Topics to be covered

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

Feature Scaling

1. Binarize
2. Min Max Scalar
3. Standard Scalar
4. Normalize

Binarize

```
Input_data = np.array([[1,2,3],[2,3,4],[1,2,2],[2,2,1]])

print("Input Data\n", Input_data)
```

Input Data
[[1 2 3]
[2 3 4]
[1 2 2]
[2 2 1]]

```
data_binarized = preprocessing.Binarizer(threshold=1).transform(Input_data)

print("\nBinarized data:\n", data_binarized)
```

Binarized data:
[[0 1 1]
[1 1 1]
[0 1 1]
[1 1 0]]

Mean Removal

Standardization or mean removal is a technique that simply centers data by removing the average value of each characteristic, and then scales it by dividing non-constant characteristics by their standard deviation. It's usually beneficial to remove the mean from each feature so that it's centered on zero. This helps us remove bias from features. The formula used to achieve this is the following:

$$x_{scaled} = \frac{x - mean}{sd}$$

Standardization results in the rescaling of features, which in turn represents the properties of a standard normal distribution:

- mean = 0
- sd = 1

Standardization is particularly useful when we do not know the minimum and maximum for data distribution. In this case, it is not possible to use other forms of data transformation. As a result of the transformation, the normalized values do not have a minimum and a fixed maximum. Moreover, this technique is not influenced by the presence of outliers, or at least not the same as other methods.

```
X = np.array([[1],[2],[3],[4],[5]])

print(X)

print("Mean =", X.mean())

print("Std deviation = ", X.std())
```

```
[[1]
 [2]
 [3]
 [4]
 [5]]
Mean = 3.0
Std deviation =  1.4142135623730951
```

```
data_scaled = preprocessing.scale(X)
print(data_scaled)
print("Mean =", data_scaled.mean())

print("Std deviation =", data_scaled.std())
```

```
[[-1.41421356]
 [-0.70710678]
 [ 0.        ]
 [ 0.70710678]
 [ 1.41421356]]
Mean = 0.0
Std deviation = 0.9999999999999999
```

```
Input_data = np.array([[1,2,3],[2,3,4],[1,2,2],[2,2,1]])

print("Input Data\n", Input_data)
```

```
Input Data
 [[1 2 3]
 [2 3 4]
 [1 2 2]
 [2 2 1]]
```

```
print("Mean =", Input_data.mean())

print("Std deviation = ", Input_data.std())
```

```
Mean = 2.0833333333333335
Std deviation =  0.8620067027323834
```

```
data_scaled = preprocessing.scale(Input_data)

print("Mean =", data_scaled.mean())

print("Std deviation =", data_scaled.std())
```

```
[[-1.        -0.57735027  0.4472136 ]
 [ 1.         1.73205081  1.34164079]
 [-1.        -0.57735027 -0.4472136 ]
 [ 1.        -0.57735027 -1.34164079]]
Mean = 1.850371707708594e-17
Std deviation = 1.0
```

```
print("Mean =", Input_data.mean(axis=0))

print("Std deviation = ", Input_data.std(axis=0))
```

```
Mean = [1.5  2.25 2.5 ]
Std deviation =  [0.5       0.4330127  1.11803399]
```

```
data_scaled = preprocessing.scale(Input_data)

#print(data_scaled)

print("Mean =", data_scaled.mean(axis=0))

print("Std deviation =", data_scaled.std(axis=0))
```

```
[[-1.        -0.57735027  0.4472136 ]
 [ 1.         1.73205081  1.34164079]
 [-1.        -0.57735027 -0.4472136 ]
 [ 1.        -0.57735027 -1.34164079]]
Mean = [ 0.00000000e+00 -5.55111512e-17  0.00000000e+00]
Std deviation = [1. 1. 1.]
```

```
print("Mean =", Input_data.mean(axis=1))

print("Std deviation = ", Input_data.std(axis=1))
```

```
Mean = [2.         3.         1.66666667 1.66666667]
Std deviation =  [0.81649658 0.81649658 0.47140452 0.47140452]
```

```
data_scaled = preprocessing.scale(Input_data)

print(data_scaled)

print("Mean =", data_scaled.mean(axis=1))

print("Std deviation =", data_scaled.std(axis=1))
```

```
[[-1.        -0.57735027  0.4472136 ]
 [ 1.         1.73205081  1.34164079]
 [-1.        -0.57735027 -0.4472136 ]
 [ 1.        -0.57735027 -1.34164079]]
Mean = [ 0.00000000e+00 -5.55111512e-17  0.00000000e+00]
Std deviation = [1. 1. 1.]
```

Min Max Scalar

```
import numpy as np

from sklearn import preprocessing
```

```
X = np.array([[1],[2],[3],[4],[5]])

X
```

```
array([[1],
       [2],
       [3],
       [4],
       [5]])
```

```
data_scaler_minmax = preprocessing.MinMaxScaler(feature_range=(0,1))
```

```
data_scaled_minmax = data_scaler_minmax.fit_transform(X)
print(data_scaled_minmax)
```

```
[[0.  ]
 [0.25]
 [0.5 ]
 [0.75]
 [1.  ]]
```

Working explained

Formula used is Y = (xi – xmin)/(xmax – xmin)

Here xmin is 1 and xmax is 5

So, answer value is (1 -1)/(5-1) =0

(2-1)/(5-1) = 0.25

(3-1)/(5-1) = .5

(4-1)/(5-1) = .75

(5-1)/(5-1) = 1

The feature range by default is (0,1), but we can change it as per our calculation

Now for 2 dimensional array

```
Input_data = np.array([[1,2,3],[2,3,4],[1,2,2],[2,2,1]])
print("Input Data\n", Input_data)
```

```
Input Data
 [[1 2 3]
 [2 3 4]
 [1 2 2]
 [2 2 1]]
```

```
data_scaled_minmax1 = data_scaler_minmax.fit_transform(Input_data)
print(data_scaled_minmax1)
```

```
[[0.        0.        0.66666667]
 [1.        1.        1.        ]
```

```
[0.         0.         0.33333333]
[1.         0.         0.        ]]
```

Working

For every column find the min and max value and do the process as 1 D array

Min for first column is 1 and Max is 2

Min for second column is 2 and max is 3

Min for third col is 1 and max is 4

Applying Mix max scaling to the data set

```
import pandas as pd

data = pd.read_csv('E:\Bharti\DCA\MScIT\Data_Visualization\Data_Analysis_MSCIT/Tips_csv.csv')

data.info()

data.head()


feature = data.iloc[:,[1,6]].values  # extracting column 1 and 6  from Tips_csv with all rows


feature # display the contents of the two columns with all rows

feature.shape
```

```
array([[ 1.01,  2.  ],
       [ 1.66,  3.  ],
       [ 3.5 ,  3.  ],
       [ 3.31,  2.  ],
       [ 3.61,  4.  ],
       [ 4.71,  4.  ],
       [ 2.  ,  2.  ],
.
.
.

(244, 2)
```

```
# applying scaling

scale_Tips_minmax = preprocessing.MinMaxScaler(feature_range=(0,1))
```

```
data_Tips_scaled_minmax = scale_Tips_minmax.fit_transform(feature)

print(data_Tips_scaled_minmax)
```

```
[[0.00111111 0.2      ]
 [0.07333333 0.4      ]
 [0.27777778 0.4      ]
 [0.25666667 0.2      ]
 [0.29       0.6      ]
 [0.41222222 0.6      ]
 [0.11111111 0.2      ]
```

L1 Normalization

Normalizer

rescales the values on individual observations to have unit norm (the sum of their lengths is 1). This type of rescaling is often used when we have many equivalent features (e.g., text classification when every word or *n*-word group is a feature).

It is also referred to as Least Absolute Deviations. This kind of normalization modifies the values so that the sum of the absolute values is always up to 1 in each row.

L1 is basically minimizing the sum of the absolute differences (S) between the target value (x) and the estimated values (x').

$x' = (x/\delta)$
where δ is the summation of all values

For example let A [1,2,3,4,5]

L1 Normalization = [1/15,2/15,3/15,4/15,5/15] where 15 =1+2+3+4=5

L1 = [0.0666,0.1333,0.2,0.266,0.333]

```
import numpy as np

from sklearn.preprocessing import normalize

normalize(np.array([1,2,3,4,5]).reshape(1, -1), norm="l1")
```

```
array([[0.06666667, 0.13333333, 0.2       , 0.26666667, 0.33333333]])
```

```
data_normalized_l1 = preprocessing.normalize(Input_data, norm='l1')

print("\nL1 normalized data:\n", data_normalized_l1)
```

Row wise

L1 normalized data:
[[0.16666667 0.33333333 0.5]
[0.22222222 0.33333333 0.44444444]
[0.2 0.4 0.4]
[0.4 0.4 0.2]]

L2 Normalization
It is also referred to as least squares. This kind of normalization modifies the values so that t he sum of the squares is always up to 1 in each row.
L2 minimizes the sum of the square of the differences (S) between the target value (x) and t he estimated values (x').

$$x' = (x/\delta)$$
where δ is the square root of the sum of square values
$$\delta = \sqrt{\sum(x^2)}$$

Or in simpler terms, just divide each value by δ. Where δ is nothing but the square root of the sum of all the squared values.

X = [1,2,3,4,5]

$\delta = \sqrt{(1^2 + 2^2 + 3^2 + 4^2 + 5^2)}$

$= \sqrt{55} = 7.416198487095663$

L2 = 1/7.416198487095663 , 2/7.416198487095663, 3/7.416198487095663, 4/7.416198487095663, 5/7.416198487095663

array([[0.13483997, 0.26967994, 0.40451992, 0.53935989, 0.67419986]])

```
import numpy as np

from sklearn.preprocessing import normalize


normalize(np.array([1,2,3,4,5]).reshape(1, -1), norm="l2")
```

array([[0.13483997, 0.26967994, 0.40451992, 0.53935989, 0.67419986]])

```
data_normalized_l2 = preprocessing.normalize(Input_data, norm='l2')

print("\nL2 normalized data:\n", data_normalized_l2)
```

Row wise

L2 normalized data:
[[0.26726124 0.53452248 0.80178373]
[0.37139068 0.55708601 0.74278135]
[0.33333333 0.66666667 0.66666667]
[0.66666667 0.66666667 0.33333333]]

Labeling the Data

Step1: Importing the useful packages

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

Step 2: Defining sample labels
After importing the packages, we need to define some sample labels so that we can create and train the label encoder. We will now define the following sample labels:

```
# Sample input labels

input_labels = ['red','black','red','green','black','yellow','white']
```

Step 3: Creating & training of label encoder object
In this step, we need to create the label encoder and train it. The following Python code will help in doing this:

```
# Creating the label encoder

encoder = preprocessing.LabelEncoder()

encoder.fit(input_labels)
```

LabelEncoder()

Step4: Checking the performance by encoding random ordered list
This step can be used to check the performance by encoding the random ordered list. Following Python code can be written to do the same:

```
# encoding a set of labels
```

```
test_labels = ['green','red','black']

encoded_values = encoder.transform(test_labels)

print("\nLabels =", test_labels)
```

Labels = ['green', 'red', 'black']

Now, we can get the list of encoded values i.e. word labels converted to numbers as follows:

```
print("Encoded values =", list(encoded_values))
```

Encoded values = [1, 2, 0]

Step 5: Checking the performance by decoding a random set of numbers:
This step can be used to check the performance by decoding the random set of numbers. Fo llowing Python code can be written to do the same:

```
# decoding a set of values
encoded_values = [3,0,4,1]
decoded_list = encoder.inverse_transform(encoded_values)
print("\nEncoded values =", encoded_values)
```

Encoded values = [3, 0, 4, 1]

```
print("\nDecoded labels =", list(decoded_list))
```

Decoded labels = ['white', 'black', 'yellow', 'green']

Suppose we wat to label the 'vaccinated' data from the file Tips_csv.csv

```
import pandas as pd

data3 =
pd.read_csv('E:\Bharti\DCA\MScIT\Data_Visualization\DataVisualization\meal_info.csv')


data_fetch = data3.iloc[:,3].values

data_fetch
```

```
array(['Thai', 'Thai', 'Thai', 'Indian', 'Indian', 'Thai', 'Italian',
       'Italian', 'Thai', 'Thai', 'Italian', 'Thai', 'Thai', 'Thai',
       'Thai', 'Italian', 'Indian', 'Indian', 'Thai', 'Thai', 'Thai',
       'Thai', 'Thai', 'Italian', 'Italian', 'Italian', 'Indian',
       'Italian', 'Italian', 'Italian', 'Continental', 'Continental',
       'Continental', 'Indian', 'Italian', 'Indian', 'Indian', 'Indian'
,
       'Indian', 'Indian', 'Indian', 'Italian', 'Continental',
```

```
       'Continental', 'Continental', 'Continental', 'Continental',
       'Continental', 'Continental', 'Continental', 'Continental'],
      dtype=object)
```

```
food = pd.unique(data_fetch)

food
```

```
array(['Thai', 'Indian', 'Italian', 'Continental'], dtype=object)
```

convert array to list

```
food_list = food.tolist()
```

```
import numpy as np

from sklearn import preprocessing

# Creating the label encoder

encoder = preprocessing.LabelEncoder()

encoder.fit(food_list)
```

```
LabelEncoder()
```

```
encoded_food = encoder.transform(food_list)

print("Encoded values =", list(encoded_food))
```

```
Encoded values = [3, 1, 2, 0]
```

```
decoded_food = encoder.inverse_transform(encoded_food)
print("\nEncoded food types =", encoded_food)
```

```
Encoded food types = [3 1 2 0]
```

```
print("\nDecoded labels =", list(decoded_food))
```

```
Decoded labels = ['Thai', 'Indian', 'Italian', 'Continental']
```

```
Same exercise can be done for day (column 5 in excel sheet)
Here
```

```
data_fetch = data.iloc[:,4].values                    # 4 instead of 3
```