Machine Learning
Unit 1 Practical

1. Drop_duplicate() - Removing duplicate entries from a customer database.

DataFrame.drop_duplicates(subset=None, *, keep='first', inplace=False, ignore_index=False)

Return DataFrame with duplicate rows removed.

Parameters:

Subset column label or sequence of labels, optional

Only consider certain columns for identifying duplicates, by default use all of the columns.

Keep {'first', 'last', False}, default 'first'

Determines which duplicates (if any) to keep.

- 'first' : Drop duplicates except for the first occurrence.
- 'last' : Drop duplicates except for the last occurrence.
- False : Drop all duplicates.

Inplace bool, default False

Whether to modify the DataFrame rather than creating a new one.

ignore_index bool, default False

If True, the resulting axis will be labeled 0, 1, …, n - 1.

Returns:

DataFrame or None

DataFrame with duplicates removed or None if inplace=True.

Demo Code:

```
import pandas as pd

# Sample customer database
data = {
   "CustomerID": [1, 2, 3, 2, 4],
   "Name": ["Alice", "Bob", "Charlie", "Bob", "Diana"],
   "Email": ["alice@mail.com", "bob@mail.com", "charlie@mail.com", "bob@mail.com",
"diana@mail.com"],
}

df = pd.DataFrame(data)
print(df)
# Removing duplicates based on all columns
df_cleaned = df.drop_duplicates()

print("After Removing Duplicates:")
print(df_cleaned)
```

Detailed Explanation

Example: Removing Duplicate Entries from a Customer Database

Suppose we have the following DataFrame:

```
python Copy code
import pandas as pd

data = {
   'CustomerID': [101, 102, 101, 103, 102, 104],
   'Name': ['Alice', 'Bob', 'Alice', 'Charlie', 'Bob', 'David'],
   'Purchase': [250, 300, 250, 400, 300, 500]
}

df = pd.DataFrame(data)
print(df)
```

Output:

	CustomerID	Name	Purchase
0	101	Alice	250
1	102	Bob	300
2	101	Alice	250
3	103	Charlie	400
4	102	Bob	300
5	104	David	500

1. drop_duplicates() :

The method removes duplicate rows from a DataFrame based on the values in one or more columns. It helps in cleaning the data by ensuring each record (row) is unique according to specific criteria.

Example 1. Remove Duplicates Based on All Columns (Default Behavior):

```
cleaned_df = df.drop_duplicates()
print(cleaned_df)
```

Output:

	CustomerID	Name	Purchase
0	101	Alice	250
1	102	Bob	300
3	103	Charlie	400
5	104	David	500

2. subset:

- Definition: Specifies the columns to consider when identifying duplicates.
- Default: If not provided, all columns are used.
- Example: If you want to check duplicates only for `CustomerID`, set `subset=['CustomerID']`.

Example2. Remove Duplicates Based on a Specific Column (`CustomerID`):

```
cleaned_df = df.drop_duplicates(subset=['CustomerID'])

print(cleaned_df)
```

- Output:

	CustomerID	Name	Purchase
0	101	Alice	250
1	102	Bob	300
3	103	Charlie	400
5	104	David	500

3. keep:

- Definition: Specifies which duplicate (if any) to keep.
- Options:
  - `'first'` (default): Keeps the first occurrence of each duplicate row.
  - `'last'`: Keeps the last occurrence of each duplicate row.
  - `False`: Drops all occurrences of duplicate rows.
- Example: If there are three identical rows and `keep='first'`, the first row remains while the others are removed.

