Assignment of Data Mining

[Scaling & Normalization, Encoding]

Submitted To-

Dr.Md.Manowarul Islam
Associate Professor

Submitted By-

Susmita Rani Saha (B180305047)
Tanvir Ahammed Hridoy (b180305020)

Department of computer Science and Engineering,

Jagannath university

Data Scaling and Normalization

Data scaling and normalization are common terms used for modifying the scale of features in a dataset.
The word “scaling” is a broader terms used for both upscaling and downscaling data, as well as for data normalization.
Data normalization, on the other hand, refers to scaling data values in such a way that the new values are within a specific range, typically -1 to 1, or 0 to 1.

➢ Why do we need data scaling?

A dataset in its raw form contains attributes of different types. A dataset in a format like this isn’t suitable for processing with many statistical algorithms. For example, the linear regression algorithm tends to assign larger weights to the features with larger values, which can affect the overall model performance. That’s why we need data scaling.

➢ Import/Create dataset

There are two techniques for dataset creation:

i. Create dataframe using panda,
ii. Import “tips” dataset from the Seaborn library

✓ Head() Method:

The head() method returns a specified number of rows, string from the top. The head() method returns the first 5 rows if a number is not specified. The column names will also be returned.

• Technique 1: Create dataframe using panda

For store the result data in dataframe we first create a dataframe from a dictionary of list using pandas and displays its header.

Example:

```
In [2]: import pandas as pd
result={'Name':['priya','riya','hridoy','riyon','tanvir','mistu','susmita','mita'],
        'id':[47,40,20,12,21,54,25,17],
        'gender':['female','female','male','male','male','female','female','female'],
        'height':[5.5,5.4,5.8,5.9,5.7,3.0,5.4,4.5],
        'weight':[45.00,44.10,76.00,50.60,70.20,23.00,46.02,50.00],
        'smoker':['no','no','no','no','yes','no','no','yes']}
df=pd.DataFrame(result)
df.head()
```

Out[2]:

	Name	id	gender	height	weight	smoker
0	priya	47	female	5.5	45.0	no
1	riya	40	female	5.4	44.1	no
2	hridoy	20	male	5.8	76.0	no
3	riyon	12	male	5.9	50.6	no
4	tanvir	21	male	5.7	70.2	yes

- ## Technique 2: Import “tips” dataset from the Seaborn library

We can also import “tips” dataset from the Seaborn library into a Pandas dataframe and displays its header.

Example:

```
In [27]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("darkgrid")

tips_ds  = sns.load_dataset('tips')
tips_ds.head()
```

Out[27]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

➢ data scaling techniques

There are some scaling techniques:

1. Standard Scaling
2. Min/Max Scaling
3. Mean Scaling
4. Maximum Absolute Scaling
5. Median and Quantile Scaling

✓ Filter() method:

Data scaling is applied to numeric columns. That's why we use filter() method for removing non-numeric columns.

Example:

```
In [3]: dff=df.filter(["id","height","weight"],axis=1)
        dff.head()
```

Out[3]:

	id	height	weight
0	47	5.5	45.0
1	40	5.4	44.1
2	20	5.8	76.0
3	12	5.9	50.6
4	21	5.7	70.2

✓ describe() method:

Now plot some statistical values for the columns in our dataset using the describe() method.

Example:

```
In [6]: dff.describe()
```

Out[6]:

	id	height	weight
count	8.000000	8.000000	8.000000
mean	29.500000	5.150000	50.615000
std	15.408717	0.969536	16.414387
min	12.000000	3.000000	23.000000
25%	19.250000	5.175000	44.775000
50%	23.000000	5.450000	48.010000
75%	41.750000	5.725000	55.500000
max	54.000000	5.900000	76.000000

The above output confirms our three columns are not scaled. The mean, minimum and maximum values, and even the standard deviation values for all three columns are very different.

This unscaled dataset is not suitable for processing by some statistical algorithms. We need to scale this data

❖ Standard Scaling

In standard scaling, a feature is scaled by subtracting the mean from all the data points and dividing the resultant values by the standard deviation of the data. Mathematically,

```
scaled = (x-u)/s
```

Here, u refers to the mean value and s corresponds to the standard deviation.

