Data Preprocessing in Scikit-learn

Neba Nfonsang
University of Denver

In [1]:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets
import sklearn.preprocessing
from sklearn.preprocessing import (StandardScaler, MinMaxScaler, nor
                                   RobustScaler, Binarizer, binarize
                                   LabelBinarizer, Imputer)

from sklearn.feature_extraction import DictVectorizer
import patsy
from sklearn.pipeline import Pipeline
```

Load data from datasets

In [2]:

```
# load the boston data
# load the boston dataset
boston = datasets.load_boston()
X = boston.data
y = boston.target
```

Data Transformation

Data transformation such as standardization or normalization are important especially is the data is to be used by algorithms that compute distances between instances. To avoid variables with larger range from dominating the results of the calculated distances and causing bias, the data needs to be transformed to be on the same scale.

- The way transformation works in sklearn is that, a scaler class is used to create an object, and the fit() method is then called on the object. The data is pass as an argument into the fit() method.
- The preprocessing module in sklearn contains various scaler classes for scaling.

Some of the classes in the preprocessing module are:

Binarizer,
FunctionTransformer,
Imputer,
KernelCenterer,
LabelBinarizer,
LabelEncoder,
MaxAbsScaler,
MinMaxScaler,
MultiLabelBinarizer,
Normalizer,
OneHotEncoder,
PolynomialFeatures,
QuantileTransformer,
RobustScaler,
StandardScaler,

The StandardScaler

Transforms features to z-scores, standardized score or to new features with a mean of 0 and standard deviation of 1

In [3]:

```
# check the mean of X features of the boston data
X.mean(axis=0)
```

Out[3]:

```
array([3.59376071e+00, 1.13636364e+01, 1.11367787e+01,
6.91699605e-02,
       5.54695059e-01, 6.28463439e+00, 6.85749012e+01,
3.79504269e+00,
       9.54940711e+00, 4.08237154e+02, 1.84555336e+01,
3.56674032e+02,
       1.26530632e+01])
```

In [4]:

```
# check the std of X features of the boston data
X.std(axis=0 )
```

Out[4]:

```
array([8.58828355e+00, 2.32993957e+01, 6.85357058e+00,
2.53742935e-01,
       1.15763115e-01, 7.01922514e-01, 2.81210326e+01,
2.10362836e+00,
       8.69865112e+00, 1.68370495e+02, 2.16280519e+00,
9.12046075e+01,
       7.13400164e+00])
```

In [5]:

```
std_sc = StandardScaler()
std_sc.fit(X)
# mean of the tranformed data
std_sc.transform(X).mean(axis=0)
```

Out[5]:

```
array([ 6.34099712e-17, -6.34319123e-16, -2.68291099e-1
5,  4.70199198e-16,
        2.49032240e-15, -1.14523016e-14, -1.40785495e-1
5,  9.21090169e-16,
        5.44140929e-16, -8.86861950e-16, -9.20563581e-1
5,  8.16310129e-15,
       -3.37016317e-16])
```

In [6]:

```
# an alternative way to fit and transform at once
std_sc = StandardScaler()
X_std_sc = std_sc.fit(X).transform(X)
# standard deviation of transformed data
X_std_sc.std(axis=0)
```

Out[6]:

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1.])
```

The MinMaxScaler

Transforms features by scaling each feature to a given range, default range is [0, 1]

In [7]:

```
MinMaxScaler().fit(X).transform(X)
```

Out[7]:

```
array([[0.00000000e+00, 1.80000000e-01, 6.78152493e-02,
...,
        2.87234043e-01, 1.00000000e+00, 8.96799117e-0
2],
       [2.35922539e-04, 0.00000000e+00, 2.42302053e-01,
...,
        5.53191489e-01, 1.00000000e+00, 2.04470199e-0
1],
       [2.35697744e-04, 0.00000000e+00, 2.42302053e-01,
...,
        5.53191489e-01, 9.89737254e-01, 6.34657837e-0
2],
       ...,
       [6.11892474e-04, 0.00000000e+00, 4.20454545e-01,
...,
        8.93617021e-01, 1.00000000e+00, 1.07891832e-0
1],
       [1.16072990e-03, 0.00000000e+00, 4.20454545e-01,
...,
        8.93617021e-01, 9.91300620e-01, 1.31070640e-0
1],
       [4.61841693e-04, 0.00000000e+00, 4.20454545e-01,
...,
        8.93617021e-01, 1.00000000e+00, 1.69701987e-0
1]])
```

In [8]:

```
# check the maximum values of the transformed data
MinMaxScaler().fit(X).transform(X).max(axis=0)
```