3. Keep the Last Duplicate (`keep='last'`):

```
cleaned_df = df.drop_duplicates(subset=['CustomerID'], keep='last')
print(cleaned_df)
```

Output:

	CustomerID	Name	Purchase
2	101	Alice	250
4	102	Bob	300
3	103	Charlie	400

5 104 David 500

3. Drop All Duplicates (`keep=False`):

```
cleaned_df = df.drop_duplicates(subset=['CustomerID'], keep=False)
print(cleaned_df)
```

Output:

	CustomerID	Name	Purchase
3	103	Charlie	400
5	104	David	500

4. inplace:

- Definition: If `True`, modifies the original DataFrame instead of creating a new one.
- Default: `False`.
- Example: If `inplace=True`, no new DataFrame is returned, and the original DataFrame is updated directly.

4. Modify DataFrame In Place (`inplace=True`):

```
df.drop_duplicates(subset=['CustomerID'], keep='first', inplace=True)
print(df)
```

Output:

	CustomerID	Name	Purchase
0	101	Alice	250
1	102	Bob	300
3	103	Charlie	400
5	104	David	500

5. ignore_index:

- Definition: If `True`, the resulting DataFrame's index is reset to sequential numbers (0, 1, 2, ...).
- Default: `False`.
- Example: If `ignore_index=True`, the new index starts from 0 even if the original DataFrame had a different index.

5. Reset Index After Removing Duplicates (`ignore_index=True`):

```
cleaned_df = df.drop_duplicates(subset=['CustomerID'], keep='first', ignore_index=True)
print(cleaned_df)
```

Output:

	CustomerID	Name	Purchase
0	101	Alice	250
1	102	Bob	300
2	103	Charlie	400
3	104	David	500

Return Value

- New DataFrame: If `inplace=False`, it returns a new DataFrame with duplicates removed.
- None: If `inplace=True`, it doesn't return anything, as the original DataFrame is modified in place.

Conclusion: `DataFrame.drop_duplicates()` is a powerful method for cleaning up duplicate data efficiently by providing options to specify columns, decide which duplicates to retain, and whether to modify the DataFrame in place.

2. Handling missing values in a dataset by imputation or deletion.

Demo Code:

```
import pandas as pd
# Example DataFrame with missing values
data = {
   'A': [1, 2, None, 4, 5],
   'B': [None, 2, 3, 4, 5],
   'C': [1, 2, 3, 4, 5]
}
df = pd.DataFrame(data)
# Delete rows with any missing values
df_cleaned_rows = df.dropna()
# Delete columns with any missing values
df_cleaned_columns = df.dropna(axis=1)
print("Original DataFrame:")
print(df)
print("\nDataFrame after dropping rows with missing values:")
print(df_cleaned_rows)
print("\nDataFrame after dropping columns with missing values:")
print(df_cleaned_columns)
```

Demo Code:

```
import pandas as pd

# Sample data with a `null` value
data = {
    "CustomerID": [1,None , 3, 2, 4],
    "Name": ["Alice", "Bob", "Charlie", "Bob", "Diana"],
    "Email": ["alice@mail.com", "bob@mail.com", "charlie@mail.com", None, "diana@mail.com"],
}

df = pd.DataFrame(data)

# Print the original DataFrame
print("Original DataFrame:")
print(df)

# Remove duplicates based on all columns
df_cleaned = df.drop_duplicates()

# Replace null values with a specified value (e.g., 'Unknown' for Email)
df_cleaned['Email'] = df_cleaned['Email'].fillna('Unknown')
df_cleaned['CustomerID'] = df_cleaned['CustomerID'].fillna(df_cleaned['CustomerID'].mean())

# Print the cleaned DataFrame
print("\nAfter Removing Duplicates and Replacing Nulls:")
print(df_cleaned)
```

Demo Code:

```
import pandas as pd

# Sample data with a `null` value
data = {
    "CustomerID": [1,None , 3, 2, 4],
    "Name": ["Alice", "Bob", "Charlie", "Bob", "Diana"],
    "Email": ["alice@mail.com", "bob@mail.com", "charlie@mail.com", None, "diana@mail.com"],
}

df = pd.DataFrame(data)

# Print the original DataFrame
print("Original DataFrame:")
print(df)
```

```
# Remove duplicates based on all columns
df_cleaned = df.drop_duplicates()

# Replace null values with a specified value (e.g., 'Unknown' for Email)
df_cleaned['Email'] = df_cleaned['Email'].fillna('Unknown')
df_cleaned['CustomerID'] = df_cleaned['CustomerID'].fillna(df_cleaned['CustomerID'].mean())

# Print the cleaned DataFrame
print("\nAfter Removing Duplicates and Replacing Nulls:")
print(df_cleaned)
```

