Data Analysis

Exploration & Cleaning

Zeham Management Technologies BootCamp
by SDAIA

July 23rd, 2024

SDAIA
الهيئة السعودية للبيانات والذكاء الاصطناعي
Saudi Data & AI Authority

Introduction to Data Analysis

Let's start together…

Agenda

- Missing Values Handling
- Handling Duplicates
- Handling Outliers
- Date and Time
- Scaling and Normalization
- Categorical Variables

Data Issues and Challenges

Missing Values: :

- Absent data due to collection issues or applicability.

Duplicates:

- Repeated entries that skew analysis results.

Outliers:

- Extreme values that may indicate errors or anomalies.

Inconsistent Formats:

- Differing date and numerical formats, or text casing

Unstandardized Data:

- Varying codes or names for the same items.

Missing Values

Data Cleaning

Data can have missing values for various reasons, such as errors in data collection, transmission losses, or simply because some information was not applicable. Handling missing data requires careful consideration to avoid introducing bias or inaccuracies.

Three Ways of handling missing data:

1. A simple option: Drop columns with missing values

- Unless most values in the dropped columns are missing, the model loses access to a lot of (potentially useful!) information with this approach. As an extreme example, consider a dataset with 10,000 rows, where one important column is missing a single entry. This approach would drop the column entirely! (Figure 1)

2. A better option: Imputation

- **Imputation** fills in the missing values with some number. For instance, we can fill in the mean value along each column.
- The imputed value won't be exactly right in most cases, but it usually leads to more accurate models than you would get from dropping the column entirely (Figure 2)

Bed	Bath
1.0	1.0
2.0	1.0
3.0	2.0
NaN	2.0

Bath
1.0
1.0
2.0
2.0

Figure 1

Bed	Bath
1.0	1.0
2.0	1.0
3.0	2.0
NaN	2.0

Bed	Bath
1.0	1.0
2.0	1.0
3.0	2.0
2.0	2.0

Figure 2

Missing Values

Data Cleaning

Data can have missing values for various reasons, such as errors in data collection, transmission losses, or simply because some information was not applicable. Handling missing data requires careful consideration to avoid introducing bias or inaccuracies.

Three Ways of handling missing data:

3. An extension to imputation

- Imputation is the standard approach, and it usually works well. However, imputed values may be systematically above or below their actual values (which weren't collected in the dataset). Or rows with missing values may be unique in some other way. In that case, your model would make better predictions by considering which values were originally missing.
- In this approach, we impute the missing values, as before. And, additionally, for each column with missing entries in the original dataset, we add a new column that shows the location of the imputed entries. In some cases, this will meaningfully improve results. In other cases, it doesn't help at all.(Figure 3)

Bed	Bath
1.0	1.0
2.0	1.0
3.0	2.0
NaN	2.0

Bed	Bath	Bed_was_missing
1.0	1.0	FALSE
2.0	1.0	FALSE
3.0	2.0	FALSE
2.0	2.0	TRUE

Figure 3

Missing Values Example:

Data Cleaning

In the example, we will work with the Melbourne Housing dataset. Our model will use information such as the number of rooms and land size to predict home price.

First, we will read the data and train the data, then we will see the difference between the 3 approaches.

```python
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_csv('melb_data.csv')
y = df.Price
melb_predictors = df.drop(['Price'], axis=1)
X = melb_predictors.select_dtypes(exclude=['object'])
X_train, X_valid, y_train, y_valid = train_test_split(X, y, train_size=0.8, test_size=0.2,random_state=0)
```

```python
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error

def score_dataset(X_train, X_valid, y_train, y_valid):
    model = RandomForestRegressor(n_estimators=10, random_state=0)
    model.fit(X_train, y_train)
    preds = model.predict(X_valid)
    return mean_absolute_error(y_valid, preds)
```

Missing Values Example:

Data Cleaning

Now we will see the differences between the three approaches, and which one is suitable for our dataset.

1. A simple option: Drop columns with missing values:

Since we are working with both training and validation sets, we are careful to drop the same columns in both DataFrames.

```
cols_with_missing = [col for col in X_train.columns
                     if X_train[col].isnull().any()]

reduced_X_train = X_train.drop(cols_with_missing, axis=1)
reduced_X_valid = X_valid.drop(cols_with_missing, axis=1)

print("MAE from Approach 1 (Drop columns with missing values):")
print(score_dataset(reduced_X_train, reduced_X_valid, y_train, y_valid))
```

✓ 1.2s

```
MAE from Approach 1 (Drop columns with missing values):
183550.22137772635
```

Missing Values Example:

Data Cleaning

Now we will see the differences between the three approaches, and which one is suitable for our dataset.