Out[8]:

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1.])
```

In [9]:

```
# change range of transformed data
MinMaxScaler(feature_range=(-3.5, 3.5)).fit(X).transform(X)
```

Out[9]:

```
array([[-3.5       , -2.24      , -3.02529326, ..., -1.
4893617 ,
         3.5       , -2.87224062],
       [-3.49834854, -3.5       , -1.80388563, ...,  0.
37234043,
         3.5       , -2.06870861],
       [-3.49835012, -3.5       , -1.80388563, ...,  0.
37234043,
         3.42816077, -3.05573951],
       ...,
       [-3.49571675, -3.5       , -0.55681818, ...,  2.
75531915,
         3.5       , -2.74475717],
       [-3.49187489, -3.5       , -0.55681818, ...,  2.
75531915,
         3.43910434, -2.58250552],
       [-3.49676711, -3.5       , -0.55681818, ...,  2.
75531915,
         3.5       , -2.31208609]])
```

The normalize() Function

Scale input vectors individually to unit norm (vector length). That is, each instance or row is divided by its norm l2.

In [10]:

```
normalize(X)
```

Out[10]:

```
array([[1.26388341e-05, 3.59966795e-02, 4.61957387e-03,
...,
        3.05971776e-02, 7.93726783e-01, 9.95908132e-0
3],
       [5.78529889e-05, 0.00000000e+00, 1.49769546e-02,
...,
        3.77071843e-02, 8.40785474e-01, 1.93620036e-0
2],
       [5.85729947e-05, 0.00000000e+00, 1.51744622e-02,
...,
        3.82044450e-02, 8.43137761e-01, 8.64965806e-0
3],
       ...,
       [1.23765824e-04, 0.00000000e+00, 2.43009593e-02,
...,
        4.27762066e-02, 8.08470305e-01, 1.14884669e-0
2],
       [2.24644719e-04, 0.00000000e+00, 2.44548909e-02,
...,
        4.30471676e-02, 8.06519433e-01, 1.32831260e-0
2],
       [9.69214289e-05, 0.00000000e+00, 2.43887924e-02,
...,
        4.29308164e-02, 8.11392431e-01, 1.61092778e-0
2]])
```

In [11]:

```
# let's demonstrate how normalize works
normalize(np.array([[7, 8, 9],[2,4,6]]))
```

Out[11]:

```
array([[0.50257071, 0.57436653, 0.64616234],
       [0.26726124, 0.53452248, 0.80178373]])
```

In [12]:

```
# divide each vector by it's norm l2
np.array([7, 8, 9])/np.sqrt(sum([7**2, 8**2, 9**2]))
```

Out[12]:

```
array([0.50257071, 0.57436653, 0.64616234])
```

In [13]:

```
np.array([2,4,6])/np.sqrt(sum([2**2, 4**2, 6**2]))
```

Out[13]:

```
array([0.26726124, 0.53452248, 0.80178373])
```

The RobustScaler

- Scale features using statistics that are robust to outliers.
- Removes the median and scale the data by the interquartile range
- The IQR is the range between the 1st quartile (25th quantile) and the 3rd quartile (75th quantile)
- That is, tranformed value = (untransformed value - median)/IQR

In [14]:

```
RobustScaler().fit(X).transform(X)[0:5]
```

Out[14]:

```
array([[-0.07017209,  1.44      , -0.57164988,  0.
   ,  0.        ,
         0.49661247, -0.25076453,  0.28577673, -0.2
   , -0.0878553 ,
        -1.33928571,  0.26190191, -0.63768116],
       [-0.06428492,  0.        , -0.20294345,  0.
   , -0.39428571,
         0.28794038,  0.0285423 ,  0.56978872, -0.15
   , -0.22739018,
        -0.44642857,  0.26190191, -0.22188906],
       [-0.06429053,  0.        , -0.20294345,  0.
   , -0.39428571,
         1.32317073, -0.3343527 ,  0.56978872, -0.15
   , -0.22739018,
        -0.44642857,  0.06667466, -0.73263368],
       [-0.06286571,  0.        , -0.5817196 ,  0.
   , -0.45714286,
         1.0697832 , -0.64627931,  0.92439084, -0.1
   , -0.27906977,
        -0.125     ,  0.15301595, -0.84157921],
       [-0.05257788,  0.        , -0.5817196 ,  0.
   , -0.45714286,
         1.27168022, -0.47502548,  0.92439084, -0.1
   , -0.27906977,
        -0.125     ,  0.26190191, -0.60269865]])
```

The Binarizer

This transforms a vector of values to binary values of 0's and 1's. A threshold is provided: if the vector value is greater than the threshold, 1 is returned, if the vector value is less than or equal to the threshold, 0 is returned.

- Let's create a binary output variable with the boston data, so we can use it for classification. We want to classify instances or cities whose median house prices (the output variable) are above the mean into class 1, otherwise class 0.

In [15]:

```
threshold = np.mean(y)
# Reshape your data either using array.reshape(-1, 1)
# if your data has a single feature or if it contains a single sampl
bin_sc = Binarizer(threshold=threshold)
bin_sc.fit(y.reshape(1, -1))
# view a 20 data points of the transformed data
bin_sc.transform(y.reshape(1, -1))[:,0:10]
```

Out[15]:

```
array([[1., 0., 1., 1., 1., 1., 1., 1., 0., 0.]])
```

In [16]:

```
bin_sc.transform(y.reshape(1, -1)).shape
```

Out[16]:

```
(1, 506)
```

In [17]:

```
# the binarize function could also be used
binarize(y.reshape(1, -1), threshold=np.mean(y))[:,0:10]
```

Out[17]:

```
array([[1., 0., 1., 1., 1., 1., 1., 1., 0., 0.]])
```

Transforming Categorical Variables

In [18]:

```
# load another suitable dataset
iris = datasets.load_iris()
X, y = iris.data, iris.target
```

In [19]:

```
# view five rows of the input data
X[0:5]
```

Out[19]:

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2]])
```

In [20]:

```
# view the output data
y
```

Out[20]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2])
```

In [21]:

```
# names of feature variables
iris.feature_names
```

Out[21]:

```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

In [22]:

```
# values in the target variable
iris.target_names
```

Out[22]:

```
array(['setosa', 'versicolor', 'virginica'], dtype='<U1
0')
```

In [23]:

```
# combine the input and output data
d = np.column_stack((X, y))

# view the first few rows of the data
d[0:5]
```

Out[23]:

```
array([[5.1, 3.5, 1.4, 0.2, 0. ],
       [4.9, 3. , 1.4, 0.2, 0. ],
       [4.7, 3.2, 1.3, 0.2, 0. ],
       [4.6, 3.1, 1.5, 0.2, 0. ],
       [5. , 3.6, 1.4, 0.2, 0. ]])
```

- The values of all variables in sklearn must be represented as numbers. So, we need to transform the values of the output variable in the iris data to numerical.
- Imagine we want to use the iris data to predict sepal width, then the specie would be used as part of the features. It is not a good idea to have values as 0, 1, 2. We would rather change these categorical values to three dummy variables using one-hot encoder.

The OneHotEncoder

Encode categorical integer features using a one-hot aka one-of-K scheme.

In [24]:

```
one_hot = OneHotEncoder()
one_hot.fit(d[:,-1:]).transform(d[:,-1:]).toarray()[0:5]
```

Out[24]:

```
array([[1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.]])
```

The DictVectorizer

Transforms lists of feature-value mappings to vectors. We can use the DictVectorizer to transform a list of dictionary items to dummy variables.

In [25]:

```
dv = DictVectorizer()
dictionary = [{"specie": iris.target_names[i]} for i in y]
dictionary[0:5]
```

Out[25]:

```
[{'specie': 'setosa'},
 {'specie': 'setosa'},
 {'specie': 'setosa'},
 {'specie': 'setosa'},
 {'specie': 'setosa'}]
```

In [26]:

```
dv.fit(dictionary).transform(dictionary).toarray()[0:5]
```

Out[26]:

```
array([[1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.]])
```

Using Patsy

In [27]:

```
# Construct a single design matrix given a formula_like and data.
species = iris.target
patsy.dmatrix("0 + C(species)", {"species": species})[0:5]
```

Out[27]:

```
array([[1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.]])
```

Binarizing Label Features

LabelBinarizer can be used to transform label features into dummy variables. This is an alternative to OneHotEncoding. The labels could be numeric code or text.

