NOTRE DAME UNIVERSITY BANGLADESH

Machine Learning Lab Report-05

Course Code: CSE4214

Course Title: Machine Learning Lab

Lab Task Topic: Naive Bayes Theorem & Feature Selection

Submitted by:

Name: Istiak Alam

ID: 0692230005101005

Batch: CSE-20

Submission Date: April 26, 2026

Submitted to:

A. H. M. Saiful Islam

Chairman, Dept of CSE

Notre Dame University Bangladesh

Table of Contents

1 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.

2 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.

3 Implementing Naive Bayes Theorem

3.1 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

3.2 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
```

3.3 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()
```

3.4 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
```

3.5 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']
```

4 Naive Bayes Classifier - Titanic

4.1 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	Name	Sex	Age	SibSp	Parch
0	892	0	3	Kelly, Mr. James	male	34.5	0	0
1	893	1	3	Wilkes, Mrs. James (Ellen Needs)	female	47.0	1	0
2	894	0	2					
3	895	0	3					
4	896	1	3					

```
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
```

4.2 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  \
0            892         0       3
1            893         1       3
2            894         0       2
3            895         0       3
4            896         1       3
..           …       …      …
413         1305         0       3
414         1306         1       1
415         1307         0       3
416         1308         0       3
417         1309         0       3

                                          Name     Sex       Age  SibSp  \
0                              Kelly, Mr. James    male  34.50000      0
1              Wilkes, Mrs. James (Ellen Needs)  female  47.00000      1
2                     Myles, Mr. Thomas Francis    male  62.00000      0
3                              Wirz, Mr. Albert    male  27.00000      0
4    Hirvonen, Mrs. Alexander (Helga E Lindqvist)  female  22.00000      1
..                                            …     …       …    …
413                           Spector, Mr. Woolf    male  30.27259      0
414                 Oliva y Ocana, Dona. Fermina  female  39.00000      0
415                 Saether, Mr. Simon Sivertsen    male  38.50000      0
416                         Ware, Mr. Frederick    male  30.27259      0
417                   Peter, Master. Michael J    male  30.27259      1

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

4.3 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

4.4 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]

4.5 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]
```

4.6 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]
```

4.7 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]
```

4.8 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
```

4.9 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

4.10 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
```

4.11 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
```

4.12 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 Feature Selection Stdm

5.1 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]
```

5.2 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 Cardiovascular Disease Count Visualization Using Countplot

Explanation

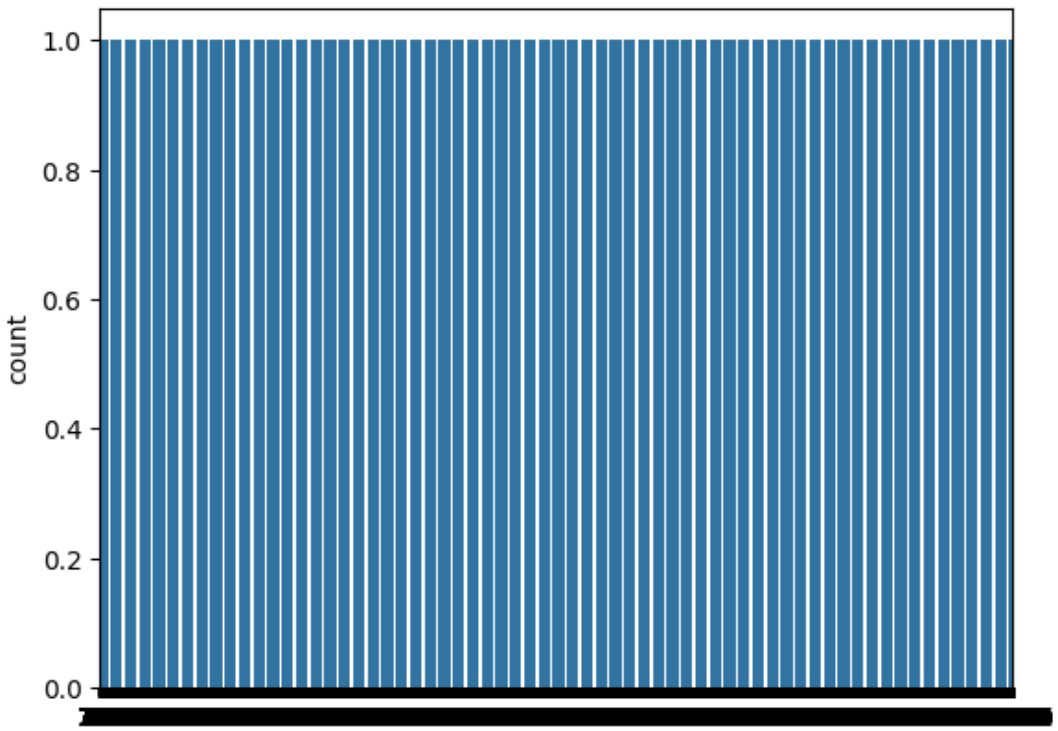
The function `sns.countplot(df["cardio"])` from Seaborn is used to visualize the frequency distribution of the `cardio` column in the dataset `df`. It counts the number of occurrences of each category (typically 0 and 1, representing absence or presence of cardiovascular disease) and displays them as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

5.4 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
```

5.5 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
```

5.6 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
```

5.7 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
```

6 Feature Selection Using Correlation

6.1 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
```

6.2 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

6.3 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
```

6.4 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
```

6.5 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

6.6 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
```

6.7 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

6.8 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())
```

6.9 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')
```

6.10 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
```

6.11 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

6.12 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
```

6.13 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.

6.14 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

6.15 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
```