2. A better option: Imputation

Next, we use Simple to replace missing values with the mean value along each column.

Although it's simple, filling in the mean value generally performs quite well (but this varies by dataset). While statisticians have experimented with more complex ways to determine imputed values (such as **regression imputation**, for instance), the complex strategies typically give no additional benefit once you plug the results into sophisticated machine learning models.

```
from sklearn.impute import SimpleImputer

my_imputer = SimpleImputer()
imputed_X_train = pd.DataFrame(my_imputer.fit_transform(X_train))
imputed_X_valid = pd.DataFrame(my_imputer.transform(X_valid))

# Imputation removed column names; put them back
imputed_X_train.columns = X_train.columns
imputed_X_valid.columns = X_valid.columns

print("MAE from Approach 2 (Imputation):")
print(score_dataset(imputed_X_train, imputed_X_valid, y_train, y_valid))
```

✓ 0.9s

```
MAE from Approach 2 (Imputation):
178166.46269899711
```

Missing Values Example:

Data Cleaning

Now we will see the differences between the three approaches, and which one is suitable for our dataset.

3. An extension to imputation

Next, we impute the missing values, while also keeping track of which values were imputed.

```
X_train_plus = X_train.copy()
X_valid_plus = X_valid.copy()

# Make new columns indicating what will be imputed
for col in cols_with_missing:
    X_train_plus[col + '_was_missing'] = X_train_plus[col].isnull()
    X_valid_plus[col + '_was_missing'] = X_valid_plus[col].isnull()

# Imputation
my_imputer = SimpleImputer()
imputed_X_train_plus = pd.DataFrame(my_imputer.fit_transform(X_train_plus))
imputed_X_valid_plus = pd.DataFrame(my_imputer.transform(X_valid_plus))

# Imputation removed column names; put them back
imputed_X_train_plus.columns = X_train_plus.columns
imputed_X_valid_plus.columns = X_valid_plus.columns

print("MAE from Approach 3 (An Extension to Imputation):")
print(score_dataset(imputed_X_train_plus, imputed_X_valid_plus, y_train, y_valid))
```

✓ 1.4s

```
MAE from Approach 3 (An Extension to Imputation):
178927.503183954
```

Missing Values Example:

Data Cleaning

<u>So why did imputation perform better than dropping the columns ?</u>

The training data has 10864 rows and 12 columns, where three columns contain missing data. For each column, less than half of the entries are missing. Thus, dropping the columns removes a lot of useful information, and so it makes sense that imputation would perform better.

```
print(X_train.shape)

# Number of missing values in each column of training data
missing_val_count_by_column = (X_train.isnull().sum())
print(missing_val_count_by_column[missing_val_count_by_column > 0])
```

✓ 0.1s

```
(10864, 12)
Car               49
BuildingArea    5156
YearBuilt       4307
dtype: int64
```

Duplicates

Data Cleaning

Data can have missing values for various reasons, such as errors in data collection, transmission losses, or simply because some information was not applicable. Handling missing data requires careful consideration to avoid introducing bias or inaccuracies.

Ways of handling Duplicated data:

- Identifying Duplicates: Finding exact/approximate repeats based on one or more key columns.
- Standardize categorical data and format numeric data
- Checking if duplicates are legitimate to avoid valid data removal.
- Drop/remove duplicated rows

Hint: In some cases, like undersampling, the duplicates will give you better results.

Duplicates Example

Data Cleaning

First, let's explore the dataset:

```
df2 = pd.read_csv('UsedCarsSA_Unclean_EN.csv')
df2.head()
```

✓ 0.1s Python

Link	Make	Type	Year	Origin	Color	Options	Engine_Size	Fuel_Type	Gear_Type	Condition	Mileage	Region
A7%D9%8A%D8...	Chrysler	C300	2018	Saudi	Black	Full	5.7	Gas	Automatic	Used	103000	Riyadh
B3%D8%A7%D9...	Nissan	Patrol	2016	Saudi	White	Full	4.8	Gas	Automatic	Used	5448	Riyadh
%8A%D8%B3%D...	Nissan	Sunny	2019	Saudi	Silver	Standard	1.5	Gas	Automatic	Used	72418	Riyadh
%88%D9%86%D...	Hyundai	Elantra	2019	Saudi	Grey	Standard	1.6	Gas	Automatic	Used	114154	Riyadh
%88%D9%86%D...	Hyundai	Elantra	2019	Saudi	Silver	Semi Full	2.0	Gas	Automatic	Used	41912	Riyadh

```
print(df.shape)
```

✓ 0.8s Python

```
(13580, 21)
```

It has 13580 rows and 21 columns.

