data_cleaning_exercise_lab2

July 16, 2025

1 Data Cleaning Exercise - Laboratory 2

AICTE Faculty ID: 1-3241967546
Faculty Name: Milav Jayeshkuamar Dabgar
Date: July 16, 2025

1.1 Objective

This laboratory exercise focuses on data cleaning and preprocessing techniques using the Car Evaluation dataset. We will perform comprehensive data analysis, handle missing values, feature engineering, and prepare the dataset for machine learning applications.

1.2 Import Libraries

- numpy
- matplotlib.pyplot
- pandas

[1]:
```python
# Import necessary libraries for data manipulation and visualization
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

print("Libraries imported successfully!")
```

```
Libraries imported successfully!
```

1.3 Import Data Set

- car_evaluation.csv
- see the given dataset

[2]:
```python
# Load the car evaluation dataset
# Note: The CSV file doesn't have headers, so we'll add them manually
data = pd.read_csv('/Users/milav/Code/qip-dl/ml/day-2/car_evaluation.csv',␣
 ↪header=None)
```

```
print("Dataset loaded successfully!")
print(f"Dataset shape: {data.shape}")
print("\nFirst few rows of the raw dataset:")
print(data.head())

print("\nLast few rows to check the data:")
print(data.tail())
```

```
Dataset loaded successfully!
Dataset shape: (1728, 7)

First few rows of the raw dataset:
       0      1  2  3      4     5      6
0  vhigh  vhigh  2  2  small   low  unacc
1  vhigh  vhigh  2  2  small   med  unacc
2  vhigh  vhigh  2  2  small  high  unacc
3  vhigh  vhigh  2  2    med   low  unacc
4  vhigh  vhigh  2  2    med   med  unacc

Last few rows to check the data:
        0    1      2     3    4     5      6
1723  low  low  5more  more  med   med   good
1724  low  low  5more  more  med  high  vgood
1725  low  low  5more  more  big   low  unacc
1726  low  low  5more  more  big   med   good
1727  low  low  5more  more  big  high  vgood
```

1.4 data analysis

- See the number of rows and column
- Need to change the column names– rename them
- See the dataset after adding new column names
- In each column/features, see the distribution of every catorical values

[3]:
```
# Data Analysis Section

# Check the number of rows and columns
print("=== DATASET DIMENSIONS ===")
print(f"Number of rows: {data.shape[0]}")
print(f"Number of columns: {data.shape[1]}")

# Add proper column names based on the car evaluation dataset documentation
column_names = ['buying_price', 'maintenance_cost', 'doors', 'persons',␣
 ↪'luggage_boot', 'safety', 'class_value']
data.columns = column_names

print("\n=== DATASET AFTER ADDING COLUMN NAMES ===")
```

```python
print(data.head(10))

print("\n=== BASIC DATASET INFO ===")
print(data.info())

print("\n=== DATASET DESCRIPTION ===")
print(data.describe(include='all'))
```

```
=== DATASET DIMENSIONS ===
Number of rows: 1728
Number of columns: 7

=== DATASET AFTER ADDING COLUMN NAMES ===
  buying_price maintenance_cost doors persons luggage_boot safety class_value
0        vhigh            vhigh     2       2        small    low       unacc
1        vhigh            vhigh     2       2        small    med       unacc
2        vhigh            vhigh     2       2        small   high       unacc
3        vhigh            vhigh     2       2          med    low       unacc
4        vhigh            vhigh     2       2          med    med       unacc
5        vhigh            vhigh     2       2          med   high       unacc
6        vhigh            vhigh     2       2          big    low       unacc
7        vhigh            vhigh     2       2          big    med       unacc
8        vhigh            vhigh     2       2          big   high       unacc
9        vhigh            vhigh     2       4        small    low       unacc

=== BASIC DATASET INFO ===
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1728 entries, 0 to 1727
Data columns (total 7 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   buying_price      1728 non-null   object
 1   maintenance_cost  1728 non-null   object
 2   doors             1728 non-null   object
 3   persons           1728 non-null   object
 4   luggage_boot      1728 non-null   object
 5   safety            1728 non-null   object
 6   class_value       1728 non-null   object
dtypes: object(7)
memory usage: 94.6+ KB
None

=== DATASET DESCRIPTION ===
       buying_price maintenance_cost doors persons luggage_boot safety  \
count          1728             1728  1728    1728         1728   1728
unique            4                4     4       3            3      3
top           vhigh            vhigh     2       2        small    low
freq            432              432   432     576          576    576
```

```
       class_value
count         1728
unique           4
top          unacc
freq          1210
```

