NITTE EDUCATION TRUST	Nitte Education Trust Nitte Meenakshi Institute of Technology
Department of Electronics and Communication Engineering	An Autonomous Institution Approved by UGC/AICTE/Govt. of Karnataka Accredited by NBA (Tier-1) and NAAC A+ Grade Affiliated to Visvesvaraya Technological University, Belagavi-590018. Post Box No. 6429, Yelahanka, Bengaluru-560064, Karnataka, India.

Data Pre-processing Techniques: Binning and One-Hot Encoding

AML LA-2 [Mini-project]

4TH SEM 'A' SECTION

By

Auchitya Jain **1NT22EC029**

Under the Guidance of

Dr. Vishwanath V

Assistant professor, ECE Dept.

NMIT

INTRODUCTION

Data pre-processing is a crucial step in machine learning that prepares raw data for use in algorithms. Two common techniques for pre-processing numerical and categorical data are binning and one-hot encoding.

Binning

Binning is a technique used to transform continuous data into discrete intervals. It essentially groups similar data points together into "bins" or "buckets."

- **Purpose:**
    - Binning simplifies complex data by reducing the number of unique values. This can improve the performance of certain machine learning algorithms that struggle with highly granular continuous data.
    - It can also help reduce the impact of noise or outliers in the data, as extreme values might be grouped with more common ones within a bin.
- **How it Works:**

1. **Define Bins:** You first divided into them. There are different approaches for defining bins:need to decide on the number of bins and how the data will be
    - **Equal-width bins:** Here, the data range is divided into a specific number of bins with equal width. This is a simple approach, but it may not be ideal for data with a skewed distribution.
    - **Quantile-based bins:** Here, the data is divided into a certain number of bins based on percentiles (quartiles, deciles, etc.). This can be more appropriate for skewed data distributions.
2. **Assign Data Points:** Each data point in the continuous variable is then assigned to a bin based on its value. The criteria for assignment depends on the chosen binning strategy (e.g., falling within a specific range for equal-width bins.

One-Hot Encoding

One-hot encoding is another data pre-processing technique used specifically for categorical data. It transforms categorical variables, which can have multiple string labels (e.g., "red", "green", "blue"), into numerical representations suitable for machine learning algorithms.

- **Purpose:**
    - Machine learning algorithms typically work better with numerical data. One-hot encoding converts categorical data into a binary vector representation, where each category has its own column. The value in a column is set to 1 if the data point belongs to that category, and 0 otherwise.

- This allows the algorithm to learn the relationships between different categories without assuming any inherent order or meaning between them (nominal data).

Key Differences:

Feature	**Binning**	**One-Hot Encoding**
Data Type	Continuous data	Categorical data
Output Format	Discrete intervals (bins)	Binary vectors
Purpose	Reduce complexity, improve algorithm	Represent categories numerically for ML

One Hot Encoder

CODE:-

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

#Building a dummy employee dataset for example
data = {'Employee id': [10, 20, 15, 25, 30],
        'Gender': ['M', 'F', 'F', 'M', 'F'],
        'Remarks': ['Good', 'Nice', 'Good', 'Great', 'Nice'],
        }
#Converting into a Pandas dataframe
df = pd.DataFrame(data)
#Print the dataframe:
print(f"Employee data : \n{df}")

#Extract categorical columns from the dataframe
#Here we extract the columns with object datatype as they are the categorical columns
categorical_columns = df.select_dtypes(include=['object']).columns.tolist()

#Initialize OneHotEncoder
encoder = OneHotEncoder(sparse_output=False)

# Apply one-hot encoding to the categorical columns
one_hot_encoded = encoder.fit_transform(df[categorical_columns])

#Create a DataFrame with the one-hot encoded columns
#We use get_feature_names_out() to get the column names for the encoded data
one_hot_df = pd.DataFrame(one_hot_encoded, columns=encoder.get_feature_names_out(categorical_columns))

