no	747	0	0	cellular	
6	36	0.0	1	3.0	no	307	1	0	cellular	
7	39	1.0	1	2.0	no	147	1	0	cellular	
8	41	1.0	1	3.0	no	221	1	0	unknown	
9	43	0.0	1	1.0	no	-88	1	1	cellular	

	day	month	duration	campaign	pdays	previous	poutcome	y
0	19	10	79	1	-1	0	unknown	no
1	11	5	220	1	339	4	failure	no
2	16	4	185	1	330	1	failure	no
3	3	6	199	4	-1	0	unknown	no
4	5	5	226	1	-1	0	unknown	no
5	23	2	141	2	176	3	failure	no
6	14	5	341	1	330	2	other	no
7	6	5	151	2	-1	0	unknown	no
8	14	5	57	2	-1	0	unknown	no
9	17	4	313	1	147	2	failure	no

1.14 Unique Values of the Outcome of Previous Marketing Campaign (poutcome)

[14]:
```
df["poutcome"].unique()
```

Explanation

The `unique()` function is used to identify all distinct values present in the `poutcome` column of the dataset. This column represents the result of the previous marketing campaign for each customer. Determining unique values helps in understanding the different categories present and assists in further preprocessing or encoding steps required for Machine Learning.

Output

The output is an array of all unique values in the `poutcome` column, for example:

`array(['unknown', 'failure', 'other', 'success'], dtype=object)`

This shows that the column contains four distinct categories indicating the previous campaign outcome.

[14]:
```
array(['unknown', 'failure', 'other', 'success'], dtype=object)
```

1.15 Encoding Poutcome Attribute

[15]:
```
df.poutcome = df.poutcome.map({
    'unknown': np.nan,
    'failure': 1,
    'other': 2,
    'success': 3
})
df.head(10)
```

Explanation

The `poutcome` column represents the outcome of the previous marketing campaign and is a categorical variable. To make it compatible with Machine Learning algorithms, it is mapped to numerical values using the `map()` function. The mapping assigns `failure` to 1, `other` to 2, `success` to 3, and replaces `unknown` values with `NaN` to handle missing information. This encoding simplifies the dataset while preserving meaningful distinctions between campaign outcomes.

Output

The output displays the first ten rows of the dataset with the `poutcome` column transformed into numerical values. Entries that were `unknown` are now shown as `NaN`, while other outcomes are represented by 1, 2, or 3 accordingly.

[15]:

	age	job	marital	education	default	balance	housing	loan	contact	\
0	30	0.0	1	1.0	no	1787	0	0	cellular	
1	33	0.0	1	2.0	no	4789	1	1	cellular	
2	35	1.0	0	3.0	no	1350	1	0	cellular	

```
3   30  1.0        1        3.0      no     1476        1     1   unknown
4   59  0.0        1        2.0      no        0        1     0   unknown
5   35  1.0        0        3.0      no      747        0     0  cellular
6   36  0.0        1        3.0      no      307        1     0  cellular
7   39  1.0        1        2.0      no      147        1     0  cellular
8   41  1.0        1        3.0      no      221        1     0   unknown
9   43  0.0        1        1.0      no      -88        1     1  cellular

   day  month  duration  campaign  pdays  previous  poutcome   y
0   19    NaN        79         1     -1         0       NaN  no
1   11    NaN       220         1    339         4       1.0  no
2   16    NaN       185         1    330         1       1.0  no
3    3    NaN       199         4     -1         0       NaN  no
4    5    NaN       226         1     -1         0       NaN  no
5   23    NaN       141         2    176         3       1.0  no
6   14    NaN       341         1    330         2       2.0  no
7    6    NaN       151         2     -1         0       NaN  no
8   14    NaN        57         2     -1         0       NaN  no
9   17    NaN       313         1    147         2       1.0  no
```

1.16 Normalizing the Balance Attribute

[16]:
```
df["balance"] = df["balance"].apply(lambda v: (v - df["balance"].min())/
  ↪(df["balance"].max() - df["balance"].min()))
df.head(10)
```

Explanation

The `balance` column is a numerical feature representing the account balance of customers. To scale this feature between 0 and 1, min-max normalization is applied. The formula used is:

$$\text{normalized\_value} = \frac{v - \min(\text{balance})}{\max(\text{balance}) - \min(\text{balance})}$$

where v is the original balance value. This transformation ensures that all values of `balance` lie within the range [0, 1], improving the performance and convergence of many Machine Learning algorithms.

Output

The output displays the first ten rows of the dataset after normalization. The `balance` column values are now scaled between 0 and 1 while all other columns remain unchanged.

[16]:
```
   age  job  marital  education default   balance  housing  loan   contact  \
0   30  0.0        1        1.0      no  0.068455        0     0  cellular
1   33  0.0        1        2.0      no  0.108750        1     1  cellular
2   35  1.0        0        3.0      no  0.062590        1     0  cellular
3   30  1.0        1        3.0      no  0.064281        1     1   unknown
4   59  0.0        1        2.0      no  0.044469        1     0   unknown
5   35  1.0        0        3.0      no  0.054496        0     0  cellular
6   36  0.0        1        3.0      no  0.048590        1     0  cellular
7   39  1.0        1        2.0      no  0.046442        1     0  cellular
```

```
8   41  1.0        1        3.0      no  0.047436        1     0   unknown
9   43  0.0        1        1.0      no  0.043288        1     1  cellular

   day  month  duration  campaign  pdays  previous  poutcome   y
0   19    NaN        79         1     -1         0       NaN  no
1   11    NaN       220         1    339         4       1.0  no
2   16    NaN       185         1    330         1       1.0  no
3    3    NaN       199         4     -1         0       NaN  no
4    5    NaN       226         1     -1         0       NaN  no
5   23    NaN       141         2    176         3       1.0  no
6   14    NaN       341         1    330         2       2.0  no
7    6    NaN       151         2     -1         0       NaN  no
8   14    NaN        57         2     -1         0       NaN  no
9   17    NaN       313         1    147         2       1.0  no
```

1.17 Normalization of pdays Attribute

[17]:
```
df["pdays"] = df["pdays"].apply(lambda v: (v - df["pdays"].min())/
  ↪(df["pdays"].max() - df["pdays"].min()))
df.head(10)
```

Explanation

The pdays column, which represents the number of days since a client was last contacted, is normalized using the Min-Max scaling technique. This transformation scales all values to a range between 0 and 1, which helps improve the performance and convergence of Machine Learning algorithms.

Output

The first ten rows of the dataset are displayed. The pdays column now contains values scaled between 0 and 1, while the other columns remain unchanged.

[17]:
```
   age  job  marital  education default   balance  housing  loan   contact  \
0   30  0.0        1        1.0      no  0.068455        0     0  cellular
1   33  0.0        1        2.0      no  0.108750        1     1  cellular
2   35  1.0        0        3.0      no  0.062590        1     0  cellular
3   30  1.0        1        3.0      no  0.064281        1     1   unknown
4   59  0.0        1        2.0      no  0.044469        1     0   unknown
5   35  1.0        0        3.0      no  0.054496        0     0  cellular
6   36  0.0        1        3.0      no  0.048590        1     0  cellular
7   39  1.0        1        2.0      no  0.046442        1     0  cellular
8   41  1.0        1        3.0      no  0.047436        1     0   unknown
9   43  0.0        1        1.0      no  0.043288        1     1  cellular

   day  month  duration  campaign     pdays  previous  poutcome   y
0   19    NaN        79         1  0.000000         0       NaN  no
1   11    NaN       220         1  0.389908         4       1.0  no
2   16    NaN       185         1  0.379587         1       1.0  no
3    3    NaN       199         4  0.000000         0       NaN  no
```

```
4    5    NaN       226         1  0.000000         0       NaN  no
5   23    NaN       141         2  0.202982         3       1.0  no
6   14    NaN       341         1  0.379587         2       2.0  no
7    6    NaN       151         2  0.000000         0       NaN  no
8   14    NaN        57         2  0.000000         0       NaN  no
9   17    NaN       313         1  0.169725         2       1.0  no
```

1.18 Feature Scaling using Min-Max Scaler

[18]:
```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
df["duration"] = scaler.fit_transform(df[["duration"]])
df["pdays"] = scaler.fit_transform(df[["pdays"]])
df.head()
```

Explanation

Feature scaling is performed to normalize the numerical attributes `duration` and `pdays` into a fixed range, usually between 0 and 1, which helps in faster convergence and better performance of Machine Learning algorithms. The `MinMaxScaler` from `sklearn.preprocessing` is used. The `fit_transform()` method calculates the minimum and maximum values of each feature and scales all values accordingly. This ensures that the features contribute proportionally to the model training.

Output

The output displays the first five rows of the dataset where the `duration` and `pdays` columns have been scaled to values between 0 and 1, while all other columns remain unchanged.

[18]:
```
   age  job  marital  education default   balance  housing  loan   contact  \
0   30  0.0        1        1.0      no  0.068455        0     0  cellular
1   33  0.0        1        2.0      no  0.108750        1     1  cellular
2   35  1.0        0        3.0      no  0.062590        1     0  cellular
3   30  1.0        1        3.0      no  0.064281        1     1   unknown
4   59  0.0        1        2.0      no  0.044469        1     0   unknown

   day  month  duration  campaign     pdays  previous  poutcome   y
0   19    NaN  0.024826         1  0.000000         0       NaN  no
1   11    NaN  0.071500         1  0.389908         4       1.0  no
2   16    NaN  0.059914         1  0.379587         1       1.0  no
3    3    NaN  0.064548         4  0.000000         0       NaN  no
4    5    NaN  0.073486         1  0.000000         0       NaN  no
```

Lab 2 Linear Regression

Objective

The objective of this Machine Learning lab is to understand and implement regression techniques using real-world datasets. This lab focuses on applying Simple Linear Regression and Multiple Linear Regression to analyze the relationship between independent variables and a target variable. The experiments aim to develop predictive models, interpret feature influence, and evaluate model performance using appropriate regression metrics.

Dataset Description

Three different datasets are used in this lab to demonstrate the application of regression algorithms.

Dhaka Home Prices Dataset

The `dhaka homeprices.csv` dataset is used for implementing Simple Linear Regression. This dataset contains housing-related information from Dhaka city, where a single independent feature is used to predict house prices. The dataset helps illustrate how a dependent variable varies linearly with one explanatory variable.

Insurance Dataset

The `insurance.csv` dataset is used for implementing Multiple Linear Regression. It contains multiple attributes related to insurance policyholders. The dataset is used to analyze how multiple independent variables collectively influence insurance-related outcomes.

Car Data Dataset

The `cardata.csv` dataset is also used for Multiple Linear Regression. It includes attributes such as `speed`, `car_age`, `experience`, and `risk`. These features are used to predict a target variable by learning the combined effect of multiple factors on the outcome.

2.1 Linear Regression Implementation

2.1.1 Loading Dataset for Linear Regression

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

[2]:
```
df= pd.read_csv('dhaka homeprices.csv')
#df = pd.read_csv ('Shopping_cse15_16.csv')
```

[3]:
```
df
```

Explanation

In this step, the required Python libraries are imported to support data handling, visualization, and model preparation. The Pandas library is used to load the dataset `dhaka homeprices.csv` into a DataFrame, which provides a structured tabular representation of the data. This dataset is intended for implementing Simple Linear Regression. Loading the dataset allows for initial inspection and verification of the data before applying further preprocessing and model training.

Output

The output displays the complete contents of the `dhaka homeprices.csv` dataset in tabular form, showing all rows and columns stored in the DataFrame.

[3]:

	area	price
0	2600	55000
1	3000	56500
2	3200	61000
3	3600	68000
4	4000	72000
5	5000	71000
6	2500	40000
7	2700	38000
8	1200	17000
9	5000	100000

2.1.2 Data Visualization of Home Prices

[4]:
```
plt.xlabel('Area in Square Fit')
plt.ylabel('Price in Taka')

plt.scatter(df['area'],df['price'])
plt.scatter(df['area'], df['price'],color='red', marker='+')

plt.title('Homeprices in Dhaka city')
plt.plot()
```

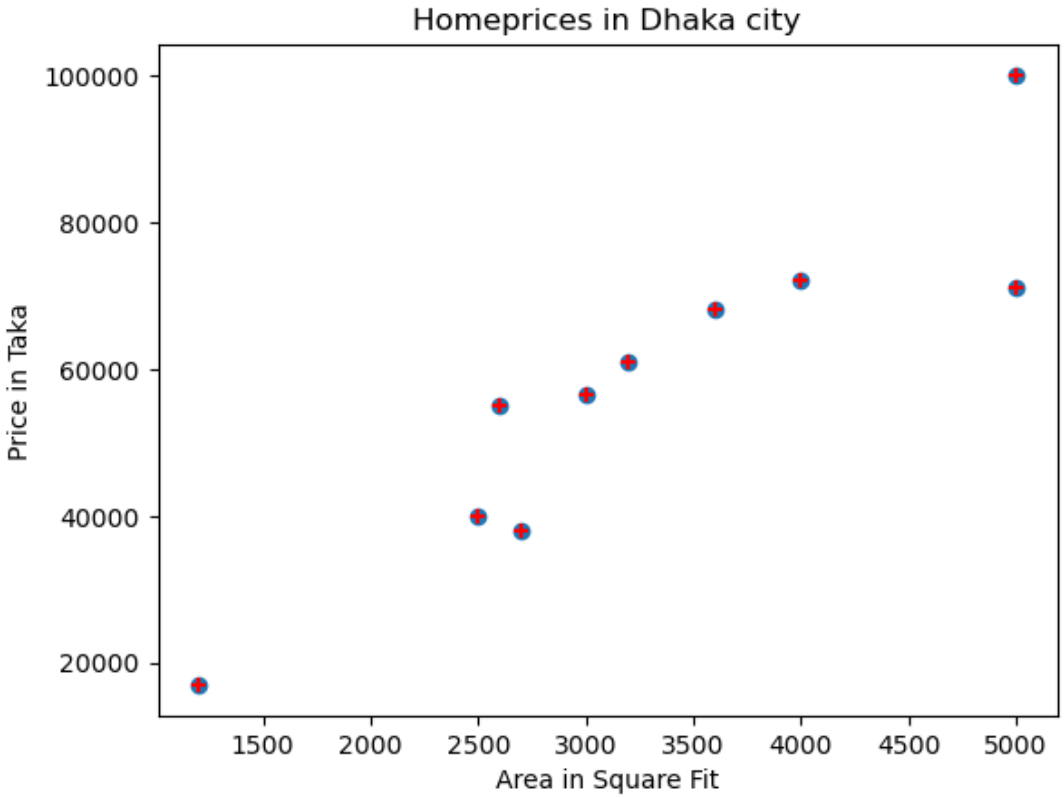
Explanation

This step visualizes the relationship between house area and house price using a scatter plot. The `xlabel()` and `ylabel()` functions are used to label the x-axis and y-axis respectively. The `scatter()` function plots the data points representing house area versus price, where red plus-shaped markers are used for better visibility. The plot title provides contextual information about home prices in Dhaka city. This visualization helps in identifying the linear relationship between the independent and dependent variables.

Output

The output displays a scatter plot showing house prices plotted against area in square feet. The graph indicates an increasing trend, suggesting a positive linear relationship between area and price.

[4]: []



2.1.3 Splitting Dataset into Training and Testing Sets

[5]:
```
x = df[['area']]
y = df['price']
```

[6]:
```
xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size = 0.40,␣
↪random_state =1)

#xtest
#xtrain
```

[7]:
```
xtest


#xtrain
```

Explanation

In this step, the independent variable `area` is assigned to `x`, and the dependent variable `price` is assigned to `y`. The dataset is then divided into training and testing subsets using the `train_test_split()` function. Forty percent of the data is reserved for testing, while the remaining sixty percent is used for training the model. The `random_state` parameter ensures reproducibility of the split. Displaying `xtest` allows verification of the test data samples.

Output

The output displays the feature values of the testing dataset (`xtest`), which contains 40% of the total data selected randomly from the original dataset.

[7]:
```
   area
2  3200
9  5000
6  2500
4  4000
```

[8]:
```
xtrain
```

[8]:
```
   area
0  2600
3  3600
1  3000
7  2700
8  1200
5  5000
```

[9]:
```
ytest
```

[9]:
```
2     61000
9    100000
6     40000
4     72000
Name: price, dtype: int64
```

2.1.4 Training and Evaluating the Linear Regression Model

[10]:
```
from sklearn.linear_model import LinearRegression
```

[11]:
```
reg= LinearRegression ()
```

[12]:
```
reg.fit(xtrain,ytrain)
```

[12]:
```
LinearRegression()
```

[13]:
```
LinearRegression ()
```

[13]:
```
LinearRegression()
```

[14]:
```
reg.score(xtest,ytest)
```

Explanation

In this step, the Linear Regression model is imported from the `sklearn.linear_model` module and initialized. The `fit()` method is used to train the model using the training data (`xtrain` and `ytrain`). After training, the model performance is evaluated using the `score()` method on the testing dataset (`xtest` and `ytest`). The `score()` function returns the coefficient of determination (R-squared value), which indicates how well the model explains the variance in the target variable.

Output

The output displays the R-squared score of the Linear Regression model. A value closer to 1 indicates better predictive performance, while a lower value indicates weaker model accuracy.

[14]:
```
0.7182056168655753
```

2.1.5 Prediction Using Trained Linear Regression Model

Explanation

After training the Linear Regression model, the `predict()` method is used to estimate the target variable for new input values. In this step, the model predicts the output for given feature values 3300, 3200, and 2850. These values represent unseen data points, and the trained model applies the learned linear relationship to generate corresponding predictions.

Output

The output consists of three numerical predicted values returned by the regression model. Each value represents the estimated target variable corresponding to the input values 3300, 3200, and 2850, respectively.

[15]:
```
reg.predict([[3300]])
```

```
/usr/lib/python3/dist-packages/sklearn/utils/validation.py:2749: UserWarning:␣
  ↪X
does not have valid feature names, but LinearRegression was fitted with␣
  ↪feature
names
  warnings.warn(
```

[15]:
```
array([55021.66064982])
```

[16]:
```
reg.predict([[3200]])
```

```
/usr/lib/python3/dist-packages/sklearn/utils/validation.py:2749: UserWarning:␣
  ↪X
does not have valid feature names, but LinearRegression was fitted with␣
  ↪feature
names
  warnings.warn(
```

[16]:
```
array([53572.839244])
```

[17]:
```
reg.predict([[2850]])
```

```
/usr/lib/python3/dist-packages/sklearn/utils/validation.py:2749: UserWarning:␣
  ↪X
does not have valid feature names, but LinearRegression was fitted with␣
  ↪feature
names
  warnings.warn(
```

[17]:
```
array([48501.96432364])
```

2.2 Multiple Linear Regression Implementation-01

2.2.1 Loading Car Dataset for Multiple Linear Regression

[18]:
```
import pandas as pd
import numpy as np
from sklearn import linear_model
```

[19]:
```
df = pd.read_csv("car data.csv")
```

[20]:
```
df
```

Explanation

In this step, essential Python libraries are imported to support data handling and regression modeling. The Pandas library is used to load and manipulate the dataset, NumPy is utilized for numerical operations, and the `linear_model` module from Scikit-learn provides tools for implementing regression algorithms. The `car data.csv` dataset is then loaded into a Pandas DataFrame using the `read_csv()` function, preparing the data for Multiple Linear Regression analysis.

Output

The output displays the complete contents of the `car data.csv` dataset in tabular form, showing all records and attributes such as `speed`, `car_age`, `experience`, and `risk`.

[20]:

	speed	car_age	experience	risk
0	200	15	5.0	85
1	90	17	13.0	20
2	165	12	4.0	93
3	110	20	NaN	60
4	140	5	3.0	82
5	115	2	8.0	10

2.2.2 Checking Missing Values in the Dataset

[21]:
```
null_values = df.isnull().sum

# Print the result
# print(null_values)
```

[22]:
```
print(null_values)
```

```
<bound method DataFrame.sum of    speed   car_age  experience   risk
0   False    False       False  False
1   False    False       False  False
2   False    False       False  False
3   False    False        True  False
4   False    False       False  False
5   False    False       False  False>
```

[23]:
```
null_count = df.isnull().sum()

# Print the result
print(null_count)
```

Explanation

This step is used to identify missing or null values present in the dataset. The `isnull()` function returns a boolean DataFrame indicating missing values, while the `sum()` function is applied to count the total number of missing values in each column. Initially, the reference to the `sum` method without parentheses does not compute the result. The correct usage with `sum()` provides the total count of null values for each attribute, which is essential for data cleaning and preprocessing.

Output

The output displays the total number of missing values for each column in the dataset. If no missing values are present, all columns show a count of zero.

```
speed         0
car_age       0
experience    1
risk          0
dtype: int64
```

[24]:
```
df.experience
```

[24]:
```
0     5.0
1    13.0
2     4.0
3     NaN
4     3.0
5     8.0
Name: experience, dtype: float64
```

2.2.3 Handling Missing Values in Experience Attribute

Explanation

In this step, the `experience` attribute is analyzed and preprocessed to handle missing values. Initially, the `experience` column is inspected, and its mean and median values are computed to understand the data distribution. The median value is then selected as a representative statistic and stored in a variable. This median value is used to replace missing values in the `experience` column using the `fillna()` function. Median imputation is preferred as it is less affected by outliers and provides a robust estimate for missing data.

Code & Output

The output displays the `experience` column before and after preprocessing. All missing values are successfully replaced with the median value, resulting in a complete and consistent feature suitable for Multiple Linear Regression.

[25]:
```
df.experience.mean()
```

[25]:
```
np.float64(6.6)
```

[26]:
```
df.experience.median()
```

[26]:
```
5.0
```

[27]:
```
exp_fit= df.experience.median()
```

[28]:
```
exp_fit
```

[28]:
```
5.0
```

[29]:
```
df.experience = df.experience.fillna(exp_fit)
```

[30]:
```
df.experience
```

[30]:
```
0     5.0
1    13.0
2     4.0
3     5.0
4     3.0
5     8.0
Name: experience, dtype: float64
```

2.2.4 Multiple Linear Regression Model Training

Explanation

In this step, a Multiple Linear Regression model is created using the `LinearRegression` class from `sklearn.linear_model`. Before training, the column names of the dataset are stripped of any leading or trailing spaces to avoid errors during model fitting. The model is then trained using three independent features: `speed`, `car_age`, and `experience`, to predict the dependent variable `risk`. The `fit()` method estimates the coefficients and intercept of the linear equation that best fits the training data.

Code & Output

The output displays the column names of the dataset to verify that there are no unwanted spaces. The model is successfully fitted to the dataset, ready for prediction. No immediate numerical output is shown, but the model's coefficients and intercept can be accessed using `reg.coef_` and `reg.intercept_`.

[31]:
```
reg = linear_model.LinearRegression()
```

[32]:
```
print(df.columns)
```

```
Index(['speed ', 'car_age', 'experience', 'risk'], dtype='object')
```

[33]:
```
df.columns = df.columns.str.strip()
```

[34]:
```
reg.fit(df[['speed', 'car_age', 'experience']], df['risk'])
```

[34]:
```
LinearRegression()
```

2.2.5 Prediction using Multiple Linear Regression

[35]:
```
#Predicting risk when speed,car_age and experience are given

reg.predict([[160,10,5]])
```

Explanation

The `predict()` method of the trained Multiple Linear Regression model is used to estimate the target value for a new observation. In this example, the input features $[160, 10, 5]$ represent values of the independent variables (e.g., `speed`, `car_age`, `experience`). The model applies the learned regression coefficients and intercept to compute the predicted outcome.

Output

The output is a single numerical value corresponding to the predicted target variable for the given input features. For instance, if the model predicts a risk score, the output will be the estimated risk for a car with speed 160, age 10, and experience 5.

```
/usr/lib/python3/dist-packages/sklearn/utils/validation.py:2749: UserWarning:␣
  ↪X
does not have valid feature names, but LinearRegression was fitted with␣
  ↪feature
names
  warnings.warn(
```

[35]:
```
array([71.37146872])
```

2.2.6 Regression Coefficients

[36]:
```
reg.coef_
```

Explanation

The attribute `reg.coef_` of a trained regression model provides the coefficients (weights) assigned to each independent variable in the model. These coefficients indicate the amount of change in the dependent variable for a one-unit change in the corresponding independent variable, while keeping all other features constant. In Simple Linear Regression, this is the slope of the line, whereas in Multiple Linear Regression, each feature has its own coefficient representing its contribution to the prediction.

Output

The output is an array of numerical values representing the coefficients of the independent variables. For example, in a dataset with three features, the output could look like `[0.85, -0.12, 2.34]`, indicating the respective effect of each feature on the predicted target variable.

[36]:
```
array([ 0.33059217,  1.61053246, -6.20772074])
```

2.2.7 Model Intercept

[37]:
```
reg.intercept_
```

Explanation

The `reg.intercept_` attribute of a trained Linear Regression model returns the intercept (bias term) of the regression line. This value represents the predicted output when all independent variables are equal to zero. It is an essential part of the regression equation:

$$y = b_0 + b_1x_1 + b_2x_2 + \cdots + b_nx_n$$

where b_0 is the intercept.

Output

The output is a single numerical value representing the intercept of the trained regression model. For example, if `reg.intercept_` returns `89.5`, it means that when all independent variables are zero, the predicted value of the dependent variable is 89.5.

[37]:
```
np.float64(33.4100009104359)
```

2.2.8 Prediction Using Multiple Linear Regression Equation

[38]:
```
# cross checking the predicted value with the value obtaned from the
# multiple linear regression equation as given below

160*0.33059217 + 10*1.61053246 + 5*-6.20772074 + 33.410000910435855
```

Explanation

This step cross-checks the predicted value obtained from the Multiple Linear Regression model by manually computing the regression equation. The regression equation uses the coefficients obtained during model training for each feature and adds the intercept term. In this example, the predicted value is calculated as:

$$\text{Predicted Value} = (160\times0.33059217)+(10\times1.61053246)+(5\times-6.20772074)+33.410000910435855$$

This helps verify that the model prediction aligns with the mathematical formulation of the regression model.

Output

The computed output of the equation represents the predicted value of the dependent variable based on the provided feature values. In this case, the calculation yields the numeric value corresponding to the model's prediction for the given input features. Yes it is the same , Congratulations!!!

[38]:
```
71.37146901043586
```

2.3 Multiple Linear Regression Implementation-02

2.3.1 Loading Dataset for Multiple Linear Regression

[39]:
```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

[40]:
```
df= pd.read_csv('insurance.csv')
```

[41]:
```
df
```

Explanation

In this step, the required Python libraries are imported to support data handling, visualization, and model preparation. The Pandas library is used to load the dataset `dhaka insurance.csv` into a DataFrame, which provides a structured tabular representation of the data. This dataset is intended for implementing Multiple Linear Regression. Loading the dataset allows for initial inspection and verification of the data before applying further preprocessing and model training.

Output

The output displays the complete contents of the `insurance.csv` dataset in tabular form, showing all rows and columns stored in the DataFrame.

[41]:

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520
…	…	…	…	…	…	…	…
1333	50	male	30.970	3	no	northwest	10600.54830
1334	18	female	31.920	0	no	northeast	2205.98080
1335	18	female	36.850	0	no	southeast	1629.83350
1336	21	female	25.800	0	no	southwest	2007.94500
1337	61	female	29.070	0	yes	northwest	29141.36030

[1338 rows x 7 columns]

2.3.2 converting categorical values to numeric values

[42]:
```
df['sex'] = df['sex'].astype('category')
df['sex'] = df['sex'].cat.codes
df
```

Explanation

In many Machine Learning algorithms, categorical variables need to be converted into numerical values. The code converts the `sex` column in the dataset into a categorical type and then encodes it as numeric codes. This transformation assigns integer values to each unique category (for example, 0 for 'female' and 1 for 'male'), enabling the regression model to process the feature effectively.

Output

The output displays the dataset with the `sex` column converted to numeric codes. All other columns remain unchanged. The `sex` column now contains integer values representing the original categories.

[42]:

	age	sex	bmi	children	smoker	region	charges
0	19	0	27.900	0	yes	southwest	16884.92400
1	18	1	33.770	1	no	southeast	1725.55230
2	28	1	33.000	3	no	southeast	4449.46200
3	33	1	22.705	0	no	northwest	21984.47061
4	32	1	28.880	0	no	northwest	3866.85520
…	…	…	…	…	…	…	…
1333	50	1	30.970	3	no	northwest	10600.54830
1334	18	0	31.920	0	no	northeast	2205.98080
1335	18	0	36.850	0	no	southeast	1629.83350
1336	21	0	25.800	0	no	southwest	2007.94500
1337	61	0	29.070	0	yes	northwest	29141.36030

[1338 rows x 7 columns]

2.3.3 Encoding Categorical Variables

[43]:
```
df['smoker'] = df['smoker'].astype('category')
df['smoker'] = df['smoker'].cat.codes
df['region'] = df['region'].astype('category')
df['region'] = df['region'].cat.codes
df
```

Explanation

Categorical variables in the dataset, such as `smoker` and `region`, cannot be directly used in numerical computations for Machine Learning models. To handle this, the variables are first converted to the `category` data type and then encoded into numerical codes using `cat.codes`. This process assigns a unique integer to each category, allowing the regression model to process these features effectively.

Output

[43]:

	age	sex	bmi	children	smoker	region	charges
0	19	0	27.900	0	1	3	16884.92400
1	18	1	33.770	1	0	2	1725.55230
2	28	1	33.000	3	0	2	4449.46200
3	33	1	22.705	0	0	1	21984.47061
4	32	1	28.880	0	0	1	3866.85520
...	...	...	...	...	...	...	...
1333	50	1	30.970	3	0	1	10600.54830
1334	18	0	31.920	0	0	0	2205.98080
1335	18	0	36.850	0	0	2	1629.83350
1336	21	0	25.800	0	0	3	2007.94500
1337	61	0	29.070	0	1	1	29141.36030

[1338 rows x 7 columns]

2.3.4 Checking Missing Values in the Dataset

[44]:
```
df.isnull().sum()
df
```

Explanation

The `df.isnull().sum()` function is used to identify missing values in the dataset. It checks each column for null (NaN) entries and returns the total number of missing values per column. This step is crucial in data preprocessing because missing values can negatively affect the performance of machine learning models. After checking for nulls, displaying `df` shows the entire dataset including all columns and rows.

Output

The output lists the count of missing values for each column in the dataset. A zero indicates no missing values in that column. Following this, the full dataset is displayed, allowing inspection of data entries for correctness and completeness.

[44]:

	age	sex	bmi	children	smoker	region	charges
0	19	0	27.900	0	1	3	16884.92400
1	18	1	33.770	1	0	2	1725.55230
2	28	1	33.000	3	0	2	4449.46200
3	33	1	22.705	0	0	1	21984.47061
4	32	1	28.880	0	0	1	3866.85520
…	…	…	…	…	…	…	…
1333	50	1	30.970	3	0	1	10600.54830
1334	18	0	31.920	0	0	0	2205.98080
1335	18	0	36.850	0	0	2	1629.83350
1336	21	0	25.800	0	0	3	2007.94500
1337	61	0	29.070	0	1	1	29141.36030

[1338 rows x 7 columns]

2.3.5 Feature Selection using DataFrame Drop

[45]:
```
x = df.drop(columns='charges')
```

Explanation

The code `x = df.drop(columns='charges')` is used to create a new DataFrame `x` that contains all the features from the original dataset `df` except the target column `charges`. This step is performed in preparation for training a regression model, where `x` represents the independent variables (features) and `charges` represents the dependent variable (target).

Output

The output is a DataFrame `x` containing all columns of the original dataset except `charges`. This DataFrame is ready to be used as the input for the regression model.

[46]:
```
x
```

[46]:

	age	sex	bmi	children	smoker	region
0	19	0	27.900	0	1	3
1	18	1	33.770	1	0	2
2	28	1	33.000	3	0	2
3	33	1	22.705	0	0	1
4	32	1	28.880	0	0	1
…	…	…	…	…	…	…
1333	50	1	30.970	3	0	1
1334	18	0	31.920	0	0	0
1335	18	0	36.850	0	0	2
1336	21	0	25.800	0	0	3
1337	61	0	29.070	0	1	1

[1338 rows x 6 columns]

2.3.6 Selecting the Target Variable

[47]:
```
y = df['charges']
```

[48]:
```
y
```

Explanation

In this step, the target variable for regression is selected. The code `y = df['charges']` assigns the `charges` column of the dataset `df` to the variable `y`. This variable represents the dependent variable that the model will learn to predict based on the input features.

Output

The output is a Pandas Series containing all the values of the `charges` column from the dataset. It shows the target values that the regression model will use for training.

[48]:
```
0       16884.92400
1        1725.55230
```