To apply standard scaling with Python, we can use the **StandardScaler** class from the **sklearn.preprocessing** module. We need to call the **fit_transform()** method from the **StandardScaler** class and pass Pandas Dataframe containing the features we want scaled.

Example:

```
In [10]: from sklearn.preprocessing import StandardScaler
ss=StandardScaler()
dfs=ss.fit_transform(dff)
print(dfs)
```

```
[[ 1.21413655e+00  3.85922492e-01 -3.65696730e-01]
 [ 7.28481929e-01  2.75658923e-01 -4.24312413e-01]
 [-6.59102697e-01  7.16713200e-01  1.65328789e+00]
 [-1.21413655e+00  8.26976770e-01 -9.76928042e-04]
 [-5.89723466e-01  6.06449631e-01  1.27554238e+00]
 [ 1.69979117e+00 -2.37066674e+00 -1.79852453e+00]
 [-3.12206541e-01  2.75658923e-01 -2.99265624e-01]
 [-8.67240391e-01 -7.16713200e-01 -4.00540497e-02]]
```

But the fit_transform() method returns a NumPy array which we have to convert to a Pandas Dataframe.

Example:

```
In [13]: from sklearn.preprocessing import StandardScaler
ss=StandardScaler()
dfs=ss.fit_transform(dff)
dfsf=pd.DataFrame(dfs,columns=dff.columns)
dfsf.head()
```

Out[13]:

	id	height	weight
0	1.214137	0.385922	-0.365697
1	0.728482	0.275659	-0.424312
2	-0.659103	0.716713	1.653288
3	-1.214137	0.826977	-0.000977
4	-0.589723	0.606450	1.275542

If we call the describe() function again, we'll see that your data columns are now uniformly scaled and will have a mean centered around 0.

Example:

```
In [14]: dfsf.describe()
```

Out[14]:

	id	height	weight
count	8.000000e+00	8.000000e+00	8.000000e+00
mean	-1.387779e-17	-2.498002e-16	-6.765422e-17
std	1.069045e+00	1.069045e+00	1.069045e+00
min	-1.214137e+00	-2.370667e+00	-1.798525e+00
25%	-7.111371e-01	2.756589e-02	-3.803507e-01
50%	-4.509650e-01	3.307907e-01	-1.696598e-01
75%	8.498956e-01	6.340155e-01	3.181529e-01
max	1.699791e+00	8.269768e-01	1.653288e+00

❖ Min/Max Scaling

Min/Max scaling normalizes the data between 0 and 1 by subtracting the overall minimum value from each data point and dividing the result by the difference between the minimum and maximum values.

The Min/Max scaler is commonly used for data scaling when the maximum and minimum values for data points are known.

We have to use the **MinMaxScaler** class from the **sklearn.preprocessing** module to perform min/max scaling. The **fit_transform()** method of the class performs the min/max scaling on the input Pandas Dataframe, Similarly, converts the NumPy array returned by the fit_transform() to a Pandas Dataframe which contains normalized values between 0 and 1.

Example:

In [15]:
```
from sklearn.preprocessing import MinMaxScaler
mms=MinMaxScaler()
dfs=mms.fit_transform(dff)
dfsf=pd.DataFrame(dfs,columns=dff.columns)
dfsf.head()
```

Out[15]:

	id	height	weight
0	0.833333	0.862069	0.415094
1	0.666667	0.827586	0.398113
2	0.190476	0.965517	1.000000
3	0.000000	1.000000	0.520755
4	0.214286	0.931034	0.890566

❖ Mean Scaling

Mean scaling is similar to min/max scaling, however in the case of mean scaling, the mean value, instead of the minimum value, is subtracted from all the data points. The result of the subtraction is divided by the range

Like Min/Max scaling, mean scaling is used for data scaling when the minimum and maximum values for the features to be scaled are known in advance.

The scikit-learn module by default doesn't contain a class for the mean scaling. However, it's very easy to implement this form of scaling in Python from scratch.