3. Correcting inconsistencies in data entry (e.g., standardizing date formats).

Pandas to_datetime() method helps to convert string Date time into Python Date time object.

Sample Code:

```
import pandas as pd

# Sample data with inconsistent date formats
data = {
    'name': ['Alice', 'Bob', 'Charlie'],
    'date_of_birth': ['01/15/1990', '1991-07-12', 'March 4, 1992']
}

# Create DataFrame
df = pd.DataFrame(data)

# Print before standardization
print("Before standardization:")
print(df)

# Convert the 'date_of_birth' column to datetime, coercing errors to NaT (Not a Time)
df['date_of_birth'] = pd.to_datetime(df['date_of_birth'], errors='coerce')

# Print after standardization
print("\nAfter standardization:")
print(df)
```

4. Filtering out irrelevant or erroneous data points based on predefined criteria.

Demo Code:

```
import pandas as pd

# Example dataset (this can be replaced with your own dataset)
data = {
    'age': [25, 27, 15, 45, 100, -5],
    'salary': [50000, 60000, 120000, 150000, 30000, None],
    'experience': [2, 5, 0, 10, 0, 20],
    'location': ['NY', 'LA', 'SF', 'NY', 'LA', 'SF']
}

# Create DataFrame
df = pd.DataFrame(data)

# Show the original data
print("Original Data:")
print(df)

# Filter based on criteria
# Filter out rows where 'age' is less than 18 or greater than 100
filtered_df = df[
    (df['age'] >= 18) & (df['age'] <= 100)

]
# Show filtered data
print("\nFiltered Data:")
print(filtered_df)
```

Sample Code:

```
import pandas as pd

# Example dataset (this can be replaced with your own dataset)
data = {
    'age': [25, 27, 15, 45, 100, -5],
    'salary': [50000, 60000, 120000, 150000, 30000, None],
    'experience': [2, 5, 0, 10, 0, 20],
    'location': ['NY', 'LA', 'SF', 'NY', 'LA', 'SF']
}

# Create DataFrame
df = pd.DataFrame(data)

# Show the original data
print("Original Data:")
```

```
print(df)

# Filter based on criteria
# Filter out rows where 'salary' is less than 30,000 or missing (None or NaN)

filtered_df = df[
    (df['salary'] >= 30000) & (~df['salary'].isna())   # Salary criteria (non-missing and > 30000)
]

# Show filtered data
print("\nFiltered Data:")
print(filtered_df)
```

Sample Code:

```
import pandas as pd

# Example dataset (this can be replaced with your own dataset)
data = {
   'age': [25, 27, 15, 45, 100, -5],
   'salary': [50000, 60000, 120000, 150000, 30000, None],
   'experience': [2, 5, 0, 10, 0, 20],
   'location': ['NY', 'LA', 'SF', 'NY', 'LA', 'SF']
}

# Create DataFrame
df = pd.DataFrame(data)

# Show the original data
print("Original Data:")
print(df)

# Filter based on criteria
# Filter out rows where 'experience' is less than 1 or greater than 30

filtered_df = df[
   (df['experience'] >= 1) & (df['experience'] <= 30) # Experience criteria
]

# Show filtered data
print("\nFiltered Data:")
print(filtered_df)
```

5 . Cleaning textual data by removing HTML tags, punctuation, and special characters.

