Python Tutorial 5

February 16, 2022

This tutorial is for Prof. Xin Tong's DSO 530 class at the University of Southern California in spring 2022. It dissusses HW1.

1 Question 3

A few students try to use `normalize()` from `sklearn.preprocessing` instead of using `MinMaxScaler()` as we taught in Tutorial 2. But `normalize()` doesn't work as we thought it might do here.

The function `normalize()` provides a quick and easy way to perform other kinds of **normalization** like *l1 norms* or *l2 norms*:

By default, `normalize()` operates on the rows of the data.

For example:

[1]:
```
from sklearn.preprocessing import normalize
X = [[ 1., -1.,  2.],
     [ 2.,  0.,  0.],
     [ 0.,  1., -1.]]
```

[2]:
```
X_normalized_1 = normalize(X, norm='l1')
X_normalized_1
```

[2]:
```
array([[ 0.25, -0.25,  0.5 ],
       [ 1.  ,  0.  ,  0.  ],
       [ 0.  ,  0.5 , -0.5 ]])
```

When `norm='l1'`, it calculates the *Absolute-value Norm*. In this example: *l1 norm* of the first row is |1| + |-1| + |2| = 4; *l1 norm* of the second row is |2| + |0| + |0| = 2; *l1 norm* of the third row is |0| + |1| + |-1| = 2. The function calculates the values that equal to the original values divided by the *l1 norm* of each row.

Therefore, it returns the array:[[1/4, -1/4, 2/4],[2/2, 0/2, 0/2],[0/2, 1/2, -1/2]]

[3]:
```
X_normalized_2 = normalize(X, norm='l2')
X_normalized_2
```

[3]:
```
array([[ 0.40824829, -0.40824829,  0.81649658],
       [ 1.        ,  0.        ,  0.        ],
```

```
 [ 0.        ,  0.70710678, -0.70710678]])
```

When `norm='l2'`, it calculates the *Euclidean Norm*. In this example: *l2 norm* of the first row is $\sqrt{1^2+(-1)^2+2^2}=\sqrt{6}$; *l2 norm* of the second row is $\sqrt{2^2+0^2+0^2}=2$; *l2 norm* of the third row is $\sqrt{0^2+1^2+(-1)^2}=\sqrt{2}$. The function calculates the values that equal to the original values divided by the *l1 norm* of each row.

Therefore, it returns the array:[[1/sqrt(6), -1/sqrt(6), 2/sqrt(6)],[2/2, 0/2, 0/2],[0/sqrt(2), 1/sqrt(2), -1/sqrt(2)]]

[4]:
```python
X_normalized_3 = normalize(X, norm='max')
X_normalized_3
```

[4]:
```
array([[ 0.5, -0.5,  1. ],
       [ 1. ,  0. ,  0. ],
       [ 0. ,  1. , -1. ]])
```

When `norm='max'`, it calculates the *Maximum Norm*. In this example: *max norm* of the first row is *max(1, -1, 2) = 2*; *max norm* of the second row is *max(2, 0, 0) = 2*; *max norm* of the third row is *max(0, 1, -1) = 1*. The function calculates the values that equal to the original values divided by the *l1 norm* of each row.

Therefore, it returns the array:[[1/2, -1/2, 2/2],[2/2, 0/2, 0/2],[0/1, 1/1, -1/1]]

We can see that it does not achieve the normalization as we taught in Tutorial 2. Most importantly, the function `normalize()` by default works on the instances as opposed to the variables.

Apart from using `normalize()` function, some students manually normalized the data. It is ok if the normalization is done in a correct way. For example, for *housing* dataset, one can do the following.