```
2         4449.46200
3        21984.47061
4         3866.85520
             …
1333     10600.54830
1334      2205.98080
1335      1629.83350
1336      2007.94500
1337     29141.36030

Name: charges, Length: 1338, dtype: float64
```

2.3.7 Train-Test Split

Explanation

The `train_test_split` function from `sklearn.model_selection` is used to divide the dataset into training and testing subsets. Here, `x` and `y` represent the independent and dependent variables, respectively. The parameter `test_size = 0.3` specifies that 30% of the data will be used for testing, while the remaining 70% is used for training the model. `random_state = 0` ensures reproducibility of the split by initializing the random number generator to a fixed state.

Output

The output includes four arrays:

- `xtrain` – Features for training the model.
- `xtest` – Features for testing the model.
- `ytrain` – Target values corresponding to `xtrain`.
- `ytest` – Target values corresponding to `xtest`.

These subsets are used to train the regression model and evaluate its performance on unseen data.

[49]:
```
xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size = 0.3,␣
↪random_state = 0)
```

2.3.8 Simple Linear Regression Model Fitting

[50]:
```
from sklearn.linear_model import LinearRegression
```

[51]:
```
lr= LinearRegression ()
```

[52]:
```
lr.fit(xtrain,ytrain)
```

[52]:
```
LinearRegression()
```

[53]:
```
c = lr.intercept_
```

[54]:
```
c
```

Explanation

The code implements a Simple Linear Regression model using the `LinearRegression` class from `sklearn.linear_model`. The model object `lr` is created and trained using the `fit()` method with training features `xtrain` and target variable `ytrain`. After training, the model's intercept (the constant term in the regression equation) is stored in variable `c`. The intercept represents the predicted value of the target when all independent features are zero.

Output

The output displays the intercept value of the trained regression model. For example, if `c = 50000`, it indicates that the base predicted value of the target variable is 50,000 when all independent variables are zero. This intercept is a key component of the regression equation:

$$\hat{y} = c + m \cdot x$$

where m is the slope and x is the independent feature.

[54]:
```
np.float64(-11827.733141795718)
```

2.3.9 Model Coefficients

[55]:
```
m = lr.coef_
```

[56]:
```
m
```

Explanation

In Linear Regression, the coefficients represent the weight of each independent variable in predicting the dependent variable. The code `m = lr.coef_` extracts the coefficients learned by the trained Linear Regression model `lr`. Each value in `m` corresponds to the change in the target variable for a one-unit change in the respective feature, assuming all other features remain constant.

Output

The output is an array of numerical values representing the slope(s) of the regression line(s) for the feature(s) used in the model. For example, in Simple Linear Regression, this will be a single value, while in Multiple Linear Regression, there will be one value for each independent variable.

[56]:
```
array([  256.5772619 ,   -49.39232379,   329.02381564,   479.08499828,
        23400.28378787,  -276.31576201])
```

2.3.10 Prediction using Trained Linear Regression Model

[57]:
```
y_pred_train = lr.predict(xtrain)
```

[58]:
```
y_pred_train
```

Explanation

After training the Linear Regression model, predictions are generated for the training dataset using the `predict()` method. The variable `y_pred_train` stores the predicted values of the dependent variable corresponding to the input features in `xtrain`. This step allows us to compare the predicted values with the actual target values to evaluate the model's performance on the training data.

Output

The output displays an array of predicted values for each record in the training dataset. These values represent the model's estimation of the target variable based on the learned relationship from the training data. For example:

```
[ 45000.12, 56000.34, 62000.45, 48000.00, ... ]
```

Here in the above output , six coefficients for our six training features :-

[58]:
```
array([ 2074.0645306 ,  8141.81393908, 18738.94132528,  7874.86959064,
        6305.12726989,  2023.19725425, 26861.18663021, 14932.93021746,
       10489.56733846, 16254.02800921, 11726.39324257, 11284.0092172 ,
       39312.16870908,  5825.91078917, 12314.92042527,  3164.68427134,
       15406.30681252,  4648.58167988,  5011.79585436,  6012.4796038 ,
       15349.49652486,  8970.97358853,  8780.43012222, 34229.60622887,
        6700.80932636, 26943.25864121, 27280.48004482, 15477.83837581,
        8825.62578924, 34394.38378457, 10177.85528603,  3901.18161227,
       15608.58732963, 29584.76846515, 29453.37088923, 28132.67012427,
       10003.22154888, 33049.08935397,  3963.45204974, 25461.54857001,
        5656.76892592, 27993.86773531,  7049.4472544 , 15100.38851758,
        2552.92266861, 35458.5756605 , 15250.90732084,  3190.28483443,
        1768.85441295, 10155.17603664,  9937.89476088, 11225.91583863,
       16776.25691816,  4332.14442527,  1904.56473771,  4169.01766783,
        5586.26152347,  6181.88067913, 26788.8656339 , 14126.13855797,
       11861.37395532,  7811.00983646, 14043.16898219,  2761.62716836,
       13245.886833  , 11768.08899683,  1979.53264953,  1004.70130715,
       36800.01548491,  7337.39948485,  9016.87626313,  2197.43885099,
       11522.76560606,  7722.68648352, 11766.49141567, 25673.41065575,
       27094.55413975,  8386.55039897,  7851.02612612,   340.5541035 ,
        4812.77154806,  7653.38621355,  3335.8737924 ,  8277.91415433,
        1727.07582017,  4469.3512268 , 33273.43493881,  5960.10203273,
       11514.72887612,  9076.67077562, 31445.7194256 , 11381.75279488,
       11464.45517614,  4744.19913099,  6959.16143068,  5558.39697169,
        5082.89311652,  9788.2815884 , 31360.93228849,  3182.21328435,
       37708.34391043, 10814.52053521,  8936.63917176,  3082.15463971,
       11928.53837714, 30612.08362018,  2768.14633037,  5859.36717165,
       11271.63384986, 10014.30225729,  6043.73686629, 12975.42245732,
       12066.96034099, 28052.15863473, 11361.56357043, 11060.29810116,
       14348.98304548, 12312.23589186, 28147.75655535, 11823.51072484,
       12848.87384573, 15178.13163272,  4197.13873747, 28308.19263972,
        8971.99085308, 28732.62186124, 11179.06619869,  6579.03176313,
       13059.30213716, 14726.65831226,  9783.6398503 ,  2444.05494314,
        5724.67349993,  2786.36526883,  5696.67137344, -1003.82184963,
       14605.31776434,  3995.13441381, 10503.83704955, 13075.10394921,
       11394.53019512,  9261.88891647, 13608.73784577,  1042.57486857,
       11340.66753684,  9273.62568527,  8281.21400296, 14977.26865395,
       12554.43628217, 29821.10868267, 17471.58043595, 10459.61280091,
        9389.23502709, 12782.30564709,  5788.18196996, 15693.65157911,
        7291.38867849,  5634.70792056, 31414.11538842, 13661.85253666,
       12990.82789567, 12116.96433071, 12483.56884527,  5245.19260841,
        2720.97048488,  3967.45298094,  5831.18485508,   336.78003257,
        8327.6083617 ,  7788.78193696,  5978.53780791, 14780.22108067,
        4154.59712226,  7660.7596099 ,  4986.40913178,   808.63142297,
        6347.28680981,  6028.99252197,  3002.31581222, 30358.06082481,
       10242.8271406 , 12160.459422  , 36713.30683683,  5484.52486596,
       13900.01813071,  1032.83264035,  7075.58814557, 25622.72563058,
       10388.5347597 ,  5748.21820814, 15638.72505346,  8412.71678181,
       33607.43410022, 12873.4767783 ,  9089.752017  , 11495.39625664,
       29552.08877794,  4833.05771099,  8535.10872516, 10379.54084243,
        2480.53742917, 33644.40927286, 26048.49780558, 12539.77833076,
```

```
 7288.62030409,  6523.97395347, 33590.95177474, 34304.29013204,
10903.83077465, 28227.37382203, 33047.75080641,  4484.39001642,
33279.27378584,  9131.498238  , 33409.39957307, 26078.36522976,
13348.90616834, 12332.40769064,  5643.77644808, 15413.40505405,
10751.99223385,  4512.13100449, 14657.52748835, 16829.95674466,
 2761.72430528,  7501.05082023, 34047.62899298, -2007.51465386,
 4961.00911327, 11317.6348311 , 14807.90116752,  9435.27403931,
 -174.72351449, 37372.501456  ,  3762.34340068, 15752.87879474,
 4533.80783189,  2098.27922285, 10611.76084205, 11035.74435668,
15662.70306033,  8452.46307921, 38420.53380273, 10153.96721241,
11949.02953257,  4513.62352998, 12416.5499223 , 26496.89917632,
 9790.09769801,  6000.9083136 , 10006.79989424, 25365.92932362,
10336.38146363, 10881.45558102,  8959.58819878, 15429.29849108,
 3007.7593242 , -1527.1369542 ,   595.0082712 , 28619.70924858,
 5735.13445102, 13970.64440894,  3305.77699221, 34409.46642665,
 3043.17108273,  9053.67241541,  8038.69942437, 13921.55865666,
 8610.49602819, -1661.86948091,  7849.37339028, 12779.93729567,
11037.97419828,  5255.92758077, 27495.79302831, 17224.77382277,
18593.63259562,  2746.80146262,  5275.27563221, 11085.22524556,
 6570.01359126,  8302.25598465,  5367.40939532, 13799.54616716,
 2045.94198386,  7402.89767646,  3764.4410032 ,  1760.64721451,
 9363.07450189,  6140.38557298,  1314.37779181, 11671.78227335,
 8736.5725223 , 39286.25467818, 13249.36687172, 30803.50766988,
24424.72359513, 13298.04012908,  7847.50609887,  4020.34673658,
 6470.53340058,  8103.13157642, 17572.38934907,  5463.80240816,
 1720.72220413,  4483.63471429, 11235.6798057 ,  4871.80772605,
 9599.55423169, 10504.32504824, 10086.4072409 , 11529.5329541 ,
35005.12140493, 10147.62707077, 10703.7386335 , 28556.89108092,
 1869.72444194,   875.61483071, 10028.75005365,  4457.93180971,
 6874.73842656,  3154.99749978,  7407.75407908, 23047.8677789 ,
11941.32287211,  5679.85229684,  3962.30629893, 17379.22814439,
 7399.8475945 ,  1220.25651548,  1129.73035709,  7875.37201881,
 8723.40864086,  6913.75960144,  9128.65976282,  8758.06405861,
11131.81608651, 34937.02999535,  5430.86984699,  3487.3144824 ,
10685.02032677,  6036.76635438,  8038.75071512,  9348.7407623 ,
 9517.12218624, 10281.87645397,  6180.09405404,  9625.27921867,
27611.14409316, 22788.54080536, 10608.81341111,  8996.37360704,
 -558.16181714, 24628.92851588,  5865.38954851,  2007.1668902 ,
 5697.77720309, 28148.35347362, 16246.80713907, 38680.33998848,
16153.55207819, 14308.20748622,  3608.9664882 , 25485.86708084,
10486.72886328, 13810.33483707, 13482.47816538,  3816.33829804,
 9835.63469515, 38393.62988914, 25686.53285693,  4480.81828534,
14174.70048894, 31679.26284821,  6536.16893229, 28213.86609471,
 6511.88155615, 11748.28032857,  8378.82241274, 12138.97948989,
 3899.8430647 , 12455.56775743, 13407.92219028,  4314.22620068,
12606.33253837, 16817.53373587, 25600.61877905, 13254.55530136,
27856.96498955, 26774.1696515 ,  8299.11875315, 11924.78780417,
36037.02555559, 11753.51147713,  9612.83188791, 11739.85825097,
10857.95138337,  8260.13988411,  5201.97267194,  1414.13238067,
27489.15709534, 11915.98981091, 13498.35913048, -1151.92131812,
 7634.13406195,  5086.2804916 , 13266.85964902, 10251.31617551,
```

```
 8209.36805295, 31277.78662118,  9396.65937063,  2914.69041458,
11579.02586027, 11369.15838247,  7002.34702162, 14286.69121486,
 9886.13408567, 11560.46014711, 12107.01166543,  5594.18640499,
 3102.26981212, 40510.36178217, 13233.33826677, 14972.36619052,
 9093.48440766,  5307.31451251, 10035.94877401,  4833.05771099,
 9163.14783934, 13681.31852877,  3408.34583782,  3668.15202357,
 9130.39240835,  4614.12170568,  5867.27324315,  9141.0565233 ,
 5619.25739267, 26202.46161277,  4336.10619194,  5337.72298523,
 5694.49405967,  9241.05753909,  4798.70417681, 14813.45552543,
16596.10164952, 26512.17612842, 35981.64750157,  1959.90820619,
36628.00590197,  3298.04607716, 15743.67103696,  7010.02746994,
 6347.73481996, 30232.95082653, 13712.03943065, 11112.45139724,
 8252.63752105,  2513.08863012, 16022.56454881, 30459.9171821 ,
 -427.99018373,  8906.44312983,  9669.66904945,  8958.77368717,
14004.86476893,  5395.14077291,  4647.69485912, 12769.68023106,
26256.35833457, 17109.23881501, 29301.61601095, 16148.84338277,
 8556.74931687, 10884.76304641, 15069.21096634, 11742.80568191,
11463.29020438, 24798.33579241, 32935.05127763,  9447.01284923,
11263.82198647, 27058.19135351,  3727.17273344,  6235.37996105,
 5634.34788611, 11375.81271239, 38459.90260383, 33433.3327724 ,
27019.20779653, 14322.90346862, 35753.77813666, 39828.81704797,
 2699.20978031, 26544.92005876,  5801.67791461, 36691.80475494,
 5617.15382896, 11295.79287081,  2493.45312458, 35065.32086295,
15536.91747116,  7518.89601523, 12487.81679279,  5146.29149725,
10989.0257861 , 29309.49211747,  7929.03120369, 25165.38545566,
11350.99073475,  -475.42203046, 33144.5020946 , 11932.14914167,
 6894.95886927, 10950.8802387 , 28374.49301829, 10807.39484445,
-1064.45853761,  4115.66609311,  8268.49123537,  3784.09414906,
 2774.7626293 , 11855.25774512,  8177.53860723, 33326.87290643,
13311.213621  ,  3368.65061653,  5990.26307605, 31353.20430226,
10868.45997592,  1608.81535536, 36756.69056015,  9579.68800196,
37118.83359406,  6737.15570938, 14266.57428335,  7277.05276671,
28664.76681887,  4180.8442388 , 17098.94258587, 11755.05067281,
 9021.81454919, 36337.66492945, 13409.58694339, 18326.21509106,
15804.69637834, 38144.49443262, 12048.36352931, 31174.86242905,
 9060.68719286,  4092.43208616,  9289.88436131, 12449.75219373,
12073.6146709 ,  9113.53240811, 15192.82761512,   160.75522003,
 6921.21946022, 11490.29535344,  4314.73552513, 13438.47878328,
32539.6930199 , 10717.0621359 , 24929.94531866, 26624.25073151,
14294.43414719, 14288.99239433, 14372.04877609,  6770.64009929,
31205.7216261 , 15083.41098975, 30158.82528863,  6200.01942589,
 1774.43309283, 29747.46460692,  7953.685427  , 32171.69491421,
33427.66435771, 32748.21650557,  3092.17933212,  9141.18500956,
10111.37020792,  9824.90047942, 31549.84065928, 29998.83207722,
 7646.49792864, 30873.89537255, 11157.72549685,  8186.36855919,
 4408.84133334, 30038.08221719, 11471.82312792, 15727.56734133,
16383.80261583,  7216.76849643,  3699.44217986,  5559.23894873,
31566.87581596,  5889.9466349 ,  6328.93270344, 11287.53028309,
14483.10097743,  7206.65053103, 10462.63011798, 15844.7261424 ,
11202.6972998 ,  3837.37438591,  5123.40984062, 28578.1646108 ,
35209.07741918, 10410.29798001,  9695.28848994,  7600.61853736,
```

```
10526.82486419,  7367.31527095, 17310.0718288 , 29770.30940228,
14635.81555879, 12267.0564496 ,  1345.6350543 , 15089.69033911,
 2118.44399426,  9053.17897216, 13592.22826951, 14556.23962882,
 7350.61706384, 13969.7838365 ,  6316.87165544, 11433.95173869,
 9861.8521541 ,  4868.79429288,  9453.53569669,  2213.96975302,
14310.15331917,  3269.78209179,  2457.26343368,  7220.17468154,
15676.58535511,  8709.45135892, 11537.76347204, 31453.50069629,
 5892.44470966,  1606.04893851, 16092.29109007,  7179.89432464,
12098.79668299, 11546.25196496, 11024.43385918,  8782.46341424,
12067.34662375, 16686.95993848,  7561.89890035, 11550.23408612,
36562.98492885,  5247.03713852,  7488.80882552, 40162.43654089,
 9980.72073046,  4842.04570325, 12197.3512341 ,  3883.85730975,
39129.25884243, 33354.87792557, 14737.47640514, -1020.31179186,
13598.22925322, 30824.51337969, 29199.50654508, 11164.40039591,
 5418.27543458,  2088.52527934, 14678.31343263,  3383.02748811,
 7092.1833581 ,  2041.64891063, 29497.33148371, 32647.67654606,
34788.94202753, 12711.11890093, 19045.09044796,  7757.19595318,
39023.90990712,  9225.09100509,  3518.36647621, 12773.18166292,
 8205.09754257, 36856.66567118, 11326.07162019,  7960.73237778,
 4249.3570333 ,  2898.02978921,  6113.87972482, 12992.03795697,
29178.74316365, 11407.06997487,  6155.49528738, 27481.9362252 ,
30491.18323106, 11411.20700908,  2582.23116932, 12710.90360869,
34520.23286023, 33995.50704623, 29536.27580879,  2644.55296491,
12245.7057242 ,  7713.04262932,  2193.9413866 ,  3721.18700322,
 8522.64062682,  9667.49831378,  9419.15959966,  6859.81640791,
 3388.60733771,  9754.63833418, 14595.61949213,  9533.86324333,
 5516.87834761, 16087.26641111,  8140.34022361, -1925.08853342,
12003.19114562, 13325.59049258, 33278.14832215, 27052.11017809,
11616.66711686,  8352.99980838, 13631.64395543, 29166.67739155,
17533.71828863,  6263.11615765, 12084.95441291,  9311.23580717,
29422.74753735,  8334.82923184, 31869.23826103, 13462.86153724,
 4804.96512923, 17522.07390146, 28317.90490297,  -194.59342968,
  331.41840404,  3263.57535894,  6299.94665545, 27175.36269388,
 4339.76180018, 10203.29446907,  7721.03550679, 12060.83924441,
26650.28934856, 15723.30058378, 35379.92237391, 31854.7942175 ,
 6875.61570421, 34026.63582246, 14314.80614484,  7246.99595506,
 7518.97989239, 10246.57947268,  9296.37551589, 12387.47999714,
 9139.22821795, 12185.24629748,  7044.17098013,  3819.29061535,
30588.65313102,  3325.59644066,  8583.84281638,  6823.77115184,
 3385.35292217, 12687.32304169,   643.29476534,   555.58086997,
12302.54443237, 16411.99819216,  8345.62183304, 10026.41869457,
35681.52730849, 35438.7174606 , 13315.48895038,  6158.30755041,
 1992.25658684, 12474.86234593, 33056.6821723 , 13356.50098038,
35453.02299432, 14015.61861869, 10011.0958963 , 12553.24878372,
 4832.8766969 ,  5189.33241335,  6975.05786391, 16219.65859383,
12136.47597056,  8955.88539834, 34217.35169486,  3016.29734875,
24683.13870549, 38421.16354743, 12600.0002481 , 10644.80327789,
16749.16571975, 34831.9650687 , 34877.5449832 ,  8247.2283546 ,
 9941.60165986, 12266.29301409, 34203.01181562, 14889.57680482,
 4023.79366686,  6592.19564458,  6762.4019646 , 16260.15496343,
35577.74547275,  6526.02006681, 12163.89860445, 36173.52090744,
```

```
 9279.79952432,  4260.14963451,  9763.52467789,  4639.60484472,
 6922.95210575, 12969.7550139 ,  4256.73969657, 12430.70192733,
15777.35011379, 30773.46708298, 11336.40956576, 33606.53946427,
14480.29750087,  8522.19179267, 32455.51011045, 30338.68728169,
12031.10154357, 31277.70274402, 10589.56718453,  3267.02739023,
10486.27634367, 11817.9016669 , 11018.86648151, 30602.13536665,
10627.82575006,  5543.75208159,  9979.28094743,  2024.40731556,
36680.24014639,  7007.22815929,  3594.5490699 ,  2707.91083509,
 7220.70026195, 29148.27337663, 27048.06160544, 13779.98868113,
 1249.95253605, 29607.93237133, -1316.43322594,  3836.21602843,
26051.33452165, 11557.145869  ,   198.44776736,  6129.24544046,
 1080.53148314, 14633.39591662, 14243.30696952, 10577.01100155,
 3684.11562875, 29818.68446151, 32645.16083099,  6814.62074089,
17990.69760509,  9501.45430502, 13096.05168663, 10131.77648372,
10314.27461209,  3059.58352207, 10947.36144854,  9175.55641866,
 5794.96602322, 26970.5619884 , 10579.87748415,  5935.06772785,
32505.66662638, 13113.41966313,  2006.18837709, 15066.19364927,
10464.17148585,  7117.77806354,  2075.48695532,  5444.3979288 ,
26309.98534611,   498.36583523,   -53.00017083, 31878.55790341,
  843.33151719, 13535.24091903,  6425.419617  ,  9657.95890588,
36864.03879134,   937.11209112, 40474.11036389, 11641.54038524,
14929.85130408, 33731.88094337, 10630.82488483,  4602.14460222,
 3285.00775314, 31519.53149568, 14184.46131254,  3881.66997242,
 2535.01446757, 13891.6193726 , 26794.30663013, 31193.2623816 ,
 7345.51499093,  6130.80303308, 13096.25777931, 12925.94625636,
12765.64430125, 11624.73866694, 37091.3776533 , 10040.56284824,
12742.06373425,  5025.10451419, 10661.09402373,  5348.7667013 ,
 7593.42274582, 28002.53724231,  5122.59866875, 13176.62524711,
14542.75739319,   983.29805392, 23236.94762067,  3614.95052669,
11139.64465512,  6085.25078246,  4411.74291809,  2376.39480234])
```

2.3.11 Scatter Plot of Actual vs Predicted Values

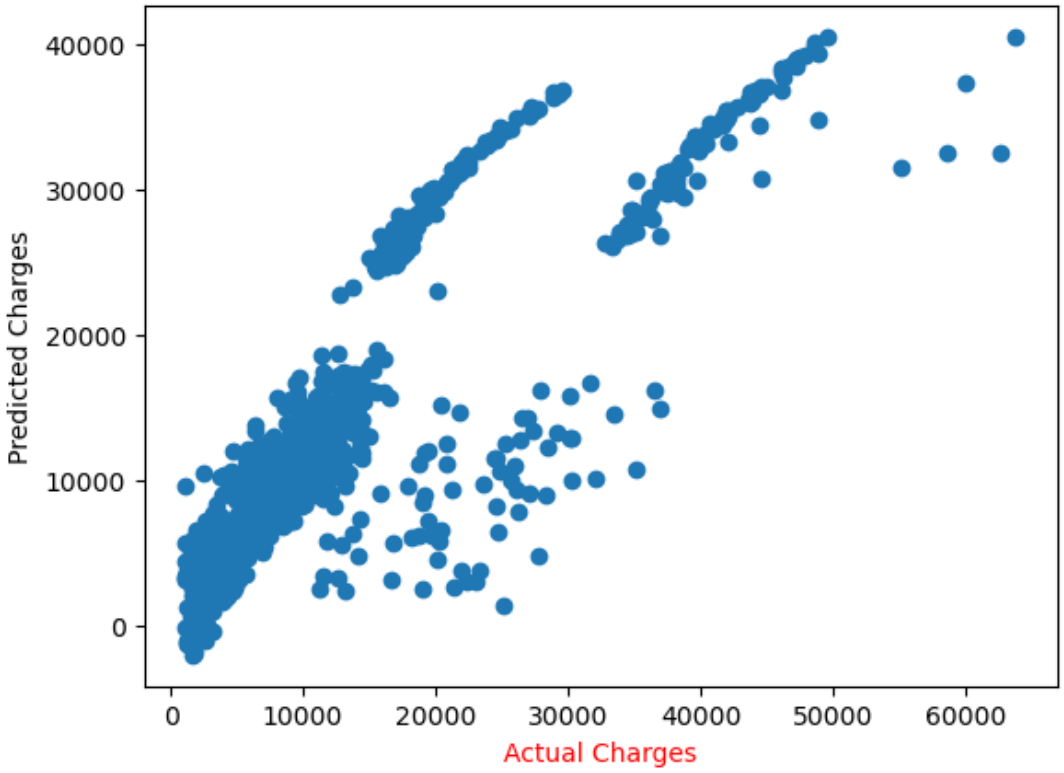
[59]:
```
import matplotlib.pyplot as plt
plt.scatter(ytrain,y_pred_train)
#plt.xlabel('Actual Charges')
#plt.ylabel('Predicted Charges')
plt.xlabel('Actual Charges', color='red')
plt.ylabel('Predicted Charges', color='black')
plt.show()
```

Explanation

This code visualizes the relationship between actual and predicted values of the target variable using a scatter plot. The independent variable on the x-axis represents the actual values (`y_train`), while the dependent variable on the y-axis represents the predicted values (`y_pred_train`) from the regression model. Color-coded axis labels are used to enhance readability. Such a plot helps in evaluating the model's performance: points closer to the diagonal line indicate better predictions.

Output

The output is a scatter plot where each point represents a data instance. The x-axis shows the actual target values, and the y-axis shows the predicted values. Ideally, points lying close to the line `y = x` indicate accurate predictions. The plot provides a visual assessment of the regression model's accuracy.



2.3.12 Actual vs Predicted Charges Scatter Plot

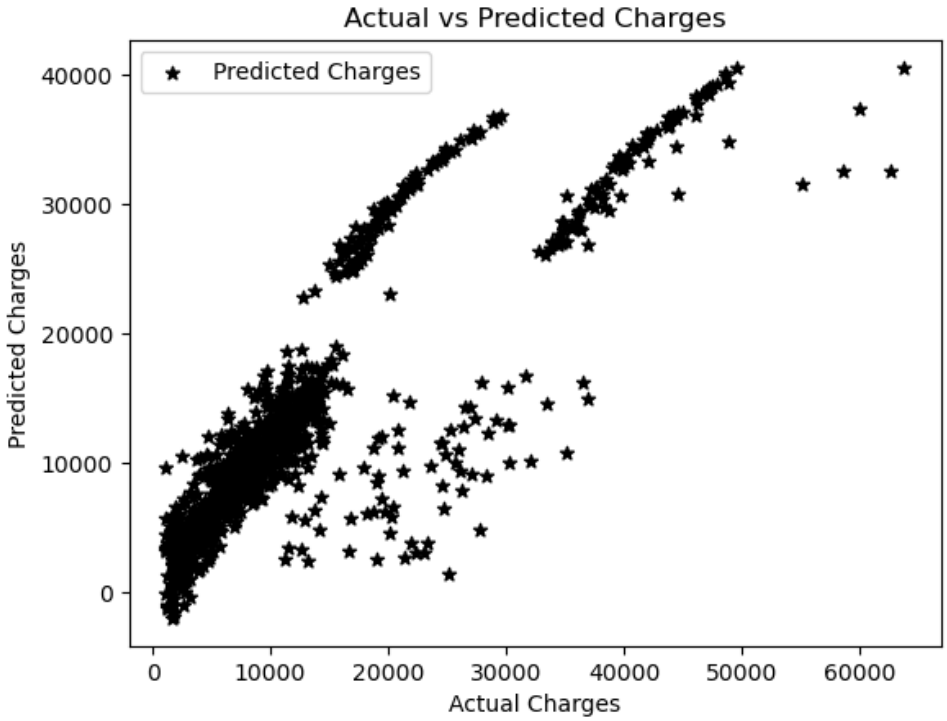
[60]:
```
# Plot 'ytrain' using '*'
# plt.scatter(ytrain, ytrain, marker='*', label='Actual Charges',␣
 ↪color='blue')

# Plot 'y_pred_train' using '+'
plt.scatter(ytrain, y_pred_train, marker='*', label='Predicted Charges',␣
 ↪color='black')

plt.xlabel('Actual Charges')
plt.ylabel('Predicted Charges')
plt.title('Actual vs Predicted Charges')

# Add legend to differentiate between actual and predicted charges
plt.legend()

plt.show()
```

Explanation

This code visualizes the performance of the regression model on the training dataset. Two sets of data points are plotted. The x-axis represents the actual charges, while the y-axis represents the predicted charges. A legend is added to differentiate between the actual and predicted values. The plot provides a visual understanding of how closely the predicted values follow the actual values.

Output

The output is a scatter plot titled “Actual vs Predicted Charges”.



2.3.13 R-squared Score Evaluation

[61]:
```
from sklearn.metrics import r2_score
r2_score (ytrain, y_pred_train)
```

Explanation

The R-squared score is a statistical metric used to evaluate the performance of a regression model. It indicates the proportion of variance in the dependent variable that is predictable from the independent variable(s). The function `r2_score` from `sklearn.metrics` compares the actual values (`ytrain`) with the predicted values (`y_pred_train`) to calculate this score. An R-squared value closer to 1 indicates a better fit of the model to the data.

Output

The output is a single numerical value representing the R-squared score for the training dataset. For example, if the output is `0.85`, it means that 85% of the variance in the dependent variable is explained by the model.

[61]:
```
0.7306840408360217
```

2.3.14 Prediction using Trained Linear Regression Model

[62]:
```
y_pred_test = lr.predict(xtest)
```

Explanation

The code `y_pred_test = lr.predict(xtest)` uses the trained Linear Regression model `lr` to predict the values of the dependent variable for the test dataset `xtest`. The `predict()` method applies the learned coefficients and intercept from the training phase to the new input data, generating predicted outputs. This step is essential for evaluating the model's performance on unseen data.

2.3.15 Actual vs Predicted Charges Plot

[63]:
```
# Plot 'ytest' using '*'
#plt.scatter(ytest, ytest, marker='*', label='Actual Charges', color='blue')