# Concatenate the one-hot encoded dataframe with the original dataframe
df_encoded = pd.concat([df, one_hot_df], axis=1)

# Drop the original categorical columns
df_encoded = df_encoded.drop(categorical_columns, axis=1)

# Display the resulting dataframe
print(f"Encoded Employee data : \n{df_encoded}")
```

Imports (Lines 1-2):

1. import pandas as pd: This line imports the pandas library and assigns it the alias pd. Pandas is a popular Python library for data manipulation and analysis.
2. from sklearn.preprocessing import OneHotEncoder: This line imports the OneHotEncoder class from the preprocessing module of the scikit-learn library. OneHotEncoder is used to convert categorical data into a one-hot encoded format.

Building Employee Dataset (Lines 4-8):

1. data = {'Employee id': [10, 20, 15, 25, 30], ... }: This line creates a dictionary named data. The dictionary stores employee information with keys like 'Employee id', 'Gender', and 'Remarks' and their corresponding values in lists.

2. df = pd.DataFrame(data): This line converts the dictionary data into a Pandas DataFrame named df. A DataFrame is a two-dimensional labeled data structure with columns and rows.

Printing the DataFrame (Line 9):

1. print(f"Employee data : \n{df}"): This line prints a formatted string that includes a message "Employee data" followed by the content of the DataFrame df with a new line character (\n) before it.

Extracting Categorical Columns (Lines 11-12):

1. categorical_columns = df.select_dtypes(include=['object']).columns.tolist():
    - This line does multiple things:
        - df.select_dtypes(include=['object']): This part selects columns from the DataFrame df that have the data type 'object'. In Python, 'object' is often used to represent string data.
        - .columns: This retrieves the column names of the selected columns.
        - .tolist(): This converts the column names from a Pandas Series to a regular Python list and stores it in the variable categorical_columns.

Initializing OneHotEncoder (Line 14):

1. encoder = OneHotEncoder(sparse_output=False): This line creates an instance of the OneHotEncoder class and assigns it to the variable encoder. The sparse_output parameter is set to False to ensure the output is a dense array (regular list of lists) instead of a sparse matrix (which can be more memory efficient for very large datasets).

Applying One-Hot Encoding (Line 16):

1. one_hot_encoded = encoder.fit_transform(df[categorical_columns]): This line applies the one-hot encoding to the categorical columns (df[categorical_columns]) of the DataFrame df using the encoder object. The .fit_transform() method both learns the possible categories from the data and then transforms the data into a one-hot encoded format. The result is stored in the variable one_hot_encoded.

Creating One-Hot Encoded DataFrame (Lines 18-19):

1. one_hot_df = pd.DataFrame(one_hot_encoded, columns=encoder.get_feature_names_out(categorical_columns)): This line creates a new DataFrame named one_hot_df.
    - pd.DataFrame(one_hot_encoded): This part creates a DataFrame from the one-hot encoded data stored in one_hot_encoded.
    - .get_feature_names_out(categorical_columns): This method of the encoder object retrieves the new column names generated for the encoded data based on the original column names in categorical_columns. These names are used to create the column labels for the new DataFrame.

Concatenating DataFrames (Line 21):

1. df_encoded = pd.concat([df, one_hot_df], axis=1): This line combines the original DataFrame df and the one-hot encoded DataFrame one_hot_df along axis 1 (which means placing the columns side-by-side) and stores the result in df_encoded.

Dropping Original Categorical Columns (Line 22):

1. df_encoded = df_encoded.drop(categorical_columns, axis=1): This line drops the original categorical columns (stored in categorical_columns) from the combined DataFrame df_encoded along axis 1 (columns) and assigns the result back to df_encoded.

Printing the Encoded DataFrame (Line 24):

1. `print(f"Encoded Employee data : \n{df_encoded}")

OUTPUT:-

```
Employee data :
   Employee id Gender Remarks
0           10      M    Good
1           20      F    Nice
2           15      F    Good
3           25      M   Great
4           30      F    Nice
Encoded Employee data :
   Employee id  Gender_F  Gender_M  Remarks_Good  Remarks_Great  Remarks_Nice
0           10       0.0       1.0           1.0            0.0           0.0
1           20       1.0       0.0           0.0            0.0           1.0
2           15       1.0       0.0           1.0            0.0           0.0
3           25       0.0       1.0           0.0            1.0           0.0
4           30       1.0       0.0           0.0            0.0           1.0
```

The output shows the original employee data alongside its one-hot encoded representation.

Original Data (Employee data):

This part displays the data you provided initially in a DataFrame format. It shows the following columns:

- Employee id: This column contains unique identifiers for each employee.
- Gender: This column indicates the employee's gender (M or F).
- Remarks: This column contains some remarks about each employee (Good, Nice, or Great).

Encoded Data (Encoded Employee data):

This part shows the result of applying one-hot encoding to the categorical columns (Gender and Remarks) in the original data. Here's a breakdown of the new columns:

- Employee id: This column remains unchanged from the original data.
- Gender_F: This column represents the gender "Female". It contains 1.0 for rows where the original Gender was "F" and 0.0 otherwise.
- Gender_M: This column represents the gender "Male". It contains 1.0 for rows where the original Gender was "M" and 0.0 otherwise.
- Remarks_Good: This column represents the remark "Good". It contains 1.0 for rows where the original Remarks was "Good" and 0.0 otherwise.
- Remarks_Great: This column represents the remark "Great". It contains 1.0 for rows where the original Remarks was "Great" and 0.0 otherwise.
- Remarks_Nice: This column represents the remark "Nice". It contains 1.0 for rows where the original Remarks was "Nice" and 0.0 otherwise.

By using one-hot encoding, we've transformed the categorical data (text labels) into numerical representations that are more suitable for certain machine learning algorithms. Each category now has its own column, and the value in that column indicates whether the corresponding category applies to that particular data point.

Binning Method

Code:-

```
import numpy as np
import math
from sklearn.datasets import load_iris
from sklearn import datasets, linear_model, metrics

# load iris data set
dataset = load_iris()
a = dataset.data
b = np.zeros(150)

# take 1st column among 4 column of data set
for i in range (150):
  b[i]=a[i,1]

b=np.sort(b) #sort the array

# create bins
bin1=np.zeros((30,5))
bin2=np.zeros((30,5))
bin3=np.zeros((30,5))

# Bin mean
for i in range (0,150,5):
  k=int(i/5)
  mean=(b[i] + b[i+1] + b[i+2] + b[i+3] + b[i+4])/5
  for j in range(5):
    bin1[k,j]=mean
print("Bin Mean: \n",bin1)

# Bin boundaries
for i in range (0,150,5):
  k=int(i/5)
  for j in range (5):
    if (b[i+j]-b[i]) < (b[i+4]-b[i+j]):
      bin2[k,j]=b[i]
    else:
      bin2[k,j]=b[i+4]
print("Bin Boundaries: \n",bin2)

# Bin median
for i in range (0,150,5):
  k=int(i/5)
  for j in range (5):
    bin3[k,j]=b[i+2]
print("Bin Median: \n",bin3)
```

Imports (Lines 1-4):

1. import numpy as np: This line imports the NumPy library and assigns it the alias np. NumPy provides powerful functions for numerical computations and array manipulation.
2. import math: This line imports the math library, but it's not used in the provided code snippet.
3. from sklearn.datasets import load_iris: This line imports the load_iris function from the datasets module of scikit-learn. This function is used to load the Iris flower dataset, a commonly used benchmark dataset in machine learning.
4. from sklearn import datasets, linear_model, metrics: This line imports several modules from scikit-learn, but only datasets is used in the provided code (for load_iris).

Loading Iris Data (Line 6):

1. dataset = load_iris(): This line loads the Iris flower dataset using the load_iris function and stores it in the variable dataset.

Extracting Specific Feature (Lines 8-10):

1. a = dataset.data: This line extracts the data from the loaded dataset and stores it in the variable a. The data is a NumPy array containing features for each data point (flower sample).
2. b = np.zeros(150): This line creates a new NumPy array named b filled with zeros. It has a size of 150, which matches the number of data points in the Iris dataset.
3. for i in range (150): b[i]=a[i,1]: This loop iterates through each data point (row) in the a array and assigns the value from the second column (index 1) to the corresponding element in the b array. This effectively extracts only the second feature from all data points.

Sorting the Feature (Line 11):

1. b=np.sort(b): This line sorts the values in the b array in ascending order using the np.sort function from NumPy.

Creating Bins (Lines 13-15):

1. bin1=np.zeros((30,5)): This line creates a NumPy array named bin1 filled with zeros. It has a shape of (30, 5), which means it has 30 rows and 5 columns. This will be used to store the bin means for 30 bins (although only 5 are calculated in this example).
2. bin2=np.zeros((30,5)): Similar to bin1, this line creates another zero-filled array bin2 with the same shape (30, 5) to store bin boundaries.
3. bin3=np.zeros((30,5)): Similar to the above, this line creates a zero-filled array bin3 with the shape (30, 5) to store bin medians.

Calculating Bin Means (Lines 17-23):

1. for i in range (0,150,5):: This loop iterates through the sorted b array with a step size of 5. This means it will process every 5th element, effectively taking groups of 5 consecutive values.
2. k=int(i/5): This line calculates the index for the current bin within the bin1 array. It divides the loop counter i by 5 and converts the result to an integer.

3. mean=(b[i] + b[i+1] + b[i+2] + b[i+3] + b[i+4])/5: This line calculates the average (mean) of the 5 values in the current group (b[i] to b[i+4]).
4. for j in range(5): bin1[k,j]=mean: This inner loop iterates through all 5 elements in the current group and assigns the calculated mean value to all corresponding elements in the same row (k) of the bin1 array. This effectively fills each row of bin1 with the same mean value, although ideally it should represent the mean of a specific bin.

Calculating Bin Boundaries (Lines 25-32):

1. for i in range (0,150,5):: This loop iterates similarly to the mean calculation loop.
2. k=int(i/5): Similar to before, this line calculates the index for the current bin within the bin2 array.

OUTPUT:-

```
Bin Mean:
 [[2.18 2.18 2.18 2.18 2.18]
 [2.34 2.34 2.34 2.34 2.34]
 [2.48 2.48 2.48 2.48 2.48]
 [2.52 2.52 2.52 2.52 2.52]
 [2.62 2.62 2.62 2.62 2.62]
 [2.7  2.7  2.7  2.7  2.7 ]
 [2.74 2.74 2.74 2.74 2.74]
 [2.8  2.8  2.8  2.8  2.8 ]
 [2.8  2.8  2.8  2.8  2.8 ]
 [2.86 2.86 2.86 2.86 2.86]
 [2.9  2.9  2.9  2.9  2.9 ]
 [2.96 2.96 2.96 2.96 2.96]
 [3.   3.   3.   3.   3.  ]
 [3.   3.   3.   3.   3.  ]
 [3.   3.   3.   3.   3.  ]
 [3.   3.   3.   3.   3.  ]
 [3.04 3.04 3.04 3.04 3.04]
 [3.1  3.1  3.1  3.1  3.1 ]
 [3.12 3.12 3.12 3.12 3.12]
 [3.2  3.2  3.2  3.2  3.2 ]
 [3.2  3.2  3.2  3.2  3.2 ]
 [3.26 3.26 3.26 3.26 3.26]
 [3.34 3.34 3.34 3.34 3.34]
 [3.4  3.4  3.4  3.4  3.4 ]
 [3.4  3.4  3.4  3.4  3.4 ]
 [3.5  3.5  3.5  3.5  3.5 ]
 [3.58 3.58 3.58 3.58 3.58]
 [3.74 3.74 3.74 3.74 3.74]
 [3.82 3.82 3.82 3.82 3.82]
 [4.12 4.12 4.12 4.12 4.12]]
```

- **Bin Mean:** This is the label for the data being displayed.
- **[[]]**: This represents a two-dimensional NumPy array. Each inner square bracket [] represents a row in the array.

Each row in the array corresponds to a bin, and there are 5 values within each row. However, due to how the code calculates the means, all five values in a row have the same value, which is the average of the middle five elements from the original sorted data (b).

For example, the first row [2.18 2.18 2.18 2.18 2.18] shows the mean value for the first bin. This represents the average of the 2nd to 6th element (inclusive) after sorting the data in array b.

The output shows that as the rows progress (representing bins with higher indices), the mean values generally increase because they are calculated based on progressively larger values in the sorted data.

It's important to note that this code snippet only calculates the means for the first 30 bins (although the arrays were created to hold data for 30 bins). Additionally, the bin boundaries and medians haven't been calculated yet in the provided code.

```
Bin Boundaries:
 [[2.  2.3 2.3 2.3 2.3]
 [2.3 2.3 2.3 2.4 2.4]
 [2.4 2.5 2.5 2.5 2.5]
 [2.5 2.5 2.5 2.5 2.6]
 [2.6 2.6 2.6 2.6 2.7]
 [2.7 2.7 2.7 2.7 2.7]
 [2.7 2.7 2.7 2.8 2.8]
 [2.8 2.8 2.8 2.8 2.8]
 [2.8 2.8 2.8 2.8 2.8]
 [2.8 2.8 2.9 2.9 2.9]
 [2.9 2.9 2.9 2.9 2.9]
 [2.9 2.9 3.  3.  3. ]
 [3.  3.  3.  3.  3. ]
 [3.  3.  3.  3.  3. ]
 [3.  3.  3.  3.  3. ]
 [3.  3.  3.  3.  3. ]
 [3.  3.  3.  3.1 3.1]
 [3.1 3.1 3.1 3.1 3.1]
 [3.1 3.1 3.1 3.1 3.2]
 [3.2 3.2 3.2 3.2 3.2]
 [3.2 3.2 3.2 3.2 3.2]
 [3.2 3.2 3.3 3.3 3.3]
 [3.3 3.3 3.3 3.4 3.4]
 [3.4 3.4 3.4 3.4 3.4]
 [3.4 3.4 3.4 3.4 3.4]
 [3.5 3.5 3.5 3.5 3.5]
 [3.5 3.6 3.6 3.6 3.6]
 [3.7 3.7 3.7 3.8 3.8]
 [3.8 3.8 3.8 3.8 3.9]
 [3.9 3.9 3.9 4.4 4.4]]
```

- **Bin Boundaries:** This is the label for the data being displayed.
- **[[]]**: This represents a two-dimensional NumPy array. Each inner square bracket [] represents a row in the array.

Each row contains 5 values, which represent the boundaries for a specific bin. The logic behind these values is:

1. The first value ([i]) in a row represents the lower boundary of the bin.
2. The last value ([i+4]) in a row represents the upper boundary of the bin.
3. The middle three values ([i+1], [i+2], [i+3]) are (supposed to be) repeated values within the bin, but the provided code assigns the same value to all five positions in the row.

For example, the first row [2. 2.3 2.3 2.3 2.3] shows the boundaries for the first bin. Here:

- The lower boundary is 2.0.
- The upper boundary is 2.3.
- Ideally, the middle three values should represent values within the bin between 2.0 and 2.3, but due to the code's logic, they are all set to 2.3.