[4]:

```python
# Analyze the distribution of categorical values in each column
print("=== DISTRIBUTION OF CATEGORICAL VALUES ===")

for column in data.columns:
    print(f"\n--- {column.upper()} ---")
    value_counts = data[column].value_counts()
    print(value_counts)

    # Calculate percentages
    percentages = (data[column].value_counts(normalize=True) * 100).round(2)
    print(f"\nPercentages:")
    for value, count in value_counts.items():
        percentage = percentages[value]
        print(f"  {value}: {count} ({percentage}%)")
    print("-" * 40)
```

```
=== DISTRIBUTION OF CATEGORICAL VALUES ===

--- BUYING_PRICE ---
buying_price
vhigh    432
high     432
med      432
low      432
Name: count, dtype: int64

Percentages:
  vhigh: 432 (25.0%)
  high: 432 (25.0%)
  med: 432 (25.0%)
  low: 432 (25.0%)
----------------------------------------

--- MAINTENANCE_COST ---
maintenance_cost
vhigh    432
high     432
med      432
low      432
Name: count, dtype: int64
```

```
Percentages:
  vhigh: 432 (25.0%)
  high: 432 (25.0%)
  med: 432 (25.0%)
  low: 432 (25.0%)
----------------------------------------

--- DOORS ---
doors
2        432
3        432
4        432
5more    432
Name: count, dtype: int64

Percentages:
  2: 432 (25.0%)
  3: 432 (25.0%)
  4: 432 (25.0%)
  5more: 432 (25.0%)
----------------------------------------

--- PERSONS ---
persons
2       576
4       576
more    576
Name: count, dtype: int64

Percentages:
  2: 576 (33.33%)
  4: 576 (33.33%)
  more: 576 (33.33%)
----------------------------------------

--- LUGGAGE_BOOT ---
luggage_boot
small    576
med      576
big      576
Name: count, dtype: int64

Percentages:
  small: 576 (33.33%)
  med: 576 (33.33%)
  big: 576 (33.33%)
----------------------------------------
```

```
--- SAFETY ---
safety
low     576
med     576
high    576
Name: count, dtype: int64

Percentages:
  low: 576 (33.33%)
  med: 576 (33.33%)
  high: 576 (33.33%)
----------------------------------------

--- CLASS_VALUE ---
class_value
unacc    1210
acc       384
good       69
vgood      65
Name: count, dtype: int64

Percentages:
  unacc: 1210 (70.02%)
  acc: 384 (22.22%)
  good: 69 (3.99%)
  vgood: 65 (3.76%)
----------------------------------------
```

1.4.1 Checking missing values

[5]:
```
# Check for missing values in the dataset
print("=== MISSING VALUES ANALYSIS ===")

# Check for null values
missing_values = data.isnull().sum()
print("Null values per column:")
print(missing_values)

# Check for empty strings
empty_strings = (data == '').sum()
print("\nEmpty strings per column:")
print(empty_strings)

# Check for whitespace-only values
whitespace_only = data.apply(lambda x: x.astype(str).str.strip().eq('').sum())
print("\nWhitespace-only values per column:")
print(whitespace_only)
```

```python
# Total missing data
total_missing = missing_values.sum() + empty_strings.sum() + whitespace_only.
 ↪sum()
print(f"\nTotal missing values in dataset: {total_missing}")

if total_missing == 0:
    print("  Great! No missing values found in the dataset.")
else:
    print("  Missing values detected and need to be handled.")
```

```
=== MISSING VALUES ANALYSIS ===
Null values per column:
buying_price        0
maintenance_cost    0
doors               0
persons             0
luggage_boot        0
safety              0
class_value         0
dtype: int64

Empty strings per column:
buying_price        0
maintenance_cost    0
doors               0
persons             0
luggage_boot        0
safety              0
class_value         0
dtype: int64

Whitespace-only values per column:
buying_price        0
maintenance_cost    0
doors               0
persons             0
luggage_boot        0
safety              0
class_value         0
dtype: int64

Total missing values in dataset: 0
  Great! No missing values found in the dataset.
```

1.5 Feature Engineering

- Convert Data type of columns (doors , persons)