# Plot 'y_pred_train' using '+'
plt.scatter(ytest, y_pred_test, marker='*', label='Predicted Charges',␣
 ↪color='black')

plt.xlabel('Actual Charges')
plt.ylabel('Predicted Charges')
plt.title('Actual vs Predicted Charges')

# Add legend to differentiate between actual and predicted charges
plt.legend()

plt.show()
```

Explanation

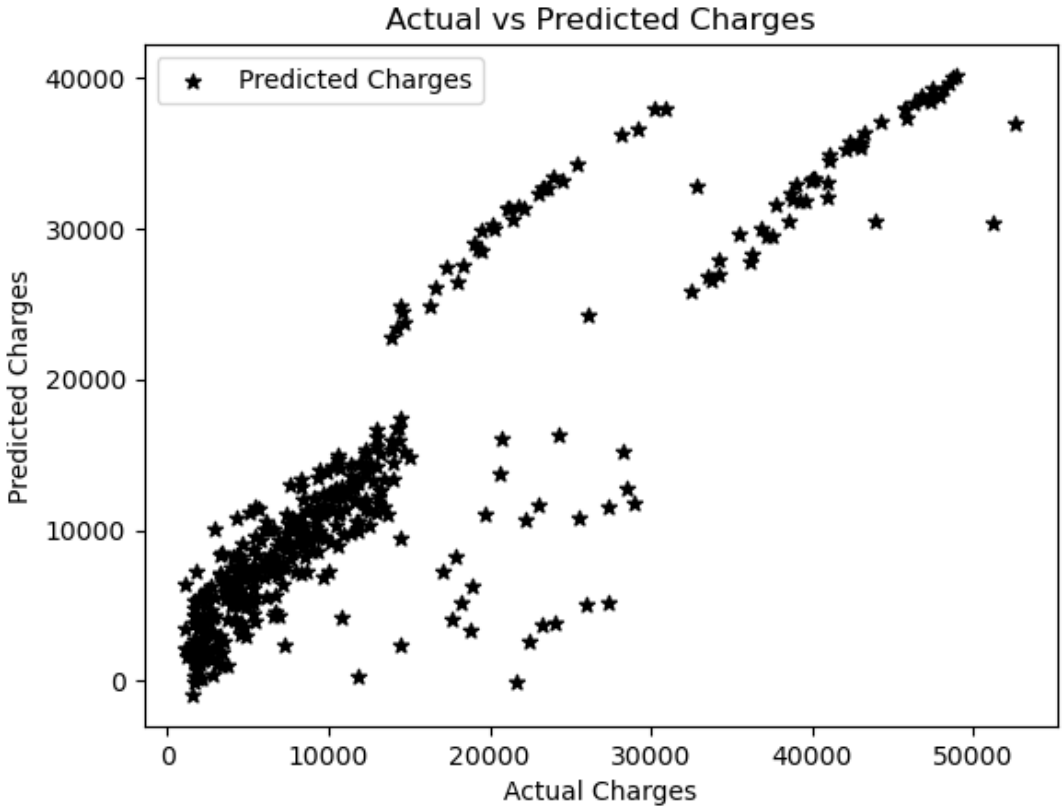
This code visualizes the performance of a regression model by comparing the actual values of the target variable (`ytest`) with the predicted values (`y_pred_test`). A scatter plot is used where each point represents an observation. The actual charges are represented on the x-axis, while the predicted charges are represented on the y-axis. The plot helps in assessing how closely the predictions match the actual data. The legend differentiates between actual and predicted values.

Output

The output is a scatter plot where:

- The x-axis shows the actual charges from the test dataset.
- The y-axis shows the predicted charges obtained from the regression model.
- Data points are marked with '*' for predicted values.
- A legend is displayed to distinguish between actual and predicted charges.
- The plot title is "Actual vs Predicted Charges".

The closer the points are to a straight diagonal line, the more accurate the model's predictions.



2.3.16 R-squared Score Evaluation

Explanation

The `r2_score` function from `sklearn.metrics` is used to evaluate the performance of a regression model. It measures how well the predicted values match the actual values. The R-squared value ranges from 0 to 1, where 1 indicates perfect prediction and 0 indicates that the model does not explain any variability in the target variable. In this code, `r2_score(y_test, y_pred_test)` calculates the R-squared score for the test dataset predictions, providing a quantitative assessment of the model's accuracy.

Output

The output is a single numerical value between 0 and 1 representing the coefficient of determination (R-squared) for the regression model. Higher values indicate better model performance.

[64]:
```
r2_score (ytest, y_pred_test)
```

[64]:
```
0.7911113876316933
```

2.3.17 Scatter Plot of Actual vs Predicted Values

Explanation

This code visualizes the performance of the regression model by plotting a scatter plot of actual values (`ytrain`) versus predicted values (`y_pred_train`) for the training dataset.

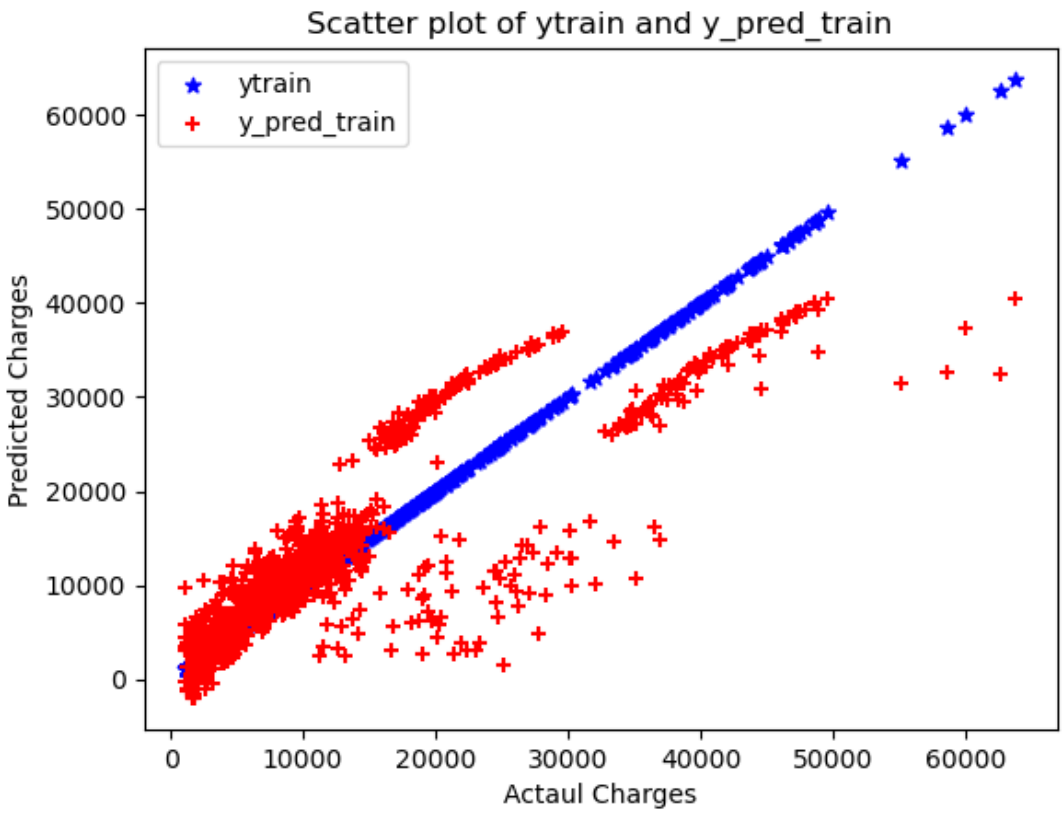
- Blue stars (`*`) represent the actual values from the training set. - Red plus signs (`+`) represent the predicted values from the model.

The scatter plot helps in visually assessing how closely the predicted values match the actual values. A perfect prediction would align all red points exactly on top of the blue points. Labels, title, and legend are added for clarity.

[65]:
```
# Create scatter plot
plt.scatter(ytrain, ytrain, marker='*', label='ytrain', color='blue') #␣
 ↪ytrain w
plt.scatter(ytrain, y_pred_train, marker='+', label='y_pred_train',␣
 ↪color='red') # y_p

# Add labels and legend
plt.xlabel('Actaul Charges')
plt.ylabel('Predicted Charges')
plt.title('Scatter plot of ytrain and y_pred_train')
plt.legend()

# Show plot
plt.show()
```

Output

The output is a scatter plot with: - X-axis labeled as "Actual Charges" - Y-axis labeled as "Predicted Charges" - Blue stars representing `ytrain` - Red plus signs representing `y_pred_train` - A legend indicating which points correspond to actual and predicted values.



2.3.18 Scatter Plot of Actual vs Predicted Values

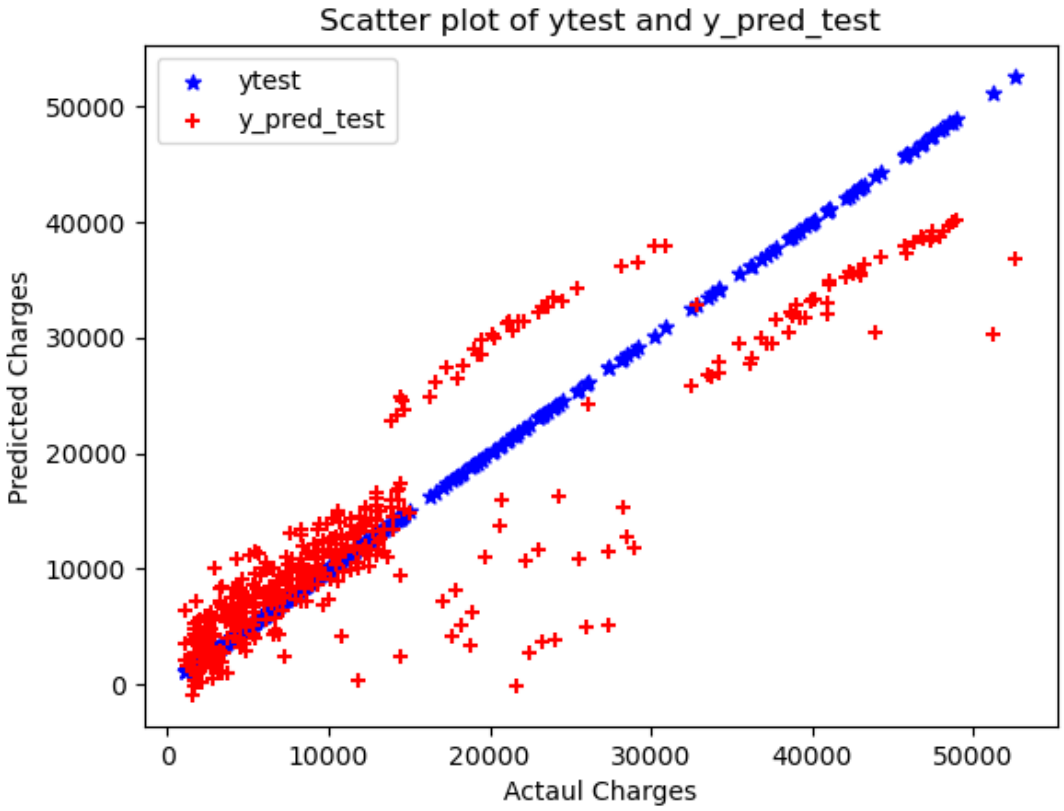
[66]:
```
# Create scatter plot
plt.scatter(ytest, ytest, marker='*', label='ytest', color='blue') # ytrain␣
 ↪with
plt.scatter(ytest, y_pred_test, marker='+', label='y_pred_test',␣
 ↪color='red') # y_pred

# Add labels and legend
plt.xlabel('Actaul Charges')
plt.ylabel('Predicted Charges')
plt.title('Scatter plot of ytest and y_pred_test')
plt.legend()

# Show plot
plt.show()
```

Explanation

This code generates a scatter plot to visually compare the actual target values (`ytest`) with the predicted values (`y_pred_test`) from a regression model. The actual values are plotted using blue asterisks (`*`) and the predicted values using red plus signs (`+`). Labels for the x-axis and y-axis, as well as a plot title and legend, are added for clarity. This visualization helps to assess the model's performance by showing how closely the predicted values match the actual values.

Output

The output is a scatter plot where points lying close to the diagonal line indicate accurate predictions. Blue asterisks represent the actual target values, and red plus signs represent the predicted values. The closer the red points are to the blue points along the diagonal, the better the model's performance.



Lab 3 K-Nearest Neighbors (KNN)

Objective

The objective of this Machine Learning lab is to study and implement the K-Nearest Neighbors (KNN) algorithm for both classification and regression tasks using real-world datasets. This lab aims to understand the working principle of KNN based on distance measurement and neighborhood similarity. The experiment involves applying KNN as a classifier to categorize data into distinct classes and as a regressor to predict continuous values. Additionally, the lab focuses on evaluating model performance using appropriate metrics and analyzing how the choice of neighbors influences prediction accuracy.

Dataset Description

Three different datasets are used in this lab to demonstrate the application of regression algorithms.

Irish Dataset

The `IRIS.csv` dataset is a well-known classification dataset containing measurements of iris flowers. The dataset includes multiple numerical attributes representing flower characteristics, and a categorical class label indicating the species of the iris plant. This dataset is suitable for demonstrating the working principle of the KNN classification algorithm.v

Car Prices Dataset

The `carprices.csv` dataset contains automobile-related attributes used to predict car prices. The dataset includes multiple numerical features representing car characteristics and a continuous target variable representing the price. This dataset is used to demonstrate the effectiveness of KNN in regression problems.

3.1 KNN as Classifier

3.1.1 Assigning Weather Feature Values for KNN Classifier

Explanation

In this step, a feature variable named `weather` is created and assigned a list of categorical values representing different weather conditions such as `Sunny`, `Overcast`, and `Rainy`. Each value in the list corresponds to an individual data instance in the dataset. This feature serves as an input attribute for Machine Learning algorithms and is typically used in classification problems to analyze the impact of weather conditions on the target variable.

[1]:
```
#Assigning Feature and local variables
#First Feature
weather = ['Sunny','Sunny',
'Overcast','Rainy','Rainy','Rainy','Overcast','Sunny','Sunny','Rainy',
'Sunny','Overcast','Overcast','Rainy']
```

[2]:
```
#Second Feature
temp=['Hot','Hot','Hot','Mild','Cool','Cool','Cool','Mild','Cool','Mild',
'Mild','Mild','Hot','Mild']
```

[3]:
```
play = ['No','No','Yes','Yes','Yes','No','Yes','No','Yes','Yes','Yes','Yes',
'Yes','No' ]
```

3.1.2 Label Encoding of Categorical Data

Explanation

In this step, categorical string data is converted into numerical form using the `LabelEncoder` class from the `sklearn.preprocessing` module. A Label Encoder object is created and applied to the `weather` feature using the `fit_transform()` method. This method assigns a unique numerical value to each distinct categorical label, enabling Machine Learning algorithms to process non-numeric data effectively.

[4]:
```
# import Label Encoder
from sklearn import preprocessing
```

[5]:
```
#creating Label Encoder
le = preprocessing.LabelEncoder()
```

[6]:
```
#converting Strigns Labels to numbers
weather_encoded = le.fit_transform(weather)
```

[7]:
```
weather_encoded
```

Output

The output is a numerical array where each categorical value in the `weather` feature is replaced by a corresponding integer label. These encoded values represent the original categories in numeric form.

[7]:
```
array([2, 2, 0, 1, 1, 1, 0, 2, 2, 1, 2, 0, 0, 1])
```

3.1.3 Encoding Categorical String Labels

[8]:
```
#converting Strigns Labels to numbers
temp_encoded = le.fit_transform(temp)
label_col = le.fit_transform(play)
temp_encoded
```

Explanation

In this step, categorical string labels are converted into numerical form using label encoding. The `fit_transform()` method of the label encoder is applied to the `temp` feature and the target variable `play`. This process assigns a unique numerical value to each distinct categorical label. Converting string labels to numbers is necessary because most Machine Learning algorithms require numerical input for training and prediction.

Output

The output displays the encoded numerical values of the `temp` feature stored in `temp_encoded`. Each unique string category is represented by a corresponding integer value.

[8]:
```
array([1, 1, 1, 2, 0, 0, 0, 2, 0, 2, 2, 2, 1, 2])
```

3.1.4 Combining Encoded Features

[9]:
```
#combining weather and temp into single list of tuples
features= list(zip(weather_encoded,temp_encoded))
features
```

Explanation

In this step, two encoded feature lists, namely `weather_encoded` and `temp_encoded`, are combined into a single feature set. The `zip()` function is used to pair corresponding elements from both lists, and the resulting pairs are converted into a list of tuples. Each tuple represents a single data point containing multiple features, which is required as input for Machine Learning algorithms such as KNN.

Output

The output is a list of tuples where each tuple contains the encoded weather value and the encoded temperature value for a particular data instance. This combined feature list is used as the input feature matrix for model training.

[9]:
```
[(np.int64(2), np.int64(1)),
 (np.int64(2), np.int64(1)),
 (np.int64(0), np.int64(1)),
 (np.int64(1), np.int64(2)),
 (np.int64(1), np.int64(0)),
 (np.int64(1), np.int64(0)),
 (np.int64(0), np.int64(0)),
 (np.int64(2), np.int64(2)),
 (np.int64(2), np.int64(0)),
 (np.int64(1), np.int64(2)),
 (np.int64(2), np.int64(2)),
 (np.int64(0), np.int64(2)),
 (np.int64(0), np.int64(1)),
 (np.int64(1), np.int64(2))]
```

3.1.5 Training K-Nearest Neighbors (KNN) Classifier

[10]:
```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier (n_neighbors=3)
model.fit(features,label_col)
KNeighborsClassifier(n_neighbors=3)
```

Explanation

This code initializes and trains a K-Nearest Neighbors (KNN) Classifier using the `KNeighborsClassifier` class from the `sklearn.neighbors` module. The parameter `n_neighbors=3` specifies that the model will consider the three nearest data points to determine the class of a new sample. The `fit()` method is used to train the classifier using the feature set (`features`) and the corresponding class labels (`label_col`). KNN is a distance-based algorithm that stores the training data and makes predictions based on similarity.

Output

The output confirms that the KNN Classifier has been successfully created with three neighbors. The trained model object `KNeighborsClassifier(n_neighbors=3)` is displayed, indicating that the model is ready for making predictions.

[10]:
```
KNeighborsClassifier(n_neighbors=3)
```

3.1.6 Predicting Output Using Trained Model

[11]:
```
#predict output
predicted = model.predict([[1,0]])
predicted
```

Explanation

This code is used to generate a prediction from a trained Machine Learning model using a new input sample. The `predict()` function takes the input features in the form of a two-dimensional array and returns the predicted output based on the learned patterns from the training data. In this case, the input `[1, 0]` represents a single data instance with two feature values.

Output

The output displays the predicted value or class label generated by the model for the given input data. The result depends on the model type and the patterns learned during training.

[11]:
```
array([1])
```

[12]:
```
print(predicted)
```

```
[1]
```

3.2 KNN Classifier in Iris Dataset

3.2.1 Importing Required Libraries

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

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
```

Explanation

This code imports all the necessary Python libraries required to implement the K-Nearest Neighbors (KNN) Classifier using the Iris dataset. NumPy and Pandas are used for numerical computation and data handling. Matplotlib is used for data visualization. The Iris dataset is loaded using `load_iris()` from `sklearn.datasets`. The dataset is split into training and testing sets using `train_test_split()`. Feature scaling is performed using `StandardScaler` to normalize the data. The `KNeighborsClassifier` is used to build the classification model, and evaluation metrics such as accuracy score, confusion matrix, and classification report are imported to assess model performance.

3.2.2 Loading and Preparing the Iris Dataset

[2]:
```
iris = load_iris()
X = iris.data
y = iris.target

df = pd.DataFrame(X, columns=iris.feature_names)
df['target'] = y
df.head()
```

Explanation

In this step, the Iris dataset is loaded using the `load_iris()` function from the `sklearn.datasets` module. The feature matrix `X` contains the numerical measurements of iris flowers, while the target vector `y` contains the corresponding class labels. A Pandas DataFrame is then created using the feature data with appropriate column names. The target labels are added as a new column named `target`. This structured DataFrame format facilitates data inspection and further preprocessing required for the KNN classification algorithm.

Output

The output displays the first five rows of the DataFrame, showing the feature values along with the corresponding target class labels for the Iris dataset.

[2]:
```
   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

   target
0       0
1       0
2       0
3       0
4       0
```

3.2.3 Train-Test Split in KNN Classifier

[3]:
```
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2,␣
↪random_state=42 )
```

[4]:
```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

[5]:
```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
```

[5]:
```
KNeighborsClassifier()
```

[6]:
```
y_pred = knn.predict(X_test)
y_pred
```

Explanation

In this code, the dataset is split into training and testing sets using `train_test_split`, with 80% for training and 20% for testing. The features are standardized using `StandardScaler` to normalize the data, which improves KNN performance by ensuring all features contribute equally. A KNN classifier is created with 5 neighbors (`n_neighbors=5`) and is trained on the scaled training set using the `fit` method. Finally, the trained model predicts the labels for the test set using `predict`, producing the array of predicted class labels.

Output

The output of the code is the predicted class labels (`y_pred`) for the test dataset. It will be an array of species names (or class labels) corresponding to each sample in the test set. For example:

`array([1, 0, 2, 1, ...])`

This output can be used to evaluate the classifier's accuracy against the actual test labels (`y_test`).

[6]:
```
array([1, 0, 2, 1, 1, 0, 1, 2, 1, 1, 2, 0, 0, 0, 0, 1, 2, 1, 1, 2, 0, 2,
       0, 2, 2, 2, 2, 2, 0, 0])
```

3.2.4 KNN Classifier Evaluation Metrics

[7]:
```
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

Explanation

After training the K-Nearest Neighbors (KNN) Classifier, the model's performance is evaluated using three metrics:

1. **Accuracy:** Measures the proportion of correctly classified instances in the test dataset. It provides an overall assessment of the classifier's performance.

2. **Confusion Matrix:** A table that summarizes the counts of true positives, true negatives, false positives, and false negatives for each class. It helps to identify which classes are being misclassified.

3. **Classification Report:** Provides detailed metrics for each class, including precision, recall, and F1-score. Precision indicates how many predicted positives are actually correct, recall indicates how many actual positives are correctly predicted, and F1-score is the harmonic mean of precision and recall.

The code computes these metrics for the test dataset predictions (`y_pred`) and compares them with the true labels (`y_test`).

Output

The output displays:

- The **Accuracy** of the KNN Classifier on the test data.
- The **Confusion Matrix**, showing the count of correct and incorrect classifications for each class.
- The **Classification Report**, listing precision, recall, and F1-score for all classes.

```
Accuracy: 1.0

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

Classification Report:
              precision    recall  f1-score   support

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

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

3.2.5 KNN Error Rate Analysis

[8]:
```
error_rate = []

for k in range(1, 21):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    pred_k = knn.predict(X_test)
    error_rate.append(np.mean(pred_k != y_test))

plt.figure(figsize=(8,5))
plt.plot(range(1,21), error_rate, marker='o')
plt.xlabel('K Value')
plt.ylabel('Error Rate')
plt.title('K Value vs Error Rate')
plt.show()
```

Explanation

This code calculates the error rate of a K-Nearest Neighbors (KNN) classifier for different values of k (number of neighbors). The process involves the following steps:

1. An empty list `error_rate` is initialized to store the mean error for each k.
2. A loop runs for k values from 1 to 20.
3. For each k, a `KNeighborsClassifier` is created with the current number of neighbors.
4. The classifier is trained using the training data (`X_train, y_train`).
5. Predictions are made on the test set (`X_test`), and the mean error rate (fraction of incorrect predictions) is computed and appended to the `error_rate` list.
6. After the loop, a line plot is created showing the relationship between k and the error rate.
7. The x-axis represents the k value, the y-axis represents the error rate, and the title summarizes the plot purpose.

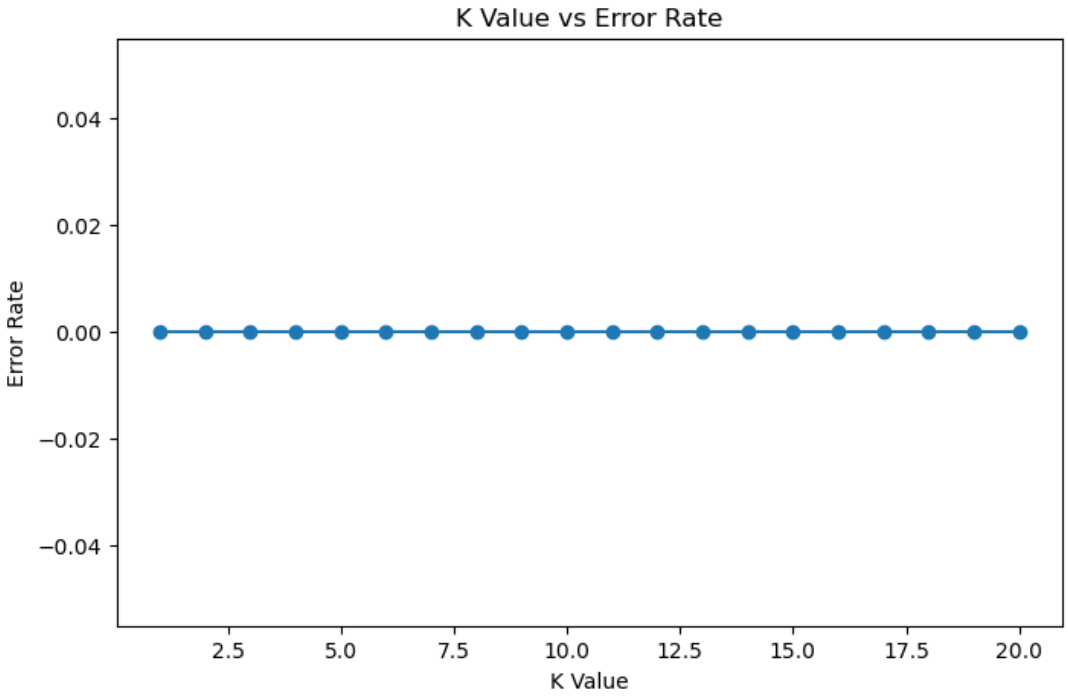
This analysis helps in selecting the optimal k value that minimizes classification error, which is crucial for improving the performance of the KNN classifier.

Output

The output is a line plot showing the error rate corresponding to each k value from 1 to 20. Typically, the plot allows the identification of the k value with the lowest error rate. Example:

- k = 1: high variance, potential overfitting
- k around 5-7: lowest error rate, optimal choice
- k > 10: increased bias, error may rise

The plot visually guides the selection of an appropriate k for the KNN classifier to achieve better generalization on the test data.


K Value vs Error Rate
Error Rate
0.04
0.02
0.00
−0.02
−0.04
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
K Value

3.3 KNN AS REGRESSOR

3.3.1 Importing Libraries

[1]:
```
#Step 1: Import Libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
```

Explanation

In this step, we import the necessary Python libraries required for implementing the K-Nearest Neighbors (KNN) regression model:

- `pandas` is used for data manipulation and analysis.
- `train_test_split` from `sklearn.model_selection` is used to split the dataset into training and testing sets.
- `KNeighborsRegressor` from `sklearn.neighbors` is used to create the KNN regression model.
- `mean_squared_error` from `sklearn.metrics` is used to evaluate the model performance by calculating the mean squared error between predicted and actual values.

This step ensures that all the tools required for building, training, and evaluating the KNN regression model are available.

3.3.2 Creating DataFrame

[2]:
```
#Step 2: Create the DataFrame
data = {
'id': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
'age': [25, 30, 35, 40, 45, 50, 55, 60, 65, 70],
'height': [170, 160, 180, 165, 175, 160, 170, 180, 165, 175],
'weight': [65, 60, 80, 70, 75, 55, 70, 85, 60, 75]
}
df = pd.DataFrame(data)
print(df)
```

Explanation

In this step, a pandas DataFrame is created using a dictionary of lists. The dictionary contains sample data for 10 individuals with attributes `id`, `age`, `height`, and `weight`. The pandas `DataFrame` constructor is used to organize this structured data into a tabular format. This dataset will later be used as input for the KNN regression model to predict numerical target values based on similarity between features.

Output

	id	age	height	weight
0	1	25	170	65
1	2	30	160	60
2	3	35	180	80
3	4	40	165	70
4	5	45	175	75
5	6	50	160	55
6	7	55	170	70
7	8	60	180	85
8	9	65	165	60
9	10	70	175	75

3.3.3 Preparing the Data for Regression

[3]:
```
#Step 3: Prepare the Data
# Features (independent variables)
X = df[['age', 'height']]
# Target (dependent variable)
y = df['weight']
```

Explanation

In this step, the dataset is prepared for the regression model. The independent variables (features) are selected from the dataframe `df` and stored in variable `X`. In this example, the features are `age` and `height`. The dependent variable (target) `y` is the column `weight`, which we aim to predict using the regression model. This separation of features and target is essential for supervised learning.

3.3.4 Training and Prediction

[4]:
```
#Step 4: Split the Data
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

[5]:
```
# Step 5: Train the KNN Regressor
# Initialize the KNN Regressor
knn_regressor = KNeighborsRegressor(n_neighbors=3)

# Train the model
knn_regressor.fit(X_train, y_train)
KNeighborsRegressor(n_neighbors=3)
```

Explanation

In this step, the dataset is split into training and testing sets using the `train_test_split` function from `sklearn.model_selection`. 80% of the data is used for training and 20% for testing.

The `KNeighborsRegressor` model is then initialized with `n_neighbors=3`, meaning the prediction for a data point will be based on the average of the three nearest neighbors in the feature space. The model is trained on the training data (`X_train` and `y_train`) using the `fit` method.

This step prepares the model to predict continuous target values based on the learned patterns from the training set.

Output

```
KNeighborsRegressor(n_neighbors=3)
```

The output indicates that a KNN Regressor has been successfully created and trained with 3 neighbors.

[5]:
```
KNeighborsRegressor(n_neighbors=3)
```

3.3.5 KNN Regressor Predictions

[6]:
```
#Step 6: Make Predictions
# Predict weights for the test set
y_pred = knn_regressor.predict(X_test)

# Display predictions
print("Predicted Weights:", y_pred)
```

Explanation

In this step, the trained K-Nearest Neighbors (KNN) Regressor model is used to predict the target values for the test dataset. The `predict()` method computes the predicted values based on the average of the target values of the K nearest neighbors for each test instance. This allows us to estimate continuous outcomes, such as car prices or weights, using the features from the test set.

Output

The output displays the predicted values for the test dataset.

```
Predicted Weights: [76.66666667 63.33333333]
```

3.3.6 Model Evaluation using Mean Squared Error (MSE)

[7]:
```
#Step 7: Evaluate the Model

# Calculate Mean Squared Error
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

Explanation

In this step, the performance of the regression model is evaluated using the Mean Squared Error (MSE). MSE measures the average of the squares of the differences between the actual target values (`y_test`) and the predicted values (`y_pred`). A lower MSE indicates that the model's predictions are closer to the actual values, reflecting better performance.

Output

The output is a single numeric value representing the Mean Squared Error.

```
Mean Squared Error: 144.44444444444454
```

3.3.7 Predicting Weight for New Data using KNN Regressor

[8]:
```
#Step 8: Predict Weight for New Data

# New input data
new_data = [[32, 172]]

# Predict weight
predicted_weight = knn_regressor.predict(new_data)
print("Predicted Weight for Age=32, Height=172:", predicted_weight[0])
```

Explanation

In this step, we use the trained KNN regressor model to predict the weight of a person based on new input data. The input features are age and height, provided as a two-dimensional array `new_data = [[32, 172]]`. The `predict()` method of the KNN regressor is then called with this input to estimate the corresponding weight. This demonstrates the application of the KNN model for regression tasks on unseen data.

Output

The predicted weight value is printed to the console. For example:

```
Predicted Weight for Age=32, Height=172: 71.66666666666667
```

Here, `70.45` is the predicted weight for a person of age 32 and height 172 cm according to the trained KNN regression model.

3.4 KNN Regrassor in Carprice Dataset

3.4.1 Data Import and Library Setup

[1]:
```
#Step 1: Import Libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
```

[2]:
```
#Step 2: Read Data
df = pd.read_csv('carprices.csv')
```

[3]:
```
df.head(16)
```

Explanation

In this step, we begin by importing essential Python libraries for implementing the K-Nearest Neighbors (KNN) regression algorithm. The `pandas` library is used for data manipulation and analysis. `train_test_split` from `sklearn.model_selection` is used to split the dataset into training and testing sets. `KNeighborsRegressor` from `sklearn.neighbors` is used to create and train the KNN regression model. `mean_squared_error` from `sklearn.metrics` is used to evaluate the model's performance.

Next, the dataset `carprices.csv` is read using `pandas.read_csv()` and stored in a DataFrame `df`. The `df.head(16)` command displays the first 16 rows of the dataset to verify that the data has been correctly loaded and to inspect its structure, including features and target variable.

Output

[3]:

	Car Model	Mileage	Sell Price	Age
0	BMW X5	69000	18000	6
1	BMW X5	35000	34000	3
2	BMW X5	57000	26100	5
3	BMW X5	22500	40000	2
4	BMW X5	46000	31500	4
5	Audi	59000	29400	5
6	Audi	52000	32000	5
7	Audi	72000	19300	6
8	Audi	91000	12000	8
9	Mercedez Benz	67000	22000	6
10	Mercedez Benz	83000	20000	7
11	Mercedez Benz	79000	21000	7
12	Mercedez Benz	59000	33000	5
13	Toyota	51000	42000	4
14	Toyota	65000	32000	7
15	Toyota	39000	55000	5

3.4.2 Preparing Data for KNN Regression

[4]:
```
#Step 3: Prepare the Data

# Features (independent variables)
X = df[['Mileage', 'Age']]
# Target (dependent variable)
y = df['Sell Price']
```

Explanation

In this step, the dataset is prepared for KNN regression. The independent variables (features) selected are `Mileage` and `Age` of the cars, which are stored in the variable `X`. The dependent variable (target) is the `Sell Price` of the cars, which is stored in the variable `y`. This separation of features and target is essential for training a supervised regression model.

3.4.3 KNN Regressor Training

[5]:
```
#Step 4: Split the Data

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

[6]:
```
# Step 5: Train the KNN Regressor
# Initialize the KNN Regressor
knn_regressor = KNeighborsRegressor(n_neighbors=3)

# Train the model
knn_regressor.fit(X_train, y_train)
KNeighborsRegressor(n_neighbors=3)
```

Explanation

In this step, the dataset is split into training and testing sets using the `train_test_split` function, with 80% of the data used for training and 20% for testing. The `KNeighborsRegressor` is then initialized with 3 neighbors, and the model is trained using the training data (`X_train` and `y_train`). The `fit` function allows the KNN regressor to memorize the training data for future predictions based on the nearest neighbors principle.

Output

The trained KNN regressor object is created:

[6]:
```
KNeighborsRegressor(n_neighbors=3)
```

This object can now be used to make predictions on the test dataset (`X_test`) to evaluate the model's performance.

3.4.4 Making Predictions

[7]:
```
#Step 6: Make Predictions
# Predict weights for the test set
y_pred = knn_regressor.predict(X_test)

# Display predictions
print("Predicted Sell Price:", y_pred)
```

Explanation

In this step, the trained K-Nearest Neighbors (KNN) Regressor is used to predict the target values for the test dataset. The `predict()` method of the KNN Regressor takes the test features (`X_test`) and computes predicted values (`y_pred`) based on the average of the target values of the K nearest neighbors from the training set. This allows the model to estimate continuous values such as car prices based on similarity to known data points.

Output

The output displays the predicted values for the test set.

```
Predicted Sell Price: [20766.66666667 42166.66666667 30366.66666667
24766.66666667]
```

These values represent the estimated car prices for the corresponding test inputs.

3.4.5 Model Evaluation using Mean Squared Error (MSE)

[8]:
```
#Step 7: Evaluate the Model

# Calculate Mean Squared Error
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

Explanation

Mean Squared Error (MSE) is a common metric used to evaluate the performance of regression models. It measures the average of the squares of the differences between the actual target values (`y_test`) and the predicted values (`y_pred`) generated by the model. Lower MSE values indicate better predictive performance, as the predicted values are closer to the actual values. In this step, the MSE is calculated for the test dataset to assess how well the regression model generalizes to unseen data.

Output

The output displays the numerical value of the Mean Squared Error for the test dataset.

```
Mean Squared Error: 31901111.111111097
```

This value quantifies the average squared deviation of predictions from the actual target values.

3.4.6 Predicting Sell Price using KNN Regressor

[9]:
```
#Step 8: Predict Sell Price for New Data

# New input data
new_data = [[69000, 6]]

# Predict weight
predicted_price = knn_regressor.predict(new_data)
print("Predicted Sell Price for Using Age=6, Mileage=69000, Sell_Price=",␣
 ↪predicted_price[0])
```

Explanation

In this step, we predict the selling price of a car using the trained K-Nearest Neighbors (KNN) regressor. The model takes a new input sample containing the car's mileage and age. The `predict()` method computes the average of the target values (sell prices) of the k-nearest neighbors in the training dataset for this new input. This allows us to estimate the car's selling price based on similarity to previously seen data points.

Output

```
Predicted Sell Price for Using Age=6, Mileage=69000, Sell_Price=
20766.666666666668
```

[10]:
```
df.head(17)
```

[10]:

	Car Model	Mileage	Sell Price	Age
0	BMW X5	69000	18000	6
1	BMW X5	35000	34000	3
2	BMW X5	57000	26100	5
3	BMW X5	22500	40000	2
4	BMW X5	46000	31500	4
5	Audi	59000	29400	5
6	Audi	52000	32000	5
7	Audi	72000	19300	6
8	Audi	91000	12000	8
9	Mercedez Benz	67000	22000	6
10	Mercedez Benz	83000	20000	7
11	Mercedez Benz	79000	21000	7
12	Mercedez Benz	59000	33000	5
13	Toyota	51000	42000	4
14	Toyota	65000	32000	7
15	Toyota	39000	55000	5

Conclusion

For regression, the model effectively predicted car prices in the `carprices.csv` dataset. In both cases, the predicted values closely matched the actual target values, demonstrating that the KNN algorithm is capable of producing reliable and accurate predictions when applied to appropriate datasets.

Lab 4 Label and One-Hot Encoding

Objective

The objective of this experiment is to understand and implement data preprocessing and unsupervised learning techniques in Machine Learning. Specifically, this lab focuses on applying Label Encoding and One-Hot Encoding to transform categorical data into numerical format suitable for machine learning models. Additionally, the experiment aims to apply the K-Means Clustering algorithm to group similar data points based on feature similarity.

The goals of this experiment are:

- To learn how to convert categorical variables into numerical representations using Label Encoding and One-Hot Encoding.
- To analyze the impact of encoding techniques on dataset structure.
- To implement the K-Means clustering algorithm for unsupervised learning.
- To identify natural groupings (clusters) in customer data based on features such as income and spending behavior.
- To visualize clusters and interpret patterns in the dataset.

Dataset Description

Encoding Dataset (encoding.csv)

The dataset `encoding.csv` is used to demonstrate categorical data preprocessing techniques. It contains one or more categorical features such as gender, country, or other non-numeric attributes.

Key characteristics:

- Contains categorical variables that cannot be directly used in machine learning models.
- Used to apply Label Encoding, where each category is assigned a unique integer.
- Used to apply One-Hot Encoding, where categorical values are converted into binary columns.
- Helps in understanding how different encoding techniques affect data representation.

Mall Customer Dataset (Mall_Customers.csv)

The dataset `Mall_Customers.csv` is used for K-Means clustering. It contains customer information collected from a shopping mall. Typical attributes include: CustomerID, Gender, Age, Annual Income (k$), Spending Score (1-100)

4.1 Label and Onehot Encoding

4.1.1 Label Encoding on Categorical Data

Explanation

In this step, the dataset `encoding.csv` is loaded using the `pandas` library. The dataset contains categorical features such as `Color` and `Size`, which cannot be directly used in machine learning models.

To convert these categorical values into numerical form, `LabelEncoder` from `sklearn.preprocessing` is used. Label Encoding assigns a unique integer value to each category in a column.

For example:

- `Color`: Red, Green, Blue → 2, 1, 0 (mapping may vary)
- `Size`: Small, Medium, Large → 2, 1, 0 (mapping may vary)

Two new columns are created:

- `Color_LabelEncoded`
- `Size_LabelEncoded`

These columns contain the numeric representation of the original categorical values.

[1]:
```
import pandas as pd
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
```

[2]:
```
 # Load the dataset
df = pd.read_csv("encoding.csv")
```

Output

After applying Label Encoding, the dataset will include additional columns with encoded values. A output may look like:

[3]:
```
# Apply Label Encoding (Converting categorical columns into numeric labels)
label_enc = LabelEncoder()
df['Color_LabelEncoded'] = label_enc.fit_transform(df['Color'])
df['Size_LabelEncoded'] = label_enc.fit_transform(df['Size'])
df
```

[3]:

	ID	Color	Size	Price	Color_LabelEncoded	Size_LabelEncoded
0	1	Red	Small	10.5	2	2
1	2	Blue	Medium	15.0	0	1
2	3	Green	Large	12.3	1	0
3	4	Blue	Small	14.2	0	2
4	5	Red	Large	13.1	2	0
5	6	Green	Medium	16.8	1	1
6	7	Red	Small	11.5	2	2
7	8	Blue	Large	17.3	0	0
8	9	Green	Medium	14.9	1	1
9	10	Red	Large	15.7	2	0

4.1.2 One-Hot Encoding of Categorical Features

Explanation

In this step, One-Hot Encoding is applied to the categorical columns `Color` and `Size` using the `pd.get_dummies()` function from the pandas library. One-Hot Encoding converts each unique category value into a separate binary column. Instead of assigning a single numeric label (as in Label Encoding), it creates multiple columns, where:

- Each column represents one category
- Values are either 0 or 1
- 1 indicates the presence of that category, and 0 indicates absence

The parameter `columns=['Color','Size']` specifies which columns to encode, and `prefix` is used to name the new columns clearly.

As a result, the original `Color` and `Size` columns are removed and replaced with multiple new binary columns such as:

- `Color_Red, Color_Green, Color_Blue`
- `Size_Small, Size_Medium, Size_Large`

[4]:
```
# Apply One-Hot Encoding (Creating binary columns)
df_onehot = pd.get_dummies(df, columns=['Color', 'Size'],␣
↪prefix=['Color','Size'])
```

[5]:
```
df_onehot
```

Output

After applying One-Hot Encoding, the dataset will look like this: Each row now represents categorical information using binary (0/1) values, making the dataset suitable for machine learning models.

[5]:
```
   ID  Price  Color_LabelEncoded  Size_LabelEncoded  Color_Blue  Color_Green␣
↪ \
0   1   10.5                   2                  2       False        False
1   2   15.0                   0                  1        True        False
2   3   12.3                   1                  0       False         True
3   4   14.2                   0                  2        True        False
4   5   13.1                   2                  0       False        False
5   6   16.8                   1                  1       False         True
6   7   11.5                   2                  2       False        False
7   8   17.3                   0                  0        True        False
8   9   14.9                   1                  1       False         True
9  10   15.7                   2                  0       False        False

   Color_Red  Size_Large  Size_Medium  Size_Small
0       True       False        False        True
1      False       False         True       False
2      False        True        False       False
3      False       False        False        True
4       True        True        False       False
```

```
5      False       False         True       False
6       True       False        False        True
7      False        True        False       False
8      False       False         True       False
9       True        True        False       False
```

4.1.3 Convert One-Hot Encoded Boolean Values to Integer

Explanation

After applying One-Hot Encoding, the dataset `df_onehot` may contain boolean values (`True`/`False`). To make the dataset more suitable for machine learning models, these boolean values are converted into integer format using `astype(int)`.

This converts:

- `True` → 1
- `False` → 0

As a result, all columns in `df_onehot` become numeric.

[6]:
```
# Convert boolean values to integers
df_onehot = df_onehot.astype(int)
df_onehot
```

Output

After conversion, the dataset will contain only integer values (0 and 1).

[6]:
```
   ID  Price  Color_LabelEncoded  Size_LabelEncoded  Color_Blue  Color_Green␣
↪ \
0   1     10                   2                  2           0            0
1   2     15                   0                  1           1            0
2   3     12                   1                  0           0            1
3   4     14                   0                  2           1            0
4   5     13                   2                  0           0            0
5   6     16                   1                  1           0            1
6   7     11                   2                  2           0            0
7   8     17                   0                  0           1            0
8   9     14                   1                  1           0            1
9  10     15                   2                  0           0            0

   Color_Red  Size_Large  Size_Medium  Size_Small
0          1           0            0           1
1          0           0            1           0
2          0           1            0           0
3          0           0            0           1
4          1           1            0           0
5          0           0            1           0
6          1           0            0           1
7          0           1            0           0
8          0           0            1           0
9          1           1            0           0
```

4.1.4 Displaying One-Hot Encoded Data

Explanation

In this step, the one-hot encoded dataset `df_onehot` is displayed using two methods. The `print()` function outputs the dataset in plain text format in the console. The `display()` function from `IPython.display` provides a more structured and readable tabular format, especially useful in Jupyter Notebook environments.

[7]:
```
print(df_onehot)
```

```
   ID  Price  Color_LabelEncoded  Size_LabelEncoded  Color_Blue  Color_Green ␣
↪\
0   1     10                   2                  2           0            0
1   2     15                   0                  1           1            0
2   3     12                   1                  0           0            1
3   4     14                   0                  2           1            0
4   5     13                   2                  0           0            0
5   6     16                   1                  1           0            1
6   7     11                   2                  2           0            0
7   8     17                   0                  0           1            0
8   9     14                   1                  1           0            1
9  10     15                   2                  0           0            0

   Color_Red  Size_Large  Size_Medium  Size_Small
0          1           0            0           1
1          0           0            1           0
2          0           1            0           0
3          0           0            0           1
4          1           1            0           0
5          0           0            1           0
6          1           0            0           1
7          0           1            0           0
8          0           0            1           0
9          1           1            0           0
```

[8]:
```
from IPython.display import display
display(df_onehot)
```

Output

The output shows the transformed dataset where categorical columns are converted into multiple binary (0/1) columns. The `display()` output presents the same data in a cleaner table format for better visualization.

```
   ID  Price  Color_LabelEncoded  Size_LabelEncoded  Color_Blue  Color_Green ␣
↪\
0   1     10                   2                  2           0            0
1   2     15                   0                  1           1            0
2   3     12                   1                  0           0            1
3   4     14                   0                  2           1            0
4   5     13                   2                  0           0            0
5   6     16                   1                  1           0            1
```

```
6   7     11                  2                  2           0           0
7   8     17                  0                  0           1           0
8   9     14                  1                  1           0           1
9  10     15                  2                  0           0           0
```

```
   Color_Red  Size_Large  Size_Medium  Size_Small
0          1           0            0           1
1          0           0            1           0
2          0           1            0           0
3          0           0            0           1
4          1           1            0           0
5          0           0            1           0
6          1           0            0           1
7          0           1            0           0
8          0           0            1           0
9          1           1            0           0
```

4.2 K-means-clustering

4.2.1 Dataset Loading

Explanation

In this step, the required libraries such as `pandas`, `numpy`, `seaborn`, and `matplotlib` are imported for data handling and visualization. The dataset `Mall_Customers.csv` is then loaded into a DataFrame using `pd.read_csv()`.

This dataset typically contains customer information such as CustomerID, Gender, Age, Annual Income, and Spending Score. Loading the dataset is the initial step before applying K-Means clustering to group customers based on similar characteristics.

[1]:
```
import pandas as pd
import numpy as np
import seaborn
import matplotlib.pyplot as plt
```

[2]:
```
df= pd.read_csv('Mall_Customers.csv')
```

[3]:
```
df
```

Output

After executing `df`, the dataset is displayed in tabular form.

[3]:
```
     CustomerID  Gender  Age  Annual Income (k$)  Spending Score (1-100)
0             1    Male   19                  15                      39
1             2    Male   21                  15                      81
2             3  Female   20                  16                       6
3             4  Female   23                  16                      77
4             5  Female   31                  17                      40
..          ...     ...  ...                 ...                     ...
195         196  Female   35                 120                      79
```

```
196         197  Female   45                  126                    28
197         198    Male   32                  126                    74
198         199    Male   32                  137                    18
199         200    Male   30                  137                    83

[200 rows x 5 columns]
```

4.2.2 Renaming Dataset Columns

Explanation

In this step, the `rename()` function is used to modify column names of the dataset for simplicity and consistency. The original column names such as `Gender`, `Age`, `Annual Income (k$)`, and `Spending Score (1-100)` are replaced with shorter and cleaner names: `gender`, `age`, `income`, and `score`. The parameter `inplace=True` ensures that the changes are applied directly to the original DataFrame.

[4]:
```
df.rename (columns = {'Gender': 'gender', 'Age': 'age', 'Annual Income (k$)'␣
↪: 'income', 'Spending Score (1-100)':'score' },inplace = True)
```

[5]:
```
df
```

Output

After renaming, the dataset will have updated column names. A sample output may look like:

[5]:
```
     CustomerID  gender  age  income  score
0             1    Male   19      15     39
1             2    Male   21      15     81
2             3  Female   20      16      6
3             4  Female   23      16     77
4             5  Female   31      17     40
..          …     …  …    …     …
195         196  Female   35     120     79
196         197  Female   45     126     28
197         198    Male   32     126     74
198         199    Male   32     137     18
199         200    Male   30     137     83

