EDA : Using Feature Scaling . When and How?

**Overview:**

Understand the requirement of feature transformation and scaling techniques

Get to know different feature transformation and scaling techniques including-

MinMax Scaler

Standard Scaler

Power Transformer Scaler

Unit Vector Scaler/Normalizer

1. **Why do we need Feature Transformation and Scaling?**

Oftentimes, we have datasets in which different columns have different units – like one column can be in kilograms, while another column can be in centimeters. Furthermore, we can have columns like income which can range from 20,000 to 100,000, and even more; while an age column which can range from 0 to 100(at the most). Thus, Income is about 1,000 times larger than age.

But how can we be sure that the model treats both these variables equally? When we feed these features to the model as is, **there is every chance that the income will influence the result more due to its larger value.** But this doesn’t necessarily mean it is more important as a predictor. So, **to give importance to both Age, and Income, we need feature scaling.**

In most examples of machine learning models, you would have observed either the Standard Scaler or MinMax Scaler. However, the powerful sklearn library offers many other feature transformations scaling techniques as well, which we can leverage depending on the data we are dealing with.

1. **MinMax Scaler**

The MinMax scaler is one of the simplest scalers to understand. It just scales all the data between 0 and 1.

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

Apply it on only the values of the features

Df\_scaled[col\_names] = scaler.fit\_transform(features.values)

1. **Standard Scaler**

Just like the MinMax Scaler, the Standard Scaler is another popular scaler that is very easy to understand and implement. For each feature, the Standard Scaler scales the values such that the mean is 0 and the standard deviation is 1(or the variance).

x\_scaled = x – mean/std\_dev

However, Standard Scaler assumes that the distribution of the variable is normal. Thus, **in case, the variables are not normally distributed, we**

1. either choose a different scaler
2. or first, convert the variables to a normal distribution and then apply this scaler

Implementing the standard scaler is much similar to implementing a min-max scaler. Just like before, we will first import StandardScaler and then use it to transform our variable.

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

df\_scaled[col\_names] = scaler.fit\_transform(features.values)

df\_scaled

**4. Robust Scaler**

If you have noticed in the scalers we used so far, each of them was using values like the mean, maximum and minimum values of the columns. **All these values are sensitive to outliers. If there are too many outliers in the data, they will influence the mean and the max value or the min value.** Thus, even if we scale this data using the above methods, we cannot guarantee a balanced data with a normal distribution.

The Robust Scaler, as the name suggests is not sensitive to outliers. This scaler-

1. removes the median and mean from the calculations/formula, and,
2. **scales the data by the InterQuartile Range(IQR), thus, should be used where we have Outliers.**

x\_scaled = (x – Q1)/(Q3 – Q1)

and wkt, IQR = Q3 – Q1

from sklearn.preprocessing import RobustScaler

scaler = RobustScaler()

df\_scaled[col\_names] = scaler.fit\_transform(features.values)

df\_scaled

**5. Log Transform**

The Log Transform is one of the most popular Transformation techniques out there. **It is primarily used to convert a**[**skewed distribution**](https://www.analyticsvidhya.com/blog/2020/07/what-is-skewness-statistics/?utm_source=blog&utm_medium=Feature_Transformation_and_Scaling_Techniques)**to a normal distribution/less-skewed distribution**. In this transform, we take the log of the values in a column and use these values as the column instead.

Why does it work? It is because the log function is equipped to deal with large numbers. Here is an example-

log(10) = 1, log(100) = 2, and log(10000) = 4.

Example:

Raw Data ------

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

%matplotlib inline

df = pd.DataFrame({

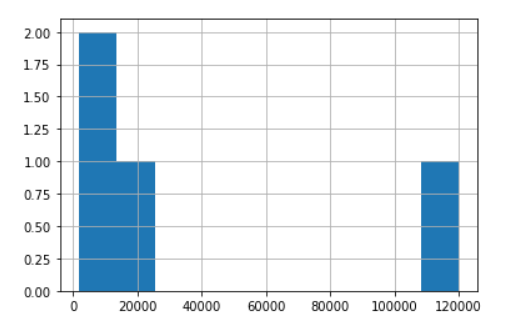
   'Income': [15000, 1800, 120000, 10000],

   'Age': [25, 18, 42, 51],

   'Department': ['HR','Legal','Marketing','Management']

})

Thus, in our example, while plotting the histogram of Income, it ranges from 0 to 1,20,000:

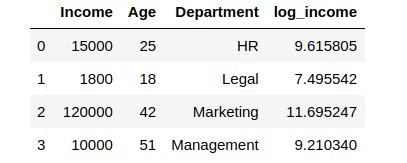


Let us see what happens when we apply log on this column:

df['log\_income'] = np.log(df['Income'])

# We created a new column to store the log values

This is how the dataframe looks like:



Wow! While our Income column had extreme values ranging from 1800 to 1,20,000 – the log values are now ranging from approximately 7.5 to 11.7! Thus, the log operation had a dual role:

* Reducing the impact of too-low values
* Reducing the impact of too-high values.

**A small caveat though** – if our data has negative values or values ranging from 0 to 1, we cannot apply log transform directly – since the log of negative numbers and numbers between 0 and 1 is undefined, we would get error or NaN values in our data. In such cases, *we can add a number to all these values to make them all greater than 1. Then, we can apply the log transform.*

**6. Power Transformer Scaler**

Like some other scalers we studied above, the Power Transformer also changes the distribution of the variable, as in, it makes it more Gaussian(normal). We are familiar with similar power transforms such as log transforms.

**However, to use them, we need to first study the original distribution, and then make a choice.** **The Power Transformer actually automates this decision making by introducing a parameter called *lambda***. It decides on a generalized power transform by finding the best value of lambda using either the:

1. [Box-Cox transform](https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwiV8dKcx9TqAhUb7XMBHRCEAoAQFjAHegQIERAG&url=https%3A%2F%2Fwww.statisticshowto.com%2Fbox-cox-transformation%2F&usg=AOvVaw0iFLbXcrz55ImaZx7cNA-2)

2. [The Yeo-Johnson transform](https://www.stat.umn.edu/arc/yjpower.pdf)

While I will not get into too much detail of how each of the above transforms works, it is helpful to know that Box-**Cox works with only positive values, while Yeo-Johnson works with both positive and negative values.**

In our case, we will use the Box-Cox transform since all our values are positive.

from sklearn.preprocessing import PowerTransformer

scaler = PowerTransformer(method = 'box-cox')

'''

parameters:

method = 'box-cox' or 'yeo-johnson'

'''

df\_scaled[col\_names] = scaler.fit\_transform(features.values)

df\_scaled

This is how the Power Transformer scales the data:

