Preprocessing Class

January 1, 2020

1 Preprocessing Class

[1]:
```
import numpy as np
```

[2]:
```
from sklearn import preprocessing as pre
from sklearn.impute import SimpleImputer
```

[3]:
```
import pandas as pd
```

1.0.1 Strategy of Replacing Missing Values

[4]:
```
REPLACE_STRATEGY_MEAN = "mean"
REPLACE_STRATEGY_MEDIAN = "median"
REPLACE_STRATEGY_MOST_FREQUENT = "most_frequent"

# Replace missing values with specific constant
# which passed to fill_value parameter
REPLACE_STRATEGY_CONSTANT = "constant"
```

1.0.2 Strategy of Dropping Missing Values

DROP_IF_ANY_NA -> Drop the row/column if any of the values is null. **DROP_IF_ALL_NA** -> Drop the row/column if all the values are missing.

[5]:
```
DROP_IF_ANY_NA = 'any'
DROP_IF_ALL_NA = 'all'
```

[6]:
```
class Preprocessing:
    def __init__(self, data):
        self.data = data
```

2 Feature Scaling

Feature scaling is a method used to normalize the range of independent variables or features of data. Since the range of values of raw data varies widely, in some data mining algorithms, objective functions will not work properly without normalization. For example, many classifiers calculate the distance between two points by the Euclidean distance. If one of the features has a broad range of values, the distance will be governed by this particular feature. Therefore, the range of all features should be normalized so that each feature contributes approximately proportionately to the final distance.

2.1 Min-Max Scaler

Transform features by scaling each feature to a given range. This estimator scales and translates each feature individually such that it is in the given range on the training set, e.g. between zero and one. The transformation is given by: **X_std = (X - X.min(axis=axis)) / (X.max(axis=axis) - X.min(axis=axis)) X_scaled = X_std * (feature_range.max - feature_range.min) + feature_range.min**

[7]:
```
def minMaxScale(self, feature_range=(0, 1), axis=0):
    self.data = pre.minmax_scale(self.data, feature_range=feature_range,␣
 ↪axis=axis)
    return self
Preprocessing.minMaxScale = minMaxScale
```

2.2 Max Scaler

This scaler is calculated by formula **X_scaled = X / X.max(axis=axis)** All entries in result matrix is less than or equal to 1

[8]:
```
def maxScale(self, axis=0):
    self.data = pre.maxabs_scale(self.data, axis=axis)
    return self
Preprocessing.maxScale = maxScale
```

2.3 Min Scaler

This scaler is calculated by formula **X_scaled = X / X.min(axis=axis)** All entries in result matrix is greater than or equal to 1

[9]:
```
def minScale(self, axis=0):
    if (axis != 0):
        for r in range(self.data.shape[0]):
            self.data[r, :] = self.data[r, :] / self.data[r, :].min()
        return self

    for c in range(self.data.shape[1]):
        self.data[:, c] = self.data[:, c] / self.data[:, c].min()
    return self
Preprocessing.minScale = minScale
```

2.4 Standard Scaler

Standardize features by removing the mean and scaling to unit variance The standard score of a sample x is calculated as:

z = (x - u) / s
where **u** is the mean of the training samples or zero if **with_mean=False**, and s is the standard deviation of the training samples or one if **with_std=False**.

[10]:
```
def standardScale(self, with_mean=True, with_std=True):
    stdScaler = pre.StandardScaler(with_mean=with_mean, with_std=with_std)
    self.data = stdScaler.fit(self.data).transform(self.data)
    return self
Preprocessing.standardScale = standardScale
```

3 Encoding

Encoding is the process of converting categorical features into numberical features - **categorical_axis**: index of axis of the categorical feature - **axis**: if 0 -> categorical axis is a column, else categorical axis is row

[11]:
```
def encode(self, categorical_axis, axis=0):
    if axis == 0:
        le = pre.LabelEncoder().fit(self.data[:,categorical_axis])
        self.data[:, categorical_axis] = le.transform(self.data[:,␣
 ↪categorical_axis])
    else:
        le = pre.LabelEncoder().fit(self.data[categorical_axis, :])
        self.data[categorical_axis, :] = le.transform(self.
 ↪data[categorical_axis, :])
    return self
Preprocessing.encode = encode
```