In [28]:

```
target = iris.target
label_binarizer = LabelBinarizer()
label_binarizer.fit(target).transform(target)[0:5]
```

Out[28]:

```
array([[1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [1, 0, 0]])
```

In [29]:

```
# the values of the feature could be text
cat = np.array(["A", "B", "A", "C", "B", "A"])
LabelBinarizer().fit(cat).transform(cat)
```

Out[29]:

```
array([[1, 0, 0],
       [0, 1, 0],
       [1, 0, 0],
       [0, 0, 1],
       [0, 1, 0],
       [1, 0, 0]])
```

Impute Missing Values

In [30]:

```
np.random.seed(2020)
dat = np.random.randint(10, 20, 60).reshape(15, 4)
dat
```

Out[30]:

```
array([[10, 18, 13, 16],
       [13, 13, 17, 18],
       [10, 10, 18, 19],
       [13, 17, 12, 13],
       [16, 15, 10, 14],
       [18, 16, 14, 11],
       [11, 15, 19, 15],
       [16, 16, 16, 15],
       [14, 16, 14, 12],
       [13, 14, 17, 11],
       [14, 19, 13, 12],
       [10, 19, 11, 12],
       [17, 11, 10, 12],
       [18, 18, 15, 16],
       [13, 13, 10, 10]])
```

In [31]:

```
# add missing value 999 to the data
dat[dat>16] = 999
dat[dat>16] = 999
dat
```

Out[31]:

```
array([[ 10, 999,  13,  16],
       [ 13,  13, 999, 999],
       [ 10,  10, 999, 999],
       [ 13, 999,  12,  13],
       [ 16,  15,  10,  14],
       [999,  16,  14,  11],
       [ 11,  15, 999,  15],
       [ 16,  16,  16,  15],
       [ 14,  16,  14,  12],
       [ 13,  14, 999,  11],
       [ 14, 999,  13,  12],
       [ 10, 999,  11,  12],
       [999,  11,  10,  12],
       [999, 999,  15,  16],
       [ 13,  13,  10,  10]])
```

In [32]:

```
impute = Imputer(missing_values=999, strategy="mean")
imputed = impute.fit_transform(dat)
imputed
```

Out[32]:

```
array([[10.        , 13.9       , 13.        , 16.        ],
       [13.        , 13.        , 12.54545455, 13.        ],
       [10.        , 10.        , 12.54545455, 13.        ],
       [13.        , 13.9       , 12.        , 13.        ],
       [16.        , 15.        , 10.        , 14.        ],
       [12.75      , 16.        , 14.        , 11.        ],
       [11.        , 15.        , 12.54545455, 15.        ],
       [16.        , 16.        , 16.        , 15.        ],
       [14.        , 16.        , 14.        , 12.        ],
       [13.        , 14.        , 12.54545455, 11.        ],
       [14.        , 13.9       , 13.        , 12.        ],
       [10.        , 13.9       , 11.        , 12.        ],
       [12.75      , 11.        , 10.        , 12.        ],
       [12.75      , 13.9       , 15.        , 16.        ],
       [13.        , 13.        , 10.        , 10.        ]])
```

Pipeline

Several preprocessing steps can be completed at once using pipeline

- After imputing the missing values, we standardize the data as follows:

In [33]:

```
standardized = StandardScaler().fit_transform(imputed)
standardized[0:5]
```

Out[33]:

```
array([[-1.50248550e+00, -1.07575702e-15,  2.63230822e-
01,
         1.67705098e+00],
       [ 1.36589591e-01, -5.45037627e-01,  0.00000000e+
00,
         0.00000000e+00],
       [-1.50248550e+00, -2.36182972e+00,  0.00000000e+
00,
         0.00000000e+00],
       [ 1.36589591e-01, -1.07575702e-15, -3.15876986e-
01,
         0.00000000e+00],
       [ 1.77566469e+00,  6.66157099e-01, -1.47409260e+
00,
         5.59016994e-01]])
```

Using pipeline, the imputation and standardization can be accomplished in one step as follows:

In [34]:

```
impute = Imputer(missing_values=999)
scaler = StandardScaler()
pipe = Pipeline([("impute", impute), ("scaler", scaler)])
pipe
```

Out[34]:

```
Pipeline(memory=None,
     steps=[('impute', Imputer(axis=0, copy=True, missi
ng_values=999, strategy='mean', verbose=0)), ('scaler',
StandardScaler(copy=True, with_mean=True, with_std=Tru
e))])
```

In [35]:

```
pipe.fit_transform(dat)[0:5]
```

Out[35]:

```
array([[-1.50248550e+00, -1.07575702e-15,  2.63230822e-
01,
         1.67705098e+00],
       [ 1.36589591e-01, -5.45037627e-01,  0.00000000e+
00,
         0.00000000e+00],
       [-1.50248550e+00, -2.36182972e+00,  0.00000000e+
00,
         0.00000000e+00],
       [ 1.36589591e-01, -1.07575702e-15, -3.15876986e-
01,
         0.00000000e+00],
       [ 1.77566469e+00,  6.66157099e-01, -1.47409260e+
00,
         5.59016994e-01]])
```

NB: Data transformation in Scikit-learn can be accomplished through the .fit() and .transform() methods or for convinience, you could use the .fit_transform() method.