Preprocessing

This notebook will give you a taste of what `scikit-learn` provides for preprocessing data.

Data used

We will be using the planets data and red wine data:

Data License for Planet Data

Citations for Red Wine Data

P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 47(4):547-553. ISSN: 0167-9236.

Available at:

- @Elsevier
- Pre-press (pdf)
- bib

Dua, D. and Karra Taniskidou, E. (2017). UCI Machine Learning Repository http://archive.ics.uci.edu/ml/index.php. Irvine, CA: University of California, School of Information and Computer Science.

Setup

```
import numpy as np
import pandas as pd

planets = pd.read_csv('data/planets.csv')
red_wine = pd.read_csv('data/winequality-red.csv')
wine = pd.concat([
    pd.read_csv('data/winequality-white.csv',
sep=';').assign(kind='white'),
    red_wine.assign(kind='red')
])
```

Train Test Split

Rather than having to write something like this every time:

```
shuffled = planets.reindex(np.random.permutation(planets.index))
train_end_index = int(np.ceil(shuffled.shape[0] * .75))
training = shuffled.iloc[:train_end_index,]
testing = shuffled.iloc[train_end_index:,]
```

We can use scikit-learn's `train_test_split()` function to get our training and testing sets. (We will discuss the validation set in chapter 10.)

```
from sklearn.model_selection import train_test_split

X = planets[['eccentricity', 'semimajoraxis', 'mass']]
y = planets.period

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=0
)
```

The original data had this shape:

```
X.shape, y.shape
```

```
((4094, 3), (4094,))
```

Our training data has this shape:

```
X_train.shape, y_train.shape
```

```
((3070, 3), (3070,))
```

Our testing data has this shape:

```
X_test.shape, y_test.shape
```

```
((1024, 3), (1024,))
```

Let's look at the first 5 entries:

```
X_train.head()
```

```
      eccentricity  semimajoraxis  mass
1390           NaN            NaN   NaN
2837           NaN            NaN   NaN
3619           NaN         0.0701   NaN
1867           NaN            NaN   NaN
1869           NaN            NaN   NaN
```

Our y data will be for the same rows as our X:

```
y_train.head()
```

```
1390     1.434742
2837    51.079263
3619     7.171000
1867    51.111024
1869    62.869161
Name: period, dtype: float64
```

Scaling data

Standardizing with `StandardScaler`

```
from sklearn.preprocessing import StandardScaler

standardized = StandardScaler().fit_transform(X_train)

# examine some of the non-NaN values
standardized[~np.isnan(standardized)][:30]
```

```
array([-5.43618156e-02,  1.43278593e+00,  1.95196592e+00,
4.51498477e-03,
       -1.96265630e-01,  7.79591646e-02, -4.74717586e-02, -
3.12856028e-01,
       -4.18101448e-01, -5.47587283e-02, -2.46399501e-01,
1.65946487e+00,
       -8.59044215e-01, -5.47511116e-02, -4.04573808e-01,
1.88194856e-01,
       -5.41905011e-02, -4.75421907e-01,  1.33077010e-01, -
3.01831439e-02,
       -1.08822831e-01,  1.62409605e-01,  1.21526007e+01,
1.73149454e+00,
       -2.25664815e-02,  9.91013258e-01, -7.48808523e-01, -
4.99260165e-02,
       -8.59044215e-01, -5.49264158e-02])
```

Normalizing with `MinMaxScaler`

```
from sklearn.preprocessing import MinMaxScaler

normalized = MinMaxScaler().fit_transform(X_train)

# examine some of the non-NaN values
normalized[~np.isnan(normalized)][:30]
```

```
array([2.28055906e-05, 1.24474091e-01, 5.33472803e-01, 1.71374569e-03,
       1.83543340e-02, 1.77824268e-01, 2.20687839e-04, 1.07593965e-02,
       8.36820084e-02, 1.14062675e-05, 1.50885109e-02, 1.39240422e-01,
       0.00000000e+00, 1.16250178e-05, 4.78471239e-03, 1.98744770e-01,
       2.77257370e-05, 1.69522066e-04, 1.88284519e-01, 7.17216379e-04,
       2.40505371e-02, 6.24847487e-03, 8.22784793e-01, 4.91631799e-01,
       9.35966714e-04, 9.56961137e-02, 2.09205021e-02, 1.50201619e-04,
       0.00000000e+00, 6.59028789e-06])
```

Using the Median and IQR with `RobustScaler`

```
from sklearn.preprocessing import RobustScaler

robust_scaled = RobustScaler().fit_transform(X_train)

# examine some of the non-NaN values
robust_scaled[~np.isnan(robust_scaled)][:30]
```

```
array([-5.64660112e-02,  3.90058747e+00,  1.87999087e+00,
4.32924097e+00,
        2.79222462e-01,  3.28542094e-01,  4.56771306e-01,
2.00431965e-02,
       -8.21355236e-02, -8.60318668e-02,  1.67775378e-01,
4.40449244e+00,
       -4.47182295e-01, -8.54645050e-02, -1.83844492e-01,
4.19803787e-01,
       -4.37048728e-02, -3.41339093e-01,  3.74172941e-01,
1.74459261e+00,
        4.73606911e-01,  1.60907419e+01,  2.77306263e+01,
1.69746749e+00,
        2.31195445e+00,  2.91853132e+00, -3.55920602e-01,
2.73954715e-01,
       -4.47182295e-01, -9.85228329e-02])
```

Encoding

Binary encoding with `np.where()`

```
np.where(wine.kind == 'red', 1, 0)
```

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

We can also use the `LabelBinarizer` class from scikit-learn. By calling the `inverse_transform()` method, we see the labels assigned to each value:

```
from sklearn.preprocessing import LabelBinarizer

binary_labels = LabelBinarizer().fit(wine.kind)
binary_labels.inverse_transform(np.array([0, 1]))
```

```
array(['red', 'white'], dtype='<U5')
```

We can use the `Binarizer` class for binary encoding of values based on a threshold. Values less than or equal to `threshold` will be 0; values greater than `threshold` will be 1:

```
from sklearn.preprocessing import Binarizer

pd.Series(

Binarizer(threshold=6).fit_transform(red_wine.quality.values.reshape(-
1, 1)).flatten()
).value_counts()
```

```
0    1382
1     217
dtype: int64
```

Ordinal Encoding with `LabelEncoder`

```
from sklearn.preprocessing import LabelEncoder

pd.Series(LabelEncoder().fit_transform(pd.cut(
    red_wine.quality,
    bins=[-1, 3, 6, 10],
    labels=['0-3 (low)', '4-6 (med)', '7-10 (high)']
))).value_counts()
```

```
1    1372
2     217
0      10
dtype: int64
```

One-hot encoding

In some cases, label encoding may yield some associations that aren't something we want the model to be trained on. A safer strategy is to use one-hot encoding.

Our planets data has a `list` column that we can one-hot encode:

```
planets.list.value_counts()
```

```
Confirmed planets                    3972
Controversial                          97
Retracted planet candidate             11
Solar System                            9
Kepler Objects of Interest              4
Planets in binary systems, S-type       1
Name: list, dtype: int64
```

We can use `pd.get_dummies()` to one-hot encode this information:

```
pd.get_dummies(planets.list).head()
```

```
   Confirmed planets  Controversial  Kepler Objects of Interest  \
0                  1              0                           0
1                  1              0                           0
2                  1              0                           0
3                  1              0                           0
4                  0              1                           0

   Planets in binary systems, S-type  Retracted planet candidate
Solar System
0                                  0                           0
0
1                                  0                           0
0
2                                  0                           0
0
3                                  0                           0
0
4                                  0                           0
0
```

This gives us a redundant column. Note that we only need one less column than the number of planet lists. Pandas makes it easy to remove one of the columns to address multicollinearity:

```
pd.get_dummies(planets.list, drop_first=True).head()
```

```
   Controversial  Kepler Objects of Interest  \
0              0                           0
1              0                           0
2              0                           0
```

```
3              0                            0
4              1                            0

   Planets in binary systems, S-type  Retracted planet candidate
Solar System
0                                  0                            0
0
1                                  0                            0
0
2                                  0                            0
0
3                                  0                            0
0
4                                  0                            0
0
```

We can also use the `LabelBinarizer` class:

```
from sklearn.preprocessing import LabelBinarizer

LabelBinarizer().fit_transform(planets.list)
```

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

Imputing

The planets data has some missing values. We can use imputing strategies to avoid having to drop them from our model.

```
planets[['semimajoraxis', 'mass', 'eccentricity']].tail()
```

```
      semimajoraxis    mass  eccentricity
4089        0.08150  1.9000         0.000
4090        0.04421  0.7090         0.038
4091            NaN  0.3334         0.310
4092            NaN  0.4000         0.270
4093            NaN  0.4200         0.160
```

`SimpleImputer`

We can fill with the `mean`, `median`, `most_frequent` (mode), or a constant value by specifiying the `strategy`. The default is the mean:

```
from sklearn.impute import SimpleImputer

SimpleImputer().fit_transform(
    planets[['semimajoraxis', 'mass', 'eccentricity']]
)
```

```
array([[ 1.29      , 19.4       ,  0.231     ],
       [ 1.54      , 11.2       ,  0.08      ],
       [ 0.83      ,  4.8       ,  0.        ],
       ...,
       [ 5.83796389,  0.3334    ,  0.31      ],
       [ 5.83796389,  0.4       ,  0.27      ],
       [ 5.83796389,  0.42      ,  0.16      ]])
```

Changing to the median is just a matter of passing that as the `strategy`:

```
from sklearn.impute import SimpleImputer

SimpleImputer(strategy='median').fit_transform(
    planets[['semimajoraxis', 'mass', 'eccentricity']]
)
```

```
array([[ 1.29  , 19.4   ,  0.231 ],
       [ 1.54  , 11.2   ,  0.08  ],
       [ 0.83  ,  4.8   ,  0.    ],
       ...,
       [ 0.1409,  0.3334,  0.31  ],
       [ 0.1409,  0.4   ,  0.27  ],
       [ 0.1409,  0.42  ,  0.16  ]])
```

`KNNImputer`

Since this data isn't something that is easily measured, assuming that the planets we don't have the data for are similar to the rest is dangerous. It could be that the ones that have missing data have something in common. Replacing missing values for the semi-major axis with the average of the ones we know is hardly a good strategy. Instead, we could try to use the mass and eccentricity columns to find similar planets and use their semi-major axes to impute the missing data. This can be done with the `KNNImputer` class. Notice the first column in the bottom 3 rows (the imputed semi-major axis is drastically different from what we got using the overall mean:

```
from sklearn.impute import KNNImputer

KNNImputer().fit_transform(
    planets[['semimajoraxis', 'mass', 'eccentricity']]
)
```

```
array([[ 1.29    , 19.4     ,  0.231   ],
       [ 1.54    , 11.2     ,  0.08    ],
       [ 0.83    ,  4.8     ,  0.      ],
       ...,
       [ 0.404726,  0.3334   ,  0.31    ],
       [ 0.85486 ,  0.4      ,  0.27    ],
       [ 0.15324 ,  0.42     ,  0.16    ]])
```

`MissingIndicator`

In some cases, we don't want to fill in a value, but rather use the fact that the data is missing as a feature in our model:

```
from sklearn.impute import MissingIndicator

MissingIndicator().fit_transform(
    planets[['semimajoraxis', 'mass', 'eccentricity']]
)

array([[False, False, False],
       [False, False, False],
       [False, False, False],
       ...,
       [ True, False, False],
       [ True, False, False],
       [ True, False, False]])
```

Additional Transformers

`FunctionTransformer`

With the `FunctionTransformer` class, we can use any function on the data. By passing `validate=True`, we will convert the result to two-dimensional NumPy array and raise an error if there is an issue:

```
from sklearn.preprocessing import FunctionTransformer

FunctionTransformer(
    np.abs, validate=True
).fit_transform(X_train.dropna())

array([[0.51   , 4.94   , 1.45   ],
       [0.17   , 0.64   , 0.85   ],
       [0.08   , 0.03727, 1.192  ],
       ...,
       [0.295  , 4.46   , 1.8    ],
       [0.34   , 0.0652 , 0.0087 ],
       [0.3    , 1.26   , 0.5    ]])
```

`ColumnTransformer`

Sometimes we don't want to perform the same transformation on all of our features, the `ColumnTransformer` class lets us specify which tranformations to use on each column. We pass a list of tuples in the form `(name, transformer object, columns to apply to)`:

```
from sklearn.compose import ColumnTransformer
from sklearn.impute import KNNImputer
from sklearn.preprocessing import MinMaxScaler, StandardScaler

ColumnTransformer([
    ('impute', KNNImputer(), [0]),
    ('standard_scale', StandardScaler(), [1]),
    ('min_max', MinMaxScaler(), [2])
]).fit_transform(X_train)[10:15]
```

```
array([[ 0.17      , -0.04747176,  0.0107594 ],
       [ 0.08      , -0.05475873,  0.01508851],
       [ 0.15585591,         nan,  0.13924042],
       [ 0.15585591,         nan,         nan],
       [ 0.        , -0.05475111,  0.00478471]])
```

We can also use the `make_column_transformer()` function, which will name the transformers for us:

```
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler

categorical = [
    col for col in planets.columns
    if col in [
        'list', 'name', 'description',
        'discoverymethod', 'lastupdate'
    ]
]
numeric = [col for col in planets.columns if col not in categorical]

make_column_transformer(
    (StandardScaler(), numeric),
    (OneHotEncoder(sparse=False), categorical)
).fit_transform(planets.dropna())
```

```
array([[ 3.09267587, -0.2351423 , -0.40487424, ...,  0.        ,
         0.        ,  0.        ],
       [ 1.432445  , -0.24215395, -0.28360905, ...,  0.        ,
         0.        ,  0.        ],
       [ 0.13665505, -0.24208849, -0.62800218, ...,  0.        ,
         0.        ,  0.        ],
       ...,
       [-0.83289954, -0.76197788, -0.84918988, ...,  0.        ,
         1.        ,  0.        ],
       [ 0.25813535,  0.38683239, -0.92873984, ...,  0.        ,
         0.        ,  0.        ],
       [-0.26827931, -0.21657671, -0.70076129, ...,  0.        ,
         0.        ,  1.        ]])
```

`Pipeline`

Using pipelines ensures the whole model training and testing process is consistent. To make a pipeline, we pass in a list of steps as tuples of `(name, object)`:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression

Pipeline([('scale', StandardScaler()), ('lr', LinearRegression())])

Pipeline(steps=[('scale', StandardScaler()), ('lr',
LinearRegression())])
```

We aren't limited to using pipelines with models — they can be used inside other `sklearn` objects. This makes it possible for us to first use k-NN imputing on the semi-major axis data and then standard scale the result:

```
from sklearn.compose import ColumnTransformer
from sklearn.impute import KNNImputer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler, StandardScaler

ColumnTransformer([
    ('impute', Pipeline([
        ('impute', KNNImputer()), ('scale', StandardScaler())
    ]), [0]),
    ('standard_scale', StandardScaler(), [1]),
    ('min_max', MinMaxScaler(), [2])
]).fit_transform(X_train)[10:15]

array([[ 0.13531604, -0.04747176,  0.0107594 ],
       [-0.7257111 , -0.05475873,  0.01508851],
       [ 0.        ,         nan,  0.13924042],
       [ 0.        ,         nan,         nan],
       [-1.49106856, -0.05475111,  0.00478471]])
```

We can then include this as part of a pipeline, which gives us tremendous flexibility in how we build our models:

```
Pipeline([
    (
        'preprocessing',
        ColumnTransformer([
            ('impute', Pipeline([
                ('impute', KNNImputer()), ('scale', StandardScaler())
            ]), [0]),
            ('standard_scale', StandardScaler(), [1]),
            ('min_max', MinMaxScaler(), [2])
        ])
    ),
    ('model', LinearRegression())
])
```

```
Pipeline(steps=[('preprocessing',
                 ColumnTransformer(transformers=[('impute',

Pipeline(steps=[('impute',

KNNImputer()),

('scale',

StandardScaler())]),
                                                  [0]),
                                                 ('standard_scale',
                                                  StandardScaler(),
[1]),
                                                 ('min_max',
MinMaxScaler(),
                                                  [2])])),
                ('model', LinearRegression())])
```

We can also use the `make_pipeline()` function to make the pipeline without naming the steps ourselves:

```
from sklearn.pipeline import make_pipeline

make_pipeline(StandardScaler(), LinearRegression())
```

```
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('linearregression', LinearRegression())])
```