`re.sub(pattern, replacement, text)`:

```
Sample Code:
import re
def remove_html_tags(text):
    clean_text = re.sub(r'<.*?>', '', text)
    return clean_text
def remove_special_characters(text):
    clean_text = re.sub(r'[^a-zA-Z0-9\s]', '', text)
    return clean_text

text = "Hello #GLS @Ahmedabad Studnets <p>  <br>  at www.glsuniversity.ac.in"
print(text)

remove_html = remove_html_tags(text)
print(remove_html)

remove_sChar = remove_special_characters(remove_html)
print(remove_sChar)
```

Explanation : clean_text = re.sub(r'<.*?>', '', text)

`<` matches the opening angle bracket of an HTML tag.

`.*?` matches any character (except newlines) as few times as possible until it finds the next `>`.

`>` matches the closing angle bracket of the HTML tag.

Data Transformation

1. Converting categorical variables into numerical representations using one-hot encoding or label encoding.

Various Machine Learning models do not work with categorical data and to fit this data into the machine learning model it needs to be converted into numerical data. For example, suppose a dataset has a *Gender* column with categorical elements like *Male and Female*.

To address this issue, one effective technique is one hot encoding. OHE in machine learning transforms categorical data into a numerical format

Fruit	Categorical value of fruit	Price
apple	1	5
mango	2	10
apple	1	15
orange	3	20

The output after applying one-hot encoding on the data is given as follows,

Fruit_apple	Fruit_mango	Fruit_orange	price
1	0	0	5

0	1	0	10
1	0	0	15
0	0	1	20

Demo Code:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler, LabelEncoder, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.feature_extraction.text import CountVectorizer
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize
from sklearn.preprocessing import FunctionTransformer

# Sample dataset (including categorical, numerical, text, and date columns)
data = {
    'Category': ['A', 'B', 'A', 'B', 'C', 'A', 'C', 'B'],
    'Age': [23, 45, 12, 67, 29, 45, 34, 22],
    'Income': [50000, 60000, 25000, 80000, 55000, 62000, 35000, 48000],
    'Transaction_Date': ['2022-12-01', '2022-12-02', '2022-12-01', '2022-12-03', '2022-12-02', '2022-12-
01', '2022-12-03', '2022-12-01'],
    'Text': ['I love machine learning', 'Python is great for data', 'Deep learning is powerful', 'I enjoy
coding', 'Data science is fun', 'Python machine learning', 'Deep learning', 'I love Python'],
    'Target': [1, 0, 1, 0, 1, 0, 1, 0]
}

# Creating a DataFrame
df = pd.DataFrame(data)
print(df)
```

```
# Step 1: Converting categorical variables into numerical representations using One-Hot Encoding
df = pd.get_dummies(df, columns=['Category'], drop_first=True)
print(df)
```

```
from sklearn.preprocessing import OneHotEncoder
import numpy as np

# Creating the encoder
enc = OneHotEncoder(handle_unknown='ignore')

# Sample data
X = [['Red'], ['Green'], ['Blue']]

# Fitting the encoder to the data
enc.fit(X)

# Transforming new data
result = enc.transform([['Red']]).toarray()

# Displaying the encoded result
print(result)
```

2. Scaling numerical features to a standard range using Min-Max scaling or Z-score normalization.

```
#The MinMax scaler is one of the simplest scalers to understand.  It just scales all the data between 0 and 1. The formula for calculating the scaled value is-
#x_scaled = (x – x_min)/(x_max – x_min)
from sklearn.preprocessing import MinMaxScaler

import numpy as np

# Create a sample dataset

data = np.array([[1, 2], [2, 4], [3, 6]])

