Scaling data to the standard normal

- A preprocessing step that is almost recommended is to scale columns to the standard normal.
- The standard normal is probably the most important distribution of all statistics.
- If you've ever been introduced to statistics, you must have almost certainly seen z-scores.
- In truth, that's all this recipe is about—transforming our features from their endowed distribution into z-scores.

Getting ready

- The act of scaling data is extremely useful. There are a lot of machine learning algorithms, which perform differently (and incorrectly) in the event the features exist at different scales.
- For example, SVMs perform poorly if the data isn't scaled because it uses a distance function in its optimization, which is biased if one feature varies from 0 to 10,000 and the other varies from 0 to 1.

The **preprocessing** module contains several useful functions to scale features. Using the boston dataset, run the following commands:

```
>>> from sklearn import preprocessing
>>> import numpy as np # we'll need it later

>>> X[:, :3].mean(axis=0) #mean of the first 3 features
array([ 3.59376071, 11.36363636, 11.13677866])
>>> X[:, :3].std(axis=0)
array([ 8.58828355, 23.29939569, 6.85357058])
```

There's actually a lot to learn from this initially. Firstly, the first feature has the smallest mean but varies even more than the third feature. The second feature has the largest mean and standard deviation—it takes the widest spread of values:

```
>>> X_2 = preprocessing.scale(X[:, :3])
>>> X_2.mean(axis=0)
array([ 6.34099712e-17, -6.34319123e-16, -2.68291099e-15])
>>> X_2.std(axis=0)
array([ 1., 1., 1.])
```

Implementation

The center and scaling function is extremely simple. It merely subtracts the mean and divides by the standard deviation:

$$x_{scaled} = \frac{x - \bar{x}}{s}$$

It's also useful for the center and scaling class to persist across individual scaling:

```
>>> my_scaler = preprocessing.StandardScaler()
>>> my_scaler.fit(X[:, :3])
>>> my_scaler.transform(X[:, :3]).mean(axis=0)
array([ 6.34099712e-17, -6.34319123e-16, -2.68291099e-15])
```

`MinMaxScaler`

Scaling features to mean 0, and standard deviation 1 isn't the only useful type of scaling.

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Preprocessing also contains a `MinMaxScaler` class, which will scale the data within a certain range:

```
>>> my_minmax_scaler = preprocessing.MinMaxScaler()
>>> my_minmax_scaler.fit(X[:, :3])
>>> my_minmax_scaler.transform(X[:, :3]).max(axis=0)
array([ 1., 1., 1.])
```

It's very simple to change the minimum and maximum values of the `MinMaxScaler` class from its default of 0 and 1, respectively:

```
>>> Z_scaler = preprocessing.MinMaxScaler(feature_range=(-3,
3))
```

Normalization

(N.B. Not related to Normal Distribution)
Furthermore, another option is ***normalization***. This will scale each sample to have a length of 1. This is different from the other types of scaling done previously, where the features were scaled.

Normalization is illustrated in the following command:

```
>>> normalized_X = preprocessing.normalize(X[:, :3])
```

- If it's not apparent why this is useful, consider the Euclidian distance (a measure of similarity) between three of the samples, where one sample has the values (1, 1, 0), another has (3, 3, 0), and the final has (1, -1, 0).
- The distance between the 1st and 3rd vector is less than the distance between the 1st and 2nd though the 1st and 3rd are orthogonal, whereas the 1st and 2nd only differ by a scalar factor of 3.
- Since distances are often used as measures of similarity, not normalizing the data first will be misleading

Creating binary features through thresholding

- We previously looked at transforming our data into the standard normal distribution.
- Now, we'll talk about another transformation that is quite different.
- Often, in what is ostensibly continuous data, there are discontinuities that can be determined via binary features.

Getting ready

Creating binary features and outcomes is a very useful method, but it should be used with caution. Let's use the boston dataset to learn how to create turn values in binary outcomes. First, load the boston dataset:

```
>>> from sklearn import datasets
>>> boston = datasets.load_boston()
>>> import numpy as np
```

Implementation

Similar to scaling, there are two ways to binarize features in scikit-learn:

1. `preprocessing.binarize`
2. `preprocessing.Binarizer`

Boston Dataset

The boston dataset's target variable is the median value of houses in thousands. This dataset is good to test regression and other continuous predictors, but consider a situation where we want to simply predict if a house's value is more than the overall mean. To do this, we will want to create a threshold value of the mean. If the value is greater than the mean, produce a 1; if it is less, produce a 0:

```
>>> from sklearn import preprocessing
>>> new_target = preprocessing.binarize(boston.target,
threshold=boston.target.mean())
```

```
>>> new_target[:5]
array([ 1., 0., 1., 1., 1.])
```

This was easy, but let's check to make sure it worked correctly:

```
>>> (boston.target[:5] > boston.target.mean()).astype(int)
array([1, 0, 1, 1, 1])
```

```
>>> bin = preprocessing.Binarizer(boston.target.mean())
>>> new_target = bin.fit_transform(boston.target)
>>> new_target[:5]
array([ 1., 0., 1., 1., 1.])
```

Sparse matrices

Sparse matrices are special in that zeros aren't stored; this is done in an effort to save space in memory. This creates an issue for the binarizer, so to combat it, a special condition for the binarizer for sparse matrices is that the threshold cannot be less than zero:

```
>>> from scipy.sparse import coo
>>> spar = coo.coo_matrix(np.random.binomial(1, .25, 100))
>>> preprocessing.binarize(spar, threshold=-1)

ValueError: Cannot binarize a sparse matrix with threshold < 0
```

Working with categorical variables

- Categorical variables are a problem. On one hand they provide valuable information; on the other hand, it's probably text—either the actual text or integers corresponding to the text—like an index in a lookup table.
- So, we clearly need to represent our text as integers for the model's sake, but we can't just use the id field or naively represent them.
- This is because we need to avoid a similar problem to the
- Creating binary features through thresholding recipe. If we treat data that is continuous, it must be interpreted as continuous.

Getting ready

- The boston dataset won't be useful for this section. While it's useful for feature binarization, it won't suffice for creating features from categorical variables.
- For this, the iris dataset will suffice. For this to work, the problem needs to be turned on its head.
- Imagine a problem where the goal is to predict the sepal width; therefore, the species of the flower will probably be useful as a feature.

Let's get the data sorted first:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> X = iris.data
>>> y = iris.target
```

Now, with X and Y being as they normally will be, we'll operate on the data as one:

```
>>> import numpy as np
>>> d = np.column_stack((X, y))
```

Implementation

Convert the text columns to three features:

```
>>> from sklearn import preprocessing
>>> text_encoder = preprocessing.OneHotEncoder()
>>> text_encoder.fit_transform(d[:, -1:]).toarray()[:5]
array([[ 1., 0., 0.],
[ 1., 0., 0.],
[ 1., 0., 0.],
[ 1., 0., 0.]])
```

Implementation

- The encoder creates additional features for each categorical variable, and the value returned is a sparse matrix.
- The result is a sparse matrix by definition; each row of the new features has 0 everywhere, except for the column whose value is associated with the feature's category.
- Therefore, it makes sense to store this data in a sparse matrix. `text_encoder` is now a standard scikit-learn model, which means that it can be used again:

```
>>> text_encoder.transform(np.ones((3, 1))).toarray()
array([[ 0., 1., 0.],
[ 0., 1., 0.],
[ 0., 1., 0.]])
```

DictVectorizer

Another option is to use DictVectorizer. This can be used to directly convert strings to features:

```
>>> from sklearn.feature_extraction import DictVectorizer
>>> dv = DictVectorizer()
>>> my_dict = [{'species': iris.target_names[i]} for i in y]
>>> dv.fit_transform(my_dict).toarray()[:5]
array([[ 1., 0., 0.],
[ 1., 0., 0.],
[ 1., 0., 0.],
[ 1., 0., 0.]])
```

Dictionaries can be viewed as a sparse matrix. They only contain entries for the nonzero values.

Binarizing label features

Now we'll look at working with categorical variables in a different way. In the event that only one or two categories of the feature are important, it might be wise to avoid the extra dimensionality, which might be created if there are several categories.

Getting ready

- There's another way to work with categorical variables.
- Instead of dealing with the categorical variables using `OneHotEncoder`, we can use `LabelBinarizer`.
- This is a combination of thresholding and working with categorical variables.

To show how this works, load the iris dataset:

```
>>> from sklearn import datasets as d
>>> iris = d.load_iris()
>>> target = iris.target
```

Implementation

Import the `LabelBinarizer()` method and create an object. Then transform the target outcomes to the new feature space.

```
>>> from sklearn.preprocessing import LabelBinarizer
>>> label_binarizer = LabelBinarizer()

>>> new_target = label_binarizer.fit_transform(target)
```

Let's look at `new_target` and the `label_binarizer` object to get a feel of what happened:

21

```
>>> new_target.shape
(150, 3)
>>> new_target[:5]
array([[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0]])
>>>
>>> new_target[-5:]
array([[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1]])
>>>
>>> label_binarizer.classes_
array([0, 1, 2])
```

Theoretical Matters

The iris target has a cardinality of 3, that is, it has three unique values. When `LabelBinarizer` converts the vector N x 1 into the vector N x C, where C is the cardinality of the N x 1 dataset, it is important to note that once the object has been fit, introducing unseen values in the transformation will throw an error:

```
>>> label_binarizer.transform([4])
[...]
ValueError: classes [0 1 2] mismatch with the labels [4] found in th
data
```

Other Remarks

- Zero and one do not have to represent the positive and negative instances of the target value.
- For example, if we want positive values to be represented by 1,000, and negative values to be represented by -1,000, we'd simply make the designation when we create `label_binarizer`:

```
>>> label_binarizer = LabelBinarizer(neg_label=-1000,
pos_label=1000)
>>> label_binarizer.fit_transform(target)[:5]
array([[ 1000, -1000, -1000],
[ 1000, -1000, -1000],
[ 1000, -1000, -1000],
[ 1000, -1000, -1000],
[ 1000, -1000, -1000]])
```

The only restriction on the positive and negative values is that they must be integers.