Using k-NN for regression

Regression is covered elsewhere in the book, but we might also want to run a regression on "pockets" of the feature space. We can think that our dataset is subject to several data processes. If this is true, only training on similar data points is a good idea.

Getting ready

- Regression can be used in the context of clustering. Regression is obviously a supervised technique, so we'll use k-Nearest Neighbors (k-NN) clustering rather than KMeans.
- For the k-NN regression, we'll use the K closest points in the feature space to build the regression rather than using the entire space as in regular regression.

Example

In this exercise, we'll use the iris dataset. If we want to predict something such as the petal width for each flower, clustering by iris species can potentially give us better results. The k-NN regression won't cluster by the species, but we'll work under the assumption that the Xs will be close for the same species, or in this case, the petal length. We'll use the iris dataset for this recipe:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> iris.feature_names
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
'petal width (cm)']
```

We'll try to predict the petal length based on the sepal length and

width. We'll also fit a regular linear regression to see how well the k-NN regression does in comparison:

```
>>> from sklearn.linear_model import LinearRegression
>>> lr = LinearRegression()
>>> lr.fit(X, y)
>>> print "The MSE is: {:.2}".format(np.power(y - lr.predict(X),
2).mean())
The MSE is: 0.15
```

Now, for the k-NN regression, use the following code:

```
>>> from sklearn.neighbors import KNeighborsRegressor
>>> knnr = KNeighborsRegressor(n_neighbors=10)
>>> knnr.fit(X, y)
>>> print "The MSE is: {:.2}".format(np.power(y - knnr.predict(X),
2).mean())
The MSE is: 0.069
```

Let's look at what the k-NN regression does when we tell it to use the closest 10 points for regression:

```
>>> f, ax = plt.subplots(nrows=2, figsize=(7, 10))
>>> ax[0].set_title("Predictions")
>>> ax[0].scatter(X[:, 0], X[:, 1], s=lr.predict(X)*80, label='LR
Predictions', color='c', edgecolors='black')
>>> ax[1].scatter(X[:, 0], X[:, 1], s=knnr.predict(X)*80, label='k-N
Predictions', color='m', edgecolors='black')
>>> ax[0].legend()
```

```
>>> ax[1].legend()
```

The following is the output: It might be completely clear that the predictions are close for the most part, but let's look at the predictions for the Setosa species as compared to the actuals:

```
>>> setosa_idx = np.where(iris.target_names=='setosa')
>>> setosa_mask = iris.target == setosa_idx[0]
>>> y[setosa_mask][:5]
    array([ 0.2, 0.2, 0.2, 0.2, 0.2])
>>>
>>> knnr.predict(X)[setosa_mask][:5]
    array([ 0.28, 0.17, 0.21, 0.2 , 0.31])
>>>
>>> lr.predict(X)[setosa_mask][:5]
    array([ 0.44636645, 0.53893889, 0.29846368, 0.27338255, 0.32612
```

Looking at the plots again, the Setosa species (upper-left cluster) is largely overestimated by linear regression, and k-NN is fairly close to the actual values.

k-NN regression

The k-NN regression is very simply calculated taking the average of the k closest point to the point being tested. Let's manually predict a single point. Then we need to get the 10 closest points to our `example_point`:

```
>>> example_point = X[0]
```

```
>>> from sklearn.metrics import pairwise
>>> distances_to_example = pairwise.pairwise_distances(X)[0]
>>>
>>> ten_closest_points = X[np.argsort(distances_to_example)][:10]
>>> ten_closest_y = y[np.argsort(distances_to_example)][:10]
>>> ten_closest_y.mean()
0.28000
```

We can see that this is very close to what was expected.