# Create a scaler object

scaler = MinMaxScaler()
```

```
# Scale the data

scaled_data = scaler.fit_transform(data)

print(scaled_data)
```

3. Transforming skewed distributions using log or square root transformations.

Data transformation is a crucial preprocessing step, especially when dealing with skewed distributions. skewness is an imbalance in the distribution of a training dataset's target variable. It can affect the accuracy of predictive models.
A skewed distribution is a data set that is not symmetrical, meaning the data points are clustered more toward one side of the scale than the other. When graphed, a skewed distribution has a peak on one side and a tail extending to the other side.

Skewness can affect the performance of certain algorithms, such as linear regression, logistic regression, and some tree-based models. Transformations like **log or square root can help reduce skewness** and make data more normally distributed, which is often more suitable for modeling.

Demo Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Example DataFrame with a skewed distribution
data = {'value': [1, 2, 3, 4, 5, 10, 20, 50, 100, 200, 500, 1000, 2000]}
df = pd.DataFrame(data)

# Visualizing the original data
```

```
plt.plot(df['value'])
plt.title('Original Distribution')
plt.show()

# Apply Log Transformation (log base e)
df['log_value'] = np.log(df['value'])

# Visualizing the log-transformed data
plt.plot(df['log_value'])
plt.title('Log-transformed Distribution')
plt.show()
```

Demo Code:

```
# Apply Square Root Transformation
df['sqrt_value'] = np.sqrt(df['value'])

# Visualizing the square root-transformed data
plt.plot(df['log_value'])
plt.title('Square Root-transformed Distribution')
plt.show()
```

4. Applying text preprocessing techniques such as tokenization, stemming, and lemmatization to textual data

Demo Code:

```
df['Text_Tokenized'] = df['Text'].apply(word_tokenize)
print("\nTokenized Text Data:")
print(df[['Text', 'Text_Tokenized']])
# Step 3: Stemming
stemmer = PorterStemmer()
df['Text_Stemmed'] = df['Text_Tokenized'].apply(lambda tokens: [stemmer.stem(word)
for word in tokens])
print("\nStemmed Text Data:")
print(df[['Text', 'Text_Stemmed']])
```

5. Encoding temporal data into meaningful features such as day of the week or time of day.

Encoding temporal data (like timestamps or datetime values) into meaningful features is an important preprocessing step in many machine learning tasks. By extracting temporal features such as **day of the week**, **hour of the day**, **month**, **season**, and others, you can help the machine learning model capture time-related patterns in the data.

First, ensure that the temporal data is in the correct `datetime` format. If it's not already, you can use `pd.to_datetime()` to convert it.

Extract meaningful features from the datetime

Once you have the `datetime` object, you can extract several features, including:

- **Day of the Week** (0-6, where 0 is Monday)
- **Hour of the Day** (0-23)
- **Month** (1-12)
- **Weekday vs. Weekend** (binary)
- **Is Holiday** (if you have holiday data)
- **Season** (categorical, based on month)
- **Day of the Month**
- **Quarter of the Year** (1, 2, 3, 4)

Demo Code:

```
import pandas as pd
import numpy as np

# Sample DataFrame with 'timestamp' column
data = {'timestamp': ['2025-01-06 08:23:45', '2025-06-15 14:12:00', '2025-12-25 18:35:00']}
df = pd.DataFrame(data)

# Convert 'timestamp' column to datetime type
df['timestamp'] = pd.to_datetime(df['timestamp'])

# Extract day of the week (0=Monday, 6=Sunday)
df['day_of_week'] = df['timestamp'].dt.dayofweek

# Extract hour of the day (0 to 23)
df['hour_of_day'] = df['timestamp'].dt.hour

# Extract month of the year (1 to 12)
```

```
df['month'] = df['timestamp'].dt.month

# Extract day of the month (1 to 31)
df['day_of_month'] = df['timestamp'].dt.day

# Extract quarter of the year (1 to 4)
df['quarter'] = df['timestamp'].dt.quarter

# Extract whether it's a weekend or weekday (1 for weekend, 0 for weekday)
df['is_weekend'] = df['timestamp'].dt.dayofweek >= 5      # 5 and 6 are Saturday and Sunday

df['season'] = df['month'].apply(get_season)

# Show the transformed DataFrame
print(df)
```