- Encode the non numerical value to numerical values

[6]:
```
# Feature Engineering Section

# Create a copy of the data for processing
processed_data = data.copy()

print("=== ORIGINAL DATA TYPES ===")
print(processed_data.dtypes)

# Convert specific columns that should be numerical
print("\n=== CONVERTING DATA TYPES ===")

# Handle 'doors' column - convert to numeric
print("Converting 'doors' column...")
doors_mapping = {'2': 2, '3': 3, '4': 4, '5more': 5}
processed_data['doors'] = processed_data['doors'].map(doors_mapping)
print(f"Doors unique values after conversion: {sorted(processed_data['doors'].
    ↪unique())}")

# Handle 'persons' column - convert to numeric
print("Converting 'persons' column...")
persons_mapping = {'2': 2, '4': 4, 'more': 6}  # Assuming 'more' means 6+ people
processed_data['persons'] = processed_data['persons'].map(persons_mapping)
print(f"Persons unique values after conversion:␣
    ↪{sorted(processed_data['persons'].unique())}")

print("\n=== DATA TYPES AFTER CONVERSION ===")
print(processed_data.dtypes)
```

```
=== ORIGINAL DATA TYPES ===
buying_price        object
maintenance_cost    object
doors               object
persons             object
luggage_boot        object
safety              object
class_value         object
dtype: object

=== CONVERTING DATA TYPES ===
Converting 'doors' column…
Doors unique values after conversion: [np.int64(2), np.int64(3), np.int64(4),
np.int64(5)]
Converting 'persons' column…
Persons unique values after conversion: [np.int64(2), np.int64(4), np.int64(6)]

=== DATA TYPES AFTER CONVERSION ===
```

```
buying_price        object
maintenance_cost    object
doors                int64
persons              int64
luggage_boot        object
safety              object
class_value         object
dtype: object
```

[7]:

```
# Encode categorical variables to numerical values
print("=== ENCODING CATEGORICAL VARIABLES ===")

# Columns that need encoding (excluding doors and persons which are now numeric)
categorical_columns = ['buying_price', 'maintenance_cost', 'luggage_boot',␣
↪'safety', 'class_value']

# Store the original values for reference
original_mappings = {}

# Initialize label encoders
label_encoders = {}

for column in categorical_columns:
    print(f"\nEncoding {column}...")

    # Store original values
    original_values = processed_data[column].unique()
    original_mappings[column] = original_values
    print(f"Original values: {list(original_values)}")

    # Apply label encoding
    le = LabelEncoder()
    processed_data[column] = le.fit_transform(processed_data[column])
    label_encoders[column] = le

    # Show the mapping
    encoded_values = processed_data[column].unique()
    print(f"Encoded values: {sorted(encoded_values)}")

    # Create mapping dictionary for clarity
    mapping = dict(zip(le.classes_, le.transform(le.classes_)))
    print(f"Mapping: {mapping}")

print("\n=== ENCODED DATASET SAMPLE ===")
print(processed_data.head(10))
```

```
=== ENCODING CATEGORICAL VARIABLES ===
```

```
Encoding buying_price…
Original values: ['vhigh', 'high', 'med', 'low']
Encoded values: [np.int64(0), np.int64(1), np.int64(2), np.int64(3)]
Mapping: {'high': np.int64(0), 'low': np.int64(1), 'med': np.int64(2), 'vhigh':
np.int64(3)}

Encoding maintenance_cost…
Original values: ['vhigh', 'high', 'med', 'low']
Encoded values: [np.int64(0), np.int64(1), np.int64(2), np.int64(3)]
Mapping: {'high': np.int64(0), 'low': np.int64(1), 'med': np.int64(2), 'vhigh':
np.int64(3)}

Encoding luggage_boot…
Original values: ['small', 'med', 'big']
Encoded values: [np.int64(0), np.int64(1), np.int64(2)]
Mapping: {'big': np.int64(0), 'med': np.int64(1), 'small': np.int64(2)}

Encoding safety…
Original values: ['low', 'med', 'high']
Encoded values: [np.int64(0), np.int64(1), np.int64(2)]
Mapping: {'high': np.int64(0), 'low': np.int64(1), 'med': np.int64(2)}

Encoding class_value…
Original values: ['unacc', 'acc', 'vgood', 'good']
Encoded values: [np.int64(0), np.int64(1), np.int64(2), np.int64(3)]
Mapping: {'acc': np.int64(0), 'good': np.int64(1), 'unacc': np.int64(2),
'vgood': np.int64(3)}

=== ENCODED DATASET SAMPLE ===
   buying_price  maintenance_cost  doors  persons  luggage_boot  safety  \
0             3                 3      2        2             2       1
1             3                 3      2        2             2       2
2             3                 3      2        2             2       0
3             3                 3      2        2             1       1
4             3                 3      2        2             1       2
5             3                 3      2        2             1       0
6             3                 3      2        2             0       1
7             3                 3      2        2             0       2
8             3                 3      2        2             0       0
9             3                 3      2        4             2       1

   class_value
0            2
1            2
2            2
3            2
4            2
5            2
```

```
6             2
7             2
8             2
9             2
```