Converts data-type of entries inside data matrix

[12]:
```
def convert(self, dtype):
    self.data = np.asarray(self.data, dtype=dtype)
    return self
Preprocessing.convert = convert
```

3.1 Drop Missings

DropMissings function is used to remove rows and columns with **Null/NaN** values. - **axis**: possible values are {0 or ‘index’, 1 or ‘columns’}, default 0. If 0, drop rows with null values. If 1, drop columns with missing values. - **how**: possible values are {‘any’, ‘all’}, default ‘any’. If ‘any’, drop the row/column if any of the values is null. If ‘all’, drop the row/column if all the values are missing. - **thresh**: an int value to specify the threshold for the drop operation. - **subset**: specifies the rows/columns to look for null values.

[13]:
```
def dropMissings(self, axis=0, how=DROP_IF_ANY_NA, thresh=None, subset=None):
    df = pd.DataFrame(self.data)
    df.dropna(axis=axis, how=how, thresh=thresh, subset=subset, inplace=True)
    self.data = df.to_numpy()
    return self
```

```
Preprocessing.dropMissings = dropMissings
```

3.2 Replace Missings

ReplaceMissings function is used to remove rows with **Null/NaN** values. - **missing_values**: The placeholder for the missing values. All occurrences of missing_values will be imputed. - **strategy**: The imputation strategy. - **fill_value**: When strategy == REPLACE_STRATEGY_CONSTANT, fill_value is used to replace all occurrences of missing_values. If left to the default, fill_value will be 0 when imputing numerical data and "missing_value" for strings or object data types. - **axis**: if axis==0 -> feature values in columns else feature values in rows

[14]:
```
def replaceMissings(self, missing_values=np.nan,␣
↪strategy=REPLACE_STRATEGY_MEAN, fill_value=0, axis=0):
    if (axis != 0):
        self.data = self.data.transpose()
    imp_mean = SimpleImputer(missing_values=missing_values,strategy=strategy,␣
↪fill_value=fill_value)
    self.data = imp_mean.fit_transform(self.data)
    if (axis != 0):
        self.data = self.data.transpose()
    return self
Preprocessing.replaceMissings = replaceMissings
```

3.3 Get Result

Returns the result matrix as numpy array

[15]:
```
def getResult(self):
    return self.data
Preprocessing.getResult = getResult
```

4 Example

[16]:
```
arr = np.array([
    [1, 2, 3, 'C1'],
    [3, None, 5, 'C2'],
    [6, 7, np.nan, 'C3'],
    [6, 7, 5, 'C2'],
    [6, 7, 5, 'C3']
])
```

[17]:
```
Preprocessing(arr).encode(3).convert(np.float64).dropMissings().getResult()
```

[17]:
```
array([[1., 2., 3., 0.],
       [6., 7., 5., 1.],
       [6., 7., 5., 2.]])
```

[18]:
```
arr2 = np.array([
    [1, 2, 3, 'C1'],
    [3, None, 5, 'C2'],
    [6, 7, np.nan, 'C3'],
    [6, 7, 5, 'C2'],
    [6, 7, 5, 'C3']
])
```

[19]:
```
arr2 = Preprocessing(arr2).encode(3).convert(np.float64).
 ↪replaceMissings(axis=1).getResult()
arr2
```

[19]:
```
array([[1., 2., 3., 0.],
       [3., 3., 5., 1.],
       [6., 7., 5., 2.],
       [6., 7., 5., 1.],
       [6., 7., 5., 2.]])
```

[20]:
```
arr2[:, 3] = arr[:, 3] + 1
```

[21]:
```
Preprocessing(arr2).minScale(axis=0).getResult()
```

[21]:
```
array([[1.        , 1.        , 1.        , 1.        ],
       [3.        , 1.5       , 1.66666667, 2.        ],
       [6.        , 3.5       , 1.66666667, 3.        ],
       [6.        , 3.5       , 1.66666667, 2.        ],
       [6.        , 3.5       , 1.66666667, 3.        ]])
```