Steps:

I. Find mean value for each column
II. Find difference between max and min
III. Then apply formula: scaled = (data point – mean)/difference

Example:

```
In [23]: dfm=dff.mean()
         diff=dff.max()-dff.min()
         dfsm=(dff-dfm)/diff
         dfsmf=pd.DataFrame(dfsm)
         dfsmf.head()
```

Out[23]:

	id	height	weight
0	0.416667	0.120690	-0.105943
1	0.250000	0.086207	-0.122925
2	-0.226190	0.224138	0.478962
3	-0.416667	0.258621	-0.000283
4	-0.202381	0.189655	0.369528

❖ Maximum Absolute Scaling

Maximum absolute scaling is another commonly used data scaling technique where the difference between the data points and the minimum value is divided by the maximum value. The maximum absolute scaling technique also normalizes the data between 0 and 1.

Maximum absolute scaling doesn't shift or center the data so it's commonly used for scaling sparse datasets.

We have to use the **MaxAbsScaler** class from the **sklearn.preprocessing** module to perform min/max scaling. The **fit_transform()** method of the class performs the min/max scaling on the input Pandas Dataframe, Similarly, converts the NumPy array returned by the fit_transform() to a Pandas Dataframe which contains normalized values between 0 and 1.

Example:

In [24]:
```
from sklearn.preprocessing import MaxAbsScaler
mas=MaxAbsScaler()
dfs=mas.fit_transform(dff)
dfsf=pd.DataFrame(dfs,columns=dff.columns)
dfsf.head()
```

Out[24]:

	id	height	weight
0	0.870370	0.932203	0.592105
1	0.740741	0.915254	0.580263
2	0.370370	0.983051	1.000000
3	0.222222	1.000000	0.665789
4	0.388889	0.966102	0.923684

❖ Median and Quantile Scaling

In median and quantile scaling, also known as robust scaling, the first step is to subtract the median value from all the data points. In the next step, the resultant values are divided by the IQR (interquartile range). The IQR is calculated by subtracting the first quartile values in our dataset from the third quartile values.

We have to use the **RobustScaler** class from the **sklearn.preprocessing** module to perform min/max scaling. The **fit_transform()** method of the class performs the min/max scaling on the input Pandas Dataframe, Similarly, converts the NumPy array returned by the fit_transform() to a Pandas Dataframe.

Example:

In [25]:
```
from sklearn.preprocessing import RobustScaler
rs=RobustScaler()
dfs=rs.fit_transform(dff)
dfsf=pd.DataFrame(dfs,columns=dff.columns)
dfsf.head()
```

Out[25]:

	id	height	weight
0	1.066667	0.090909	-0.280653
1	0.755556	-0.090909	-0.364569
2	-0.133333	0.636364	2.609790
3	-0.488889	0.818182	0.241492
4	-0.088889	0.454545	2.068998

How to Deal with Categorical Data for Machine Learning?

- ✓ Categorical data is a type of data that represents characteristics or qualities rather than numerical values. Some examples of categorical data include gender, color, occupation, and nationality. Machine learning models typically require numerical inputs, so dealing with categorical data is an important preprocessing step in building machine learning models.
- ✓ There are several techniques for dealing with categorical data in machine learning. The choice of technique depends on the nature of the data and the requirements of the machine learning model.

✓ Why do we need encoding?

- Most machine learning algorithms cannot handle categorical variables unless we convert them to numerical values
- Many algorithm's performances even vary based upon how the categorical variables are encoded

✓ **Categorical variables can be divided into two categories:**

I. Nominal: no particular order
II. Ordinal: there is some order between values

Method 1: Using Python's Category Encoder Library

Python's Category Encoder library is a powerful tool for encoding categorical data in machine learning. The library provides various encoding techniques, including Ordinal Encoding, One-Hot Encoding, Binary Encoding, Target Encoding, CatBoost Encoding, and Hashing Encoding.