1.6 Seperate dependent and independent variables

[8]:
```python
# Separate dependent and independent variables
print("=== SEPARATING FEATURES AND TARGET ===")

# Independent variables (features) - all columns except the target
X = processed_data.drop('class_value', axis=1)
print("Independent variables (Features):")
print(f"Feature columns: {list(X.columns)}")
print(f"Features shape: {X.shape}")

# Dependent variable (target)
y = processed_data['class_value']
print(f"\nDependent variable (Target): 'class_value'")
print(f"Target shape: {y.shape}")

# Show feature statistics
print("\n=== FEATURE STATISTICS ===")
print(X.describe())

# Show target distribution
print("\n=== TARGET VARIABLE DISTRIBUTION ===")
target_distribution = y.value_counts().sort_index()
print(target_distribution)

# Show target percentages
target_percentages = (y.value_counts(normalize=True) * 100).round(2).
 ↪sort_index()
print("\nTarget percentages:")
for value, percentage in target_percentages.items():
    count = target_distribution[value]
    print(f"Class {value}: {count} samples ({percentage}%)")

print(f"\nDataset is ready for machine learning!")
print(f"Total samples: {X.shape[0]}")
print(f"Total features: {X.shape[1]}")
print(f"Target classes: {len(y.unique())}")
```

```
=== SEPARATING FEATURES AND TARGET ===
Independent variables (Features):
Feature columns: ['buying_price', 'maintenance_cost', 'doors', 'persons',
'luggage_boot', 'safety']
Features shape: (1728, 6)
```

```
Dependent variable (Target): 'class_value'
Target shape: (1728,)

=== FEATURE STATISTICS ===
       buying_price  maintenance_cost        doors      persons  luggage_boot  \
count   1728.000000       1728.000000  1728.000000  1728.000000   1728.000000
mean       1.500000          1.500000     3.500000     4.000000      1.000000
std        1.118358          1.118358     1.118358     1.633466      0.816733
min        0.000000          0.000000     2.000000     2.000000      0.000000
25%        0.750000          0.750000     2.750000     2.000000      0.000000
50%        1.500000          1.500000     3.500000     4.000000      1.000000
75%        2.250000          2.250000     4.250000     6.000000      2.000000
max        3.000000          3.000000     5.000000     6.000000      2.000000

            safety
count  1728.000000
mean      1.000000
std       0.816733
min       0.000000
25%       0.000000
50%       1.000000
75%       2.000000
max       2.000000

=== TARGET VARIABLE DISTRIBUTION ===
class_value
0     384
1      69
2    1210
3      65
Name: count, dtype: int64

Target percentages:
Class 0: 384 samples (22.22%)
Class 1: 69 samples (3.99%)
Class 2: 1210 samples (70.02%)
Class 3: 65 samples (3.76%)

Dataset is ready for machine learning!
Total samples: 1728
Total features: 6
Target classes: 4
```

1.7 Split the data into training and test data

[9]:
```python
# Split the data into training and testing sets
print("=== SPLITTING DATA INTO TRAIN AND TEST SETS ===")

# Perform the split (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42,
    stratify=y  # Ensure balanced split across all classes
)

print("Data split completed successfully!")
print(f"\nTraining set:")
print(f"  X_train shape: {X_train.shape}")
print(f"  y_train shape: {y_train.shape}")

print(f"\nTesting set:")
print(f"  X_test shape: {X_test.shape}")
print(f"  y_test shape: {y_test.shape}")

# Verify the split ratios
print(f"\nSplit ratios:")
print(f"  Training: {len(X_train)/len(X)*100:.1f}%")
print(f"  Testing: {len(X_test)/len(X)*100:.1f}%")

# Check class distribution in training and testing sets
print(f"\n=== CLASS DISTRIBUTION AFTER SPLIT ===")
print("Training set class distribution:")
train_dist = y_train.value_counts().sort_index()
for class_val, count in train_dist.items():
    percentage = (count / len(y_train) * 100)
    print(f"  Class {class_val}: {count} samples ({percentage:.1f}%)")

print("\nTesting set class distribution:")
test_dist = y_test.value_counts().sort_index()
for class_val, count in test_dist.items():
    percentage = (count / len(y_test) * 100)
    print(f"  Class {class_val}: {count} samples ({percentage:.1f}%)")

print("\n Data preprocessing completed successfully!")
print("The dataset is now ready for machine learning algorithms.")
```

```
=== SPLITTING DATA INTO TRAIN AND TEST SETS ===
Data split completed successfully!

Training set:
```

```
  X_train shape: (1382, 6)
  y_train shape: (1382,)

Testing set:
  X_test shape: (346, 6)
  y_test shape: (346,)

Split ratios:
  Training: 80.0%
  Testing: 20.0%

=== CLASS DISTRIBUTION AFTER SPLIT ===
Training set class distribution:
  Class 0: 307 samples (22.2%)
  Class 1: 55 samples (4.0%)
  Class 2: 968 samples (70.0%)
  Class 3: 52 samples (3.8%)

Testing set class distribution:
  Class 0: 77 samples (22.3%)
  Class 1: 14 samples (4.0%)
  Class 2: 242 samples (69.9%)
  Class 3: 13 samples (3.8%)

 Data preprocessing completed successfully!
The dataset is now ready for machine learning algorithms.
```