Handling Categorical Data

December 16, 2021

[2]:
```
# Handling Categorical Data
#Encoding Nominal categorical features

import numpy as np
from sklearn.preprocessing import LabelBinarizer, MultiLabelBinarizer

feature = np.array([["T"],["Cali"],["T"],["Dela"],["T"]])

one_hot = LabelBinarizer()
print(feature)
print(one_hot.fit_transform(feature))

one_hot.classes_
```

```
[['T']
 ['Cali']
 ['T']
 ['Dela']
 ['T']]
[[0 0 1]
 [1 0 0]
 [0 0 1]
 [0 1 0]
 [0 0 1]]
```

[2]:
```
array(['Cali', 'Dela', 'T'], dtype='<U4')
```

[3]:
```
import pandas as pd

pd.get_dummies(feature[:,0])
```

[3]:

	Cali	Dela	T
0	0	0	1
1	1	0	0
2	0	0	1
3	0	1	0
4	0	0	1

[4]:

```python
# Create Multiclass feature

multiclass_feature =␣
↪[("T","Flor"),("Cali","Alab"),("T","Flor"),("Del","Flor"),("T","Alab")]

one_hot_multiclass = MultiLabelBinarizer()

print(one_hot_multiclass.fit_transform(multiclass_feature))

one_hot_multiclass.classes_
```

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

[4]:

```
array(['Alab', 'Cali', 'Del', 'Flor', 'T'], dtype=object)
```

[5]:

```python
#Encoding Ordinal Categorical Features

dataframe = pd.DataFrame({"Score":["Low","Med","Low","High","Med"]})

scale_mapper = {"Low":1,"Med":2,"High":3}

#Replace feature values with scale
dataframe["Score"].replace(scale_mapper)
```

[5]:

```
0    1
1    2
2    1
3    3
4    2
Name: Score, dtype: int64
```

[6]:

```python
dataframe = pd.DataFrame({"Score": ["Low",
"Low",
"Medium",
"Medium",
"High",
"Barely More Than Medium"]})

scale_mapper = {"Low":1,
"Medium":2,
"Barely More Than Medium": 3,
"High":4}
```

```python
print(dataframe["Score"].replace(scale_mapper))

'''In this example, the distance between Low and Medium is the same as the␣
↪distance
between Medium and Barely More Than Medium , which is almost certainly not accu-
rate. The best approach is to be conscious about the numerical values mapped to
classes:'''

scale_mapper = {"Low":1,
"Medium":2,
"Barely More Than Medium": 2.1,
"High":3}
dataframe["Score"].replace(scale_mapper)
```

```
0    1
1    1
2    2
3    2
4    4
5    3
Name: Score, dtype: int64
```

[6]:

```
0    1.0
1    1.0
2    2.0
3    2.0
4    3.0
5    2.1
Name: Score, dtype: float64
```

[10]:

```python
# Encoding Dictionaries of Features

from sklearn.feature_extraction import DictVectorizer

data_dict = [{"Red":2,"Blue":4},{"Red":4,"Blue":3},{"Red":1,"Yellow":2},{"Red":
↪2,"Yellow":2}]

dictvectorizer = DictVectorizer(sparse = False)

features = dictvectorizer.fit_transform(data_dict)

print(features)
```

```
[[4. 2. 0.]
 [3. 4. 0.]
 [0. 1. 2.]
 [0. 2. 2.]]
```

[12]:
```
# Get feature names
feature_names = dictvectorizer.get_feature_names()

feature_names

pd.DataFrame(features,columns=feature_names)
```

[12]:

	Blue	Red	Yellow
0	4.0	2.0	0.0
1	3.0	4.0	0.0
2	0.0	1.0	2.0
3	0.0	2.0	2.0

[13]:
```
# Create word counts dictionaries for four documents
doc_1_word_count = {"Red": 2, "Blue": 4}
doc_2_word_count = {"Red": 4, "Blue": 3}
doc_3_word_count = {"Red": 1, "Yellow": 2}
doc_4_word_count = {"Red": 2, "Yellow": 2}
# Create list
doc_word_counts = [doc_1_word_count,
doc_2_word_count,
doc_3_word_count,
doc_4_word_count]
# Convert list of word count dictionaries into feature matrix
dictvectorizer.fit_transform(doc_word_counts)
```

[13]:
```
array([[4., 2., 0.],
       [3., 4., 0.],
       [0., 1., 2.],
       [0., 2., 2.]])
```

[18]:
```
# Imputing Missing class values
import numpy as np
from sklearn.neighbors import KNeighborsClassifier

X = np.array([[0,2.1,1.45],[1,1.18,1.33],[0,1.22,1.27],[1,-0.21,-1.19]])
X_with_nan = np.array([[np.nan,0.87,1.31],[np.nan,-0.67,-0.22]])

# Train KNN learner
clf = KNeighborsClassifier(3,weights="distance")
trained_model = clf.fit(X[:,1:],X[:,0])

# Predict missing values
imputed_values = trained_model.predict(X_with_nan[:,1:])

# Join column of predicted class with their other features
X_with_imputed = np.hstack((imputed_values.reshape(-1,1),x_with_nan[:,1:]))
```

```
# Join two feature matrices
np.vstack((X_with_imputed,X))
```

[18]:
```
array([[ 0.  ,  0.87,  1.31],
       [ 1.  , -0.67, -0.22],
       [ 0.  ,  2.1 ,  1.45],
       [ 1.  ,  1.18,  1.33],
       [ 0.  ,  1.22,  1.27],
       [ 1.  , -0.21, -1.19]])
```

[23]:
```
'''An alternative solution is to fill in missing values with the feature's most␣
↪frequent
value:'''
from sklearn.impute import SimpleImputer

X_complete = np.vstack((X_with_nan,X))

imputer = SimpleImputer(strategy="most_frequent")

imputer.fit_transform(X_complete)
```

[23]:
```
array([[ 0.  ,  0.87,  1.31],
       [ 0.  , -0.67, -0.22],
       [ 0.  ,  2.1 ,  1.45],
       [ 1.  ,  1.18,  1.33],
       [ 0.  ,  1.22,  1.27],
       [ 1.  , -0.21, -1.19]])
```

[25]:
```
# Handling Imbalanced Classes
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris

iris = load_iris()

features = iris.data

target = iris.target
#Remove first 40 obs
features = features[40:,:]
target = target[40:]
# Create binary target vector indicating if class 0
target = np.where((target == 0),0,1)
# imbalanced target vector
target
```

[25]: array([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, 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])

[28]:

```
# Weights
weights = {0:9,1:0.1}

# Create random forest classifier with weights
RandomForestClassifier(class_weight = weights)

# Train a random forest with balanced class weights
RandomForestClassifier(class_weight="balanced")
```

[28]: RandomForestClassifier(class_weight='balanced')