➢ Importing libraries & Creating a dataframe:

In [1]:
```
pip install category_encoders
```

```
Collecting category_encoders
  Downloading category_encoders-2.6.0-py2.py3-none-any.whl (81 kB)
     ---------------------------------------- 81.2/81.2 kB 92.7 kB/s eta 0:00:00
```

In [5]:
```
import pandas as pd
import sklearn
import category_encoders as ce
```

In [8]:
```
data = pd.DataFrame({ 'gender' : ['Male', 'Female', 'Male', 'Female', 'Female'],
                      'class' : ['A','B','C','D','A'],
                       'city' : ['Dhaka','Khulna','Dhaka','Dhaka','Khulna'] })
data.head()
```

Out[8]:

	gender	class	city
0	Male	A	Dhaka
1	Female	B	Khulna
2	Male	C	Dhaka
3	Female	D	Dhaka
4	Female	A	Khulna

➢ Implementing one-hot encoding through category_encoder:

✓ In this method, each category is mapped to a vector that contains 1 and 0 denoting the presence or absence of the feature. The number of vectors depends on the number of categories for features.

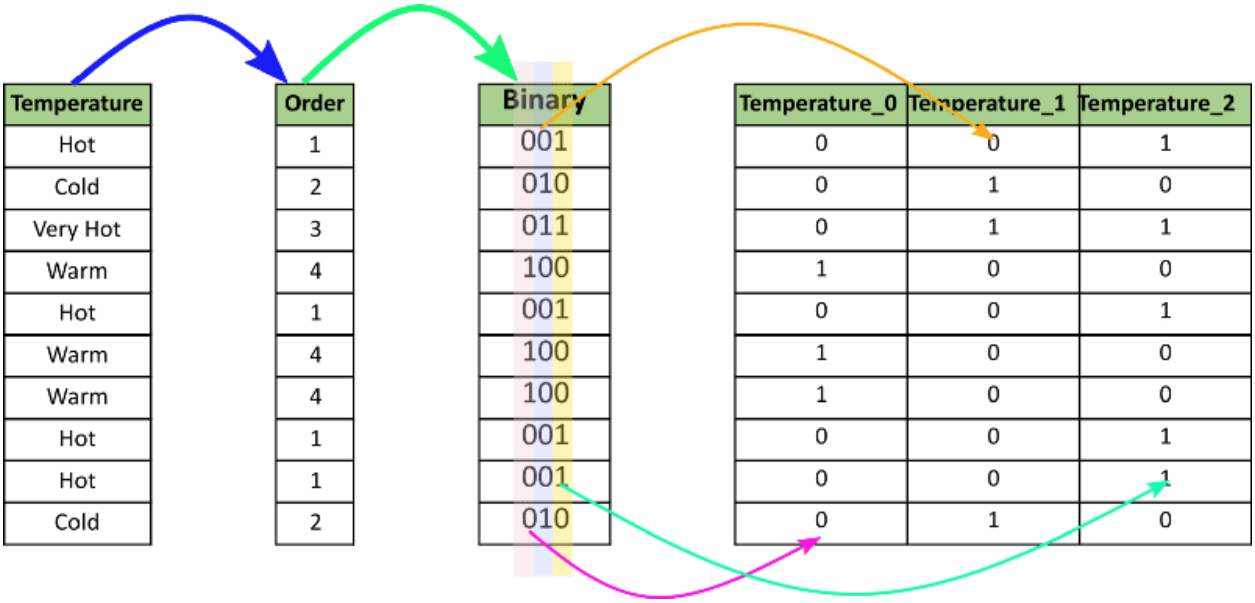
In [10]:
```
ce_OHE = ce.OneHotEncoder(cols=['gender','city','class'])
data1 = ce_OHE.fit_transform(data)
data1.head()
```

Out[10]:

	gender_1	gender_2	class_1	class_2	class_3	class_4	city_1	city_2
0	1	0	1	0	0	0	1	0
1	0	1	0	1	0	0	0	1
2	1	0	0	0	1	0	1	0
3	0	1	0	0	0	1	1	0
4	0	1	1	0	0	0	0	1

➢ Binary Encoding:

- ✓ Binary Encoding is another technique for encoding categorical data in machine learning. It is similar to One-Hot Encoding in that it creates a new binary variable for each category in the original categorical variable.



