Premodel Workflow 22

0.0.1 Imputing missing values through various strategies

Data imputation is critical in practice, and thankfully there are many ways to deal with it. In this recipe, we'll look at a few of the strategies. However, be aware that there might be other approaches that fit your situation better. This means scikit-learn comes with the ability to perform fairly common imputations; it will simply apply some transformations to the existing data and fill the NAs. However, if the dataset is missing data, and there's a known reason for this missing data—for example, response times for a server that times out after 100ms—it might be better to take a statistical approach through other packages such as the Bayesian treatment via PyMC, the Hazard Models via Lifelines, or something home-grown.

0.0.2 Getting ready

The first thing to do to learn how to input missing values is to create missing values. NumPy's masking will make this extremely simple:

```
>>> from sklearn import datasets
>>> import numpy as np
>>> iris = datasets.load_iris()
>>> iris_X = iris.data
>>> masking_array = np.random.binomial(1, .25,iris_X.sha
>>> iris_X[masking_array] = np.nan
```

To unravel this a bit, in case NumPy isn't too familiar, it's possible to index arrays with other arrays in NumPy. So, to create the random missing data, a random Boolean array is created, which is of the same shape as the iris dataset. Then, it's possible to make an assignment via the masked array. It's important to note that because a random array is used, it is likely your `masking_array` will be different from what's used here. To make sure this works, use the following command (since we're using a random mask, it might not match directly):

```
>>> masking_array[:5]
array([[False, False, False, False],
[False, False, False, False],
[False, False, False, False],
[ True, False, False, False],
[False, False, False, False]], dtype=bool)
>>> iris_X [:5]
array([[ 5.1, 3.5, 1.4, 0.2],
[ 4.9, 3. , 1.4, 0.2],

[ 4.7, 3.2, 1.3, 0.2],
[ nan, 3.1, 1.5, 0.2],
[ 5. , 3.6, 1.4, 0.2]])
```

How to do it... A theme prevalent throughout this book (due to the theme throughout scikit-learn) is reusable classes that fit and transform datasets and that can subsequently be used to transform unseen datasets. This is illustrated as follows:

```
>>> from sklearn import preprocessing
>>> impute = preprocessing.Imputer()
>>> iris_X_prime = impute.fit_transform(iris_X)
>>> iris_X_prime[:5]
array([[ 5.1 , 3.5 , 1.4 , 0.2 ],
[ 4.9 , 3. , 1.4 , 0.2 ],
[ 4.7 , 3.2 , 1.3 , 0.2 ],
[ 5.87923077, 3.1 , 1.5 , 0.2 ],
[ 5. , 3.6 , 1.4 , 0.2 ]])
```

Notice the difference in the position [3, 0]:

```
>>> iris_X_prime[3, 0]
5.87923077
>>> iris_X[3, 0]
```

```
nan
```

How it works... The imputation works by employing different strategies. The default is mean, but in total there are:

1. `mean` (default)
2. `median`
3. `most_frequent` (the mode)

scikit-learn will use the selected strategy to calculate the value for each non-missing value in the dataset. It will then simply fill the missing values. For example, to redo the iris example with the median strategy, simply reinitialize impute with the new strategy:

```
>>> impute = preprocessing.Imputer(strategy='median')
>>> iris_X_prime = impute.fit_transform(iris_X)
>>> iris_X_prime[:5]
>>> impute = preprocessing.Imputer(strategy='median')
>>> iris_X_prime = impute.fit_transform(iris_X)
>>> iris_X_prime[:5]
array([[ 5.1, 3.5, 1.4, 0.2],
[ 4.9, 3. , 1.4, 0.2],
[ 4.7, 3.2, 1.3, 0.2],
[ 5.8, 3.1, 1.5, 0.2],
[ 5. , 3.6, 1.4, 0.2]])
```

If the data is missing values, it might be inherently dirty in other places. For instance, in the example in the preceding How to do it... section, np.nan (the default missing value) was used as the missing value, but missing values can be represented in many ways. Consider a situation where missing values are -1. In addition to the strategy to compute the missing value, it's also possible to specify the missing value for the imputer. The default is Nan, which will handle np.nan values. To see an example of this, modify `iris_X` to have -1 as the missing value. It sounds crazy, but since the iris dataset contains measurements that are always possible, many people will fill the missing values with -1 to signify they're not there:

```
>>> iris_X[np.isnan(iris_X)] = -1
>>> iris_X[:5]
array([[ 5.1, 3.5, 1.4, 0.2],
[ 4.9, 3. , 1.4, 0.2],
[ 4.7, 3.2, 1.3, 0.2],
[-1. , 3.1, 1.5, 0.2],
[ 5. , 3.6, 1.4, 0.2]])
```

Filling these in is as simple as the following:

```
>>> impute = preprocessing.Imputer(missing_values=-1)
>>> iris_X_prime = impute.fit_transform(iris_X)
>>> iris_X_prime[:5]
array([[ 5.1 , 3.5 , 1.4 , 0.2 ],
[ 4.9 , 3. , 1.4 , 0.2 ],
[ 4.7 , 3.2 , 1.3 , 0.2 ],
[ 5.87923077, 3.1 , 1.5 , 0.2 ],
[ 5. , 3.6 , 1.4 , 0.2 ]])
```

There's more... pandas also provides a functionality to fill missing data. It actually might be a bit more flexible, but it is less reusable:

```
>>> import pandas as pd
>>> iris_X[masking_array] = np.nan
>>> iris_df = pd.DataFrame(iris_X, columns=iris.feature_
>>> iris_df.fillna(iris_df.mean())['sepal length (cm)'].
0 5.100000
1 4.900000
2 4.700000
3 5.879231
4 5.000000

Name: sepal length (cm), dtype: float64
```

To mention its flexibility, fillna can be passed any sort of statistic, that is, the strategy is more arbitrarily defined:

```
>>> iris_df.fillna(iris_df.max())['sepal length (cm)'].h
0 5.1
1 4.9
2 4.7
3 7.9
4 5.0
Name: sepal length (cm), dtype: float64
```