```
Bin Median:
 [[2.2 2.2 2.2 2.2 2.2]
 [2.3 2.3 2.3 2.3 2.3]
 [2.5 2.5 2.5 2.5 2.5]
 [2.5 2.5 2.5 2.5 2.5]
 [2.6 2.6 2.6 2.6 2.6]
 [2.7 2.7 2.7 2.7 2.7]
 [2.7 2.7 2.7 2.7 2.7]
 [2.8 2.8 2.8 2.8 2.8]
 [2.8 2.8 2.8 2.8 2.8]
 [2.9 2.9 2.9 2.9 2.9]
 [2.9 2.9 2.9 2.9 2.9]
 [3.  3.  3.  3.  3. ]
 [3.  3.  3.  3.  3. ]
 [3.  3.  3.  3.  3. ]
 [3.  3.  3.  3.  3. ]
 [3.  3.  3.  3.  3. ]
 [3.  3.  3.  3.  3. ]
 [3.1 3.1 3.1 3.1 3.1]
 [3.1 3.1 3.1 3.1 3.1]
 [3.2 3.2 3.2 3.2 3.2]
 [3.2 3.2 3.2 3.2 3.2]
 [3.3 3.3 3.3 3.3 3.3]
 [3.3 3.3 3.3 3.3 3.3]
 [3.4 3.4 3.4 3.4 3.4]
 [3.4 3.4 3.4 3.4 3.4]
 [3.5 3.5 3.5 3.5 3.5]
 [3.6 3.6 3.6 3.6 3.6]
 [3.7 3.7 3.7 3.7 3.7]
 [3.8 3.8 3.8 3.8 3.8]
 [4.1 4.1 4.1 4.1 4.1]]
```

- **Bin Median:** This is the label for the data being displayed.
- **[[]]**: This represents a two-dimensional NumPy array. Each inner square bracket [] represents a row in the array.

Similar to the bin boundaries output, each row contains 5 values, which are supposed to represent the medians for a specific bin. However, due to the way the code calculates them, all five values in a row have the same value, which is the the median of the middle three elements from the original sorted data (b) for that particular bin.

For example, the first row [2.2 2.2 2.2 2.2 2.2] shows the medians for the first bin. Here, ideally, all the values should represent the median of the data points between the lower and upper boundaries of this bin (which weren't calculated correctly in the provided code). However, the code assigns the same median value (calculated from the middle three elements) to all five positions in the row.

Conclusion:-

In this report, we explored two fundamental data pre-processing techniques: binning and one-hot encoding. Binning transforms continuous data into discrete intervals, simplifying complex datasets and potentially improving machine learning model performance by reducing the impact of noise or outliers. One-hot encoding tackles categorical data, converting it into a numerical format suitable for machine learning algorithms. This allows the algorithms to learn relationships between categories without assuming any inherent order.

The choice between these techniques depends on the data type (continuous vs. categorical) and specific characteristics like the number of categories or the data distribution. By understanding these techniques, we can effectively prepare data for machine learning tasks, potentially leading to more robust and accurate models.

However, it's important to consider the limitations of each technique. Binning can lead to information loss if we choose too few bins, and a high number of bins can increase computational cost. One-hot encoding can introduce the "curse of dimensionality" and create sparse data, which might impact certain algorithms.

Further exploration could involve investigating alternative binning strategies like cluster-based or entropy-based methods. Additionally, techniques like feature scaling can be used alongside binning or one-hot encoding to ensure all features contribute equally to the model. Finally, incorporating domain knowledge when choosing binning strategies or defining categories for one-hot encoding can lead to more meaningful data representations for specific problems.

Overall, binning and one-hot encoding are valuable tools for data pre-processing in machine learning. By understanding their applications and limitations, we can leverage these techniques to unlock the full potential of our data and create more successful machine learning models.