Duplicates Example

Data Cleaning

Here we have 20 duplicates, and we can show them as well:

```python
df2.duplicated().sum()
```

✓ 0.1s Python

```
20
```

```python
dups = df2.duplicated()
df2[dups]
```

✓ 0.1s Python

Link	Make	Type	Year	Origin	Color	Options	Engine_Size	Fuel_Type	Gear_Type	Condition	Mileage	Region
n/%D8%B4%D9%8A%D9%81%D8%B1%D9...	Chevrolet	Impala	2016	Gulf Arabic	White	Standard	3.5	Gas	Automatic	Used	114000	Riyadh
n/%D9%81%D9%88%D8%B1%D8%AF_%D...	Ford	Explorer	2016	Gulf Arabic	Black	Standard	3.5	Gas	Automatic	Used	111000	Riyadh
/%D8%AA%D9%88%D9%8A%D9%88%D8...	Toyota	Camry	2016	Gulf Arabic	White	Standard	2.5	Gas	Automatic	Used	81000	Riyadh
/%D8%AA%D9%88%D9%8A%D9%88%D8...	Toyota	Camry	2016	Gulf Arabic	White	Semi Full	2.4	Gas	Automatic	Used	78000	Riyadh
n/%D8%B4%D9%8A%D9%81%D8%B1%D9...	Chevrolet	Tahoe	2016	Gulf Arabic	Golden	Semi Full	NaN	Gas	Automatic	Used	97000	Riyadh
n/%D8%B4%D9%81%D8%B1%D9%88%D9...	Chevrolet	Spark	2019	Saudi	Silver	Standard	1.4	Gas	Automatic	Used	63744	Riyadh
n/%D8%B4%D9%81%D8%B1%D9%88%D9...	Chevrolet	Caprice	2013	Saudi	Silver	Standard	6.2	Gas	Automatic	Used	55769	Riyadh

Duplicates Example

Data Cleaning

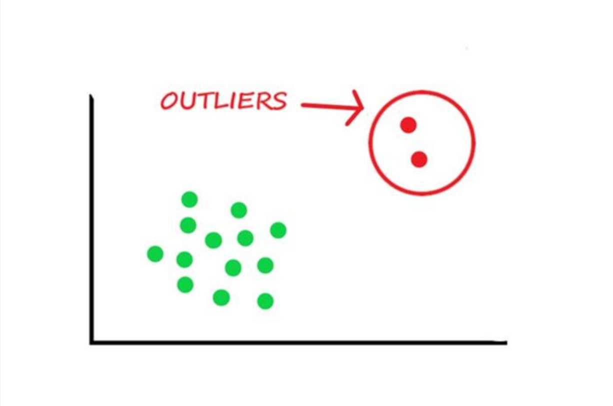
We dropped the duplicates and as we see there are no duplicates.

```
df2=df2.drop_duplicates()
✓ 0.1s

df2.duplicated().sum()
✓ 0.6s

0
```

Outliers

Data Cleaning

Outliers are data points that significantly differ from the rest of the dataset. They can indicate variability in measurement, experimental errors, or novel phenomena. Outliers can skew data analysis and may need to be examined separately or removed for accurate results.



Detection:

- Statistical Tests: Z-score, IQR (Interquartile Range) method.
- Visualization: Box plots, scatter plots to visually identify outliers.

Handling:

- Removal: Eliminate outliers if they result from errors.
- Adjustment: Transform data to reduce the impact of outliers.
- Separate Analysis: Analyze outliers separately if they represent valuable information.

Outliers Example

Data Cleaning

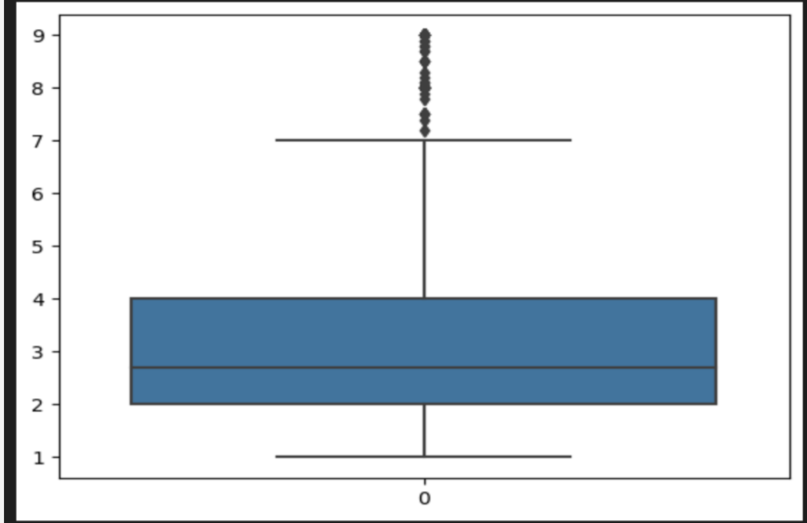
Here is an example for outliers using boxplot:

As we can see there are outliers above 7

```
import seaborn as sns
sns.boxplot(df2['Engine_Size'])
```

✓ 0.7s

<Axes: >



Outliers Detection IQR

Data Cleaning

IQR (Interquartile Range):

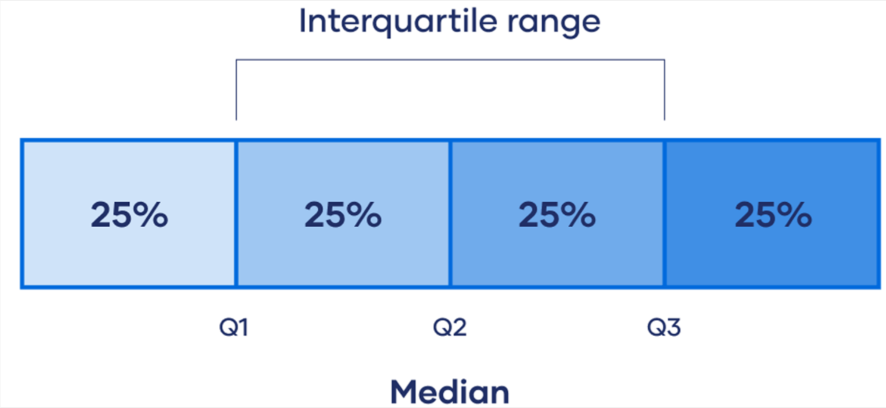
- Measures the spread of the middle 50% of data.

Calculation:

- Subtract the first quartile (Q1, 25th percentile) from the third quartile (Q3, 75th percentile).

Use:

- Identifies outliers by setting thresholds (typically 1.5*IQR above Q3 and below Q1) beyond which data points are considered outliers.



Data Quality Issues

Data Cleaning

Challenge:

- Variations in date formats (DD/MM/YYYY vs. MM/DD/YYYY), numerical representations (1,000 vs. 1000), and text casing (lowercase, UPPERCASE).

Impact:

- Complicates data parsing, aggregation, and analysis.

Solution Strategies:

- Standardize formats upon data ingestion.
- Implement validation rules to enforce consistency.

Challenge:

- Differing codes or names referring to the same items (e.g., "US" vs. "USA", "New York" vs. "NY").

Impact:

- Leads to fragmented datasets, making it difficult to aggregate or compare data accurately.

Solution Strategies:

- Develop a data dictionary or mapping table for custom business domain standardization.
- Use data standardization such as NLTK tools to normalize data based on established conventions.

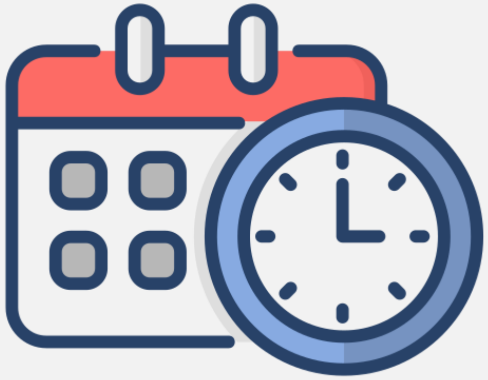
Introduction to Date and Time in Pandas

Handling date and time data effectively is crucial in data analysis and manipulation. Pandas offers robust tools to work with date and time data, which helps in ensuring accuracy and consistency. Here’s an explanation of the small subjects under this topic:

1. Importance of Handling Date and Time Data:

Date and time data are often essential in various analyses, such as time series forecasting, trend analysis, and event tracking. Correctly handling these data types allows you to:

- Perform accurate time-based calculations and aggregations.
- Analyze data over specific time periods.
- Identify trends and patterns that depend on the temporal aspect.
- Ensure consistency and correctness in reports and visualizations.