[5]:
```python
import pandas as pd
housing = pd.read_csv('housing.csv')
ptratio_max = max(housing['ptratio'])
ptratio_min = min(housing['ptratio'])
rm_max = max(housing['rm'])
rm_min = min(housing['rm'])
housing['ptratio_nc']= (housing['ptratio']-ptratio_min)/
 ↪(ptratio_max-ptratio_min)
housing['rm_nc']= (housing['rm']-rm_min)/(rm_max-rm_min)
housing[['ptratio_nc','rm_nc']].head()
```

[5]:

	ptratio_nc	rm_nc
0	0.287234	0.577505
1	0.553191	0.547998
2	0.553191	0.694386
3	0.648936	0.658555
4	0.648936	0.687105

However, some students forgot the parentheses on the denominators like the following

[6]:
```
housing['ptratio_nw']= (housing['ptratio']-ptratio_min)/ptratio_max-ptratio_min
housing['rm_nw']= (housing['rm']-rm_min)/rm_max-rm_min
housing[['ptratio_nw','rm_nw']].head()
```

[6]:

	ptratio_nw	rm_nw
0	-12.477273	-3.217720
1	-12.363636	-3.235260
2	-12.363636	-3.148244
3	-12.322727	-3.169542
4	-12.322727	-3.152572

It is clear the two ways yield different results. However, the final answer to Question 3 is not affacted by it. That is, the R^2 is still correct even if the parenthese are forgotten.

[7]:
```
from sklearn.linear_model import LinearRegression
X_c = housing[['ptratio_nc','rm_nc']].values
y= housing['medv'].values
model_c = LinearRegression()
model_c.fit(X_c,y)
r_sq_c = model_c.score(X_c,y)
print('coefficient of determination with correct normalization:', r_sq_c)
X_w = housing[['ptratio_nw','rm_nw']].values
model_w = LinearRegression()
model_w.fit(X_w,y)
r_sq_w = model_w.score(X_w,y)
print('coefficient of determination with wrong normalization:', r_sq_w)
```

```
coefficient of determination with correct normalization: 0.5612534621272915
coefficient of determination with wrong normalization: 0.5612534621272919
```

Thus, having the correct final result does not necessarily mean the intermediate steps are correct.

2 Question 4

This part aims to address a counterintuitive observation in question 4 of HW1. We will review and solve this question first.

Question 4 in HW1 is as follows:

4. Split the housing data into two parts with 30% as test data. Use `random_state = 2` in this split. Because this is a regression problem, you don't want to use stratify = y part of the code from our Python tutorial. Regress medv on `river` and `rm` using the training data. Compute R2 on both the training data and the test data.

We first import the *housing* dataset:

[8]:
```
import pandas as pd
import numpy as np
```

```
housing = pd.read_csv("Housing.csv")
```

Then we split the housing data into training and test datasets:

[9]:
```
from sklearn.model_selection import train_test_split

X, y = housing[['river','rm']].values, housing['medv'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
↪random_state=2)
```

At last we use linear regression to regress `medv` on `river` and `rm` and we calculate the R^2:

[10]:
```
# regress medv on river and rm

from sklearn.linear_model import LinearRegression

linear_model_4 = LinearRegression()
linear_model_4.fit(X_train, y_train)

r_sq_train = linear_model_4.score(X_train,y_train)
print(f'R-square on training data: {r_sq_train}')

r_sq_test = linear_model_4.score(X_test, y_test)
print(f'R-square on test data: {r_sq_test}')
```

```
R-square on training data: 0.4652498065943519
R-square on test data: 0.5582472793500368
```

Note that here R-squared on test data (i.e., out-of-sample R-squared) is larger than R-squared on training data (i.e., in-sample R-squared). Generally, the R-squared on test data should be smaller than the R-squaredd on training data. But test data itself involves randomness, and some random seeds might just result in a test data that is somewhat more linearly dependent than the training set. Now, we run the whole process 1000 times and calculate the average R-squared on training data and the average R-squared on test data. We can see that the average in-sample R-squared is larger than the average out-of-sample R-squared.

[11]:
```
N = 1000
linear_model = LinearRegression()
r_sq_train = np.zeros(N)
r_sq_test = np.zeros(N)
for i in range(N):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
↪random_state=i)
    linear_model.fit(X_train, y_train)
    r_sq_train[i] = linear_model.score(X_train,y_train)
    r_sq_test[i] = linear_model.score(X_test, y_test)
```

[12]:
```
r_sq_train.mean()
```

[12]: 0.498215503619291

[13]:
```
r_sq_test.mean()
```

[13]: 0.4760423696254844