Using Ridge Regression

In this section, we'll learn about ridge regression, and how to use it to overcome linear regression's shortfalls.

Ridge Regression is different from classical OLS linear regression; by introducing ***regularization parameter*** to "*shrink*" the coefficients.

This is useful when the dataset has collinear factors.

What is Multicollinearity?
multicollinearity (also collinearity) is a phenomenon in which two or more predictor variables in a multiple regression model are highly correlated, meaning that one can be linearly predicted from the others with a substantial degree of accuracy
Severe multicollinearity is a problem because it can increase the variance of the coefficient estimates and make the estimates very sensitive to minor changes in the model. The result is that the coefficient estimates are unstable and difficult to interpret.

Multicollinearity saps the statistical power of the analysis, can cause the coefficients to switch signs, and makes it more difficult to specify the correct model.

Getting ready

- Let's load a dataset that has a low effective rank and compare ridge regression with linear regression by way of the coefficients.
- If you're not familiar with ***rank***, it's the smaller of the linearly independent columns and the linearly independent rows.
- One of the assumptions of OLS linear regression is that the data matrix is of "full rank".

Implementation

First, use `make_regression` to create a simple dataset with three predictors, but an effective rank of 2. Effective rank means that while technically the matrix is of full rank, many of the columns have a high degree of colinearity:

```
>>> from sklearn.datasets import make_regression
>>> reg_data, reg_target = make_regression(
n_samples=2000,
n_features=3, effective_rank=2, noise=10)
```

First, let's take a look at regular linear regression:

```
import numpy as np
n_bootstraps = 1000

len_data = len(reg_data)
subsamp_n = np.int(0.75*len_data)
subsamp = lambda: np.random.choice(np.arange(0, len_data), size=sul
coefs = np.ones((n_bootstraps, 3))
for i in range(n_bootstraps):
    subsamp_idx = subsamp()
    subsamp_X = reg_data[subsamp_idx]
    subsamp_y = reg_target[subsamp_idx]
    lr.fit(subsamp_X, subsamp_y)

coefs[i][0] = lr.coef_[0]
coefs[i][1] = lr.coef_[1]
coefs[i][2] = lr.coef_[2]
```

The following is the output that gets generated:

Follow the same procedure with Ridge, and have a look at the output:

```
>>> r = Ridge()
>>> n_bootstraps = 1000
>>> len_data = len(reg_data)
>>> subsample_size = np.int(0.75*len_data)
>>> subsample = lambda: np.random.choice(np.arange(0,      len_data
size=subsample_size)
coefs_r = np.ones((n_bootstraps, 3))
# carry out the same procedure from above
```

The following is the output that gets generated: Don't let the similar width of the plots fool you; the coefficients for ridge regression are much closer to 0. Let's look at the average spread between the coefficients:

```
>>> np.mean(coefs - coefs_r, axis=0)
#coefs_r stores the ridge regression coefficients
array([ 22.19529525, 49.54961002, 8.27708536])
```

So, on an average, the coefficients for linear regression are much higher than the ridge regression coefficients. This difference is the bias in the coefficients (forgetting, for a second, the potential bias of the linear regression coefficients). So then, what is the advantage of ridge regression? Well, let's look at the variance of our coefficients:

```
>>> np.var(coefs, axis=0)
array([ 184.50845658, 150.16268077, 263.39096391])
>>> np.var(coefs_r, axis=0)
array([ 21.35161646, 23.95273241, 17.34020101])
```

Bias-Variance Trade Off

The variance has been dramatically reduced. This is the ***bias-variance trade-off*** that is so often discussed in machine learning.

The next section will introduce how to tune the regularization parameter in ridge regression, which is at the heart of this trade-off.

0.1 Theoretical Background

Optimizing the ridge regression parameter

- Once you start using ridge regression to make predictions or learn about relationships in the system you're modeling, you'll start thinking about the choice of alpha.
- For example, using OLS regression might show some relationship between two variables; however, when regularized by some alpha, the relationship is no longer significant.
- This can be a matter of whether a decision needs to be taken.

Getting ready

This is the first recipe where we'll tune the parameters for a model. This is typically done by cross-validation. There will be recipes laying out a more general way to do this in later recipes, but here we'll walkthrough to be able to tune ridge regression. If you remember, in ridge regression, the gamma parameter is typically represented as alpha in scikit-learn when calling RidgeRegression; so, the question that arises is what the best alpha is. Create a regression dataset, and then let's get started:

```
>>> from sklearn.datasets import make_regression
>>> reg_data, reg_target = make_regression(n_samples=100,
n_features=2, effective_rank=1, noise=10)
```

Implementation

- In the `linear_models` module, there is an object called `RidgeCV`, which stands for ridge cross-validation.
- This performs a cross-validation similar to ***leave-one-out cross-validation (LOOCV)***.

- Under the hood, it's going to train the model for all samples except one.

It'll then evaluate the error in predicting this one test case:

```
>>> from sklearn.linear_model import RidgeCV
>>> rcv = RidgeCV(alphas=np.array([.1, .2, .3, .4]))
>>> rcv.fit(reg_data, reg_target)
RidgeCV(alphas=array([ 0.1, 0.2, 0.3, 0.4]), cv=None,
fit_intercept=True, gcv_mode=None, loss_func=None,
normalize=False, score_func=None, scoring=None,
store_cv_values=False)
```

After we fit the regression, the alpha attribute will be the best alpha choice:

```
>>> rcv.alpha_
0.10000000000000001
```

In the previous example, it was the first choice. We might want to hone in on something around .1:

```
>>> rcv2 = RidgeCV(alphas=np.array([.08, .09, .1, .11, .12]))
>>> rcv2.fit(reg_data, reg_target)
RidgeCV(alphas=array([ 0.08, 0.09, 0.1 , 0.11, 0.12]), cv=None,
fit_intercept=True, gcv_mode=None,
loss_func=None, normalize=False,
score_func=None, scoring=None,
store_cv_values=False)
```

```
>>> rcv2.alpha_
0.08
```

We can continue this hunt, but hopefully, the mechanics are clear.

How it works

- The mechanics might be clear, but we should talk a little more about the why and define what was meant by "best".
- At each step in the cross-validation process, the model scores an error against the test sample.
- By default, it's essentially a squared error.

We can force the `RidgeCV` object to store the cross-validation values; this will let us visualize what it's doing:

```
>>> alphas_to_test = np.linspace(0.01, 1)
>>> rcv3 = RidgeCV(alphas=alphas_to_test, store_cv_values=True)
>>> rcv3.fit(reg_data, reg_target)
```

As you can see, we test a bunch of points (50 in total) between 0.01 and 1. Since we passed `store_cv_values` as true, we can access these values:

```
>>> rcv3.cv_values_.shape
(100, 50)
```

So, we had 100 values in the initial regression and tested 50 different alpha values. We now have access to the errors of all 50 values. So, we can now find the smallest mean error and choose it as alpha:

```
>>> smallest_idx = rcv3.cv_values_.mean(axis=0).argmin()
>>> alphas_to_test[smallest_idx]
```

The question that arises is "*Does RidgeCV agree with our choice*?" Use the following command to find out:

```
>>> rcv3.alpha_
0.01
```

It's also worthwhile to visualize what's going on. In order to do that, we'll plot the mean for all 50 test alphas.

Other Remarks

If we want to use our own scoring function, we can do that as well. Since we looked up MAD before, let's use it to score the differences. First, we need to define our loss function:

```
>>> def MAD(target, predictions):
absolute_deviation = np.abs(target - predictions)
return absolute_deviation.mean()
```

- After we define the loss function, we can employ the `make_scorer` function in sklearn.
- This will take care of standardizing our function so that scikit's objects know how to use it.
- Also, because this is a loss function and not a score function, the lower the better, and thus the need to let sklearn to flip the sign

to turn this from a maximization problem into a minimization problem:

```
>>> import sklearn
>>> MAD = sklearn.metrics.make_scorer(MAD, greater_is_better=False)
>>> rcv4 = RidgeCV(alphas=alphas_to_test, store_cv_values=True,
scoring=MAD)
>>> rcv4.fit(reg_data, reg_target)
>>> smallest_idx = rcv4.cv_values_.mean(axis=0).argmin()
>>> alphas_to_test[smallest_idx]
0.2322
```