Temperature	Order	Binary
Hot	1	001
Cold	2	010
Very Hot	3	011
Warm	4	100
Hot	1	001
Warm	4	100
Warm	4	100
Hot	1	001
Hot	1	001
Cold	2	010

Temperature_0	Temperature_1	Temperature_2
0	0	1
0	1	0
0	1	1
1	0	0
0	0	1
1	0	0
1	0	0
0	0	1
0	0	1
0	1	0

- ✓ The idea behind Binary Encoding is to represent each category as a binary number, where each bit of the number represents the presence or absence of a particular attribute of the category. For example, if we have a categorical variable "Color" with categories "Red", "Green", and "Blue", we can represent each category as follows:
    - "Red": 001
    - "Green": 010
    - "Blue": 100

  We can then create three new binary variables "Color_1", "Color_2", and "Color_3", where each variable represents one bit of the binary number. If an observation belongs to the "Red" category, then the values of "Color_1", "Color_2", and "Color_3" will be 0, 0, and 1, respectively.

In [12]:
```
ce_be = ce.BinaryEncoder(cols=['class']);
data_binary = ce_be.fit_transform(data);
data_binary
```

Out[12]:

	gender	class_0	class_1	class_2	city
0	Male	0	0	1	Dhaka
1	Female	0	1	0	Khulna
2	Male	0	1	1	Dhaka
3	Female	1	0	0	Dhaka
4	Female	0	0	1	Khulna

In [13]:
```
ce_be = ce.BinaryEncoder(cols=['class']);
data_binary = ce_be.fit_transform(data["class"]);
data_binary
```

Out[13]:

	class_0	class_1	class_2
0	0	0	1
1	0	1	0
2	0	1	1
3	1	0	0
4	0	0	1

Method 2: Using Pandas' Get Dummies

- ✓ The get_dummies function in Pandas also supports many other parameters to control the encoding process, such as handling NaN values, dropping one of the encoded columns to avoid multicollinearity, and prefixing the encoded column names with a custom string. You can refer to the Pandas documentation for more information on these parameters.

- ✓ We can assign a prefix if we want to, if we do not want the encoding to use the default. In this example, we use gender as gen and city as it is.

```
In [14]: 1 pd.get_dummies(data,columns=["gender","city"])
```

Out[14]:

	class	gender_Female	gender_Male	city_Dhaka	city_Khulna
0	A	0	1	1	0
1	B	1	0	0	1
2	C	0	1	1	0
3	D	1	0	1	0
4	A	1	0	0	1

```
In [15]: 1 pd.get_dummies(data,prefix=["gen","city"],columns=["gender","city"])
```

Out[15]:

	class	gen_Female	gen_Male	city_Dhaka	city_Khulna
0	A	0	1	1	0
1	B	1	0	0	1
2	C	0	1	1	0
3	D	1	0	1	0
4	A	1	0	0	1

Method 3: Using Scikit-learn

✓ Scikit-learn is a popular machine learning library in Python that provides several methods for encoding categorical data. Scikit-learn also has 15 different types of built-in encoders, which can be accessed from sklearn.preprocessing.

➢ Scikit-learn One-hot Encoding

✓ Applying on the gender column it will show female as 0-number column and where there is female present it show 1 otherwise 0 and vice-versa.

In [16]:
```
s = (data.dtypes == 'object')
cols = list(s[s].index)
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder(handle_unknown='ignore',sparse=False)
```

In [17]:
```
data_gender = pd.DataFrame(ohe.fit_transform(data[["gender"]]))
data_gender
```

Out[17]:

	0	1
0	0.0	1.0
1	1.0	0.0
2	0.0	1.0
3	1.0	0.0
4	1.0	0.0

✓ Applying on the city column it will show Dhaka as 0-number column and where there is Dhaka present it show 1 otherwise 0 and vice-versa.

In [18]:
```
data_city = pd.DataFrame(ohe.fit_transform(data[["city"]]))
data_city
```

Out[18]:

	0	1
0	1.0	0.0
1	0.0	1.0
2	1.0	0.0
3	1.0	0.0
4	0.0	1.0

✓ Applying on the class column it will show A as 0-number column and where there is A present it show 1 otherwise 0 and same for B,C and D. This is because the class column has 4 unique values.

In [19]:
```
data_class = pd.DataFrame(ohe.fit_transform(data[["class"]]))

data_class
```

Out[19]:

	0	1	2	3
0	1.0	0.0	0.0	0.0
1	0.0	1.0	0.0	0.0
2	0.0	0.0	1.0	0.0
3	0.0	0.0	0.0	1.0
4	1.0	0.0	0.0	0.0

✓ Applying to the list of categorical variables.Here (0-1) number column use to represent Gender and (2-5) number of column use to represent class finally (6-7) column use to represent city.

In [20]:
```
data_cols = pd.DataFrame(ohe.fit_transform(data[cols]))
data_cols
```

Out[20]:

	0	1	2	3	4	5	6	7
0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0
1	1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0
2	0.0	1.0	0.0	0.0	1.0	0.0	1.0	0.0
3	1.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0
4	1.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0

➢ Scikit-learn Label Encoding:

In label encoding, each category is assigned a value from 1 through N where N is the number of categories for the feature. There is no relation or order between these assignments.

In [24]:
```
1 from sklearn.preprocessing import LabelEncoder
2
3 le = LabelEncoder()
4 Label encoder takes no arguments
5 le_class = le.fit_transform(data[["class"]])
```

```
  File "C:\Users\My AsUs\AppData\Local\Temp\ipykernel_16832\8
21644767.py", line 4
    Label encoder takes no arguments
          ^
SyntaxError: invalid syntax
```

❖ Comparing with one-hot encoding:

In [25]:
```
1 data_class
```

Out[25]:

	0	1	2	3
0	1.0	0.0	0.0	0.0
1	0.0	1.0	0.0	0.0
2	0.0	0.0	1.0	0.0
3	0.0	0.0	0.0	1.0
4	1.0	0.0	0.0	0.0

Method 3: Ordinal Encoding

- ✓ Ordinal encoding's encoded variables retain the ordinal (ordered) nature of the variable. It looks similar to label encoding, the only difference being that label coding doesn't consider whether a variable is ordinal or not; it will then assign a sequence of integers.
- ✓ Example: Ordinal encoding will assign values as Very Good(1) < Good(2) < Bad(3) < Worse(4)
- ✓ First, we need to assign the original order of the variable through a dictionary.

In [26]:
```
temp = {'temperature' :['very cold', 'cold', 'warm', 'hot', 'very hot']}
df=pd.DataFrame(temp,columns=["temperature"])
temp_dict = {'very cold': 1,'cold': 2,'warm': 3,'hot': 4,"very hot":5}
df
```

Out[26]:

	temperature
0	very cold
1	cold
2	warm
3	hot
4	very hot

✓ Then we can map each row for the variable as per the dictionary.

In [27]:
```
df["temp_ordinal"] = df.temperature.map(temp_dict)
df
```

Out[27]:

	temperature	temp_ordinal
0	very cold	1
1	cold	2
2	warm	3
3	hot	4
4	very hot	5

Method 4: Frequency Encoding

✓ The category is assigned as per the frequency of values in its total lot.

In [28]:
```
data_freq = pd.DataFrame({'class' : ['A','B','C','D','A',"B","E","E","D","C","C","C","E","A","A"]})
```

✓ **Grouping by class column:**

```
In [29]:  1  fe = data_freq.groupby("class").size()
```

✓ **Dividing by length:**

```
In [30]:  1  fe_ = fe/len(data_freq)
```

✓ **Mapping and rounding off:**

```
In [31]:  1  data_freq["data_fe"] = data_freq["class"].map(fe_).round(2)
          2  data_freq
```

Out[31]:

	class	data_fe
0	A	0.27
1	B	0.13
2	C	0.27
3	D	0.13
4	A	0.27
5	B	0.13
6	E	0.20
7	E	0.20
8	D	0.13
9	C	0.27
10	C	0.27
11	C	0.27
12	E	0.20
13	A	0.27
14	A	0.27

✓ **There is Only 5 types of encoding Schemes but there are also 10 types of encoding techniques.**

Which Encoding Method is Best?

- ✓ There is no single method that works best for every problem or dataset. I personally think that the get_dummies method has an advantage in its ability to be implemented very easily.

- ✓ Here is a cheat sheet on when to use what type of encoding: