Lab Report
On

“Data Scaling, Normalization and Encoding of Categorical Data”

Course Title: Data Mining and Data Warehousing Lab
Course Code: CSEL-4108

Submitted To,

Dr. Md. Manowarul Islam
Associate Professor
Department of CSE
Jagannath University, Dhaka

Submitted By,

Momtaj Hossain Mow
(B180305025)
&
Md. Abu Yousuf Siam
(B180305012)

Department of Computer Science & Engineering
Jagannath University, Dhaka
Date of Submission: 23/03/2023

Data Scaling and Normalization

Scaling is part of data pre-processing as this technique brings data points that are far from each other closer in order to increase the algorithm effectiveness. Scaling of the data makes it easy for a model to learn and understand the problem. Because in machine learning algorithms if the values of the features are closer to each other there are chances for the algorithm to get trained well and faster instead of the data set where the data points or features values have high differences with each other will take more time to understand the data and the accuracy will be lower.

So, scaling is used for making data points generalized so that the distance between them will be lower.

Some data scaling techniques are:

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

On the other hand, data normalization 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. This can be useful in algorithms that do not assume any distribution of the data, like K-Nearest Neighbors and Neural Networks.

Normalization helps in scaling the input features to a fixed range, typically [0, 1], to ensure that no single feature disproportionately impacts the results. It preserves the relationship between the minimum and maximum values of each feature, which can be important for some algorithms. It also improves the convergence and stability of some machine learning algorithms, particularly those that use gradient-based optimization.

Importing Dataset:

To learn these scaling techniques, we are going to use the ‘tips’ dataset from the “Seaborn” library. After importing the dataset into a Pandas dataframe we display its header.

```
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()
```

Output:

	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 is applied to numeric columns. The script below filters numeric columns from the dataset.

```
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_numeric = tips_ds.filter(["total_bill", "tip", "size"], axis=1)
tips_ds_numeric.head()
```

Output:

	total_bill	tip	size
0	16.99	1.01	2
1	10.34	1.66	3
2	21.01	3.50	3
3	23.68	3.31	2
4	24.59	3.61	4

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

```
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_numeric = tips_ds.filter(["total_bill", "tip", "size"], axis=1)
tips_ds_numeric.describe()
```

Output:

	total_bill	tip	size
count	244.000000	244.000000	244.000000
mean	19.785943	2.998279	2.569672
std	8.902412	1.383638	0.951100
min	3.070000	1.000000	1.000000
25%	13.347500	2.000000	2.000000
50%	17.795000	2.900000	2.000000
75%	24.127500	3.562500	3.000000
max	50.810000	10.000000	6.000000

So we can see that the output table confirms that our table is not scaled,the value of minimum , maximum and standard deviation value are different for three columns.

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 this is written as:

scaled = (x-u) / s

where,
s = standard deviation
u = mean value

StandardScaler class from the sklearn.preprocessing package must be used for applying the standard deviation in python. The fit_transform() method from the StandardScaler class must be called, and a dataframe containing the feature we want to scale must be passed to Pandas.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler

sns.set_style("darkgrid")
tips_ds = sns.load_dataset('tips')
tips_ds_numeric = tips_ds.filter(["total_bill", "tip", "size"], axis=1)

ss = StandardScaler()
tips_ds_scaled = ss.fit_transform(tips_ds_numeric)
tips_ds_scaled_df = pd.DataFrame(tips_ds_scaled, columns = tips_ds_numeric.columns)
tips_ds_scaled_df.head()
```

Output:

	total_bill	tip	size
0	-0.314711	-1.439947	-0.600193
1	-1.063235	-0.969205	0.453383
2	0.137780	0.363356	0.453383
3	0.438315	0.225754	-0.600193
4	0.540745	0.443020	1.506958

Now if we use describe() function we can see all the columns are uniformly scaled.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler

sns.set_style("darkgrid")
tips_ds = sns.load_dataset('tips')
tips_ds_numeric = tips_ds.filter(["total_bill", "tip", "size"], axis=1)

ss = StandardScaler()
tips_ds_scaled = ss.fit_transform(tips_ds_numeric)
tips_ds_scaled_df = pd.DataFrame(tips_ds_scaled, columns = tips_ds_numeric.columns)
tips_ds_scaled_df.describe()
```

Output:

	total_bill	tip	size
count	2.440000e+02	2.440000e+02	2.440000e+02
mean	-7.871663e-17	2.839259e-16	-5.824121e-17
std	1.002056e+00	1.002056e+00	1.002056e+00
min	-1.881547e+00	-1.447189e+00	-1.653768e+00
25%	-7.247111e-01	-7.229713e-01	-6.001926e-01
50%	-2.241005e-01	-7.117518e-02	-6.001926e-01
75%	4.886857e-01	4.086192e-01	4.533829e-01
max	3.492068e+00	5.070772e+00	3.614110e+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. For instance, we can use the min/max scaler to normalize image pixels having values between 0 to 255.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler

sns.set_style("darkgrid")
tips_ds = sns.load_dataset('tips')
tips_ds_numeric = tips_ds.filter(["total_bill", "tip", "size"], axis=1)

mms = MinMaxScaler()
tips_ds_mms = mms.fit_transform(tips_ds_numeric)
tips_ds_mms_df = pd.DataFrame(tips_ds_mms, columns = tips_ds_numeric.columns)
tips_ds_mms_df.head()
```

Output:

	total_bill	tip	size
0	0.291579	0.001111	0.2
1	0.152283	0.073333	0.4
2	0.375786	0.277778	0.4
3	0.431713	0.256667	0.2
4	0.450775	0.290000	0.6

Mean Scaling:

Mean scaling is similar to min/max scaling, however in the case of mean scaling, the mean value is subtracted from all the data points. The result of the subtraction is divided by the range (difference between the minimum and maximum values).

Find mean value of numeric column,

```
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_numeric = tips_ds.filter(["total_bill", "tip", "size"], axis=1)

tips_means = tips_ds_numeric.mean(axis=0)
print(tips_means)
```

Output:

```
total_bill    19.785943
tip            2.998279
size           2.569672
dtype: float64
```

Find the range for all the features by subtracting min from the max value,

```
#mean scaling
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_numeric = tips_ds.filter(["total_bill", "tip", "size"], axis=1)
#finding the mean
tips_means = tips_ds_numeric.mean(axis=0)
#finding the range
min_max_range = tips_ds_numeric.max(axis=0) - tips_ds_numeric.min(axis=0)
print(min_max_range)
```

Output:

```
total_bill    47.74
tip            9.00
size           5.00
dtype: float64
```

The mean scaling can be performed by subtracting the data points by the mean values and dividing the result by the range, as shown below:

```
#mean scaling
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_numeric = tips_ds.filter(["total_bill", "tip", "size"], axis=1)
#finding the mean
tips_means = tips_ds_numeric.mean(axis=0)
#finding the range
min_max_range = tips_ds_numeric.max(axis=0) - tips_ds_numeric.min(axis=0)
#mean scaling
tips_scaled_mean = (tips_ds_numeric - tips_means) / min_max_range
tips_scaled_mean_df = pd.DataFrame(tips_scaled_mean, columns = tips_ds_numeric.columns)
tips_scaled_mean_df.head()
```

Output:

	total_bill	tip	size
0	-0.058566	-0.220920	-0.113934
1	-0.197862	-0.148698	0.086066
2	0.025640	0.055747	0.086066
3	0.081568	0.034636	-0.113934
4	0.100630	0.067969	0.286066

Maximum Absolute Scaling:

Maximum absolute scaling is a data scaling technique where the difference between the data point 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.

```
#maximum absolute scaling -- 1
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import MaxAbsScaler

sns.set_style("darkgrid")
tips_ds = sns.load_dataset('tips')
tips_ds_numeric = tips_ds.filter(["total_bill", "tip", "size"], axis=1)

mas = MaxAbsScaler()
tips_ds_mas = mas.fit_transform(tips_ds_numeric)
tips_ds_mas_df = pd.DataFrame(tips_ds_mas, columns = tips_ds_numeric.columns)
tips_ds_mas_df.head()
```

Output:

	total_bill	tip	size
0	0.334383	0.101	0.333333
1	0.203503	0.166	0.500000
2	0.413501	0.350	0.500000
3	0.466050	0.331	0.333333
4	0.483960	0.361	0.666667

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.

It is the most commonly used scaling technique for dataset with a large number of outliers.

Median and quantile scaling can be implemented via the RobustScaler class from the **sklearn.preprocessing** module.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import RobustScaler

sns.set_style("darkgrid")
tips_ds = sns.load_dataset('tips')
tips_ds_numeric = tips_ds.filter(["total_bill", "tip", "size"], axis=1)

rs = RobustScaler()
tips_ds_rs = rs.fit_transform(tips_ds_numeric)
tips_ds_rs_df = pd.DataFrame(tips_ds_rs, columns = tips_ds_numeric.columns)
tips_ds_rs_df.head()
```

Output:

	total_bill	tip	size
0	-0.074675	-1.2096	0.0
1	-0.691558	-0.7936	1.0
2	0.298237	0.3840	1.0
3	0.545918	0.2624	0.0
4	0.630334	0.4544	2.0

Categorical Data

Categorical data is a type of data that is used to group information with similar characteristics, while numerical data is a type of data that expresses information in the form of numbers.
Some examples of categorical data are gender, city, day etc.
Categorical variables can be divided into two categories:

- **Nominal:** It has no particular order
- **Ordinal:** There is some order between values

Encoding:

Encoding is a technique of converting categorical variables into numerical values so that it could be easily fitted to a machine learning model.Since most machine learning models only accept numeric variables, preprocessing the categorical variables becomes a necessary step.

Importance of 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.

Various types of encoding techniques are available. We will discuss some of them here.

Method 1: Using Python's Category Encoder Library

For encoding categorical data, we have a python package category_encoders. The following code helps to install it easily:

```
pip install category_encoders
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: category_encoders in /usr/local/lib/python3.9/dist-packages (2.6.0)
Requirement already satisfied: statsmodels>=0.9.0 in /usr/local/lib/python3.9/dist-packages (from category_encoders) (0.13.5)
Requirement already satisfied: patsy>=0.5.1 in /usr/local/lib/python3.9/dist-packages (from category_encoders) (0.5.3)
Requirement already satisfied: scikit-learn>=0.20.0 in /usr/local/lib/python3.9/dist-packages (from category_encoders) (1.2.2)
Requirement already satisfied: numpy>=1.14.0 in /usr/local/lib/python3.9/dist-packages (from category_encoders) (1.22.4)
Requirement already satisfied: scipy>=1.0.0 in /usr/local/lib/python3.9/dist-packages (from category_encoders) (1.10.1)
Requirement already satisfied: pandas>=1.0.5 in /usr/local/lib/python3.9/dist-packages (from category_encoders) (1.4.4)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.9/dist-packages (from pandas>=1.0.5->category_encoders) (2022.7.1)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.9/dist-packages (from pandas>=1.0.5->category_encoders) (2.8.2)
Requirement already satisfied: six in /usr/local/lib/python3.9/dist-packages (from patsy>=0.5.1->category_encoders) (1.16.0)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.9/dist-packages (from scikit-learn>=0.20.0->category_encoders) (1.1.1)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.9/dist-packages (from scikit-learn>=0.20.0->category_encoders) (3.1.0)
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.9/dist-packages (from statsmodels>=0.9.0->category_encoders) (23.0)
```

Category_encoders is an amazing Python library that provides 15 different encoding schemes.
Here is the list of the 15 types of encoding the library supports:

1. One-hot Encoding
2. Label Encoding
3. Ordinal Encoding
4. Helmert Encoding
5. Binary Encoding
6. Frequency Encoding
7. Mean Encoding
8. Weight of Evidence Encoding
9. Probability Ratio Encoding
10. Hashing Encoding
11. Backward Difference Encoding
12. Leave One Out Encoding
13. James-Stein Encoding
14. M-estimator Encoding
15. Thermometer Encoder

Importing Libraries:

```
import pandas as pd
import sklearn
import category_encoders as ce
```

Creating DataFrame:

```
import pandas as pd
import sklearn
import category_encoders as ce

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

Output:

	gender	class	city
0	Male	A	Delhi
1	Female	B	Gurugram
2	Male	C	Delhi
3	Female	D	Delhi
4	Female	A	Gurugram

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 the feature

Create an object of the **one-hot** encoder:

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

Output:

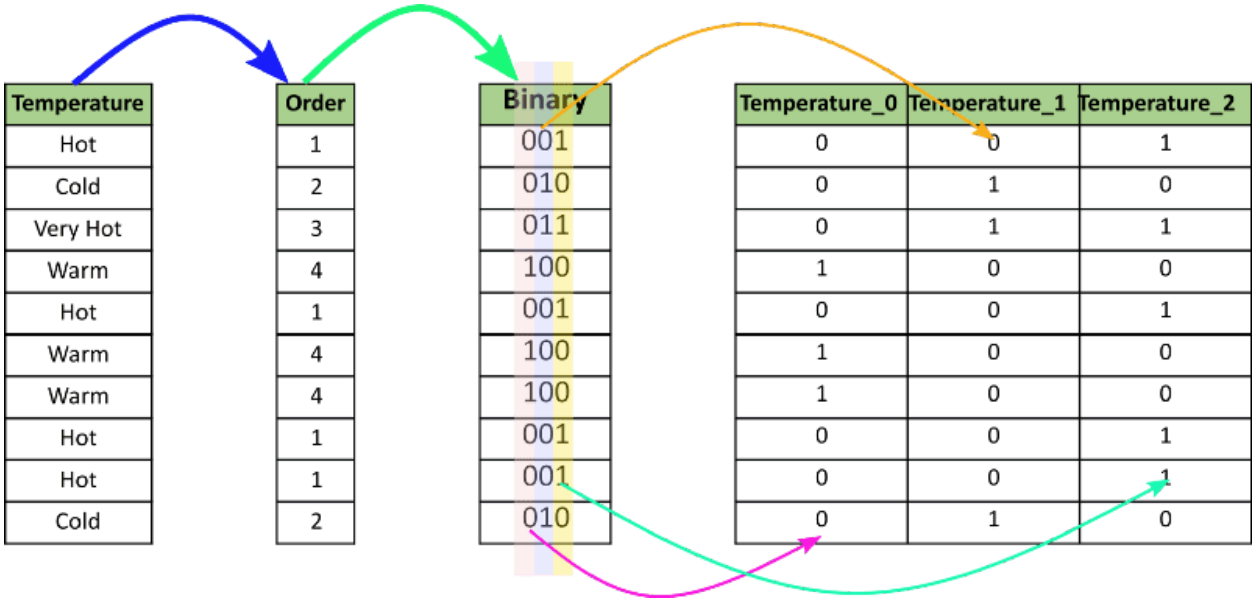
	gender_1	gender_2	class	city_1	city_2
0	1	0	A	1	0
1	0	1	B	0	1
2	1	0	C	1	0
3	0	1	D	1	0
4	0	1	A	0	1

Binary Encoding:

Binary encoding converts a category into binary digits. Each digit creates one feature column.

Binary encoding is a combination of Hash encoding and one-hot encoding. In this encoding scheme, the categorical feature is first converted into numerical using an ordinal encoder. Then the numbers are transformed into binary numbers. After that, the binary value is split into different columns. Binary encoding works really well when there are a high number of categories.

For example the types of temperature felt in a day.



Temperature
Hot
Cold
Very Hot
Warm
Hot
Warm
Warm
Hot
Hot
Cold

Order
1
2
3
4
1
4
4
1
1
2

Binary
001
010
011
100
001
100
100
001
001
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 following code binary encodes the data,

```
ce_be = ce.BinaryEncoder(cols=['class']);
#transform the data
data_binary = ce_be.fit_transform(data["class"]);
data_binary
```

Output:

	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

Similarly, there are another 14 types of encoding provided by this library.

Method 2: Using Pandas' Get Dummies

Categorical variables may be transformed into dummy or indicator variables with the help of the **get_dummies()** function in Pandas. It produces a new set of columns with binary values indicating the presence or absence of a certain category from a categorical column or collection of columns.

```
pd.get_dummies(data, columns=["gender","city"])
```

	class	gender_Female	gender_Male	city_Delhi	city_Gurugram
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

We can assign a prefix if we want to, if we do not want the encoding to use the default.

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

	class	gen_Female	gen_Male	city_Delhi	city_Gurugram
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 also has 15 different types of built-in encoders, which can be accessed from sklearn.preprocessing.

Scikit-learn One-hot Encoding

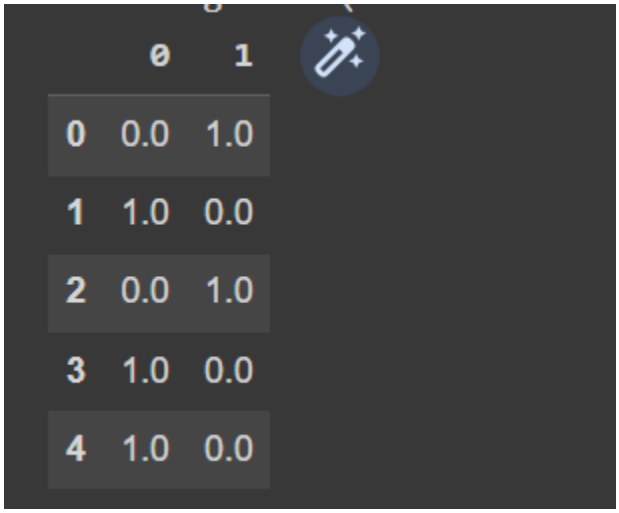
Let's first get the list of categorical variables from our data:

```
s = (data.dtypes == 'object')
cols = list(s[s].index)

from sklearn.preprocessing import OneHotEncoder

ohe = OneHotEncoder(handle_unknown='ignore', sparse=False)
```

Applying on the gender column:

```
data_gender = pd.DataFrame(ohe.fit_transform(data[['gender']]))
data_gender
```

Output:



	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:

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

Output:

	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:

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

data_class
```

Output:

	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

This is because the class column has 4 unique values.

Applying to the list of categorical variables:

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

data_cols
```

Output:

	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

Here, the first 2 columns represent gender, the next 4 columns represent class, and the remaining 2 represent city.

Scikit-learn Label Encoding:

In level 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.

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
le_class = le.fit_transform(data[["class"]])
```

Comparing with one-hot encoding:

```
    data_class
```

Output:

	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

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:

```
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
```

Output:

	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:

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

Output:

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

Frequency Encoding:

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

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

Grouping by class column:

```
fe = data_freq.groupby("class").size()
```

Dividing by length:

```
fe_ = fe/len(data_freq)
```

Mapping and rounding off:

```
data_freq["data_fe"] = data_freq["class"].map(fe_).round(2)
data_freq
```

Output:

	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

Which Encoding Method is the Best?

For every issue or dataset, there isn't a single approach that performs the best. Personally, I believe that the **get_dummies** technique has an advantage due to how simple it is to develop.