Introduction to Date and Time in Pandas

Data Cleaning

2. Common Issues with Date and Time Data

When working with date and time data, you might encounter several issues, including:

- **Inconsistent Formats**: Dates may come in various formats (e.g., YYYY-MM-DD, MM/DD/YYYY, DD-MM-YYYY), making it difficult to standardize and parse them correctly.
- **Missing Values**: Missing or null date values can cause errors in calculations and aggregations.
- **Time Zones**: Date and time data from different sources may use different time zones, requiring conversion for accurate analysis.
- **Ambiguities**: Some dates may be ambiguous (e.g., 01/02/2023 could mean January 2nd or February 1st, depending on the format).

Introduction to Date and Time in Pandas

Data Cleaning

3. Overview of Pandas Datetime Capabilities

Pandas provides powerful and flexible tools for handling date and time data. Some of its key capabilities include:

- **Conversion**: Easily convert strings and other formats to pandas datetime objects using pd.to_datetime.
- **Extraction**: Extract components like year, month, day, hour, minute, and second from datetime objects.
- **Time Zones**: Localize datetime objects to specific time zones and convert between time zones.
- **Date Arithmetic**: Perform arithmetic operations like addition and subtraction with dates and times.
- **Handling Missing Dates**: Use methods to fill or drop missing date values.
- **Resampling and Frequency Conversion**: Resample time series data to different frequencies (e.g., daily, monthly).
- **Working with Periods**: Use periods for representing spans of time and performing period-specific operations.

Parsing Dates

Data Cleaning

Using pd.to_datetime: Convert strings to datetime objects.

```
import pandas as pd

date_series = pd.Series(['2023-01-01', '2023-01-02', '2023-01-03'])
print(date_series)
date_series = pd.to_datetime(date_series)
print(date_series)
```

✓ 0.7s

```
0    2023-01-01
1    2023-01-02
2    2023-01-03
dtype: object
0   2023-01-01
1   2023-01-02
2   2023-01-03
dtype: datetime64[ns]
```

Handling Different Date Formats: Specifying the format.

```
date_series = pd.to_datetime(date_series, format='%Y/%m/%d')
```

Extracting Date and Time Components

Data Cleaning

- Extracting components like year, month, day, hour, minute, and second.

```
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day'] = df['date'].dt.day
df['hour'] = df['date'].dt.hour
df['minute'] = df['date'].dt.minute
df['second'] = df['date'].dt.second
df
```

✓ 0.1s

	transaction_id	date	sales_amount	year	month	day	hour	minute	second
0	1	2023-01-01 10:00:00	100	2023	1	1	10	0	0
1	2	2023-01-02 12:30:00	200	2023	1	2	12	30	0
2	3	2023-01-03 14:45:00	150	2023	1	3	14	45	0
3	4	2023-01-04 16:00:00	300	2023	1	4	16	0	0
4	5	2023-01-05 18:30:00	250	2023	1	5	18	30	0

Converting Time Zones

Data Cleaning

- Use *dt.tz_localize* and *dt.tz_convert*

```
df['date_utc'] = df['date'].dt.tz_localize('UTC')
df['date_eastern'] = df['date_utc'].dt.tz_convert('US/Eastern')

df
```

✓ 0.6s Python Python

	transaction_id	date	sales_amount	year	month	day	hour	minute	second	date_utc	date_eastern
0	1	2023-01-01 10:00:00	100	2023	1	1	10	0	0	2023-01-01 10:00:00+00:00	2023-01-01 05:00:00-05:00
1	2	2023-01-02 12:30:00	200	2023	1	2	12	30	0	2023-01-02 12:30:00+00:00	2023-01-02 07:30:00-05:00
2	3	2023-01-03 14:45:00	150	2023	1	3	14	45	0	2023-01-03 14:45:00+00:00	2023-01-03 09:45:00-05:00
3	4	2023-01-04 16:00:00	300	2023	1	4	16	0	0	2023-01-04 16:00:00+00:00	2023-01-04 11:00:00-05:00
4	5	2023-01-05 18:30:00	250	2023	1	5	18	30	0	2023-01-05 18:30:00+00:00	2023-01-05 13:30:00-05:00

Introduction to Scaling and Normalization

What is scaling ?

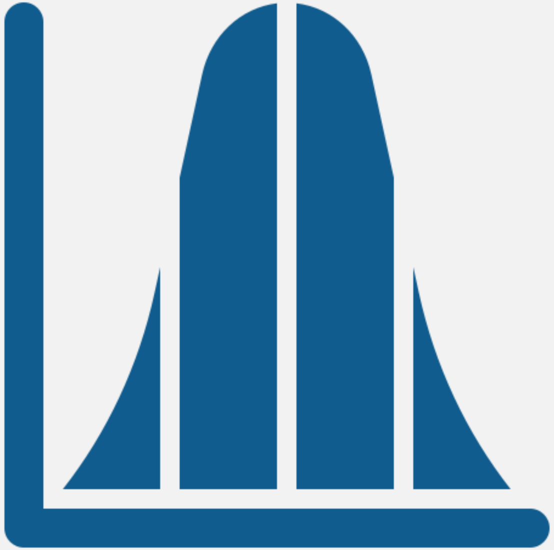
- Scaling refers to the process of transforming data to fit within a specific range, often 0 to 1 or -1 to 1.
- It is essential for algorithms that compute distances between data points, like k-nearest neighbors and clustering.

What is Normalization ?

- Normalization refers to adjusting the values in the dataset to a common scale, without distorting differences in the ranges of values.
- It ensures that no feature dominates due to its scale.

What is standardization ?

- Standardization (or Z-score normalization) transforms the data to have a mean of zero and a standard deviation of one.
- It centers the data around the mean and scales by the standard



Why Scaling and Normalization are Important

Consistency Across Features

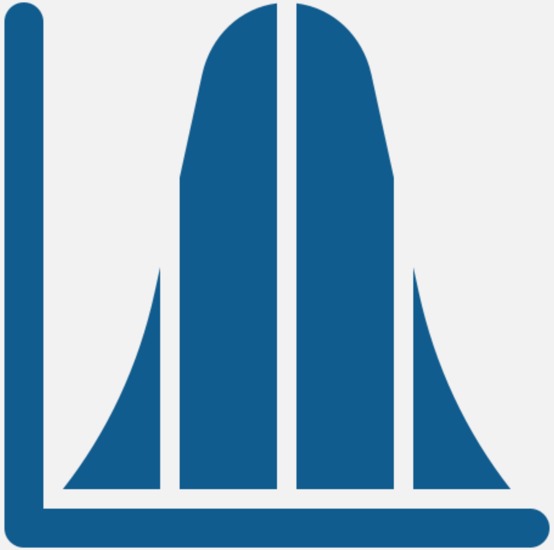
- Ensures that all features contribute equally to the result, preventing any single feature from disproportionately influencing the model.

Improved model performance

- Many machine learning algorithms perform better or converge faster when features are on a similar scale.

Handling Outliers

- Scaling and normalization can reduce the impact of outliers on the model.



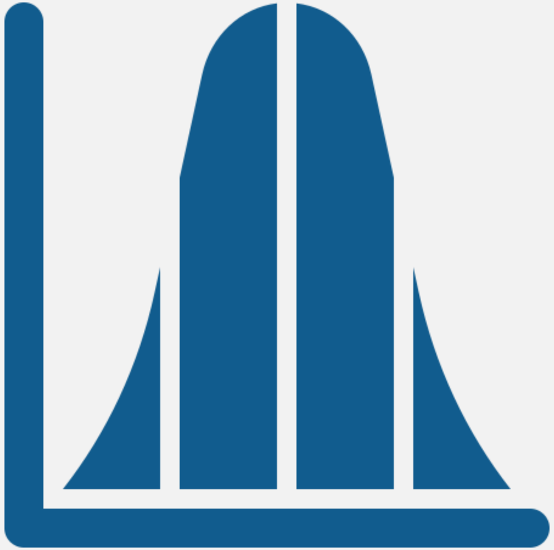
Types of Scaling Techniques

Min-Max scaling

- Rescales the feature to a fixed range, usually 0 to 1.
- Formula: $x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$

Standardization (Z-score normalization)

- Centers the feature by subtracting the mean and scales by the standard deviation.
- Formula: $Z = \frac{X - \mu}{\sigma}$



Implementing Min-Max Scaling

Data Cleaning

```python
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

# Sample DataFrame
data = {'Feature1': [10, 20, 30, 40, 50],
        'Feature2': [100, 200, 300, 400, 500]}
df = pd.DataFrame(data)
#print the data before scaling
print(df)
# Initialize the MinMaxScaler
scaler = MinMaxScaler()

# Fit and transform the data
scaled_df = pd.DataFrame(scaler.fit_transform(df), columns=df.columns)

print(scaled_df)
```

✓ 0.6s

```
   Feature1  Feature2
0        10       100
1        20       200
2        30       300
3        40       400
4        50       500
   Feature1  Feature2
0      0.00      0.00
1      0.25      0.25
2      0.50      0.50
3      0.75      0.75
4      1.00      1.00
```

Implementing Standardization

Data Cleaning

```
# Sample DataFrame
data = {'Feature1': [10, 20, 30, 40, 50],
        'Feature2': [100, 200, 300, 400, 500]}
df = pd.DataFrame(data)
#print the data before scaling
print(df)

# Initialize the StandardScaler
scaler = StandardScaler()

# Fit and transform the data
standardized_df = pd.DataFrame(scaler.fit_transform(df), columns=df.columns)
print("\n")
print(standardized_df)
```

✓ 0.5s

```
   Feature1  Feature2
0        10       100
1        20       200
2        30       300
3        40       400
4        50       500


   Feature1  Feature2
0 -1.414214 -1.414214
1 -0.707107 -0.707107
2  0.000000  0.000000
3  0.707107  0.707107
4  1.414214  1.414214
```

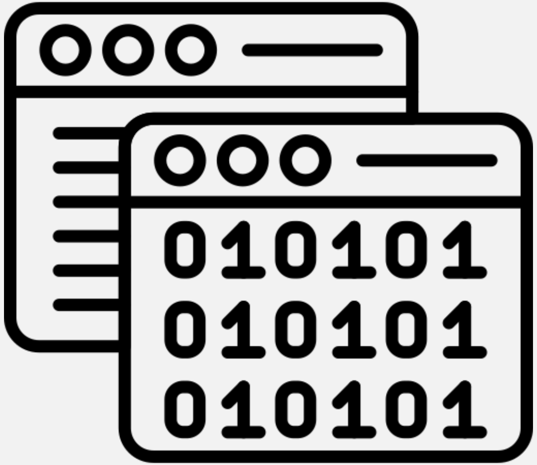
Introduction to Categorical Variables

What are Categorical Variables?

- Variables that represent categories or groups.
- Examples include gender, country, product type, etc.

Why encode categorical variable ?

- Many machine learning algorithms require numerical input.
- Encoding converts categorical data into a format that can be used in machine learning models.



Types of Encoding Techniques

Data Cleaning

One-Hot encoding:

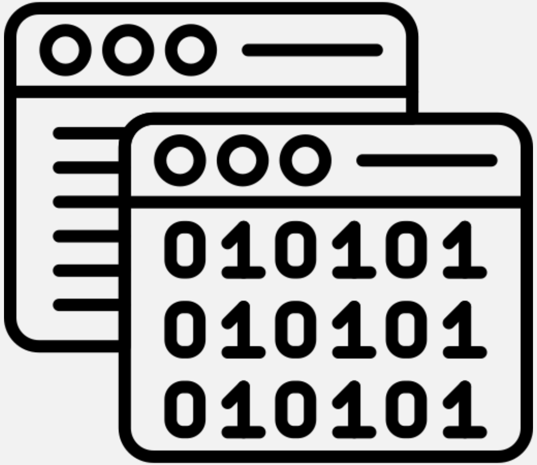
- Converts each category into a new binary column.
- Each column represents one category, with 1 indicating the presence and 0 indicating absence.

Label encoding:

- Assigns a unique integer to each category.
- Suitable for ordinal data where the order of categories is meaningful.

Ordinal Encoding:

- Similar to label encoding but maintains the order of categories.



Implementing One-Hot Encoding

Data Cleaning

```python
import pandas as pd

# Sample DataFrame
data = {'Color': ['Red', 'Blue', 'Green', 'Blue', 'Red']}
df = pd.DataFrame(data)

# One-Hot Encoding
print(df,"\n")
one_hot_encoded_df = pd.get_dummies(df, columns=['Color'])

print(one_hot_encoded_df)
```

✓ 0.8s

```
   Color
0    Red
1   Blue
2  Green
3   Blue
4    Red


   Color_Blue  Color_Green  Color_Red
0       False        False       True
1        True        False      False
2       False         True      False
3        True        False      False
4       False        False       True
```

Implementing Label Encoding

Data Cleaning

```python
import pandas as pd
from sklearn.preprocessing import LabelEncoder

# Sample DataFrame
data = {'Color': ['Red', 'Blue', 'Green', 'Blue', 'Red']}
df = pd.DataFrame(data)

# Initialize the LabelEncoder
label_encoder = LabelEncoder()

# Fit and transform the data
df['Color_Encoded'] = label_encoder.fit_transform(df['Color'])

print(df)
```

✓ 0.9s

```
   Color  Color_Encoded
0    Red              2
1   Blue              0
2  Green              1
3   Blue              0
4    Red              2
```

Implementing Ordinal Encoding

Data Cleaning

```python
import pandas as pd

# Sample DataFrame
data = {'Size': ['Small', 'Medium', 'Large', 'Medium', 'Small']}
df = pd.DataFrame(data)

# Define the mapping for ordinal encoding
size_mapping = {'Small': 1, 'Medium': 2, 'Large': 3}

# Apply the mapping
df['Size_Encoded'] = df['Size'].map(size_mapping)

print(df)
```

✓ 0.9s

```
     Size  Size_Encoded
0   Small             1
1  Medium             2
2   Large             3
3  Medium             2
4   Small             1
```

Comparing Encoding Techniques

One-Hot encoding:

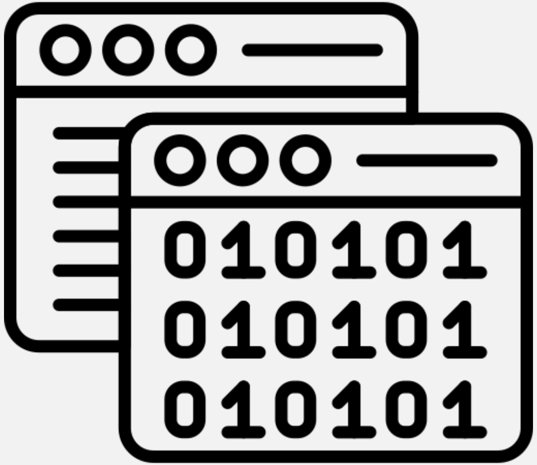
- Pros: No assumption about the order, handles non-ordinal data well.
- Cons: Increases the dimensionality of the data.

Label encoding:

- Pros: Simple, retains ordinal information.
- Cons: Can introduce unintended ordinal relationships.

Ordinal Encoding:

- Pros: Maintains the order of categories.
- Cons: Assumes the order is meaningful.

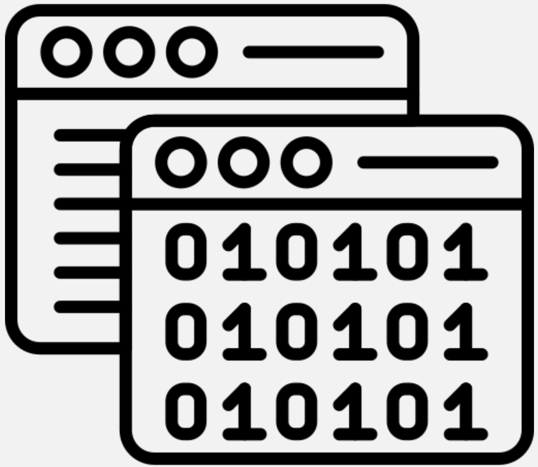


Conclusion

Data Cleaning

Key Takeaways:

- Encoding categorical variables is a crucial step in data preprocessing.
- Choose the encoding technique based on the nature of the data and the requirements of the machine learning algorithm.
- Proper encoding can significantly improve model performance.



Let’s Practice

Dataset Path:

3- Data Cleaning and Preprocessing with Python

/LAB/Titanic_Dataset.csv

Notebook Path:

3- Data Cleaning and Preprocessing with

/LAB/Data_Cleaning_Tutorial.ipynb

شكراً لكم

SDAIA
الهيئة السعودية للبيانات والذكاء الاصطناعي
Saudi Data & AI Authority