Processing_Data

December 25, 2021

1 Processing Complex Data

So far we have assumed that the intput is in the form of numerical vectors to which we can apply directly the algorithms we have. Often the data will be more complex. For example what if we want to cluster categorical data, itemsets, or text? Python provides libraries for processing the data and transforming them to a format that we can use.

Python offers a set of tools for extracting features:http://scikit-learn.org/stable/modules/feature_extraction.html

[32]:
```
import numpy as np
import scipy as sp
import scipy.sparse as sp_sparse
import scipy.spatial.distance as sp_dist

import matplotlib.pyplot as plt

import sklearn as sk
import sklearn.datasets as sk_data
import sklearn.metrics as metrics
from sklearn import preprocessing
import sklearn.cluster as sk_cluster
import sklearn.feature_extraction.text as sk_text


import scipy.cluster.hierarchy as hr

import time
import seaborn as sns

%matplotlib inline
```

1.0.1 Ordinal Encoder

The Ordinal encoder enables us to envode categorical attributes as numerical. You can read more here:

https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OrdinalEncoder.html#sklearn.preproces

[39]:
```
from sklearn.preprocessing import OrdinalEncoder

X = [['married','Yes','Athens'],
     ['single','No', 'Ioannina'],
     ['married','No', 'Thessaloniki'],
     ['divorced', 'Yes', 'Athens']]
enc = OrdinalEncoder(handle_unknown = 'use_encoded_value', unknown_value = np.
 ↪nan)
enc.fit(X)
print(enc.categories_)
print(enc.transform(X))
Y = [['married','No','Athens'],
     ['single','Yes', 'Ioannina'],
     ['single','Yes', 'Patras']
    ]
enc.transform(Y)
```

```
[array(['divorced', 'married', 'single'], dtype=object), array(['No', 'Yes'],
dtype=object), array(['Athens', 'Ioannina', 'Thessaloniki'], dtype=object)]
[[1. 1. 0.]
 [2. 0. 1.]
 [1. 0. 2.]
 [0. 1. 0.]]
```

[39]:
```
array([[ 1.,  0.,  0.],
       [ 2.,  1.,  1.],
       [ 2.,  1., nan]])
```

[40]:
```
X = [['married','Yes',30000],
     ['single','No', 24000],
     ['divorced', 'Yes', 50000]]
enc = OrdinalEncoder(handle_unknown = 'use_encoded_value',unknown_value = -1)
enc.fit(X)
print(enc.categories_)
print(enc.transform(X))
Y = [['married','No',10000],
     ['single','Yes', 24000]]
print(enc.transform(Y))
```

```
[array(['divorced', 'married', 'single'], dtype=object), array(['No', 'Yes'],
dtype=object), array([24000, 30000, 50000], dtype=object)]
[[1. 1. 1.]
 [2. 0. 0.]
 [0. 1. 2.]]
[[ 1.  0. -1.]
 [ 2.  1.  0.]]
```

1.0.2 DictVectorizer

The DictVectorizer feature extraction: http://scikit-learn.org/stable/modules/generated/sklearn.feature_extractio

The DictVectorizer takes a dictionary of attribute-value pairs and transforms them into numerical vectors. Real values are preserved, while categorical attributes are transformed into binary. The vectorizer produces a *sparse representation*.

[41]:
```
from sklearn.feature_extraction import DictVectorizer

measurements = [
{'city': 'Dubai', 'temperature': 45},
{'city': 'London', 'temperature': 12},
{'city': 'San Fransisco', 'temperature': 23},
]
vec = DictVectorizer()
print(type(vec.fit_transform(measurements)))
print(vec.fit_transform(measurements).toarray())
vec.get_feature_names_out()
```

```
<class 'scipy.sparse.csr.csr_matrix'>
[[ 1.  0.  0. 45.]
 [ 0.  1.  0. 12.]
 [ 0.  0.  1. 23.]]
```

[41]:
```
array(['city=Dubai', 'city=London', 'city=San Fransisco', 'temperature'],
      dtype=object)
```

[42]:
```
from sklearn.feature_extraction import DictVectorizer

measurements = [
{'city': 'Dubai', 'temperature': 45, 'dummy': 3},
{'city': 'London', 'temperature': 12},
{'city': 'San Fransisco', 'temperature': 23},
]
vec = DictVectorizer()
vec.fit(measurements)
print(vec.get_feature_names_out())
print(vec.transform(measurements).toarray())
x = {'city': 'Athens', 'temperature': 32, 'dummy2': 2}
print(vec.transform(x).toarray())
```

```
['city=Dubai' 'city=London' 'city=San Fransisco' 'dummy' 'temperature']
[[ 1.  0.  0.  3. 45.]
 [ 0.  1.  0.  0. 12.]
 [ 0.  0.  1.  0. 23.]]
[[ 0.  0.  0.  0. 32.]]
```

[43]:
```
measurements = [
    {'refund' : 'No','marital_status': 'married', 'income' : 100},
    {'refund' : 'Yes','marital_status': 'single', 'income' : 120},
    {'refund' : 'No','marital_status':'divorced', 'income' : 80},
]
vec = DictVectorizer()
print(vec.fit_transform(measurements))
vec.get_feature_names_out()
```

```
  (0, 0)        100.0
  (0, 2)        1.0
  (0, 4)        1.0
  (1, 0)        120.0
  (1, 3)        1.0
  (1, 5)        1.0
  (2, 0)        80.0
  (2, 1)        1.0
  (2, 4)        1.0
```

[43]:
```
array(['income', 'marital_status=divorced', 'marital_status=married',
       'marital_status=single', 'refund=No', 'refund=Yes'], dtype=object)
```

1.0.3 OneHotEncoder

The **OneHotEncoder** can be used for categorical data to transform them into binary, where for each attribute value we have 0 or 1 depending on whether this value appears in the feature vector. It works with numerical categorical values.

You can read more about it here: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHot

[45]:
```
X = [[0,1,2],
     [1,2,3],
     [0,1,4]]
enc = preprocessing.OneHotEncoder(handle_unknown='ignore')
enc.fit(X)
enc.transform([[0,2,4],[1,1,2]]).toarray()
enc.transform([[2,2,4],[1,1,2]]).toarray()
```

[45]:
```
array([[0., 0., 0., 1., 0., 0., 1.],
       [0., 1., 1., 0., 1., 0., 0.]])
```

In this example every number in every column defines a separate feature

[46]:
```
enc.categories_
```

[46]:
```
[array([0, 1]), array([1, 2]), array([2, 3, 4])]
```

[48]:
```
X = [['married','Yes',30000],
     ['single','No', 24000],
     ['divorced', 'Yes', 50000]]
enc = preprocessing.OneHotEncoder(handle_unknown='ignore')
enc.fit_transform(X).toarray()
```

[48]:
```
array([[0., 1., 0., 0., 1., 0., 1., 0.],
       [0., 0., 1., 1., 0., 1., 0., 0.],
       [1., 0., 0., 0., 1., 0., 0., 1.]])
```

You can keep binary categories as binary. In the following example note that we used only a single column for the fist two attributes

[50]:
```
X = [[0,1,2],
     [1,2,3],
     [0,1,4]]
enc = preprocessing.OneHotEncoder(drop = 'if_binary')
enc.fit(X)
print(enc.categories_)
print(enc.transform([[0,2,4],[1,1,2]]).toarray())
```

```
[array([0, 1]), array([1, 2]), array([2, 3, 4])]
[[0. 1. 0. 0. 1.]
 [1. 0. 1. 0. 0.]]
```

1.1 Text processing

Feature extraction from text: http://scikit-learn.org/stable/modules/classes.html#text-feature-extraction-ref

1.1.1 CountVectorizer

The CountVectorizer can be used to extract features in the form of bag of words. It is typically used for text, but you could use it to represent also a collection of itemsets (where each itemset will become a word).

[52]:
```
import sklearn.feature_extraction.text as sk_text

corpus = ['This is the first document.',
          'this is the second second document.',
          'And the third one.',
          'Is this the first document?',
         ]

vectorizer = sk_text.CountVectorizer(min_df=1)
X = vectorizer.fit_transform(corpus)
print(X.toarray())
vectorizer.get_feature_names_out()
```

```
[[0 1 1 1 0 0 1 0 1]
 [0 1 0 1 0 2 1 0 1]
 [1 0 0 0 1 0 1 1 0]
 [0 1 1 1 0 0 1 0 1]]
```

[52]:
```
array(['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third',
       'this'], dtype=object)
```

[54]:
```
import sklearn.feature_extraction.text as sk_text

corpus = ['This is the first document.',
          'this is the second second document.',
          'And the third one.',
          'Is this the first document?',
         ]

vectorizer = sk_text.CountVectorizer(min_df=2)
X = vectorizer.fit_transform(corpus)
print(X.toarray())
vectorizer.get_feature_names_out()
```

```
[[1 1 1 1 1]
 [1 0 1 1 1]
 [0 0 0 1 0]
 [1 1 1 1 1]]
```

[54]:
```
array(['document', 'first', 'is', 'the', 'this'], dtype=object)
```

[53]:
```
vectorizer = sk_text.CountVectorizer(min_df=1,stop_words = 'english')

X2 = vectorizer.fit_transform(corpus)
print(X2.toarray())
vectorizer.get_feature_names()
```

```
[[1 0]
 [1 2]
 [0 0]
 [1 0]]
```

[53]:
```
['document', 'second']
```

1.1.2 TfIdfVectorizer

TfIdfVectorizer transforms text into a sparse matrix where rows are text and columns are words, and values are the tf-dif values. It performs tokenization, normalization, and removes stop-words. More here: http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html#sklearn.feature_extrac

[55]:
```
vectorizer = sk_text.TfidfVectorizer(min_df=1)
X = vectorizer.fit_transform(corpus)
print(X.toarray())
print (vectorizer.get_feature_names_out())
```

```
[[0.         0.43877674 0.54197657 0.43877674 0.         0.
  0.35872874 0.         0.43877674]
 [0.         0.27230147 0.         0.27230147 0.         0.85322574
  0.22262429 0.         0.27230147]
 [0.55280532 0.         0.         0.         0.55280532 0.
  0.28847675 0.55280532 0.        ]
 [0.         0.43877674 0.54197657 0.43877674 0.         0.
  0.35872874 0.         0.43877674]]
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\deprecation.py:87:
FutureWarning: Function get_feature_names is deprecated; get_feature_names is
deprecated in 1.0 and will be removed in 1.2. Please use get_feature_names_out
instead.
  warnings.warn(msg, category=FutureWarning)
```

Removing stop-words

[56]:
```
vectorizer = sk_text.TfidfVectorizer(stop_words = 'english',min_df=1)
X = vectorizer.fit_transform(corpus)
print(X.toarray())
print (vectorizer.get_feature_names_out())
```

```
[[1.         0.        ]
 [0.30403549 0.9526607 ]
 [0.         0.        ]
 [1.         0.        ]]
['document', 'second']
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\deprecation.py:87:
FutureWarning: Function get_feature_names is deprecated; get_feature_names is
deprecated in 1.0 and will be removed in 1.2. Please use get_feature_names_out
instead.
  warnings.warn(msg, category=FutureWarning)
```

SciKit datasets: http://scikit-learn.org/stable/datasets/

We will use the 20-newsgroups datasets which consists of postings on 20 different newsgroups.

More information here: http://scikit-learn.org/stable/datasets/#the-20-newsgroups-text-dataset

[57]:
```
from sklearn.datasets import fetch_20newsgroups

categories = ['comp.os.ms-windows.misc', 'sci.space','rec.sport.baseball']
#categories = ['alt.atheism', 'sci.space','rec.sport.baseball']
news_data = sk_data.fetch_20newsgroups(subset='train',
```

```
                                remove=('headers', 'footers', 'quotes'),
                                categories=categories)
print (news_data.target)
print (len(news_data.target))
```

```
[2 0 0 … 2 1 2]
1781
```

[72]:
```
print (type(news_data))
print (news_data.filenames)
print (news_data.target[:10])
print (news_data.data[1])
print (len(news_data.data))
```

```
<class 'sklearn.utils.Bunch'>
['C:\\Users\\tsapa\\scikit_learn_data\\20news_home\\20news-bydate-
train\\sci.space\\60940'
 'C:\\Users\\tsapa\\scikit_learn_data\\20news_home\\20news-bydate-
train\\comp.os.ms-windows.misc\\9955'
 'C:\\Users\\tsapa\\scikit_learn_data\\20news_home\\20news-bydate-
train\\comp.os.ms-windows.misc\\9846'
 …
 'C:\\Users\\tsapa\\scikit_learn_data\\20news_home\\20news-bydate-
train\\sci.space\\60891'
 'C:\\Users\\tsapa\\scikit_learn_data\\20news_home\\20news-bydate-
train\\rec.sport.baseball\\104484'
 'C:\\Users\\tsapa\\scikit_learn_data\\20news_home\\20news-bydate-
train\\sci.space\\61110']
[2 0 0 2 0 0 1 2 2 1]
Recently the following problem has arrisen.  The first time I turn on my
computer when windows starts (from my autoexec) after the win31 title screen
the computer reboots on its own.  Usually the second time (after reboot) or
from the DOS prompt everything works fine.

 s far as I remember I have not changed my config.sys or autoxec.bat or
win.ini.  I can't remember whether this problem occured before I
optimized/defragmented my disk and created a larger swap file (Thank you
MathCAD 4 :(  )

System 386sx, 4MB, stacker 2.0, win31, DOS 5

---
--------------------------------------------------------------------------
1781
```

[60]:
```
vectorizer = sk_text.TfidfVectorizer(stop_words='english',
                                #max_features = 100,
```

```
                                min_df=4, max_df=0.8)
data = vectorizer.fit_transform(news_data.data)
print(type(data))
print(vectorizer.get_feature_names_out())
print(data[0])
```

```
<class 'scipy.sparse.csr.csr_matrix'>
['00' '02' '04' '0d' '0t' '10' '14' '145' '17' '1d9' '1t' '2di' '2tm' '34'
 '34u' '3t' '45' '5u' '6ei' '6um' '75u' '7ey' '7u' '9v' 'a86' 'ah' 'air'
 'available' 'ax' 'b8f' 'better' 'bhj' 'bj' 'bxn' 'c_' 'chz' 'ck' 'cx'
 'd9' 'data' 'did' 'does' 'don' 'dos' 'earth' 'edu' 'file' 'files' 'g9v'
 'game' 'giz' 'gk' 'good' 'got' 'information' 'just' 'know' 'launch'
 'like' 'lk' 'll' 'lunar' 'make' 'max' 'mq' 'mv' 'nasa' 'need' 'new'
 'orbit' 'people' 'pl' 'problem' 'program' 'qq' 'really' 'right' 'run'
 'satellite' 'shuttle' 'sl' 'space' 't7' 'team' 'thanks' 'think' 'time'
 'use' 'used' 'using' 'uw' 've' 'w7' 'way' 'win' 'windows' 'wm' 'work'
 'year' 'years']
  (0, 86)       0.20827202576766465
  (0, 52)       0.07120433322618305
  (0, 68)       0.9733274956243896
  (0, 85)       0.06470156469515784
```

1.2 Feature normalization

Python provides some functionality for normalizing and standardizing the data. Be careful though, some operations work only with dense data.

http://scikit-learn.org/stable/modules/preprocessing.html#preprocessing

Use the function **preprocessing.scale** to normalize by removing the mean and dividing by the standard deviation. This is done per **feature**, that is, per column of the dataset.

[61]:
```
from sklearn import preprocessing

X = np.array([[ 1., -1.,  2.],
              [ 2.,  0.,  1.],
              [ 0.,  1., -1.]])
print("column means: ",X.mean(axis = 0))
print("column std: ",X.std(axis = 0))
X_scaled = preprocessing.scale(X)
print("after feature normalization")
print(X_scaled)
print("normalized column means: ",X_scaled.mean(axis=0))
print("normalized column std: ",X_scaled.var(axis = 0))
```

```
column means:  [1.         0.         0.66666667]
column std:  [0.81649658 0.81649658 1.24721913]
after feature normalization
[[ 0.         -1.22474487  1.06904497]
```

```
 [ 1.22474487  0.          0.26726124]
 [-1.22474487  1.22474487 -1.33630621]]
normalized column means:  [0.00000000e+00 0.00000000e+00 1.48029737e-16]
normalized column std:  [1. 1. 1.]
```

[64]:
```
print("row means: ",X.mean(axis = 1))
print("row std: ",X.std(axis = 1))
X_scaled = preprocessing.scale(X, axis = 1)
print("after row normalization")
print(X_scaled)
print("normalized row means: ",X_scaled.mean(axis=1))
print("normalized row std: ",X_scaled.var(axis = 1))
```

```
row means:  [0.66666667 1.         0.        ]
row std:  [1.24721913 0.81649658 0.81649658]
after row normalization
[[ 0.26726124 -1.33630621  1.06904497]
 [ 1.22474487 -1.22474487  0.        ]
 [ 0.          1.22474487 -1.22474487]]
normalized row means:  [1.48029737e-16 0.00000000e+00 0.00000000e+00]
normalized row std:  [1. 1. 1.]
```

Feature normalization will not work with sparse data. In this case, the zeros are treated as values, so the sparse matrix will become non-sparse after normalization.

[62]:
```
import scipy.sparse
cX = scipy.sparse.csc_matrix(X)
cX_scaled = preprocessing.scale(cX)
print(cX_scaled)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-62-961e7864f1cd> in <module>
      1 import scipy.sparse
      2 cX = scipy.sparse.csc_matrix(X)
----> 3 cX_scaled = preprocessing.scale(cX)
      4 print(cX_scaled)

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\preprocessing\_data.py in␣
 ↪scale(X, axis, with_mean, with_std, copy)
    204         if with_mean:
    205             raise ValueError(
--> 206                 "Cannot center sparse matrices: pass `with_mean=False`␣
 ↪instead"
    207                 " See docstring for motivation and alternatives."
    208             )
```

```
ValueError: Cannot center sparse matrices: pass `with_mean=False` instead See␣
↪docstring for motivation and alternatives.
```

The same can be done with the **StandardScaler** from the preprocessing library of sklearn.

The function **fit()** computes the parameters for scaling, and **transform()** applies the scaling

[63]:
```
from sklearn import preprocessing
std_scaler = preprocessing.StandardScaler()
std_scaler.fit(X)
print(std_scaler.mean_)
print(std_scaler.scale_)
X_std = std_scaler.transform(X)
print("scaled data:")
print(X_std)
```

```
[1.         0.         0.66666667]
[0.81649658 0.81649658 1.24721913]
scaled data:
[[ 0.         -1.22474487  1.06904497]
 [ 1.22474487  0.          0.26726124]
 [-1.22474487  1.22474487 -1.33630621]]
```

The advantage is the we can now apply the transform to new data.

For example, we compute the parameters for the training data and we apply the scaling to the test data.

[66]:
```
y = np.array([[2.,3.,1.],
              [1.,2.,1.]])
print(std_scaler.transform(y))
```

```
[[1.22474487 3.67423461 0.26726124]
 [0.         2.44948974 0.26726124]]
```

The **MinMaxScaler** subbtracts from each column the minimum and then divides by the max-min.

[67]:
```
min_max_scaler = preprocessing.MinMaxScaler()
X_minmax = min_max_scaler.fit_transform(X)
print(X_minmax)
print(min_max_scaler.transform(y))
```

```
[[0.5        0.         1.        ]
 [1.         0.5        0.66666667]
 [0.         1.         0.        ]]
[[1.         2.         0.66666667]
 [0.5        1.5        0.66666667]]
```

The **MaxAbsScaler** divides with the maximum absolute value.

The MaxAbsScaler can work with sparse data, since it does not destroy the data sparseness. For the other datasets, removing the mean (or min) can destroy the sparseness of the data.

Sometimes we may choose to normalize only the non-zero values. This should be done manually.

[68]:
```
max_abs_scaler = preprocessing.MaxAbsScaler()
X_maxabs = max_abs_scaler.fit_transform(X)
X_maxabs
```

[68]:
```
array([[ 0.5, -1. ,  1. ],
       [ 1. ,  0. ,  0.5],
       [ 0. ,  1. , -0.5]])
```

[69]:
```
# works with sparse data
cX_scaled = max_abs_scaler.transform(cX)
print(cX_scaled)
```

```
  (0, 0)        0.5
  (1, 0)        1.0
  (0, 1)        -1.0
  (2, 1)        1.0
  (0, 2)        1.0
  (1, 2)        0.5
  (2, 2)        -0.5
```

The **normalize** function normalizes the **rows** so that they become unit vectors in some norm that we specify. It can be applied to sparse matrices without destroying the sparsity.

[70]:
```
#works with sparse data

X_normalized = preprocessing.normalize(X, norm='l2')

X_normalized
```

[70]:
```
array([[ 0.40824829, -0.40824829,  0.81649658],
       [ 0.89442719,  0.        ,  0.4472136 ],
       [ 0.        ,  0.70710678, -0.70710678]])
```

[90]:
```
crX = scipy.sparse.csr_matrix(X)
crX_scaled = preprocessing.normalize(crX,norm='l1')
print(crX_scaled)
```

```
  (0, 0)        0.25
  (0, 1)        -0.25
  (0, 2)        0.5
  (1, 0)        0.6666666666666666
  (1, 2)        0.3333333333333333
  (2, 1)        0.5
  (2, 2)        -0.5
```