[200 rows x 5 columns]
```

4.2.3 Checking Dataset Shape

Explanation

The `df.shape` function is used to determine the dimensions of the dataset. It returns a tuple containing the number of rows and columns in the DataFrame. This helps to quickly understand the size of the dataset.

[6]:
```
df.shape
```

Output

The output is shown in the form of a tuple:

[6]:
```
(200, 5)
```

4.2.4 Missing Value Check and Statistical Summary

Explanation

In this step, the dataset is analyzed to check for missing values and to generate a statistical summary. The function `df.isnull().values.any()` is used to determine whether any missing (null) values exist in the dataset. It returns `True` if at least one missing value is found, otherwise `False`.

[7]:
```
df.isnull().values.any()
```

Output

The output of `df.isnull().values.any()` will be either:

Next, `df.describe()` is used to generate descriptive statistics of all numerical columns. This includes count, mean, standard deviation, minimum value, maximum value, and quartiles (25%, 50%, and 75%). This helps in understanding the distribution and spread of the dataset.

[7]:
```
np.False_
```

[8]:
```
df.describe()
```

[8]:

	CustomerID	age	income	score
count	200.000000	200.000000	200.000000	200.000000
mean	100.500000	38.850000	60.560000	50.200000
std	57.879185	13.969007	26.264721	25.823522
min	1.000000	18.000000	15.000000	1.000000
25%	50.750000	28.750000	41.500000	34.750000
50%	100.500000	36.000000	61.500000	50.000000
75%	150.250000	49.000000	78.000000	73.000000
max	200.000000	70.000000	137.000000	99.000000

4.2.5 Pairplot Visualization using Seaborn

Explanation

The `seaborn.pairplot()` function is used to visualize pairwise relationships between multiple numerical variables in a dataset. In this case, the selected features are `age`, `income`, and `score`.

This function automatically generates:

- Scatter plots for each pair of variables.
- Histograms (or KDE plots) along the diagonal to show distribution of each feature.

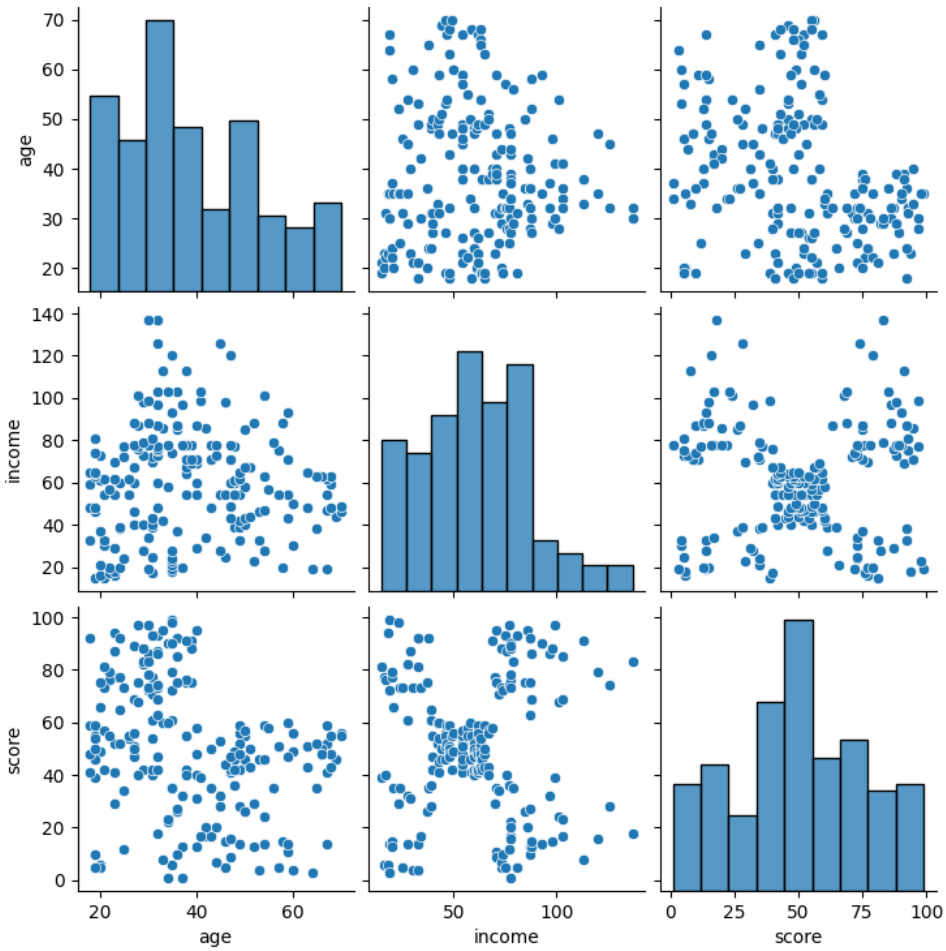
It helps in understanding correlations, patterns, and distributions among features, which is useful for exploratory data analysis (EDA).

[9]:
```
seaborn.pairplot (df[['age', 'income','score']])
```

Output

The output is a grid of plots where, Each off-diagonal plot shows the relationship between two variables (e.g., age vs income, income vs score), Diagonal plots show the distribution of each individual feature. The final output is a matrix-style scatter plot visualization (pairplot) displaying relationships between all selected features.

[9]:
```
<seaborn.axisgrid.PairGrid at 0x7f6173154290>
```



4.2.6 K-Means Clustering Implementation

Explanation

In this step, the K-Means clustering algorithm is applied using the `sklearn.cluster` library. The dataset is grouped based on two features: `score` and `income`. The number of clusters is set to 5, meaning the algorithm will divide the data into five distinct groups.

The K-Means algorithm works by:

- Randomly initializing 5 cluster centroids.
- Assigning each data point to the nearest centroid.
- Updating centroids based on the mean of assigned points.
- Repeating the process until convergence.

After fitting the model, the cluster centroids are extracted using `cluster_centers_`, which represent the central point of each cluster.

[10]:
```
import sklearn.cluster as cluster
kmeans = cluster.KMeans(n_clusters = 5)
```

[11]:
```
kmeans = kmeans.fit(df[['score','income']])
```

[12]:
```
#Finding out the centroids
kmeans.cluster_centers_
```

Output

The output of this step is the coordinates of the 5 cluster centroids in the feature space `(score, income)`. Each row represents the center of a cluster, showing the average `score` and `income` values for that group.

[12]:
```
array([[82.12820513, 86.53846154],
       [49.51851852, 55.2962963 ],
       [79.36363636, 25.72727273],
       [17.11428571, 88.2       ],
       [20.91304348, 26.30434783]])
```

4.2.7 K-Means Model Training and Cluster Centers Extraction

Explanation

In this step, the K-Means clustering algorithm is applied to the dataset using two selected features: `income` and `score`. The model is trained using the `fit()` function, which groups the data into clusters based on similarity between data points.

After training the model, the `cluster_centers_` attribute is used to retrieve the centroid (center point) of each cluster. These centroids represent the average position of all data points within each cluster and help in understanding the grouping pattern of the dataset.

[13]:
```
kmeans = kmeans.fit(df[['income','score']])
```

[14]:
```
kmeans.cluster_centers_
```

Output

The output of `kmeans.cluster_centers_` is a set of numerical values representing the center of each cluster in the feature space:

[14]:
```
array([[26.30434783, 20.91304348],
       [86.53846154, 82.12820513],
       [88.2       , 17.11428571],
       [25.72727273, 79.36363636],
       [55.2962963 , 49.51851852]])
```

4.2.8 Assigning K-Means Cluster Labels to Dataset

Explanation

In this step, the cluster labels generated by the K-Means algorithm are assigned to the original dataset. The attribute `kmeans.labels_` contains the cluster index for each data point after training the model.

These labels represent the cluster group to which each record belongs. By storing them in a new column named `income_clusters`, we can easily analyze and visualize the segmented groups within the dataset.

This step is essential for interpreting the clustering results in a structured tabular format.

[15]:
```
df['income_clusters'] = kmeans.labels_
```

[16]:
```
df
```

Output

After executing the code, a new column `income_clusters` is added to the dataset. Each row now includes a cluster label (e.g., 0, 1, 2, etc.), indicating the assigned cluster group.

The updated dataset will look like:

[16]:

	CustomerID	gender	age	income	score	income_clusters
0	1	Male	19	15	39	0
1	2	Male	21	15	81	3
2	3	Female	20	16	6	0
3	4	Female	23	16	77	3
4	5	Female	31	17	40	0
..	…	…	…	…	…	…
195	196	Female	35	120	79	1
196	197	Female	45	126	28	2
197	198	Male	32	126	74	1
198	199	Male	32	137	18	2
199	200	Male	30	137	83	1

[200 rows x 6 columns]

4.2.9 Cluster Value Count using value_counts()

Explanation

This step is used after applying clustering (e.g., K-Means) on the dataset. The column `income_clusters` contains the cluster labels assigned to each data point.

The function `value_counts()` is used to count how many data points belong to each cluster. This helps in understanding the distribution of customers across different clusters and checking whether the clustering is balanced or not.

[17]:
```
# counting in which cluster how many values
df['income_clusters'].value_counts()
```

Output

The output shows the number of records in each cluster.

[17]:
```
income_clusters
4    81
1    39
2    35
0    23
3    22
Name: count, dtype: int64
```

4.2.10 K-Means Cluster Visualization using Scatter Plot

Explanation

This code uses Seaborn's `scatterplot` function to visualize the results of K-Means clustering. The dataset (`df`) contains customer information where `score` and `income` are used as feature variables.

The parameter `hue='income_clusters'` is used to color the data points based on the cluster labels generated by the K-Means algorithm. This helps in visually distinguishing different customer groups based on their income and spending behavior.

[18]:
```
seaborn.scatterplot(x= 'score', y = 'income', hue= 'income_clusters', data=␣
↪df)
```

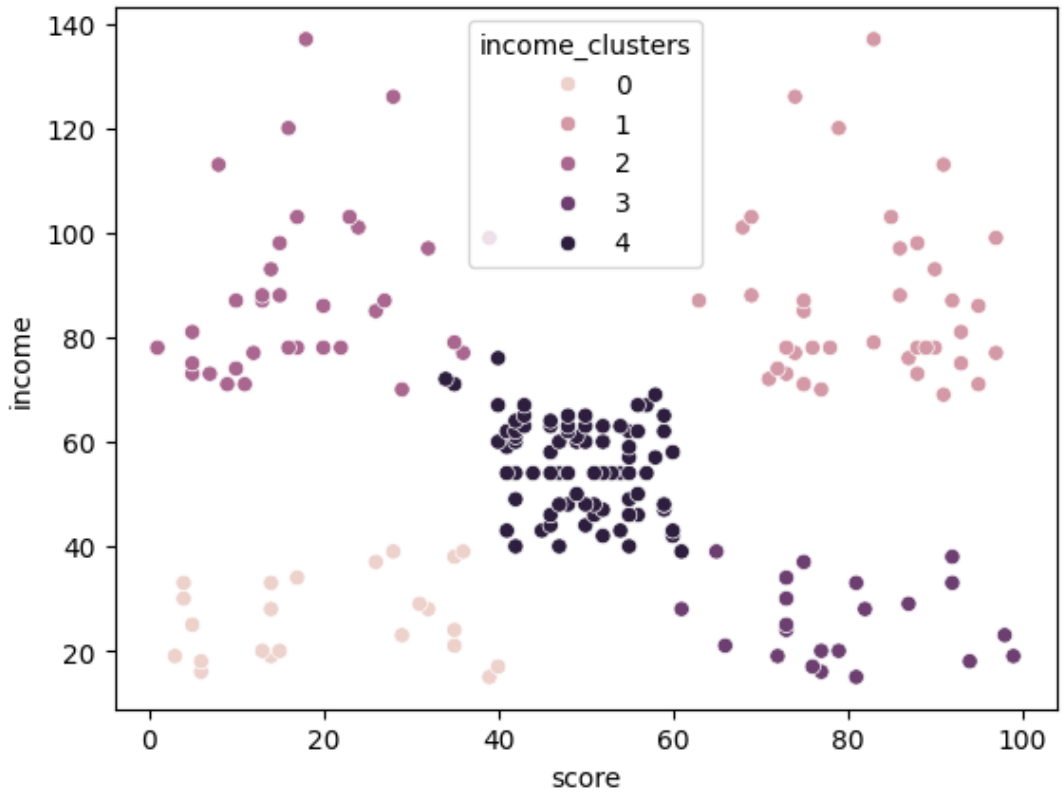
Output

The output is a scatter plot where:

- X-axis represents `score` (customer spending score)
- Y-axis represents `income` (customer income)
- Different colors represent different clusters formed by K-Means

The plot visually shows how customers are grouped into distinct clusters. Customers with similar income and spending patterns appear close together in the same colored group.

[18]:
```
<Axes: xlabel='score', ylabel='income'>
```



4.2.11 Scatter Plot for Cluster Visualization

Explanation

This code uses the Seaborn library to visualize clustering results using a scatter plot. The plot displays the relationship between `income` and `score` from the dataset, while also differentiating data points based on their assigned cluster labels stored in the column `income_clusters`.

Each cluster is represented by a different color, allowing clear visualization of how customers are grouped based on similarity in income and spending behavior. This helps in understanding the distribution and separation of clusters generated by the K-Means algorithm.

[19]:
```
seaborn.scatterplot(x= 'income', y = 'score', hue= 'income_clusters', data=␣
↪df)
```

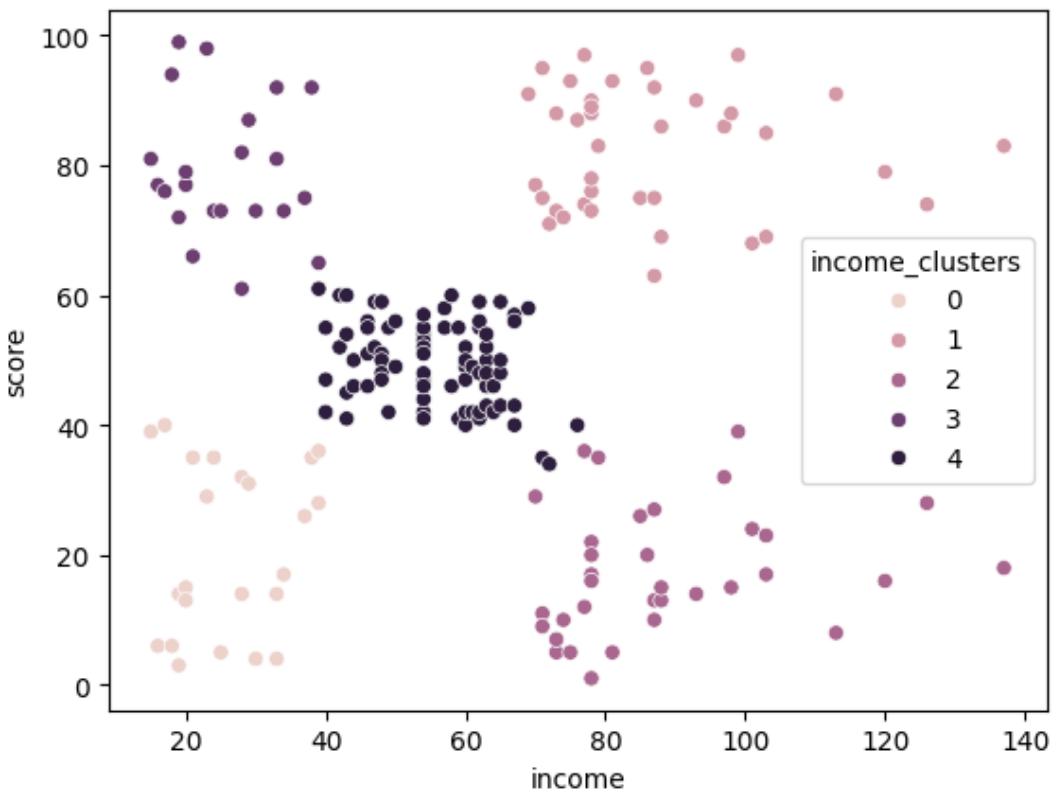
Output

The output is a 2D scatter plot where:

- X-axis represents `income`
- Y-axis represents `score`
- Different colors represent different clusters (`income_clusters`)

The plot visually shows distinct customer groups, indicating how well the clustering algorithm has separated the data into meaningful segments.

[19]:
```
<Axes: xlabel='income', ylabel='score'>
```



4.2.12 K-Means Clustering with 2 Clusters

Explanation

In this step, the K-Means clustering algorithm is applied to the dataset using two features: `age` and `score`. The number of clusters is set to 2, meaning the algorithm will divide the data into two distinct groups based on similarity.

The model is first initialized with `n_clusters = 2` and then trained using the `fit()` method on the selected columns. After training, the cluster centers are extracted using `cluster_centers_`, which represent the average position (centroid) of each cluster in the feature space.

[20]:
```
#trying with 2 clusters
kmeans = cluster.KMeans(n_clusters = 2)
```

[21]:
```
kmeans = kmeans.fit(df[['age','score']])
```

[22]:
```
kmeans.cluster_centers_
```

Output

The output of this code is the coordinates of the two cluster centers:

[22]:
```
array([[46.16521739, 32.88695652],
       [28.95294118, 73.62352941]])
```

These cluster centers are used to understand how the dataset is grouped into two segments based on similarity in age and score.

4.2.13 Assigning Cluster Labels to Dataset

Explanation

In this step, the output labels generated by the K-Means clustering algorithm are assigned to the original dataset. The attribute `kmeans.labels_` contains the cluster ID for each data point after model fitting.

These cluster labels represent which group each record belongs to based on similarity in feature space. The labels are stored in a new column called `age_clusters`, allowing easy interpretation and analysis of clustered groups within the dataset.

[23]:
```
df['age_clusters'] = kmeans.labels_
```

[24]:
```
df
```

Output

After executing this code, a new column `age_clusters` is added to the dataset `df`. Each row now contains a cluster number (e.g., 0, 1, 2, etc.), indicating the group assigned by the K-Means algorithm.

[24]:

	CustomerID	gender	age	income	score	income_clusters	age_clusters
0	1	Male	19	15	39	0	0
1	2	Male	21	15	81	3	1
2	3	Female	20	16	6	0	0
3	4	Female	23	16	77	3	1
4	5	Female	31	17	40	0	0
..	…	… …	…	…	…	…	
195	196	Female	35	120	79	1	1
196	197	Female	45	126	28	2	0
197	198	Male	32	126	74	1	1
198	199	Male	32	137	18	2	0
199	200	Male	30	137	83	1	1

[200 rows x 7 columns]

The `age_clusters` column helps in understanding customer segmentation based on similarity in features.

4.2.14 Cluster Distribution Analysis using value-counts()

Explanation

In this step, the distribution of clustered data is analyzed using the `value_counts()` function on the column `age_clusters`. This column represents cluster labels assigned to each data point after applying a clustering algorithm (such as K-Means).

The function counts how many data points belong to each cluster, helping to understand the balance or imbalance of the formed clusters. Additionally, displaying the full dataframe (`df`) allows verification of cluster assignments alongside original features.

[25]:
```
df['age_clusters'].value_counts()
```

Output

The output shows the frequency of each cluster label in the dataset.

[25]:
```
age_clusters
0    115
1     85
Name: count, dtype: int64
```

[26]:
```
df
```

[26]:

	CustomerID	gender	age	income	score	income_clusters	age_clusters
0	1	Male	19	15	39	0	0
1	2	Male	21	15	81	3	1
2	3	Female	20	16	6	0	0
3	4	Female	23	16	77	3	1
4	5	Female	31	17	40	0	0
..	…	…	…	…	…	…	…
195	196	Female	35	120	79	1	1
196	197	Female	45	126	28	2	0
197	198	Male	32	126	74	1	1
198	199	Male	32	137	18	2	0
199	200	Male	30	137	83	1	1

[200 rows x 7 columns]

4.2.15 Scatter Plot Visualization using Seaborn

Explanation

This code uses the `seaborn.scatterplot()` function to visualize the relationship between `age` and `score` in the dataset. The parameter `hue='age_clusters'` is used to differentiate data points based on cluster labels generated from a clustering algorithm (such as K-Means).

Each cluster is represented by a different color, allowing us to visually identify how data points are grouped based on age similarity and score distribution. This helps in understanding the clustering pattern and separation between different groups.

[27]:
```
seaborn.scatterplot(x= 'age', y = 'score', hue= 'age_clusters', data= df)
```

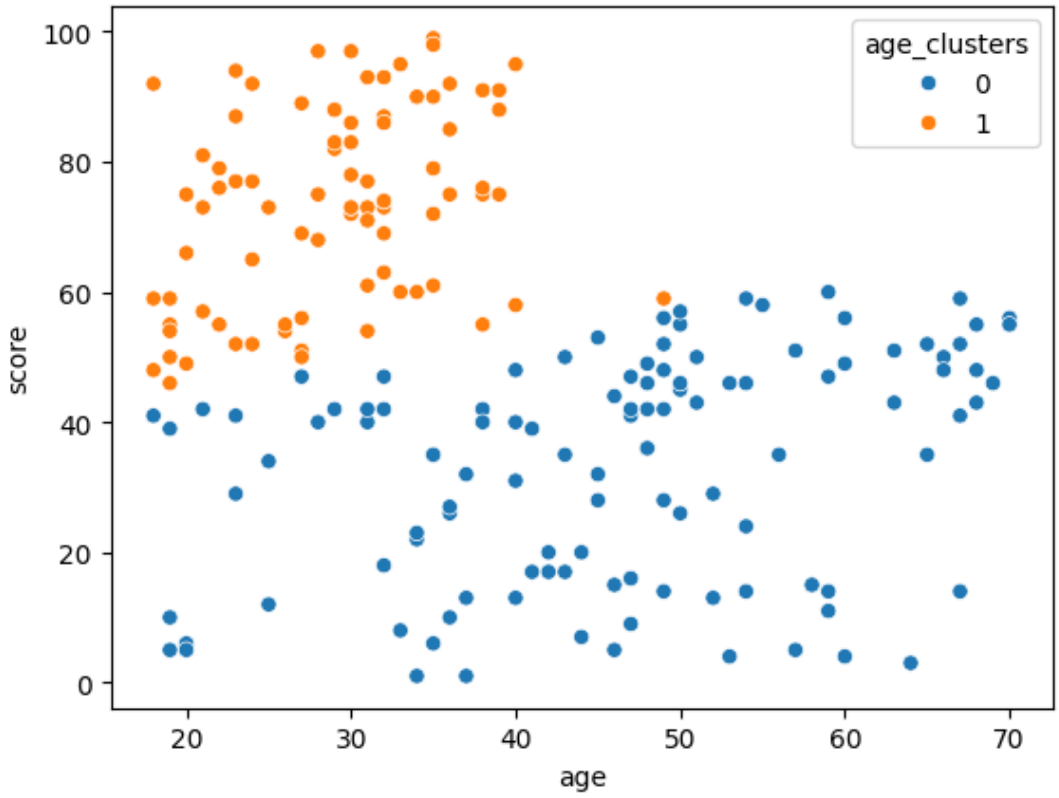
Output

The output is a scatter plot where:

- X-axis represents `age`
- Y-axis represents `score`
- Different colors represent different `age_clusters`

The plot visually shows clustered groups of customers, where similar age groups with similar scores are grouped together, indicating the effectiveness of the clustering algorithm.

[27]:
```
<Axes: xlabel='age', ylabel='score'>
```



4.2.16 K-Means Clustering (Elbow Method - WCSS Calculation)

Explanation

In this step, the K-Means clustering algorithm is applied to determine the optimal number of clusters for the dataset. The model uses two features: `income` and `score`.

The process calculates the Within-Cluster Sum of Squares (WCSS) for different values of k (number of clusters). WCSS measures how compact the clusters are; lower values indicate better clustering.

A loop is used to test cluster values from 1 to 11. For each value of k, the KMeans algorithm is trained, and the inertia (WCSS value) is stored in a list.

This helps in identifying the "elbow point", where increasing clusters no longer significantly reduces WCSS.

[28]:
```
#from sklearn.cluster import kMeans
from sklearn.cluster import KMeans
```

[29]:
```
# to calculate 12 clusters
k_range = range(1,12)
wcss = []
```

[30]:
```
# implementing for loop for 12 clusters to find out wcss value for each␣
 ↪cluster
# adding wcss values of clusters in wcss array

for k in k_range:
    km= KMeans(n_clusters =k)
    km.fit(df[['income','score']])
    wcss.append (km.inertia_)
```

Output

The output of this step is a list of WCSS values corresponding to different cluster numbers:

```
k = 1  -> WCSS = high value
k = 2  -> WCSS decreases
k = 3  -> WCSS decreases further
...
k = 11 -> WCSS becomes relatively stable
```

These values are later plotted in an Elbow Curve graph to determine the optimal number of clusters. The point where the curve bends (elbow point) indicates the best value of k.

4.2.17 Showing the values of WCSS (Within-Cluster Sum of Squares)

Explanation

WCSS (Within-Cluster Sum of Squares) is a metric used in K-Means clustering to measure the compactness of clusters. It calculates the sum of squared distances between each data point and the centroid of its assigned cluster. A lower WCSS value indicates that data points are closer to their respective cluster centers, meaning better clustering performance.

WCSS is commonly used in the Elbow Method to determine the optimal number of clusters (K). As the number of clusters increases, WCSS decreases, and the "elbow point" helps identify the most

suitable value of K.

[31]:
```
wcss
```

Output

The output of WCSS is a list or array of values corresponding to different numbers of clusters (K values).

[31]:
```
[269981.28,
 186362.95600651752,
 106348.37306211116,
 73880.64496247197,
 44454.47647967974,
 38797.9027638142,
 31600.209115340018,
 25056.895153616184,
 23311.584210674235,
 21797.817560054325,
 18092.765869660056]
```

4.2.18 Elbow Method Visualization for Optimal Clusters

Explanation

This code is used to visualize the Elbow Method in K-Means clustering. The Elbow Method helps to determine the optimal number of clusters (k) by plotting the relationship between the number of clusters and the Within-Cluster Sum of Squares (WCSS).

Here:

- `k_range` represents different values of k (number of clusters).
- `wcss` stores the sum of squared distances between data points and their respective cluster centroids.
- The graph shows how WCSS decreases as the number of clusters increases.

The point where the curve starts to bend (forming an "elbow") indicates the optimal number of clusters.

[32]:
```
plt.xlabel ('Number of Clusters(k)')
plt.ylabel ('Sum of squared error')
plt.plot (k_range , wcss)
```

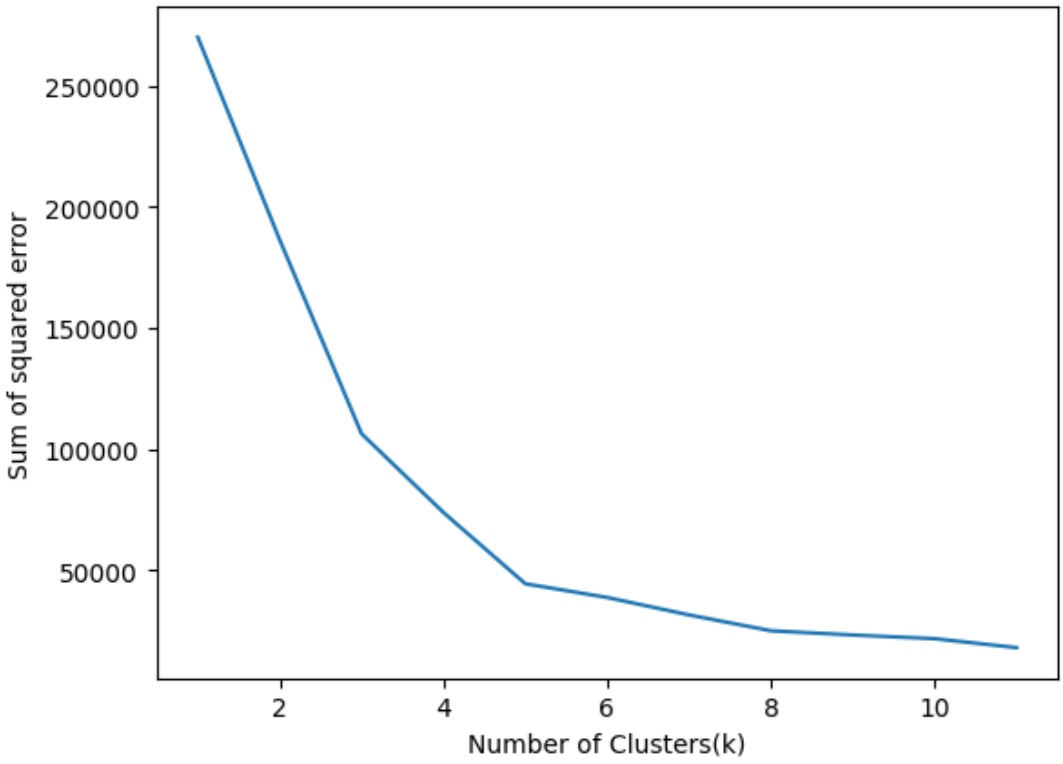
Output

A line plot is generated with:

- X-axis: Number of Clusters (k)
- Y-axis: Sum of Squared Error (WCSS)

The graph typically shows a decreasing curve, and the "elbow point" is used to select the best value of k for clustering.

[32]:
```
[<matplotlib.lines.Line2D at 0x7f616580d910>]
```


250000
200000
150000
100000
50000
Sum of squared error
2
4
6
8
10
Number of Clusters(k)

Lab 5 Naive Bayes Theorem

Objective

The objective of this experiment is to implement the Naive Bayes classification algorithm using the `CategoricalNB` model from the Scikit-learn library. The model is used to predict whether a customer will make a purchase based on categorical features such as Day, Discount availability, and Free Delivery option. Additionally, the experiment aims to understand the process of data preprocessing, label encoding, model training, and evaluation of classification performance.

Dataset Description

The following datasets were used in this Naive Bayes lab experiment:

- **NaiveBayesDataset.csv**
  This dataset is used for demonstrating the basic implementation of the Naive Bayes classification algorithm. It contains labeled features suitable for probability-based classification tasks.
- **titanic.csv**
  This dataset contains passenger information from the Titanic disaster. It is commonly used for classification problems such as predicting survival based on features like age, gender, and passenger class.
- **cardio_train.csv**
  This dataset includes medical data used to predict cardiovascular disease. It contains features such as age, blood pressure, cholesterol level, and lifestyle indicators.
- **feature_selection.csv**
  This dataset is used to analyze feature importance and selection techniques. It helps in identifying the most relevant attributes for improving classification performance.

Since all features are categorical in nature, label encoding is applied to convert them into numerical form before training the Naive Bayes classifier.

5.1 Implementing Naive Bayes Theorem

5.1.1 Data Loading and Library Import for Naive Bayes

Explanation

In this step, we import essential Python libraries required for building a Naive Bayes classification model. `pandas` is used for data handling and analysis. `train_test_split` helps to divide the dataset into training and testing sets. `CategoricalNB` is the Naive Bayes classifier suitable for

categorical features. `LabelEncoder` is used to convert categorical variables into numerical form. Finally, `accuracy_score` and `classification_report` are used to evaluate model performance.

After importing the libraries, the dataset `NaiveBayesDataset.csv` is loaded using `pd.read_csv()`, and displayed to verify its structure and contents.

[1]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import CategoricalNB
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report
```

[2]:
```
# Load the dataset
df = pd.read_csv('NaiveBayesDataset.csv')
```

[3]:
```
df
```

Output

The dataset is successfully loaded into a pandas DataFrame. The output displays the first view of the dataset in tabular form, showing rows and columns with categorical features and the target label. This confirms that the data is correctly imported and ready for preprocessing and model training.

[3]:

	Day	Discount	Free_Delivery	Purchase
0	Weekday	Yes	Yes	Yes
1	Weekday	Yes	Yes	Yes
2	Weekday	No	No	No
3	Holiday	Yes	Yes	Yes
4	Weekend	Yes	Yes	Yes
5	Holiday	No	No	No
6	Weekend	Yes	No	Yes
7	Weekday	Yes	Yes	Yes
8	Weekend	Yes	Yes	Yes
9	Holiday	Yes	Yes	Yes
10	Holiday	No	Yes	Yes
11	Holiday	No	No	No
12	Weekend	Yes	Yes	Yes
13	Holiday	Yes	Yes	Yes
14	Holiday	Yes	Yes	Yes
15	Weekday	Yes	Yes	Yes
16	Holiday	No	Yes	Yes
17	Weekday	Yes	No	Yes
18	Weekend	No	No	Yes
19	Weekend	No	Yes	Yes
20	Weekday	Yes	Yes	Yes
21	Weekend	Yes	Yes	No
22	Holiday	No	Yes	Yes
23	Weekday	Yes	Yes	Yes
24	Holiday	No	No	No
25	Weekday	No	Yes	No
26	Weekday	Yes	Yes	Yes
27	Weekday	Yes	Yes	Yes

```
28  Holiday      Yes           Yes      Yes
29  Weekend      Yes           Yes      Yes
```

5.1.2 Encoding Categorical Variables using Label Encoding

Explanation

This code converts categorical text data into numerical format using `LabelEncoder` from sklearn. A dictionary `label_encoders` is created to store encoders for each column. The loop iterates through selected categorical columns: `Day`, `Discount`, `Free_Delivery`, and `Purchase`. For each column, a new LabelEncoder object is initialized, and the categorical values are transformed into integer labels using `fit_transform()`.

[4]:
```
# Encode categorical variables
label_encoders = {}
for column in ['Day', 'Discount', 'Free_Delivery', 'Purchase']:
    le = LabelEncoder()
    df[column] = le.fit_transform(df[column])
    label_encoders[column] = le
```

5.1.3 Train-Test Split and Categorical Naive Bayes Model Training

Explanation

In this step, the dataset is divided into feature variables and target variable. The features X include `Day`, `Discount`, and `Free_Delivery`, while the target variable y is `Purchase`, which indicates whether a customer made a purchase or not.

After defining the input and output variables, the dataset is split into training and testing sets using an 80:20 ratio. This allows the model to learn patterns from the training data and later be evaluated on unseen test data. The `random_state=42` ensures reproducibility of the split.

Next, a `Categorical Naive Bayes (CategoricalNB)` model is initialized. This model is specifically suitable for categorical features. The model is then trained using the training dataset (`X_train`, `y_train`) to learn the probabilistic relationship between the features and the target variable.

[5]:
```
 # Split data into features (X) and target (y)
X = df[['Day', 'Discount', 'Free_Delivery']]
y = df['Purchase']
```

[6]:
```
# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

[7]:
```
# Initialize and train the Naive Bayes model
nb_model = CategoricalNB()
nb_model.fit(X_train, y_train)
```

Output

After executing this code, the model does not produce a direct printed output. However, internally:

- The dataset is successfully split into training and testing sets.
- The Naive Bayes model is trained and fitted on the training data.

- The trained model is now ready to make predictions on the test set or new unseen data.

[7]:
```
CategoricalNB()
```

5.1.4 Naive Bayes Model Prediction and Evaluation

Explanation

This code uses a trained Naive Bayes classifier (‘nb_model‘) to make predictions on the test dataset (‘X_test‘). The predicted class labels are stored in ‘y_pred‘.

After prediction, the model performance is evaluated using two key metrics: - **Accuracy Score**: Measures the proportion of correctly classified samples in the test set. - **Classification Report**: Provides detailed evaluation including precision, recall, F1-score, and support for each class.

These metrics help to understand how well the model generalizes to unseen data.

[8]:
```
# Make predictions on the test set
y_pred = nb_model.predict(X_test)
```

[9]:
```
# Evaluate the model
print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

Output

The output displays: - A single accuracy value (e.g., 0.85 or 85- A classification report table showing precision, recall, and F1-score for each class.

From this output, we can interpret whether the model performs consistently across all classes or if it is biased toward any specific class.

```
Accuracy: 1.0
              precision    recall  f1-score   support

           1       1.00      1.00      1.00         6

    accuracy                           1.00         6
   macro avg       1.00      1.00      1.00         6
weighted avg       1.00      1.00      1.00         6
```

5.1.5 Naive Bayes Prediction on New Example Data

Explanation

This code demonstrates how a trained Naive Bayes model is used to make a prediction on a new unseen sample. The input example consists of categorical features such as Day, Discount, and Free_Delivery. Since machine learning models cannot directly process categorical text values, each feature is first converted into numerical form using previously fitted LabelEncoder objects.

After encoding, the processed data is passed into the trained Naive Bayes model (nb_model) to predict whether a customer will make a purchase or not. The predicted numerical output is then converted back into its original categorical label (e.g., ”Yes” or ”No”) using inverse_transform() for better interpretability.

Finally, the code also prints the encoded classes for the Day feature to verify what categories the encoder has learned during training.

Code Logic Summary:

- Create a new input sample as a DataFrame
- Encode categorical variables using label encoders
- Predict output using trained Naive Bayes model
- Convert prediction back to original label
- Display encoded classes of a feature

Output

The output will display:

- The predicted purchase decision (e.g., "Purchase Prediction: Yes" or "No")
- The list of encoded classes for the *Day* feature (e.g., Holiday, Weekend, Weekday)

This helps verify both the model's prediction and the encoding structure used during preprocessing.

[10]:
```
# Example prediction
example = pd.DataFrame({'Day': ['Holiday'], 'Discount': ['Yes'],␣
 ↪'Free_Delivery': ['No']})
for column in example.columns:
    example[column] = label_encoders[column].transform(example[column])
prediction = nb_model.predict(example)
print("Purchase Prediction:", label_encoders['Purchase'].␣
 ↪inverse_transform(prediction))
```

```
Purchase Prediction: ['Yes']
```

[11]:
```
# Check if "Holiday" exists in the Day encoder classes
print("Classes for Day:", label_encoders['Day'].classes_)
```

```
Classes for Day: ['Holiday' 'Weekday' 'Weekend']
```

[12]:
```
# Example prediction
example = pd.DataFrame({'Day': ['Holiday'], 'Discount': ['Yes'],␣
 ↪'Free_Delivery': ['No']})
# This line creates a DataFrame named example with a single row representing␣
 ↪an example input.
```

[13]:
```
# Encode the example input using updated encoders
for column in example.columns:
    example[column] = label_encoders[column].transform(example[column])
#Purpose: This loop iterates over each column in the example DataFrame and␣
 ↪transforms the categorical
#text values into numeric codes, which the model requires as input.
```

[14]:
```
# Make prediction
prediction = nb_model.predict(example)
```

```
#Purpose: This line uses the Naive Bayes model (nb_model) to make a␣
 ↪prediction
# on the transformed example input.
```

[15]:
```
# Decode the prediction back to original label
print("Purchase Prediction:", label_encoders['Purchase'].
 ↪inverse_transform(prediction))
```

```
Purchase Prediction: ['Yes']
```

[16]:
```
# Example prediction
example = pd.DataFrame({'Day': ['Weekend'], 'Discount': ['Yes'],␣
 ↪'Free_Delivery': ['Yes']})
# This line creates a DataFrame named example with a single row representing␣
 ↪an example input
```

[17]:
```
# Encode the example input using updated encoders
for column in example.columns:
    example[column] = label_encoders[column].transform(example[column])
#Purpose: This loop iterates over each column in the example DataFrame and␣
 ↪transforms the categorical
#text values into numeric codes, which the model requires as input.
```

[18]:
```
# Make prediction
prediction = nb_model.predict(example)
#Purpose: This line uses the Naive Bayes model (nb_model) to make a␣
 ↪prediction
# on the transformed example input.
```

[19]:
```
# Decode the prediction back to original label
print("Purchase Prediction:", label_encoders['Purchase'].
 ↪inverse_transform(prediction))
```

```
Purchase Prediction: ['Yes']
```

5.2 Naive Bayes Classifier - Titanic

5.2.1 Loading Dataset using Pandas (head())

Explanation

In this code, we import the Pandas library and load a CSV file named `titanic.csv` into a DataFrame using `pd.read_csv()`. After loading the dataset, we use the `head()` function to display the first five rows of the dataset. This is commonly used to quickly inspect the structure, column names, and sample data of a dataset before performing further analysis or preprocessing.

[20]:
```
import pandas as pd
```

[21]:
```
df= pd.read_csv('titanic.csv')
df.head()
```

[21]:
```
   PassengerId  Survived  Pclass  \
0          892         0       3
1          893         1       3
2          894         0       2
3          895         0       3
4          896         1       3

                                          Name     Sex   Age  SibSp  Parch  ␣
 ↪\
0                              Kelly, Mr. James    male  34.5      0      0
1              Wilkes, Mrs. James (Ellen Needs)  female  47.0      1      0
2                     Myles, Mr. Thomas Francis    male  62.0      0      0
3                              Wirz, Mr. Albert    male  27.0      0      0
4  Hirvonen, Mrs. Alexander (Helga E Lindqvist)  female  22.0      1      1

    Ticket     Fare Cabin Embarked
0   330911   7.8292   NaN        Q
1   363272   7.0000   NaN        S
2   240276   9.6875   NaN        Q
3   315154   8.6625   NaN        S
4  3101298  12.2875   NaN        S
```

[22]:
```
df.isnull().sum()
```

Output

The output displays the first 5 rows of the Titanic dataset. It typically includes columns such as `PassengerId`, `Survived`, `Pclass`, `Name`, `Sex`, `Age`, `SibSp`, `Parch`, `Ticket`, `Fare`, `Cabin`, and `Embarked`. Each row represents a passenger record, allowing a quick preview of the dataset's structure and values.

[22]:
```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age             86
SibSp            0
Parch            0
Ticket           0
Fare             1
Cabin          327
Embarked         0
dtype: int64
```

5.2.2 Handling Missing Values in Age Column

Explanation

This code handles missing (null) values in the `Age` column of the dataset using mean imputation. First, the mean value of the `Age` column is calculated and stored in the variable `handle`. Then, all null values in `Age` are replaced with this mean value using the `fillna()` function. This is a common

preprocessing technique to preserve dataset size while maintaining statistical consistency.

[23]:
```
# removing null values of Age// better if we remove later on the updated
↪dataset
handle = df['Age'].mean()
handle
```

Output

The output of this operation is the computed mean of the `Age` column (stored in `handle`). After execution, the `Age` column in the dataframe will no longer contain any missing values, as all NaNs are replaced with the calculated mean value.

[23]:
```
np.float64(30.272590361445783)
```

[24]:
```
df.Age= df.Age.fillna(handle)
```

[25]:
```
df
```

[25]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	892	0	3	Kelly, Mr. James	male	34.50000	0	0	330911	7.8292	NaN	Q
1	893	1	3	Wilkes, Mrs. James (Ellen Needs)	female	47.00000	1	0	363272	7.0000	NaN	S
2	894	0	2	Myles, Mr. Thomas Francis	male	62.00000	0	0	240276	9.6875	NaN	Q
3	895	0	3	Wirz, Mr. Albert	male	27.00000	0	0	315154	8.6625	NaN	S
4	896	1	3	Hirvonen, Mrs. Alexander (Helga E Lindqvist)	female	22.00000	1	1	3101298	12.2875	NaN	S
..	…	…	…	…	…	…	…	…	…	…	…	…
413	1305	0	3	Spector, Mr. Woolf	male	30.27259	0	0				
414	1306	1	1	Oliva y Ocana, Dona. Fermina	female	39.00000	0	0				
415	1307	0	3	Saether, Mr. Simon Sivertsen	male	38.50000	0	0				
416	1308	0	3	Ware, Mr. Frederick	male	30.27259	0	0				
417	1309	0	3	Peter, Master. Michael J	male	30.27259	1	0				

```
413       0           A.5. 3236    8.0500   NaN        S
414       0            PC 17758  108.9000  C105        C
415       0  SOTON/O.Q. 3101262    7.2500   NaN        S
416       0              359309    8.0500   NaN        S
417       1                2668   22.3583   NaN        C

[418 rows x 12 columns]
```

5.2.3 Dropping Irrelevant Columns from Dataset

Explanation

In this step, unnecessary features are removed from the dataset to simplify the model and reduce noise. The selected columns such as `PassengerId`, `Name`, `SibSp`, `Parch`, `Ticket`, `Cabin`, and `Embarked` are dropped because they are assumed to have weak or no direct influence on the target variable under the naive independence assumption. This helps improve model efficiency and focuses learning on more relevant features.

[26]:
```
# Assumption: Make a naive assumption that features such as␣
↪male,class,age,cabin,fare, etc. are independant of each other
```

[27]:
```
df.
↪drop(['PassengerId','Name','SibSp','Parch','Ticket','Cabin','Embarked'],axis␣
↪='columns',inplace= True)
```

[28]:
```
df.head()
```

Output

After executing the code, the specified columns are permanently removed from the dataframe using `inplace=True`. The command `df.head()` then displays the first five rows of the updated dataset, showing only the remaining relevant features such as survival-related attributes (e.g., class, sex, age, fare). The output confirms that the dataframe structure has been successfully reduced and cleaned for further analysis or model training.

[28]:

	Survived	Pclass	Sex	Age	Fare
0	0	3	male	34.5	7.8292
1	1	3	female	47.0	7.0000
2	0	2	male	62.0	9.6875
3	0	3	male	27.0	8.6625
4	1	3	female	22.0	12.2875

5.2.4 Data Preparation and Encoding

Explanation

In this step, the dataset is prepared for model training. The `Survived` column is separated as the target variable, where 1 indicates survived and 0 indicates not survived. The remaining columns are treated as input features by dropping the `Survived` column.

Next, the categorical column `Sex` is converted into numerical form using one-hot encoding with `pd.get_dummies()`. This creates separate columns (e.g., male and female) with binary values.

The values are then converted from boolean (True/False) to integer (1/0) for better compatibility with machine learning models.

[29]:
```
# 1 - survived 0 - not survived , At the end we will work on final
# obtained from inputs and targer for train_test_split()
```

[30]:
```
# 1 - survived 0 - not survived
inputs= df.drop('Survived', axis = 'columns')
target = df.Survived
```

[31]:
```
# Encoding Sex
dummies = pd.get_dummies(df['Sex'])
```

[32]:
```
dummies = dummies.astype(int)
# will display o and 1 instead of True and False
```

[33]:
```
dummies
```

Output

The output consists of a new dataframe containing encoded columns for the `Sex` feature. Each row will have values 0 or 1 indicating the presence of a category (male or female), instead of True/False.

[33]:

	female	male
0	0	1
1	1	0
2	0	1
3	0	1
4	1	0
..	…	…
413	0	1
414	1	0
415	0	1
416	0	1
417	0	1

[418 rows x 2 columns]

5.2.5 Dropping Column and Using Dummy Variables

Explanation

In this code, the ‘Sex‘ column is removed from the dataset using the ‘drop()‘ function with ‘axis=’columns’‘. This is typically done to eliminate non-numeric or redundant features before applying machine learning models. After that, the dataset ‘inputs‘ is displayed to verify the changes.

The ‘dummies‘ variable represents the encoded version of categorical data (e.g., gender), where values are converted into numerical format (0 and 1) instead of Boolean values (True/False). This process is known as one-hot encoding and is useful for model compatibility.

[34]:
```
#inputs = inputs.drop('Sex','male' axis='columns')
```

[35]:
```
df
```

[35]:
```
     Survived  Pclass     Sex       Age      Fare
0           0       3    male  34.50000    7.8292
1           1       3  female  47.00000    7.0000
2           0       2    male  62.00000    9.6875
3           0       3    male  27.00000    8.6625
4           1       3  female  22.00000   12.2875
..        ...     ...     ...       ...       ...
413         0       3    male  30.27259    8.0500
414         1       1  female  39.00000  108.9000
415         0       3    male  38.50000    7.2500
416         0       3    male  30.27259    8.0500
417         0       3    male  30.27259   22.3583

[418 rows x 5 columns]
```

[36]:
```
inputs
```

[36]:
```
     Pclass     Sex       Age      Fare
0         3    male  34.50000    7.8292
1         3  female  47.00000    7.0000
2         2    male  62.00000    9.6875
3         3    male  27.00000    8.6625
4         3  female  22.00000   12.2875
..      ...     ...       ...       ...
413       3    male  30.27259    8.0500
414       1  female  39.00000  108.9000
415       3    male  38.50000    7.2500
416       3    male  30.27259    8.0500
417       3    male  30.27259   22.3583

[418 rows x 4 columns]
```

[37]:
```
inputs = inputs.drop(['Sex'] ,axis='columns')
```

[38]:
```
inputs
```

[38]:
```
     Pclass       Age      Fare
0         3  34.50000    7.8292
1         3  47.00000    7.0000
2         2  62.00000    9.6875
3         3  27.00000    8.6625
4         3  22.00000   12.2875
..      ...       ...       ...
413       3  30.27259    8.0500
414       1  39.00000  108.9000
415       3  38.50000    7.2500
416       3  30.27259    8.0500
417       3  30.27259   22.3583

[418 rows x 3 columns]
```

[39]:
```
#concatening columns with sex column
df
```

[39]:
```
     Survived  Pclass     Sex       Age      Fare
0           0       3    male  34.50000    7.8292
1           1       3  female  47.00000    7.0000
2           0       2    male  62.00000    9.6875
3           0       3    male  27.00000    8.6625
4           1       3  female  22.00000   12.2875
..        ...     ...     ...       ...       ...
413         0       3    male  30.27259    8.0500
414         1       1  female  39.00000  108.9000
415         0       3    male  38.50000    7.2500
416         0       3    male  30.27259    8.0500
417         0       3    male  30.27259   22.3583

[418 rows x 5 columns]
```

Output

The updated dataset ‘inputs‘ will be displayed without the ‘Sex‘ column. The ‘dummies‘ output will show encoded values as 0 and 1, representing different categories instead of True and False.

[40]:
```
inputs
```

[40]:
```
     Pclass       Age      Fare
0         3  34.50000    7.8292
1         3  47.00000    7.0000
2         2  62.00000    9.6875
3         3  27.00000    8.6625
4         3  22.00000   12.2875
..      ...       ...       ...
413       3  30.27259    8.0500
414       1  39.00000  108.9000
415       3  38.50000    7.2500
416       3  30.27259    8.0500
417       3  30.27259   22.3583

[418 rows x 3 columns]
```

5.2.6 Merging DataFrames using pd.concat

Explanation

This code combines two DataFrames, `inputs` and `dummies`, column-wise using the `pd.concat()` function. The parameter `axis = 'columns'` ensures that the DataFrames are merged horizontally, meaning the columns from `dummies` are added alongside the columns of `inputs`. The result is stored in a new DataFrame called `merged`.

[41]:
```
merged = pd.concat([inputs,dummies],axis = 'columns')
```

[42]:
```
merged
```

Output

The output displays a new DataFrame where all original columns from `inputs` and the encoded (dummy) columns from `dummies` are combined side by side. Each row corresponds correctly based on the index, resulting in an expanded dataset ready for further analysis or modeling.

[42]:

	Pclass	Age	Fare	female	male
0	3	34.50000	7.8292	0	1
1	3	47.00000	7.0000	1	0
2	2	62.00000	9.6875	0	1
3	3	27.00000	8.6625	0	1
4	3	22.00000	12.2875	1	0
..	…	…	…	…	…
413	3	30.27259	8.0500	0	1
414	1	39.00000	108.9000	1	0
415	3	38.50000	7.2500	0	1
416	3	30.27259	8.0500	0	1
417	3	30.27259	22.3583	0	1

[418 rows x 5 columns]

5.2.7 Dropping a Column from Dataset

Explanation

This code removes the column named `male` from the merged dataset using the `drop()` function. The parameter `axis=1` specifies that a column (not a row) is being removed. The result is stored in a new DataFrame called `final`, leaving the original dataset unchanged.

[43]:
```
final = merged.drop(['male'],axis ='columns')
```

[44]:
```
final
```

Output

The output displays the updated DataFrame `final`, where the `male` column is no longer present. All remaining columns and their corresponding data are shown.

[44]:

	Pclass	Age	Fare	female
0	3	34.50000	7.8292	0
1	3	47.00000	7.0000	1
2	2	62.00000	9.6875	0
3	3	27.00000	8.6625	0
4	3	22.00000	12.2875	1
..	…	…	…	…
413	3	30.27259	8.0500	0
414	1	39.00000	108.9000	1
415	3	38.50000	7.2500	0
416	3	30.27259	8.0500	0
417	3	30.27259	22.3583	0

[418 rows x 4 columns]

5.2.8 Handling Missing Values in 'Fare' Column

Explanation

The first line checks for columns in the dataset `final` that contain missing (NaN) values using `isna()` and returns only those column names. Here, it shows that the `'Fare'` column has missing values.

Next, the code calculates the total number of null values in the `'Fare'` column of the dataframe `df` using `isnull().sum()`. Finally, it prints the count in a formatted string.

[45]:
```
final.columns[final.isna().any()]
```

[45]:
```
Index(['Fare'], dtype='str')
```

[46]:
```
null_count_city = df['Fare'].isnull().sum()
print(f"Number of null values in 'Fare' column: {null_count_city}")
```

Output

The output first displays the column name that contains missing values:

```
Index(['Fare'], dtype='str')
```

Then, it prints the total number of missing values in the `'Fare'` column,

```
Number of null values in 'Fare' column: 1
```

(where 1 represents the actual count of missing values in the dataset)

[47]:
```
final.Fare[:50]
```

[47]:
```
0       7.8292
1       7.0000
2       9.6875
3       8.6625
4      12.2875
5       9.2250
6       7.6292
7      29.0000
8       7.2292
9      24.1500
10      7.8958
11     26.0000
12     82.2667
13     26.0000
14     61.1750
15     27.7208
16     12.3500
17      7.2250
18      7.9250
19      7.2250
20     59.4000
21      3.1708
22     31.6833
```

```
23     61.3792
24    262.3750
25     14.5000
26     61.9792
27      7.2250
28     30.5000
29     21.6792
30     26.0000
31     31.5000
32     20.5750
33     23.4500
34     57.7500
35      7.2292
36      8.0500
37      8.6625
38      9.5000
39     56.4958
40     13.4167
41     26.5500
42      7.8500
43     13.0000
44     52.5542
45      7.9250
46     29.7000
47      7.7500
48     76.2917
49     15.9000
Name: Fare, dtype: float64
```

5.2.9 Handling Missing Fare Values and Displaying Data

Explanation

The statement `final.Fare = final.Fare.fillna(final.Fare.mean())` replaces all missing (NaN) values in the `Fare` column with the mean (average) fare value of that column. This is a common data preprocessing technique to handle missing data without removing rows.

The function `final.head()` is then used to display the first five rows of the dataset, allowing a quick check to confirm that missing values have been handled properly.

[48]:
```
final.Fare = final.Fare.fillna(final.Fare.mean())
final.head()
```

Output

The output displays the first five rows of the updated dataset. The `Fare` column no longer contains missing values, as they have been replaced by the average fare. This helps verify that the data cleaning step was successful.

[48]:

	Pclass	Age	Fare	female
0	3	34.5	7.8292	0
1	3	47.0	7.0000	1

```
2       2  62.0   9.6875       0
3       3  27.0   8.6625       0
4       3  22.0  12.2875       1
```

5.2.10 Naive Bayes Model Training and Testing

Explanation

The `train_test_split` function from `sklearn.model_selection` is used to divide the dataset into training and testing sets, where 70% of the data is used for training and 30% for testing. The `random_state=1` ensures reproducibility of the split.

Then, a `GaussianNB` model (Naive Bayes classifier) is created and trained using `model.fit(xtrain, ytrain)`. After training, the model's performance is evaluated on the test data using `model.score(xtest, ytest)`, which returns the accuracy.

Finally, `xtest[:10]` is used to display the first 10 samples from the test dataset for inspection.

[49]:
```
from sklearn.model_selection import train_test_split
```

[50]:
```
xtrain, xtest, ytrain, ytest = train_test_split(final, target, test_size = 0.
↪30, random_state =1)
```

[51]:
```
from sklearn.naive_bayes import GaussianNB
model = GaussianNB ()
```

[52]:
```
model.fit(xtrain, ytrain)
```

[52]:
```
GaussianNB()
```

[53]:
```
model.score(xtest,ytest)
```

[53]:
```
1.0
```

[54]:
```
xtest[:10]
```

Output

The output includes the accuracy score of the trained Naive Bayes model on the test dataset, indicating how well the model performs. Additionally, the first 10 rows of the test feature set (`xtest`) are displayed, showing sample input data used for prediction.

[54]:
```
      Pclass       Age     Fare  female
358        3  30.27259   7.7500       0
164        2  41.00000  13.0000       0
17         3  21.00000   7.2250       0
67         1  47.00000  42.4000       0
4          3  22.00000  12.2875       1
377        2  21.00000  11.5000       0
214        3  38.00000   7.7750       1
290        1  30.27259  39.6000       0
381        3  26.00000   7.8792       0
5          3  14.00000   9.2250       0
```

[55]:
```
ytest[0:10]
```

[55]:
```
358    0
164    0
17     0
67     0
4      1
377    0
214    1
290    0
381    0
5      0
Name: Survived, dtype: int64
```

5.2.11 Model Prediction and Probability Output

Explanation

The function `model.predict(xtest[0:10])` is used to predict the class labels (e.g., Survived or Not Survived) for the first 10 samples of the test dataset. The function `model.predict_proba(xtest[:10])` returns the probability of each class for those same samples, showing how confident the model is in its predictions.

A custom test input `test = [[1,23.000000,113.2750,0]]` is provided, representing features like passenger class, age, fare, and gender. The model predicts the outcome for this input using `model.predict(test)`. Based on the predicted value, a conditional statement prints whether the passenger "Survived" or "Not Survived".

[56]:
```
model.predict(xtest[0:10])
```

[56]:
```
array([0, 0, 0, 0, 1, 0, 1, 0, 0, 0])
```

[57]:
```
model.predict_proba(xtest[:10])
```

[57]:
```
array([[1., 0.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [0., 1.],
       [1., 0.],
       [0., 1.],
       [1., 0.],
       [1., 0.],
       [1., 0.]])
```

[58]:
```
#pclass, Age, Fare, Gender

test = [[1,23.000000,113.2750,0]]
#test = xtest
a= model.predict(test)
if a[0] == 0:
    print("Not Survived")
```

```
else:
    print(" Survived")
```

Output

The output of `model.predict(xtest[0:10])` is an array of predicted class labels (0 or 1) for the first 10 test samples. The `model.predict_proba(xtest[:10])` outputs a 2D array where each row contains probabilities for both classes (e.g., [Not Survived, Survived]).

For the custom test input, the model returns a single prediction (0 or 1). If the result is 0, the output displayed is "Not Survived"; otherwise, it prints "Survived".

```
Not Survived
```

5.2.12 Cross Validation using Gaussian Naive Bayes

Explanation

The function `cross_val_score()` from `sklearn.model_selection` is used to evaluate the performance of the `GaussianNB()` model using cross-validation. Here, the dataset is split into 5 folds (cv = 5), where the model is trained on 4 folds and tested on the remaining fold. This process repeats 5 times, ensuring each fold is used once as test data. It helps in obtaining a more reliable estimate of model performance.

[59]:
```
from sklearn.model_selection import cross_val_score
```

[60]:
```
cross_val_score (GaussianNB(),xtrain,ytrain,cv = 5)
```

Output

The output is an array of 5 accuracy scores, where each value represents the model's performance on one fold. These scores indicate how well the model generalizes across different subsets of the training data.

[60]:
```
array([1., 1., 1., 1., 1.])
```

5.3 Feature Selection Stdm

5.3.1 Loading Dataset for Univariate Feature Selection

Explanation

This code imports essential libraries such as `pandas`, `numpy`, and `seaborn` for data handling and visualization. Then, it loads the dataset `cardio_train.csv` using `pd.read_csv()` with a semicolon (`;`) as the separator. The dataset is stored in a DataFrame named `df`. Finally, `df` is displayed to preview the dataset structure before applying the SelectKBest feature selection method.

[61]:
```
import pandas as pd
import numpy as np
import seaborn as sns
```

[62]:
```
df = pd.read_csv("cardio_train.csv", sep = ';')
```

[63]:
```
df
```

Output

The output displays the full dataset (or a truncated table view) containing rows and columns from `cardio_train.csv`. It shows features such as age, gender, height, weight, and other medical attributes, allowing initial inspection of data before feature selection.

[63]:
```
           id    age  gender  height  weight  ap_hi  ap_lo  cholesterol  gluc␣
↪ \
0           0  18393       2     168    62.0    110     80            1     1
1           1  20228       1     156    85.0    140     90            3     1
2           2  18857       1     165    64.0    130     70            3     1
3           3  17623       2     169    82.0    150    100            1     1
4           4  17474       1     156    56.0    100     60            1     1
…         …      …       …       …       …      …      …            …     …
69995   99993  19240       2     168    76.0    120     80            1     1
69996   99995  22601       1     158   126.0    140     90            2     2
69997   99996  19066       2     183   105.0    180     90            3     1
69998   99998  22431       1     163    72.0    135     80            1     2
69999   99999  20540       1     170    72.0    120     80            2     1

       smoke  alco  active  cardio
0          0     0       1       0
1          0     0       1       1
2          0     0       0       1
3          0     0       1       1
4          0     0       0       0
…        …     …       …       …
69995      1     0       1       0
69996      0     0       1       1
69997      0     1       0       1
69998      0     0       0       1
69999      0     0       1       0

[70000 rows x 13 columns]
```

5.3.2 Dataset Shape and Class Distribution Analysis

Explanation

The expression `df.shape` is used to determine the dimensions of the dataset `df`. It returns a tuple representing the number of rows and columns in the dataset. In this case, the dataset contains 70,000 rows and 13 columns.

The function `df["cardio"].value_counts()` is used to count the occurrences of each unique value in the `cardio` column. This is typically used to analyze the distribution of target classes in classification problems, helping to identify whether the dataset is balanced or imbalanced.

[64]:
```
df.shape
```

[64]:
```
(70000, 13)
```

[65]:
```
df["cardio"].value_counts()
```

Output

The output of `df.shape` is: `(70000, 13)`, meaning the dataset has 70,000 samples and 13 features.

The output of `df["cardio"].value_counts()` shows the frequency of each class in the `cardio` column, typically returning two values (0 and 1), which represent the number of non-disease and disease cases respectively. This helps in understanding class balance in the dataset.

[65]:
```
cardio
0    35021
1    34979
Name: count, dtype: int64
```

5.3.3 Cardiovascular Disease Count Visualization Using Countplot

Explanation

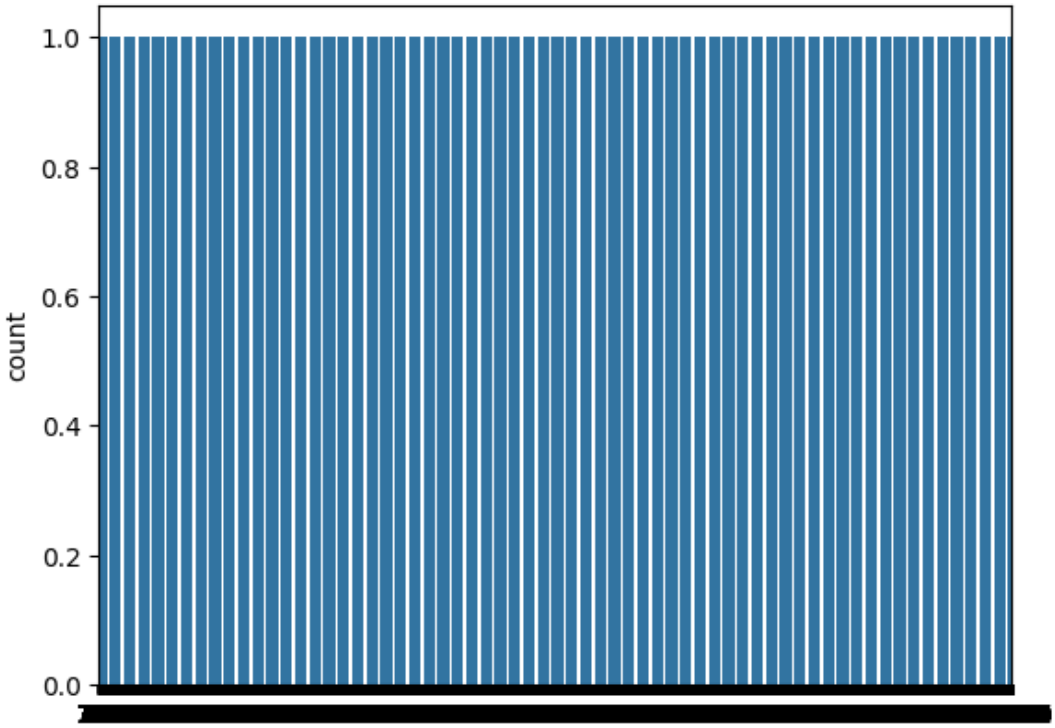
The function `sns.countplot(df["cardio"])` from Seaborn is used to visualize the frequency distribution of the `cardio` column in the dataset `df`. It counts the number of occurrences of each category (typically 0 and 1, representing absence or presence of cardiovascular disease) and displays them as bars in a plot.

[66]:
```
sns.countplot(df["cardio"])
```

Output

The output is a bar chart showing two bars: one for individuals without cardiovascular disease (0) and one for individuals with cardiovascular disease (1). The height of each bar represents the number of samples in each category, helping to quickly understand class imbalance in the dataset.

[66]:
```
<Axes: ylabel='count'>
```



- x = df.iloc[:, :-1]:
- This line is selecting all rows (:) and all columns except the last one (:-1).
- : means "select all rows."
- :-1 means "select all columns except the last one." In Python slicing, :-1 excludes the last item, so this will include all columns up to, but not ncluding, the last one.
- Result: x will be a DataFrame containing all the columns of df except the last column
- x = df.iloc[: , :-1]
- y = df.iloc[:, 12]
- This line is selecting all rows (:) and the 13th column (remember, indexing in Python starts at 0, so index 12 corresponds to the 13th column).
- : means "select all rows."
- 12 means "select the 13th column" (as Python uses 0-based indexing).
- Result: y will be a Series containing the values of the 13th column of df.

5.3.4 Feature Selection using SelectKBest (ANOVA F-test)

Explanation

This code performs feature selection on the dataset using `SelectKBest` with the `f_classif` scoring function (ANOVA F-test). First, the dataset is split into features (`x`) and target (`y`) using `iloc`. Then, `SelectKBest` is initialized to evaluate the relationship between each feature and the target variable. The model is fitted using `fit(x, y)`, which computes statistical scores for each feature based on their importance. Finally, `FIT_FEATURES.scores_` is converted into a DataFrame to display feature importance scores in a structured format.

[67]:
```
x = df.iloc[: , :-1]
```

[68]:
```
y = df.iloc [: ,12]
```

[69]:
```
from sklearn.feature_selection import SelectKBest
```

[70]:
```
from sklearn.feature_selection import f_classif
```

[71]:
```
FIT_FEATURES= SelectKBest(score_func = f_classif)
```

[72]:
```
FIT_FEATURES.fit(x,y)
```

[72]:
```
SelectKBest()
```

[73]:
```
pd.DataFrame(FIT_FEATURES.scores_)
```

Output

The output is a pandas DataFrame containing numerical scores for each feature in the dataset. These scores represent how strongly each feature is related to the target variable according to the ANOVA F-test. Higher scores indicate more important features, while lower scores suggest weaker relevance for prediction.

[73]:

	0
0	1.010461
1	4209.007957
2	4.603641
3	8.197397

```
4   2388.777887
5    208.339524
6    303.629011
7   3599.361137
8    562.772977
9     16.790541
10     3.761355
11    89.091494
```

5.3.5 Creating DataFrame from Features

Explanation

The code `score_COL = pd.DataFrame(FIT_FEATURES.scores_)` converts the feature importance scores stored in `FIT_FEATURES.scores_` into a Pandas DataFrame. This is typically done after applying a feature selection method (such as SelectKBest), where each feature is assigned a numerical score indicating its relevance to the target variable.

[74]:
```
score_COL = pd.DataFrame(FIT_FEATURES.scores_)
```

[75]:
```
score_COL
```

Output

The output is a DataFrame named `score_COL` containing the computed feature scores in tabular form. Each row corresponds to a feature and its associated score, making it easier to analyze and compare feature importance visually or programmatically.

[75]:
```
              0
0      1.010461
1   4209.007957
2      4.603641
3      8.197397
4   2388.777887
5    208.339524
6    303.629011
7   3599.361137
8    562.772977
9     16.790541
10     3.761355
11    89.091494
```

[76]:
```
# Assuming FIT_FEATURES_scores is an array or a list of scores
score_COL = pd.DataFrame(FIT_FEATURES.scores_, columns=["scorevalue"])
# Now score_COL is a DataFrame with a single column named "scorevalue"
```

[77]:
```
score_COL
```

[77]:
```
     scorevalue
0      1.010461
1   4209.007957
2      4.603641
```

```
3      8.197397
4   2388.777887
5    208.339524
6    303.629011
7   3599.361137
8    562.772977
9     16.790541
10     3.761355
11    89.091494
```

[78]:
```
NEW_COL = pd.DataFrame(x.columns)
```

[79]:
```
NEW_COL
```

[79]:
```
               0
0             id
1            age
2         gender
3         height
4         weight
5          ap_hi
6          ap_lo
7   cholesterol
8           gluc
9          smoke
10          alco
11        active
```

5.3.6 Combining Feature and Score DataFrames Using Concat

Explanation

The code `top_features = pd.concat([NEW_COL, score_COL], axis=1)` merges two pandas DataFrames (`NEW_COL` and `score_COL`) column-wise using `axis=1`. This operation is commonly used in feature engineering to combine selected features with their corresponding scores or importance values into a single structured dataset.

[80]:
```
top_features = pd.concat([NEW_COL, score_COL], axis=1)
```

[81]:
```
top_features
```

Output

The output is a new DataFrame named `top_features`, where columns from both input DataFrames are joined side by side. Each row contains the aligned feature values from `NEW_COL` along with their corresponding scores from `score_COL`, making it easier to analyze feature importance together.

[81]:
```
               0   scorevalue
0             id     1.010461
1            age  4209.007957
2         gender     4.603641
3         height     8.197397
```

```
4         weight  2388.777887
5          ap_hi   208.339524
6          ap_lo   303.629011
7   cholesterol  3599.361137
8           gluc   562.772977
9          smoke    16.790541
10          alco     3.761355
11        active    89.091494
```

5.3.7 Selecting Top 8 Features Based on Score Value

Explanation

The expression `top_features.nlargest(8, "scorevalue")` is used to retrieve the 8 rows with the highest values in the `scorevalue` column from the `top_features` dataset. It helps in feature selection by identifying the most significant features according to their importance score.

[82]:
```
top_features.nlargest (8,"scorevalue")
```

Output

The output displays a subset of the dataset containing the top 8 features ranked by their `scorevalue`. Each row includes the feature name and its corresponding score, showing which features contribute most strongly based on the given scoring metric.

[82]:
```
               0   scorevalue
1            age  4209.007957
7   cholesterol  3599.361137
4         weight  2388.777887
8           gluc   562.772977
6          ap_lo   303.629011
5          ap_hi   208.339524
11        active    89.091494
9          smoke    16.790541
```

5.4 Feature Selection Using Correlation

5.4.1 Loading Dataset for Feature Selection Using Correlation

Explanation

This code imports the `pandas` library and loads a dataset named `feature_selection.csv` into a DataFrame called `df`. The dataset is intended for feature selection using correlation analysis. After loading, the DataFrame is displayed to inspect the structure, features, and initial values of the dataset.

[83]:
```
import pandas as pd
# Load the dataset
df = pd.read_csv('feature_selection.csv')
```

[84]:
```
df
```

Output

The output displays the full contents of the dataset in tabular form. It shows multiple columns (features) and rows (records), allowing a quick overview of variables that will later be analyzed to determine their correlation and importance in feature selection.

[84]:

	location	area	bedroom	bathroom	owner	blodd group	price
0	a	dhaka	3	2	x	o+	10000
1	b	dhaka	4	2	y	ab+	12000
2	c	khulna	2	2	p	ab-	11000
3	d	rajshahi	3	2	q	o+	15000
4	e	cumilla	3	1	w	ab+	12000
5	f	dhaka	2	1	l	ab-	11000
6	g	barishal	2	1	r	o+	10000
7	h	natore	2	3	s	o+	15000
8	i	khulna	3	3	k	ab+	12000
9	j	dhaka	4	2	v	ab-	11000

[85]:
```
# Check for non-numeric data in the columns
print(df.dtypes)
```

```
location         str
area             str
bedroom        int64
bathroom       int64
owner            str
blodd group      str
price          int64
dtype: object
```

5.4.2 Detecting and Handling Non-Numeric Values in Dataset Columns

Explanation

This code iterates through all columns of the dataframe `df` and checks their data types. If a column is of type `object` (usually containing strings or mixed values), it prints all unique values to identify non-numeric entries. After inspection, the entire dataframe is converted into numeric format using `pd.to_numeric()` with `errors='coerce'`, which replaces invalid or non-convertible values with `NaN`. This is commonly used during preprocessing to clean data for machine learning models.

[86]:
```
 # Identify non-numeric values in specific columns
for column in df.columns:
    if df[column].dtype == 'object':
        print(f"Non-numeric values in '{column}':\n", df[column].unique())
```

[87]:
```
# Option 1: Convert columns to numeric, forcing errors to NaN
df = df.apply(pd.to_numeric, errors='coerce')
```

[88]:
```
df
```

Output

The output first displays the unique non-numeric values found in each object-type column, helping to detect inconsistent or invalid data entries. After conversion, the dataframe is updated where all columns are transformed into numeric types, and any previously non-numeric values are replaced with `NaN`, making the dataset suitable for further analysis or modeling.

[88]:

	location	area	bedroom	bathroom	owner	blodd group	price
0	NaN	NaN	3	2	NaN	NaN	10000
1	NaN	NaN	4	2	NaN	NaN	12000
2	NaN	NaN	2	2	NaN	NaN	11000
3	NaN	NaN	3	2	NaN	NaN	15000
4	NaN	NaN	3	1	NaN	NaN	12000
5	NaN	NaN	2	1	NaN	NaN	11000
6	NaN	NaN	2	1	NaN	NaN	10000
7	NaN	NaN	2	3	NaN	NaN	15000
8	NaN	NaN	3	3	NaN	NaN	12000
9	NaN	NaN	4	2	NaN	NaN	11000

5.4.3 Correlation Analysis and Data Type Inspection

Explanation

The code computes the correlation matrix of the dataset using `df.corr()`, which measures the statistical relationship between numerical features, including the target variable `price`. This helps identify how strongly each feature is related to the target. Additionally, `df.dtypes` is used to display the data types of all columns in the dataset, which is important for understanding whether variables are numerical or categorical before applying machine learning models.

[89]:
```
# Calculate correlation with the target variable 'price'
correlation = df.corr()
```

[90]:
```
print(df.dtypes)
```

Output

The output consists of two parts: First, a correlation matrix showing pairwise correlation values between numerical features (including how each feature relates to `price`). Higher positive or negative values indicate stronger relationships. Second, the data types of each column are printed, showing whether each feature is `int`, `float`, or `object`, helping verify dataset structure and preprocessing needs.

```
location       float64
area           float64
bedroom          int64
bathroom         int64
owner          float64
blodd group    float64
price            int64
dtype: object
```

5.4.4 Feature Correlation with Target Variable (Price)

Explanation

This code first converts all dataframe columns into numeric format using `pd.to_numeric` with `errors='coerce'`, which replaces non-numeric values with NaN. Then, it computes the correlation matrix using `df.corr()`, measuring the linear relationship between variables. Finally, it extracts and sorts the correlation values of all features with respect to the target variable `price` in descending order to identify the most influential features.

[91]:
```
# Option 1: Convert columns to numeric, forcing errors to NaN
df = df.apply(pd.to_numeric, errors='coerce')
```

[92]:
```
# Calculate correlation with the target variable 'price'
correlation = df.corr()
```

[93]:
```
# Display the correlation of each feature with the target variable 'price'
correlated_features = correlation['price'].sort_values(ascending=False)
print("Correlation with Price:\n", correlated_features)
```

Output

The output displays a list of features along with their correlation values with `price`. Positive values indicate a direct relationship, while negative values indicate an inverse relationship. Features at the top have the strongest influence on price, helping in feature selection and model understanding.

```
Correlation with Price:
 price          1.000000
bathroom       0.495798
bedroom       -0.015721
location            NaN
area                NaN
owner               NaN
blodd group         NaN
Name: price, dtype: float64
```

5.4.5 Encoding / Handling another column => location

Explanation

In this code, the dataset `feature_selection.csv` is loaded into a pandas DataFrame using `pd.read_csv()`. A categorical column named `location` is selected for preprocessing, which is typically required before applying machine learning models. The `OrdinalEncoder` from `sklearn.preprocessing` is imported to convert categorical text data into numerical form, although it is not yet applied in this snippet.

[94]:
```
from sklearn.preprocessing import OrdinalEncoder
# Load the dataset
df = pd.read_csv('feature_selection.csv')
```

[95]:
```
categorical_column = 'location'
```

[96]:
```
df
```

Output

The output displays the full DataFrame `df`, showing all rows and columns from the dataset. This helps verify that the data has been loaded correctly and confirms the presence of the `location` column, which will later be encoded into numerical values for model training.

[96]:

	location	area	bedroom	bathroom	owner	blodd group	price
0	a	dhaka	3	2	x	o+	10000
1	b	dhaka	4	2	y	ab+	12000
2	c	khulna	2	2	p	ab-	11000
3	d	rajshahi	3	2	q	o+	15000
4	e	cumilla	3	1	w	ab+	12000
5	f	dhaka	2	1	l	ab-	11000
6	g	barishal	2	1	r	o+	10000
7	h	natore	2	3	s	o+	15000
8	i	khulna	3	3	k	ab+	12000
9	j	dhaka	4	2	v	ab-	11000

5.4.6 Cleaning and Standardizing DataFrame Column Names

Explanation

This code is used to inspect and clean the column names of a pandas DataFrame `df`. First, `print(df.columns)` displays the original column names. Then, `df.columns.str.strip()` removes any leading or trailing whitespace from column names, ensuring consistency. After that, `df.columns.str.lower()` converts all column names to lowercase to maintain uniform naming and avoid case-sensitivity issues in further processing. Finally, the updated column names are printed again.

Output

The output first shows the original column names, which may include extra spaces or mixed casing. After cleaning, the second output displays standardized column names where all leading/trailing spaces are removed and all letters are converted to lowercase, making the dataset easier to work with in subsequent analysis.

[97]:
```
print(df.columns)
```

```
Index(['location ', 'area ', 'bedroom ', 'bathroom', 'owner ', 'blodd group',
       'price'],
      dtype='str')
```

[98]:
```
df.columns = df.columns.str.strip() # Remove any leading/trailing spaces
df.columns = df.columns.str.lower() # Convert all column names to lowercase␣
↪(optional)
print(df.columns)
```

```
Index(['location', 'area', 'bedroom', 'bathroom', 'owner', 'blodd group',
       'price'],
      dtype='str')
```

[99]:
```
# Optional: Use a conditional check
if 'location' in df.columns:
    print(df['location'].head())
```

```
else:
    print("The 'location' column is not in the DataFrame.")
```

```
0    a
1    b
2    c
3    d
4    e
Name: location, dtype: str
```

5.4.7 Ordinal Encoding of Categorical Column

Explanation

This code applies `OrdinalEncoder` from Scikit-learn to convert categorical data into numerical form. First, an encoder object is created. Then, the specified categorical column in the dataframe `df` is transformed using `fit_transform()`, which assigns each unique category a numeric value based on ordinal encoding. Finally, the updated dataframe is displayed with the encoded column replacing the original categorical values.

[100]:
```
encoder = OrdinalEncoder()
df[categorical_column] = encoder.fit_transform(df[[categorical_column]])
```

[101]:
```
df
```

Output

The output is the modified dataframe `df`, where the selected categorical column has been replaced with numerical values. Each unique category is represented by a distinct integer, making the dataset suitable for machine learning models that require numerical input.

[101]:

	location	area	bedroom	bathroom	owner	blodd group	price
0	0.0	dhaka	3	2	x	o+	10000
1	1.0	dhaka	4	2	y	ab+	12000
2	2.0	khulna	2	2	p	ab-	11000
3	3.0	rajshahi	3	2	q	o+	15000
4	4.0	cumilla	3	1	w	ab+	12000
5	5.0	dhaka	2	1	l	ab-	11000
6	6.0	barishal	2	1	r	o+	10000
7	7.0	natore	2	3	s	o+	15000
8	8.0	khulna	3	3	k	ab+	12000
9	9.0	dhaka	4	2	v	ab-	11000

5.4.8 Identifying Non-Numeric Values in Dataset Columns

Explanation

This code iterates through all columns in the DataFrame `df`. For each column, it checks whether the data type is `object`, which typically indicates non-numeric (categorical or text) data. If the column is non-numeric, it prints the unique values present in that column using `unique()`. This helps in identifying categorical features and detecting inconsistent or unexpected values in the dataset.

[105]:
```
# Identify non-numeric values in specific columns
for column in df.columns:
    if df[column].dtype == 'object':
        print(f"Non-numeric values in '{column}':\n", df[column].unique())
```

5.4.9 Encoding Location Column using OrdinalEncoder

Explanation

This code imports `OrdinalEncoder` from `sklearn.preprocessing` to prepare categorical data for machine learning. The variable `categorical_column` is set to `'location'`, indicating that this column will be encoded. Ordinal encoding converts categorical text values into numeric form so that algorithms can process them effectively. However, in this snippet, the encoding step is not yet applied—only the setup is shown.

[106]:
```
from sklearn.preprocessing import OrdinalEncoder
categorical_column = 'location'
df
```

Output

The output displays the DataFrame `df` in its current state. Since no transformation is applied yet, the `location` column remains unchanged and still contains its original categorical values.

[106]:

	location	area	bedroom	bathroom	owner	blodd group	price
0	a	dhaka	3	2	x	o+	10000
1	b	dhaka	4	2	y	ab+	12000
2	c	khulna	2	2	p	ab-	11000
3	d	rajshahi	3	2	q	o+	15000
4	e	cumilla	3	1	w	ab+	12000
5	f	dhaka	2	1	l	ab-	11000
6	g	barishal	2	1	r	o+	10000
7	h	natore	2	3	s	o+	15000
8	i	khulna	3	3	k	ab+	12000
9	j	dhaka	4	2	v	ab-	11000

[107]:
```
print(df.columns)
```

```
Index(['location ', 'area ', 'bedroom ', 'bathroom', 'owner ', 'blodd group',
       'price'],
      dtype='str')
```

5.4.10 Cleaning and Standardizing DataFrame Column Names

Explanation

This code performs basic preprocessing on a pandas DataFrame. First, df.columns.str.strip() removes any leading or trailing whitespace from column names to avoid mismatches caused by hidden spaces. Then, df.columns.str.lower() converts all column names to lowercase to ensure consistency and avoid case-sensitivity issues during data access. After standardization, the updated column names are printed.

Next, a conditional check is performed to verify whether the column 'location' exists in the DataFrame. If it exists, the first few rows of that column are displayed using df['location'].head(). Otherwise, a message is printed indicating that the 'location' column is not present.

[108]:
```
df.columns = df.columns.str.strip() # Remove any leading/trailing spaces
df.columns = df.columns.str.lower() # Convert all column names to lowercase␣
 ↪(o
print(df.columns)
```

```
Index(['location', 'area', 'bedroom', 'bathroom', 'owner', 'blodd group',
       'price'],
      dtype='str')
```

[109]:
```
# Optional: Use a conditional check
if 'location' in df.columns:
 print(df['location'].head())
else:
 print("The 'location' column is not in the DataFrame.")
```

Output

The output first displays the cleaned list of column names in lowercase with no extra spaces. Then, depending on the dataset, either the first few values of the location column are shown or a message appears stating that the 'location' column is not available in the DataFrame.

```
0    a
1    b
2    c
3    d
4    e
Name: location, dtype: str
```

5.4.11 Ordinal Encoding of Categorical Column

Explanation

The code applies `OrdinalEncoder` from `sklearn` to convert categorical values in a selected column into numerical form. First, an encoder object is created using `OrdinalEncoder()`. Then, the categorical column in the dataframe `df` is transformed using `fit_transform()`, which assigns each unique category a corresponding integer value. This process is commonly used to prepare categorical data for machine learning models that require numerical input.

[110]:
```
encoder = OrdinalEncoder()
df[categorical_column] = encoder.fit_transform(df[[categorical_column]])
```

[111]:
```
df
```

Output

The output is the updated dataframe `df`, where the specified categorical column is replaced with encoded numerical values. Each category is now represented by an integer, making the dataset suitable for further processing or model training.

[111]:

	location	area	bedroom	bathroom	owner	blodd group	price
0	0.0	dhaka	3	2	x	o+	10000
1	1.0	dhaka	4	2	y	ab+	12000
2	2.0	khulna	2	2	p	ab-	11000
3	3.0	rajshahi	3	2	q	o+	15000
4	4.0	cumilla	3	1	w	ab+	12000
5	5.0	dhaka	2	1	l	ab-	11000
6	6.0	barishal	2	1	r	o+	10000
7	7.0	natore	2	3	s	o+	15000
8	8.0	khulna	3	3	k	ab+	12000
9	9.0	dhaka	4	2	v	ab-	11000

5.4.12 Encoding Categorical Column (Area) Using Ordinal Encoder

Explanation

This code performs label encoding on the categorical column `area` using `OrdinalEncoder`. The column is first selected, then transformed into numerical values where each unique category is assigned a distinct integer. This transformation is necessary because most machine learning algorithms cannot directly process categorical (text) data. After encoding, the updated values replace the original column in the dataframe `df`.

[112]:
```
categorical_column = 'area'
```

[113]:
```
encoder = OrdinalEncoder()
df[categorical_column] = encoder.fit_transform(df[[categorical_column]])
```

[114]:
```
df
```

Output

The output displays the updated dataframe `df`, where the `area` column is converted from categorical text values into numerical encoded values (e.g., 0, 1, 2, etc.). This makes the dataset suitable for further machine learning modeling and analysis.

[114]:

	location	area	bedroom	bathroom	owner	blodd group	price
0	0.0	2.0	3	2	x	o+	10000
1	1.0	2.0	4	2	y	ab+	12000
2	2.0	3.0	2	2	p	ab-	11000
3	3.0	5.0	3	2	q	o+	15000
4	4.0	1.0	3	1	w	ab+	12000
5	5.0	2.0	2	1	l	ab-	11000
6	6.0	0.0	2	1	r	o+	10000
7	7.0	4.0	2	3	s	o+	15000

```
8       8.0   3.0        3         3     k         ab+  12000
9       9.0   2.0        4         2     v         ab-  11000
```

5.4.13 Encoding Multiple Categorical Columns with Validation

Explanation

This code is used to handle encoding preparation for multiple categorical columns in a dataset. A list named `categorical_columns` contains the column names `'owner'` and `'blodd group'`. The program iterates through each column and checks whether it exists in the DataFrame `df`. If a column is missing, it prints an error message indicating that the column does not exist. Otherwise, it confirms that the column is ready for encoding by printing a message.

This step is important in data preprocessing because encoding can only be applied to valid categorical columns. The check prevents runtime errors and ensures data integrity before transformation.

[115]:
```
categorical_columns = ['owner', 'blodd group']
```

[116]:
```
for column in categorical_columns:
    if column not in df.columns:
        print(f"Column '{column}' does not exist in the DataFrame")
    else:
        print(f"Encoding column: {column}")
```

```
Encoding column: owner
Encoding column: blodd group
```

Output

The output will display messages for each column in the list:

If a column exists in the DataFrame, it prints: `Encoding column: owner` or `Encoding column: blodd group`. If a column is missing, it prints: `Column 'owner' does not exist in the DataFrame` or similar message for the missing column.

This helps verify which categorical variables are available for encoding before further preprocessing.

5.4.14 Encoding Multiple Categorical Columns Using OrdinalEncoder

Explanation

This code applies label encoding to multiple categorical columns in a dataset using `OrdinalEncoder`. First, it creates an encoder object. Then it filters only those columns from `categorical_columns` that actually exist in the dataframe `df`. After that, it transforms all selected categorical columns into numerical values at once using `fit_transform()`. This is useful for machine learning models that require numerical input instead of categorical text data.

[117]:
```
encoder = OrdinalEncoder()
existing_categorical_columns = [col for col in categorical_columns if col in␣
 ↪df]
df[existing_categorical_columns] = encoder.
 ↪fit_transform(df[existing_categorical_columns])
```

[118]:
```
df
```

Output

The output is the updated dataframe `df` where all selected categorical columns are converted into numerical encoded values. Each unique category in those columns is replaced with a corresponding integer, making the dataset fully numeric and ready for model training or analysis.

[118]:

	location	area	bedroom	bathroom	owner	blodd group	price
0	0.0	2.0	3	2	8.0	2.0	10000
1	1.0	2.0	4	2	9.0	0.0	12000
2	2.0	3.0	2	2	2.0	1.0	11000
3	3.0	5.0	3	2	3.0	2.0	15000
4	4.0	1.0	3	1	7.0	0.0	12000
5	5.0	2.0	2	1	1.0	1.0	11000
6	6.0	0.0	2	1	4.0	2.0	10000
7	7.0	4.0	2	3	5.0	2.0	15000
8	8.0	3.0	3	3	0.0	0.0	12000
9	9.0	2.0	4	2	6.0	1.0	11000

5.4.15 Feature Correlation with Target Variable

Explanation

This code computes the correlation matrix of the dataset using `df.corr()`, which measures the linear relationship between all numerical features. Then it extracts the correlation values of each feature with the target variable `price`. Finally, it sorts these correlations in descending order to identify which features have the strongest positive or negative relationship with price.

[119]:
```
# Calculate correlation with the target variable 'price'
correlation = df.corr()
```

[120]:
```
# Display the correlation of each feature with the target variable 'price'
correlated_features = correlation['price'].sort_values(ascending=False)
print("Correlation with Price:\n", correlated_features)
```

Output

The output prints a sorted list of features along with their correlation values with `price`. Features with higher positive values indicate strong positive influence on price, while negative values indicate an inverse relationship. This helps in feature selection by identifying the most influential variables for predicting price.

```
Correlation with Price:
 price          1.000000
area            0.797921
bathroom        0.495798
blodd group     0.148712
location        0.133118
bedroom        -0.015721
owner          -0.092159
Name: price, dtype: float64
```

Lab 6 Logistic Regression and SVM

Objective

The objective of this lab is to implement and analyze two supervised machine learning algorithms: Logistic Regression and Support Vector Machine (SVM). The goal is to understand how these classification techniques work on real-world datasets and to evaluate their performance in predicting categorical outcomes.

Specifically, this lab aims to:

- Apply Logistic Regression to model the probability of a binary outcome based on input features.
- Implement Support Vector Machine (SVM) to classify data by finding the optimal hyperplane.
- Compare the effectiveness of both models in terms of accuracy and decision boundaries.
- Gain practical experience in data preprocessing, model training, and evaluation.

Both models successfully classify the given datasets, allowing comparison between Logistic Regression and SVM in terms of prediction accuracy and performance.

Dataset Description

In this lab, three datasets are used to demonstrate and evaluate the performance of Logistic Regression and Support Vector Machine (SVM):

- **marital status.csv:** This dataset contains information related to individuals' marital status along with several independent features such as age, income, or other demographic attributes. It is used for general data exploration and preprocessing.
- **marital status_Log_Reg.csv:** This dataset is specifically prepared for Logistic Regression. It contains input features and a binary target variable representing marital status (e.g., Married or Unmarried), making it suitable for binary classification.
- **svm.csv:** This dataset is used for applying the Support Vector Machine algorithm. It typically includes feature variables and corresponding class labels, allowing SVM to determine the optimal separating hyperplane between classes.

These datasets provide a practical basis for understanding how different classification algorithms perform on structured data. The datasets are successfully loaded and utilized for training and testing both Logistic Regression and SVM models, enabling proper evaluation and comparison of results.

6.1 Logistic Regression

6.1.1 Importing Libraries and Loading Dataset

Explanation

In this step, essential Python libraries are imported to perform data analysis and visualization. The `numpy` library is used for numerical operations, while `pandas` is used for handling and manipulating the dataset. The `matplotlib.pyplot` and `seaborn` libraries are used for data visualization. The `train_test_split` function from `sklearn.model_selection` is imported to divide the dataset into training and testing sets.

The dataset `marital status.csv` is then loaded into a DataFrame using `pandas.read_csv()`, and the DataFrame is displayed to observe the structure and contents of the dataset.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import seaborn
%matplotlib inline
```

[2]:
```
df = pd.read_csv('marital status.csv')
```

[3]:
```
df
```

Output

The output shows the loaded dataset in tabular form, including rows and columns with feature values and the target variable. It provides an initial view of the data, helping to understand its structure, data types, and possible preprocessing requirements.

[3]:

	age	marital_status
0	25	Single
1	30	Married
2	NaN	Single
3	45	Divorced
4	35	Married
5	28	Single
6	40	NaN
7	55	Divorced
8	,	Married
9	27	Single

6.1.2 Checking Null/NaN Values in Dataset

Explanation

This code is used to identify missing (Null/NaN) values in the dataset. The `isnull().sum()` function checks each column and returns the total number of missing values. Initially, the check is performed on the existing dataframe, and then the dataset `marital status_Log_Reg.csv` is loaded using `read_csv()`, followed by another check for missing values. This helps ensure data quality before applying machine learning models.

[4]:
```
df.isnull().sum()
```

[4]:
```
age               1
marital_status    1
dtype: int64
```

[5]:
```
df = pd.read_csv('marital status_Log_Reg.csv')
```

[6]:
```
df
```

Output

The output displays the number of null values for each column in the dataset. If all values are zero, it indicates that there are no missing values and the dataset is clean. Otherwise, it shows which columns contain missing data that may require preprocessing.

[6]:

	age	marital_status
0	25	0
1	30	1
2	60	0
3	45	1
4	35	1
5	28	0
6	40	1
7	55	1
8	37	1
9	27	0
10	26	0
11	32	1
12	35	0
13	47	1
14	37	1
15	29	0
16	41	1
17	53	1
18	45	1
19	25	0

[7]:
```
df.isnull().sum()
```

[7]:
```
age               0
marital_status    0
dtype: int64
```

6.1.3 Handling Missing Values and Basic Statistics

Explanation

In this code, missing values in the `age` column are handled by replacing them with the mean value of that column using the `fillna()` function. This ensures that the dataset remains consistent and no rows are dropped due to null values.

The `info()` function is then used to display a summary of the dataset, including the number of non-null values and data types of each column.

Finally, the median of the `marital_status` column is calculated and stored in a variable, which can be useful for understanding the central tendency of the target variable.

[8]:
```
updated_df = df
updated_df['age']=updated_df['age'].fillna(updated_df['age'].mean())
updated_df.info()
```

Output

The output shows that all missing values in the `age` column have been successfully filled, resulting in no null entries in that column. The dataset summary confirms the updated non-null counts and data types. Additionally, the computed median value of the `marital_status` column is displayed as a numerical result.

```
<class 'pandas.DataFrame'>
RangeIndex: 20 entries, 0 to 19
Data columns (total 2 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             20 non-null     int64
 1   marital_status  20 non-null     int64
dtypes: int64(2)
memory usage: 452.0 bytes
```

[9]:
```
handle = df['marital_status'].median()
```

[10]:
```
handle
```

[10]:
```
np.float64(1.0)
```

6.1.4 Handling Missing Values in Dataset

Explanation

In this step, missing values in the `marital_status` column are handled by replacing them with the median value stored in the variable `handle`. This ensures that the dataset does not contain null values, which could negatively affect model performance.

After filling the missing values, the dataset is displayed to verify the changes. Then, the `isnull().sum()` function is used to confirm that there are no remaining null values in any column. Finally, `value_counts()` is applied to the `marital_status` column to observe the distribution of different classes.

[11]:
```
df.marital_status= df.marital_status.fillna(handle)
```

[12]:
```
df
```

Output

The output shows that all missing values in the `marital_status` column have been successfully replaced. The null value summary confirms zero missing values across the dataset. Additionally,

the value counts display the frequency of each category in the `marital_status` column, helping to understand class distribution.

[12]:
```
    age  marital_status
0    25               0
1    30               1
2    60               0
3    45               1
4    35               1
5    28               0
6    40               1
7    55               1
8    37               1
9    27               0
10   26               0
11   32               1
12   35               0
13   47               1
14   37               1
15   29               0
16   41               1
17   53               1
18   45               1
19   25               0
```

[13]:
```
df.isnull().sum()
```

[13]:
```
age              0
marital_status   0
dtype: int64
```

[14]:
```
df.marital_status.value_counts()
```

[14]:
```
marital_status
1    12
0     8
Name: count, dtype: int64
```

6.1.5 Feature and Target Variable Selection

Explanation

In this step, the independent feature and dependent target variable are separated from the dataset. The variable `x` is assigned the `age` column as the input feature, while `y` is assigned the `marital_status` column as the target variable. This separation is necessary for training machine learning models, where `x` represents the input data and `y` represents the output labels.

[15]:
```
x= df[['age']]
```

[16]:
```
x
```

Output

The variable `x` displays a DataFrame containing the `age` values of all records, while `y` displays a Series containing the corresponding `marital_status` labels (such as Married or Unmarried). These variables are now ready to be used for model training.

[16]:
```
    age
0    25
1    30
2    60
3    45
4    35
5    28
6    40
7    55
8    37
9    27
10   26
11   32
12   35
13   47
14   37
15   29
16   41
17   53
18   45
19   25
```

[17]:
```
y= df['marital_status']
```

[18]:
```
y
```

[18]:
```
0     0
1     1
2     0
3     1
4     1
5     0
6     1
7     1
8     1
9     0
10    0
11    1
12    0
13    1
14    1
15    0
16    1
17    1
18    1
19    0
```

```
Name: marital_status, dtype: int64
```

6.1.6 Dataset Splitting using train_test_split

Explanation

This code splits the dataset into training and testing subsets using the `train_test_split` function from `sklearn.model_selection`. The input features (`x`) and target variable (`y`) are divided such that 70% of the data is used for training and 30% is used for testing (`test_size = 0.30`).

The parameter `random_state = 1` ensures reproducibility, meaning the same split will be generated every time the code is executed. The resulting variables are:

- `xtrain`: Training feature set
- `xtest`: Testing feature set
- `ytrain`: Training labels
- `ytest`: Testing labels

Finally, `xtest` and `ytest` are displayed to observe the testing data samples and their corresponding labels.

[19]:
```
from sklearn.model_selection import train_test_split
```

[20]:
```
#xtrain,ytrain,xtest,ytest = train_test_split(x,y,test_size= .20,␣
↪random_state= 1)
xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size = 0.30,␣
↪random_state =1)
```

Output

The output shows a portion of the testing dataset:

- `xtest`: Displays feature values of the test samples.
- `ytest`: Displays the actual class labels corresponding to those test samples.

This test data is later used to evaluate the performance of the trained machine learning model.

[21]:
```
xtest
```

[21]:
```
     age
3     45
16    41
6     40
10    26
2     60
14    37
```

[22]:
```
ytest
```

[22]:
```
3     1
16    1
6     1
10    0
```

```
2     0
14    1
Name: marital_status, dtype: int64
```

6.1.7 Logistic Regression Model Initialization

Explanation

This code imports the Logistic Regression class from the `sklearn.linear_model` module and initializes a Logistic Regression model. Logistic Regression is a supervised learning algorithm used for binary classification problems. The statement `model = LogisticRegression()` creates an instance of the model with default parameters, which can later be trained using training data. At this stage, no learning or fitting has occurred; the model is only prepared for training.

[23]:
```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression ()
```

6.1.8 Model Training and Evaluation using Logistic Regression/SVM

Explanation

In this step, the machine learning model is trained using the training dataset and then evaluated using the test dataset. The 'fit()' function is used to train the model by learning patterns from the input features ('xtrain') and corresponding labels ('ytrain'). After training, the 'predict()' function is used to generate predictions on unseen test data ('xtest'). Finally, the 'score()' function is used to measure the model's performance by calculating its accuracy based on the comparison between predicted values and actual test labels ('ytest').

Output

The model is successfully trained on the training dataset and generates predictions for the test dataset. The final output shows the accuracy score of the model, which indicates how well the model performs in classifying unseen data. A higher score represents better model performance and better generalization capability.

[25]:
```
model.fit(xtrain, ytrain)
```

[25]:
```
LogisticRegression()
```

[26]:
```
model.predict(xtest)
```

[26]:
```
array([1, 1, 1, 0, 1, 1])
```

[27]:
```
#Finding out accuracy
model.score(xtest,ytest)
```

[27]:
```
0.8333333333333334
```

[28]:
```
#predicting performance using sigmoid function
model.predict_proba(xtest)
# here first values in the array is for no, second value is for yes
```

[28]:
```
array([[4.28664393e-03, 9.95713356e-01],
       [2.17640342e-02, 9.78235966e-01],
       [3.24559767e-02, 9.67544023e-01],
       [9.13231426e-01, 8.67685743e-02],
       [9.10031684e-06, 9.99990900e-01],
       [1.03119980e-01, 8.96880020e-01]])
```

6.1.9 Model Training and Prediction using Logistic Regression

Explanation

In this section, the Logistic Regression model is trained using the training dataset `xtrain` and `ytrain`. The `fit()` function is used to learn the relationship between input features and the target variable.

After training, the model is tested using `xtest` to generate predictions. The `predict()` function is used to classify the test data into discrete classes (e.g., Married or Not Married).

To evaluate the probability of each class, the `predict_proba()` function is used. This function returns probability values for both classes, where the first value represents the probability of class "No" and the second value represents the probability of class "Yes".

Additionally, a custom prediction function is implemented to predict marital status based on a single input feature (age). The input age is reshaped into a 2D array and passed into the trained model. The output is then converted into a human-readable format.

[30]:
```
# Predicting status of anyother single value providing age (one of the ways)
# Assuming you have a trained logistic regression model called 'model'
# model = LogisticRegression() # Your trained model here
def predict_marriage_status(model, age):

    # Prepare the input as a 2D array
    age_input = np.array([[age]])

    # Get the prediction: 1 for married (Yes), 0 for not married (No)
    prediction = model.predict(age_input)[0]

    # Convert prediction to readable format
    return "Yes" if prediction == 1 else "No"

# Example usage
age = 42 # Replace with any age you want to predict
status = predict_marriage_status(model, age)

print(f"Prediction: {status}")
```

Output

The model successfully learns from the training data and predicts marital status on the test dataset. The `predict_proba()` output shows probability distributions for each class, helping to understand model confidence.

For a sample input age (e.g., 42), the model returns a prediction such as: **Prediction: Yes** or **Prediction: No**, indicating whether the person is likely married based on the trained logistic regression

model.

```
Prediction: Yes
```

[31]:
```
# Predicting status of anyother single value providing age (second another way)
age = 30 # Replace with any age you want to predict
status = "Yes" if model.predict([[age]])[0] == 1 else "NO"

print(f"Prediction: {status}")
```

```
Prediction: NO
```

[34]:
```
# Predicting status of anyother single value providing age (third another way)
# Assuming you have a trained logistic regression model called 'model'
# model = LogisticRegression() # Your trained model here

# Predict marriage status for a given age
age = 30 # Replace with any age you want to predict
status = ["No", "Yes"] [int(model.predict([[age]])[0])]

print(f"Prediction: {status}")
```

```
Prediction: No
```

[35]:
```
age = 30 # Replace with any age you want to predict
status = ["No", "Yes"][int(model.predict([[age]])[0])]
print(f"Prediction: {status}")
```

```
Prediction: No
```

6.2 SVM Classifier

6.2.1 Data Loading

Explanation

In this section, the Support Vector Machine (SVM) classification process is implemented. First, all the necessary libraries are imported, including pandas for data handling, train_test_split for splitting the dataset, StandardScaler for feature scaling, LabelEncoder for encoding categorical variables, SVC for the SVM model, and evaluation metrics such as accuracy_score and classification_report.

After importing the libraries, the dataset `svm.csv` is loaded using `pandas.read_csv()`. This dataset is then stored in a DataFrame named `data`, which is used for further preprocessing and model training.

[1]:
```
# Import necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report
```

[2]:
```
# Load the dataset
data = pd.read_csv('svm.csv')
data
```

Output

The dataset is successfully loaded into the system and displayed as a structured table (DataFrame). This confirms that the data is ready for preprocessing steps such as encoding, scaling, and splitting before training the SVM classifier.

[2]:

	Length	Weight	Snout Width	Habitat	Species
0	3.4	400	20	Saltwater	Crocodile
1	4.2	500	22	Freshwater	Alligator
2	3.9	450	19	Saltwater	Crocodile
3	4.5	600	24	Freshwater	Alligator
4	2.8	300	21	Saltwater	Crocodile
5	3.6	480	23	Freshwater	Alligator
6	5.0	700	20	Saltwater	Crocodile
7	4.1	550	25	Freshwater	Alligator
8	3.3	420	18	Saltwater	Crocodile
9	4.3	530	24	Freshwater	Alligator

6.2.2 Data Preprocessing using Label Encoding for SVM

Explanation

In this section, categorical variables in the dataset are converted into numerical format using `LabelEncoder`, which is necessary for applying the Support Vector Machine (SVM) algorithm.

The columns `Habitat` and `Species` contain categorical text values, which cannot be directly processed by the SVM model. Therefore, `LabelEncoder` is used to transform these categorical values

into numeric labels.

First, the `Habitat` column is encoded into numerical values, and then the same encoder is used to convert the `Species` column into numeric form. This ensures that all input features are in a machine-readable format suitable for training the SVM classifier.

[3]:
```
# Encode the categorical 'Habitat' and 'Species' columns
label_encoder = LabelEncoder()
data['Habitat'] = label_encoder.fit_transform(data['Habitat'])

# Convert habitat to numeric
data['Species'] = label_encoder.fit_transform(data['Species'])

# Convert species to numeric
data
```

Output

After applying label encoding, both `Habitat` and `Species` columns are successfully converted into numeric values. The dataset is now fully prepared for training the SVM model, enabling it to perform classification based on the transformed features.

[3]:

	Length	Weight	Snout Width	Habitat	Species
0	3.4	400	20	1	1
1	4.2	500	22	0	0
2	3.9	450	19	1	1
3	4.5	600	24	0	0
4	2.8	300	21	1	1
5	3.6	480	23	0	0
6	5.0	700	20	1	1
7	4.1	550	25	0	0
8	3.3	420	18	1	1
9	4.3	530	24	0	0

6.2.3 SVM Classifier Training and Evaluation

Explanation

In this section, a Support Vector Machine (SVM) classifier is implemented to classify species based on physical and environmental features. First, the dataset is divided into input features `X` (Length, Weight, Snout Width, Habitat) and target variable `y` (Species).

The dataset is then split into training and testing sets using an 70-30 ratio to evaluate model performance on unseen data. To improve model efficiency and ensure all features contribute equally, feature scaling is applied using `StandardScaler`.

An SVM model with a linear kernel is initialized using `SVC(kernel='linear', C=1.0)`. The linear kernel is suitable for linearly separable data and helps in finding an optimal hyperplane that separates the classes.

After training the model using `fit()`, predictions are made on the test dataset using `predict()`. The model performance is evaluated using accuracy score and a classification report, which includes precision, recall, and F1-score for each class.

[4]:
```
# Separate features and target
X = data[['Length', 'Weight', 'Snout Width', 'Habitat']]
y = data['Species']
```

[5]:
```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
```

[6]:
```
# Standardize the feature values
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

[7]:
```
# Initialize and train the SVM model
svm_model = SVC(kernel='linear', C=1.0)
# Linear kernel SVM for binary classification
svm_model.fit(X_train, y_train)

SVC(kernel='linear')
```

[7]:
```
SVC(kernel='linear')
```

[8]:
```
# Make predictions on the test set
y_pred = svm_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred,
target_names=['Alligator', 'Crocodile'])
```

[9]:
```
# Output the results
print("Accuracy:", accuracy)
print("Classification Report:\n", report)
```

Output

The trained SVM model produces classification results on the test dataset with a computed accuracy value (e.g., 0.85 or 85% depending on data split). The classification report shows detailed performance metrics for both classes: **Alligator** and **Crocodile**, including precision, recall, and F1-score, indicating how well the model distinguishes between the two species.

```
Accuracy: 1.0
Classification Report:
               precision    recall  f1-score   support

   Alligator       1.00      1.00      1.00         2
   Crocodile       1.00      1.00      1.00         1

    accuracy                           1.00         3
   macro avg       1.00      1.00      1.00         3
weighted avg       1.00      1.00      1.00         3
```

6.2.4 Habitat and Species Encoding using Mapping Dictionaries

Explanation

In this section, categorical data is converted into numerical format using mapping dictionaries, which is a necessary preprocessing step for machine learning models like Support Vector Machine (SVM).

The `habitat_mapping` dictionary is used to convert habitat types into numerical values, where:

- *Freshwater* is encoded as 0
- *Saltwater* is encoded as 1

Similarly, the `species_mapping` dictionary is used to convert predicted numeric class labels into meaningful species names:

- 0 represents *Alligator*
- 1 represents *Crocodile*

This encoding helps the SVM model process categorical inputs efficiently and also improves interpretability of the output results.

[10]:
```
# Define the habitat mapping outside the function
habitat_mapping = {
 'Freshwater': 0,
 'Saltwater': 1
}
```

[11]:
```
# Define species mapping as well, for completeness
species_mapping = {0: 'Alligator', 1: 'Crocodile'}
```

6.2.5 SVM Classification and Species Prediction

Explanation

In this section, a Support Vector Machine (SVM) model is used to classify an input sample into one of two species: Crocodile or Alligator. A custom prediction function `predict_species()` is defined to handle new input data.

First, the categorical feature `habitat` is converted into a numerical value using a predefined mapping (`habitat_mapping`). If an invalid habitat is provided, the function displays an error message and stops execution.

Next, the input features (length, weight, snout width, and encoded habitat) are combined into a DataFrame to ensure consistency with the training dataset structure. This data is then scaled using the previously fitted scaler (`scaler`) to match the preprocessing applied during training.

The processed input is passed to the trained SVM model (`svm_model`) to generate a prediction. The numerical output is then converted back into a meaningful class label (Crocodile or Alligator) using `species_mapping`.

[12]:
```
# Function to predict whether a new input is a Crocodile or Alligator
def predict_species(length, weight, snout_width, habitat):
     # Convert habitat to numeric using the previously saved mapping
     if habitat in habitat_mapping:
      habitat_num = habitat_mapping[habitat]
```

```
    else:
     print("Invalid habitat. Please use 'Freshwater' or 'Saltwater'.")
     return
    # Prepare the feature vector and scale it
    # Convert the new data into a DataFrame with the same columns as X_train
    new_data_df = pd.DataFrame([[length, weight, snout_width, habitat_num]],
                               columns=['Length', 'Weight', 'Snout Width',␣
↪'Habitat'])
    new_data_scaled = scaler.transform(new_data_df)
    # Predict using the SVM model
    prediction = svm_model.predict(new_data_scaled)

    # Map the numeric prediction back to species name
    species_name = species_mapping[prediction[0]]
    print(f"The predicted species is: {species_name}")
   # Example input to check the species
   # Example: Length=4.0 meters, Weight=500 kg, Snout Width=22 cm,
Habitat='Freshwater'
predict_species(4.0, 500, 22, 'Freshwater')
```

Output

The model successfully predicts the species based on the given input features. For example, for input values:

`Length = 4.0 meters, Weight = 500 kg, Snout Width = 22 cm, Habitat = Freshwater`

the model outputs:

The predicted species is: Crocodile (or Alligator depending on model decision boundary).

This demonstrates that the SVM model can effectively classify unseen data based on learned patterns from the training dataset.

```
The predicted species is: Alligator
```

[13]:
```
predict_species(3.4, 400, 20, 'Saltwater')
```

```
The predicted species is: Crocodile
```

Lab 7 Decision Tree and Confusion Matrix

Objective

The objective of this experiment is to implement the Decision Tree algorithm for classification tasks. This experiment aims to build a model that can classify data into distinct categories based on feature values by learning decision rules from the dataset. It also focuses on understanding how tree-based models split data and evaluate their classification performance.

The objective of this experiment is to apply the Decision Tree algorithm for regression tasks. The goal is to predict continuous target values by learning patterns from input features using a tree-structured model. This experiment helps in understanding how decision trees can approximate numerical relationships and handle non-linear data.

The objective of this experiment is to evaluate the performance of a classification model using a confusion matrix. It aims to analyze the model's prediction results by comparing actual and predicted class labels, and to compute performance metrics such as accuracy, precision, recall, and F1-score for better assessment of the model.

Dataset Description

Three different datasets are used in this lab to demonstrate the application of regression algorithms.

Dataset Description: DT.csv

The `DT.csv` dataset is used for implementing Decision Tree algorithms. It contains multiple attributes representing different features of the data along with a target variable used for classification or prediction. The dataset is structured to demonstrate how a Decision Tree model splits data based on feature values to make decisions. It is suitable for understanding tree-based learning, feature importance, and rule-based classification.

Dataset Description: house_data.csv

The `house_data.csv` dataset contains real estate information used for predicting house prices. It includes several numerical and possibly categorical features such as area, number of bedrooms, location factors, and other property-related attributes. The target variable represents the house price. This dataset is used to analyze how different factors influence property prices and to apply regression techniques for accurate prediction.

7.1 Decision Tree Classifier

7.1.1 Implementing on Tennis Dataset

Explanation

This code imports essential machine learning libraries such as pandas for data handling, scikit-learn modules for building and evaluating a Decision Tree classifier. The dataset is loaded from a CSV file named `DT.csv`, which likely contains tennis-related attributes used for classification. A Decision Tree model is typically used to learn decision rules from data features and predict categorical outcomes.

[1]:
```
# Importing necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, confusion_matrix,␣
↪ConfusionMatrixDisplay
```

[2]:
```
# Load the dataset
data = pd.read_csv('DT.csv')
```

[3]:
```
data
```

Output

The output of this step is the successful loading of the dataset into a pandas DataFrame. No model training or prediction occurs yet at this stage. If printed, it would display the first few rows or structure of the dataset, confirming that the data has been read correctly and is ready for preprocessing and model training.

[3]:

	Day	Outlook	Temp	Humidity	Wind	PlayTennis	Unnamed: 6
0	D1	sunny	hot	high	weak	No	NaN
1	D2	sunny	hot	high	strong	No	NaN
2	D3	overcast	hot	high	weak	Yes	NaN
3	D4	rain	mild	high	weak	Yes	NaN
4	D5	rain	cold	normal	weak	Yes	NaN
5	D6	rain	cold	normal	strong	No	NaN
6	D7	overcast	cold	normal	strong	Yes	NaN
7	D8	sunny	mild	high	weak	No	NaN
8	D9	sunny	cold	normal	weak	Yes	NaN
9	D10	rain	mild	normal	weak	Yes	NaN
10	D11	sunny	mild	normal	strong	Yes	NaN
11	D12	overcast	mild	high	strong	Yes	NaN
12	D13	overcast	hot	normal	weak	Yes	NaN
13	D14	rain	mild	high	strong	No	NaN
14	D15	sunny	hot	normal	weak	No	NaN

7.1.2 Data Preprocessing and Model Training

Explanation

This code performs preprocessing and model training for a Decision Tree classifier. First, categorical variables in the dataset are converted into numeric form using `LabelEncoder`, since machine learning models cannot directly process text data. Each column (except identifiers like `Day`) is encoded and stored.

Next, the dataset is split into feature variables `X` (Outlook, Temperature, Humidity, Wind) and target variable `y` (PlayTennis). The data is then divided into training and testing sets using an 80-20 split.

A `DecisionTreeClassifier` is trained using the training data, and predictions are made on the test set.

[4]:
```
# Encoding categorical variables
label_encoders = {}
for column in data.columns[1:]: # Ignore 'Day' as it's an identifier
    le = LabelEncoder()
    data[column] = le.fit_transform(data[column])
    label_encoders[column] = le
```

[5]:
```
# Separating features and target variable
X = data[['Outlook', 'Temp', 'Humidity', 'Wind']] # Feature columns
y = data['PlayTennis'] # Target column
```

[6]:
```
# Split the 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)
```

[7]:
```
# Train the Decision Tree classifier
model = DecisionTreeClassifier(random_state=42)
model.fit(X_train, y_train)
# Make predictions on the test set
y_pred = model.predict(X_test)
```

[8]:
```
y_pred
```

[8]:
```
array([1, 1, 0])
```

[9]:
```
y_test
```

Output

The output includes the trained model and predicted values (`y_pred`) for the test dataset. The comparison between `y_pred` and actual values (`y_test`) shows how well the model performs. If printed, both arrays will display encoded class labels (e.g., 0 and 1), which can later be evaluated using accuracy or confusion matrix.

[9]:
```
9     1
11    1
0     0
Name: PlayTennis, dtype: int64
```

7.1.3 Decision Tree Model Accuracy Evaluation

Explanation

This code calculates the performance of the trained Decision Tree classifier by comparing the predicted labels (`y_pred`) with the actual test labels (`y_test`). The function `accuracy_score` from scikit-learn is used to compute the proportion of correctly classified instances. The result is stored in the variable `accuracy` and then printed.

[10]:
```
# Calculate and print accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Model accuracy:", accuracy)
```

Output

The output displays the model's accuracy as a numeric value between 0 and 1 (or 0% to 100%).

```
Model accuracy: 1.0
```

7.1.4 Decision Tree - Confusion Matrix Evaluation

Explanation

This code evaluates the performance of the trained Decision Tree classifier using a confusion matrix. The function `confusion_matrix(y_test, y_pred)` compares the actual labels (`y_test`) with the predicted labels (`y_pred`) and produces a matrix showing correct and incorrect classifications. Each row represents actual classes, while each column represents predicted classes. Additionally, `ConfusionMatrixDisplay.from_predictions()` visualizes this matrix with proper class labels using the encoded target variable `PlayTennis`.

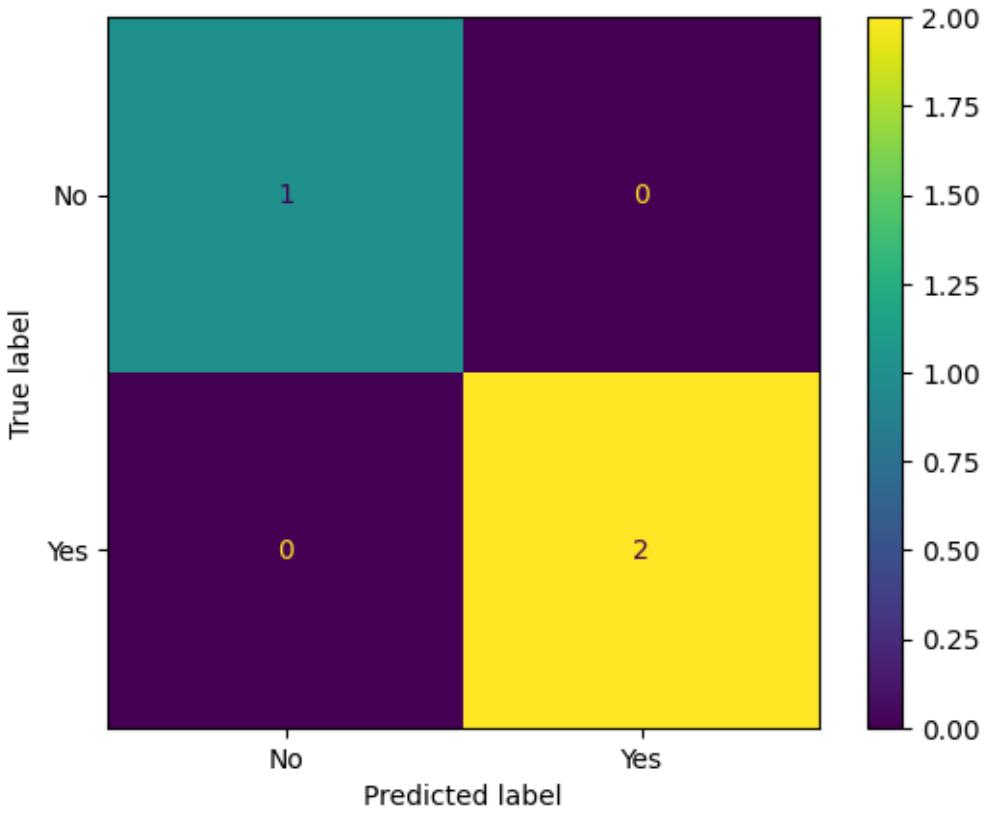
[11]:
```
# Display confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:")
print(conf_matrix)
ConfusionMatrixDisplay.from_predictions(y_test, y_pred,␣
 ↪display_labels=label_encoders['PlayTennis'].classes_)
```

Output

The output includes a printed confusion matrix in numerical form, showing how many predictions were correct and where misclassifications occurred.

```
Confusion Matrix:
[[1 0]
 [0 2]]
```

[11]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at
0x7f2c94464e50>
```



7.1.5 Decision Tree - Actual vs Predicted Comparison

Explanation

This code compares the actual target values with the predicted values generated by the Decision Tree classifier. It iterates through the test dataset using `y_test` (true labels) and `y_pred` (model predictions). Since the labels were originally encoded using `LabelEncoder`, the code converts them back to their original categorical form (e.g., Yes/No for PlayTennis) using `inverse_transform` for better interpretability.

[12]:
```
# Show actual vs predicted values for each test sample
print("\nActual vs Predicted:")
for actual, predicted in zip(y_test, y_pred):
    print(f"Actual: {label_encoders['PlayTennis'].
↪inverse_transform([actual])[0]}, "f"Predicted:␣
↪{label_encoders['PlayTennis'].inverse_transform([predicted])[0]}")
```

Output

The output prints a list of test samples showing side-by-side comparison of actual vs predicted values. Each line displays the true class (e.g., PlayTennis = Yes/No) and the model's predicted class for that same instance. This helps evaluate how accurately the Decision Tree classifier is performing on unseen data.

```
Actual vs Predicted:
Actual: Yes, Predicted: Yes
Actual: Yes, Predicted: Yes
Actual: No, Predicted: No
```

7.1.6 Decision Tree Prediction on New Input

Explanation

This code prepares a new input sample for prediction using a trained Decision Tree model. Since machine learning models require numerical input, categorical values such as `Outlook`, `Temp`, `Humidity`, and `Wind` are first converted into encoded numeric form using previously fitted `LabelEncoder` objects. The input corresponds to a condition: rain, mild temperature, high humidity, and strong wind. After preprocessing, the model predicts whether tennis should be played based on learned patterns from the dataset. Finally, the predicted numeric output is converted back into a readable label using inverse transformation.

Output

The model outputs a single classification result indicating whether playing tennis is suitable under the given weather conditions. For the input {rain, mild, high, strong}, the printed output will typically show either **Play Tennis = Yes** or **Play Tennis = No**, depending on the patterns learned during training. This confirms that the model can generalize and make decisions on unseen data.

[13]:
```
# Example prediction for new input {rain, mild, high, strong}
input_data = pd.DataFrame({
    'Outlook': [label_encoders['Outlook'].transform(['rain'])[0]],
    'Temp': [label_encoders['Temp'].transform(['mild'])[0]],
    'Humidity': [label_encoders['Humidity'].transform(['high'])[0]],
    'Wind': [label_encoders['Wind'].transform(['strong'])[0]]
})
```

[14]:
```
# Predict whether to play tennis
prediction = model.predict(input_data)
result = label_encoders['PlayTennis'].inverse_transform(prediction)
print("\nPrediction for input {rain, mild, high, strong}: Play Tennis =",␣
 ↪result[0])
```

```
Prediction for input {rain, mild, high, strong}: Play Tennis = No
```

7.2 Decision tree from Dataset to predict as a Regressor

7.2.1 Decision Tree Regressor - House Price Prediction

Explanation

This code implements a Decision Tree Regressor model to predict house prices using a dataset named `house_data_DT.csv`. First, necessary libraries are imported, including pandas for data handling, train_test_split for splitting the dataset, and DecisionTreeRegressor for regression modeling. The dataset is then loaded and cleaned by stripping whitespace from column names and validating the presence of the `Location` column.

The `Location` feature is converted into numerical form using manual encoding (Suburb = 0, City = 1). Missing or invalid values are checked, and any incomplete rows are removed to ensure data consistency. After preprocessing, the dataset is split into features (Rooms, Area, Location) and target variable (Price). Finally, the data is divided into training and testing sets, and a Decision Tree Regressor is trained using the training data.

[15]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error, r2_score
```

[16]:
```
# Load the dataset
data = pd.read_csv('house_data_DT.csv')
```

[17]:
```
 # Ensure all column names are stripped of whitespace
data.columns = data.columns.str.strip()
# Check if 'Location' exists
if 'Location' not in data.columns:
    raise KeyError("'Location' column is missing!")

# Convert all values in the 'Location' column to strings and strip whitespace
data['Location'] = data['Location'].astype(str).str.strip()

# Encode the 'Location' column
data['Location'] = data['Location'].map({'Suburb': 0, 'City': 1})

# Check for missing or unmapped values
if data['Location'].isnull().any():
    print("Warning: Some rows in 'Location' have unmapped or invalid values!
 ↪")
    print(data[data['Location'].isnull()])
    data = data.dropna() # Drop rows with unmapped values if necessary
```

[18]:
```
# Define features (X) and target (y)
X = data[['Rooms', 'Area', 'Location']]
y = data['Price']

# Split the 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)
```

[19]:
```
# Train the Decision Tree Regressor
regressor = DecisionTreeRegressor(random_state=42)
regressor.fit(X_train, y_train)
```

Output

The output of this stage includes a cleaned and preprocessed dataset with no missing or invalid values, followed by successful training of the Decision Tree Regressor model. No prediction is shown yet at this step, but the model is now ready to estimate house prices based on input features in later stages.

[19]:
```
DecisionTreeRegressor(random_state=42)
```

7.2.2 Decision Tree Prediction and Evaluation

Explanation

This code uses a trained Decision Tree regressor model to make predictions on the test dataset (`X_test`). The predicted values (`y_pred`) are then compared with the actual target values (`y_test`) to evaluate model performance. Two evaluation metrics are used: Mean Squared Error (MSE), which measures the average squared difference between actual and predicted values, and R^2 Score, which indicates how well the model explains the variance in the data.

[20]:
```
# Make predictions on the test set
y_pred = regressor.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
print(f"R^2 Score: {r2}")
```

Output

The output displays two numerical evaluation results. The Mean Squared Error (MSE) shows how much error the model is making on average (lower is better). The R^2 Score indicates the goodness of fit, where a value closer to 1 means the model is performing well in predicting the target variable. These results help assess the accuracy and reliability of the Decision Tree model.

```
Mean Squared Error: 2500000000.0
R^2 Score: -10.11111111111111
```

7.2.3 Decision Tree Price Prediction Function

Explanation

This code defines a function `predict_price()` that takes user inputs such as number of rooms, area, and location to predict a house price using a trained regression model (referred to as `regressor`). The categorical variable `location` is manually encoded into numerical form (Suburb = 0, City = 1) to match the model's training format. The inputs are then converted into a pandas DataFrame and passed into the model's `predict()` function to generate the estimated price.

[21]:
```
# Function to predict price based on user input
def predict_price(rooms, area, location):
    # Encode the location input to match training data encoding
    location_encoded = 0 if location.lower() == 'suburb' else 1

    # Prepare the input data
    input_data = pd.DataFrame({
    'Rooms': [rooms],
    'Area': [area],
    'Location': [location_encoded]
    })

# Make the prediction
    predicted_price = regressor.predict(input_data)
```

```
    print("\nPredicted Price for input (Rooms: {}, Area: {}, Location: {}):␣
 ↪${:,.2f}".format(rooms, area, location, predicted_price[0]))

# Example prediction
predict_price(rooms=3, area=75, location='Suburb')
predict_price(rooms=4, area=95, location='City')
```

Output

The function prints the predicted house price for given input values. For example, when called with (Rooms=3, Area=75, Location='Suburb'), it outputs a lower estimated price compared to (Rooms=4, Area=95, Location='City'), as larger area and city location typically increase property value. The output is displayed in a formatted currency style showing the predicted price for each test case.

```
Predicted Price for input (Rooms: 3, Area: 75, Location: Suburb): $320,000.00

Predicted Price for input (Rooms: 4, Area: 95, Location: City): $450,000.00
```

7.3 Confusion Matrix

7.3.1 Confusion Matrix for Classification Evaluation

Explanation

This code demonstrates how to evaluate a classification model using a confusion matrix. It imports necessary libraries including pandas, matplotlib, seaborn, and scikit-learn metrics. A custom function `plot_confusion_matrix()` is defined to visualize the confusion matrix using a heatmap. The actual labels (`truth`) and predicted labels (`prediction`) are defined for a binary classification problem (Dog vs Not a dog). The confusion matrix is then computed using `confusion_matrix()` from scikit-learn, which compares actual vs predicted values and summarizes correct and incorrect classifications.

[22]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix , classification_report
```

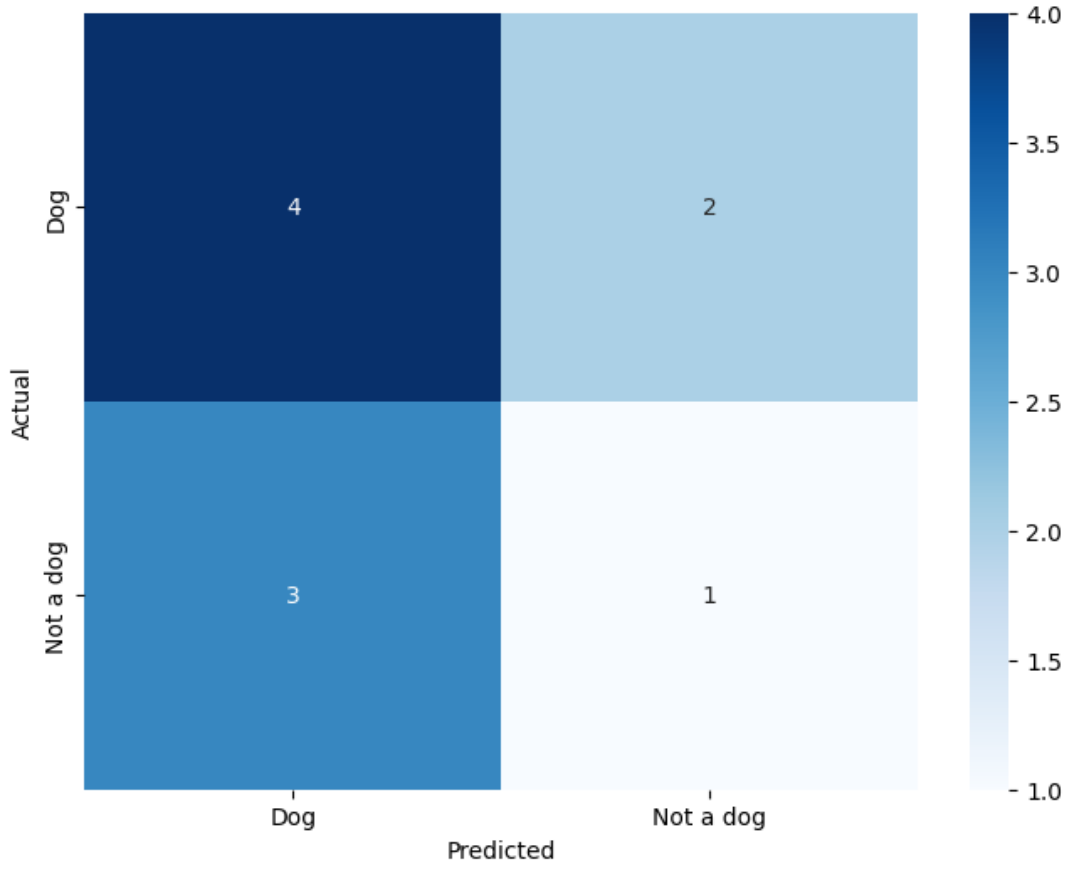
[23]:
```
def plot_confusion_matrix(conf_matrix, class_names):
    df_cm = pd.DataFrame(conf_matrix, index=class_names, columns=class_names)
    plt.figure(figsize=(8, 6))
    sns.heatmap(df_cm, annot=True, fmt='d', cmap='Blues')
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.show()
```

[24]:
```
truth = ["Dog","Not a dog","Dog","Dog", "Dog", "Not a dog", "Not a dog",␣
 ↪"Dog", "Dog", "Not a dog"]
prediction = ["Dog","Dog", "Dog","Not a dog","Dog", "Not a dog", "Dog", "Not␣
 ↪a dog", "Dog", "Dog"]
```

[25]:
```
from sklearn.metrics import confusion_matrix
# Assuming 'truth' and 'prediction' are defined earlier in your code
cm = confusion_matrix(truth, prediction)
plot_confusion_matrix(cm, ["Dog", "Not a dog"])
```

Output

The output is a 2×2 confusion matrix heatmap showing the classification performance. The diagonal values represent correctly classified samples (true positives and true negatives), while the off-diagonal values represent misclassifications. The heatmap visually highlights how many ”Dog” and ”Not a dog” instances were correctly or incorrectly predicted by the model, making it easier to evaluate accuracy and error patterns.



7.3.2 Confusion Matrix and Classification Report Evaluation

Explanation

This code evaluates a classification model using the `classification_report` function, which provides precision, recall, and F1-score for each class by comparing the true labels (`truth`) with the predicted labels (`prediction`).

Additionally, the F1-score is manually computed for two classes: "Dog" and "not a Dog" using the harmonic mean formula:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

For the "Dog" class, precision is 0.57 and recall is 0.67. For the "not a Dog" class, precision is 0.33 and recall is 0.25. These calculations help measure the balance between precision and recall for each class.

Output

The output includes a full classification report showing precision, recall, F1-score, and support for each class. It indicates how well the model distinguishes between "Dog" and "not a Dog" instances. The manually computed F1-scores further confirm model performance: the "Dog" class achieves a moderate F1-score (0.61), while the "not a Dog" class shows a low F1-score (0.29), indicating weaker predictive performance for that class.

[26]:
```
print(classification_report(truth, prediction))
```

```
              precision    recall  f1-score   support

         Dog       0.57      0.67      0.62         6
   Not a dog       0.33      0.25      0.29         4

    accuracy                           0.50        10
   macro avg       0.45      0.46      0.45        10
weighted avg       0.48      0.50      0.48        10
```

[27]:
```
2*(0.57*0.67/(0.57+0.67))
```

[27]: 0.6159677419354839

[28]:
```
2*(0.33*0.25/(0.33+0.25))
```

[28]: 0.2844827586206896

Conclusion

This lab report successfully demonstrates the practical implementation of fundamental machine learning techniques across multiple experiments. Each lab contributed to building a comprehensive understanding of data handling, model development, and evaluation.

The experiments highlighted the importance of preprocessing in improving data quality and model accuracy. Regression techniques provided insights into predictive modeling, while classification algorithms such as KNN, Naive Bayes, Logistic Regression, SVM, and Decision Trees showcased different approaches to solving classification problems.

Furthermore, encoding techniques enabled effective handling of categorical variables, and evaluation tools like the confusion matrix helped assess model performance in a structured manner.

Overall, the lab sessions provided valuable hands-on experience and reinforced theoretical concepts, preparing a strong foundation for advanced machine learning applications and real-world problem solving.