In [1]:
```
import mglearn
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
```

In [3]:
```
X, y = mglearn.datasets.load_extended_boston()
##complex dataset
```

Let's take a look at how LinearRe gression performs on a more complex dataset, like the Boston Housing dataset. Remember that this dataset has 506 samples and 105 derived features.

Linear

In [4]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
lr = LinearRegression().fit(X_train, y_train)
```

In [5]:
```
print("Training set score: {:.2f}".format(lr.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lr.score(X_test, y_test)))
```

```
Training set score: 0.95
Test set score: 0.61
```

The test set is a clear sign of overfitting, and therefore we should try to find a model that allows us to con- trol complexity.

In [6]:
```
from sklearn.linear_model import Ridge
ridge = Ridge().fit(X_train, y_train) #Defalt alpha = 1
```

In [7]:
```
print("Training set score: {:.2f}".format(ridge.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge.score(X_test, y_test)))
```

```
Training set score: 0.89
Test set score: 0.75
```

As you can see, the training set score of Ridge is lower. Ridge is a more restricted model, so we are less likely to overfit.

Ridge (L2/^2)

In [8]:
```
ridge10 = Ridge(alpha=10).fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge10.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge10.score(X_test, y_test)))
```

```
Training set score: 0.79
Test set score: 0.64
```

Increasing alpha forces coefficients to move more toward zero, which decreases training set performance but might help generalization.

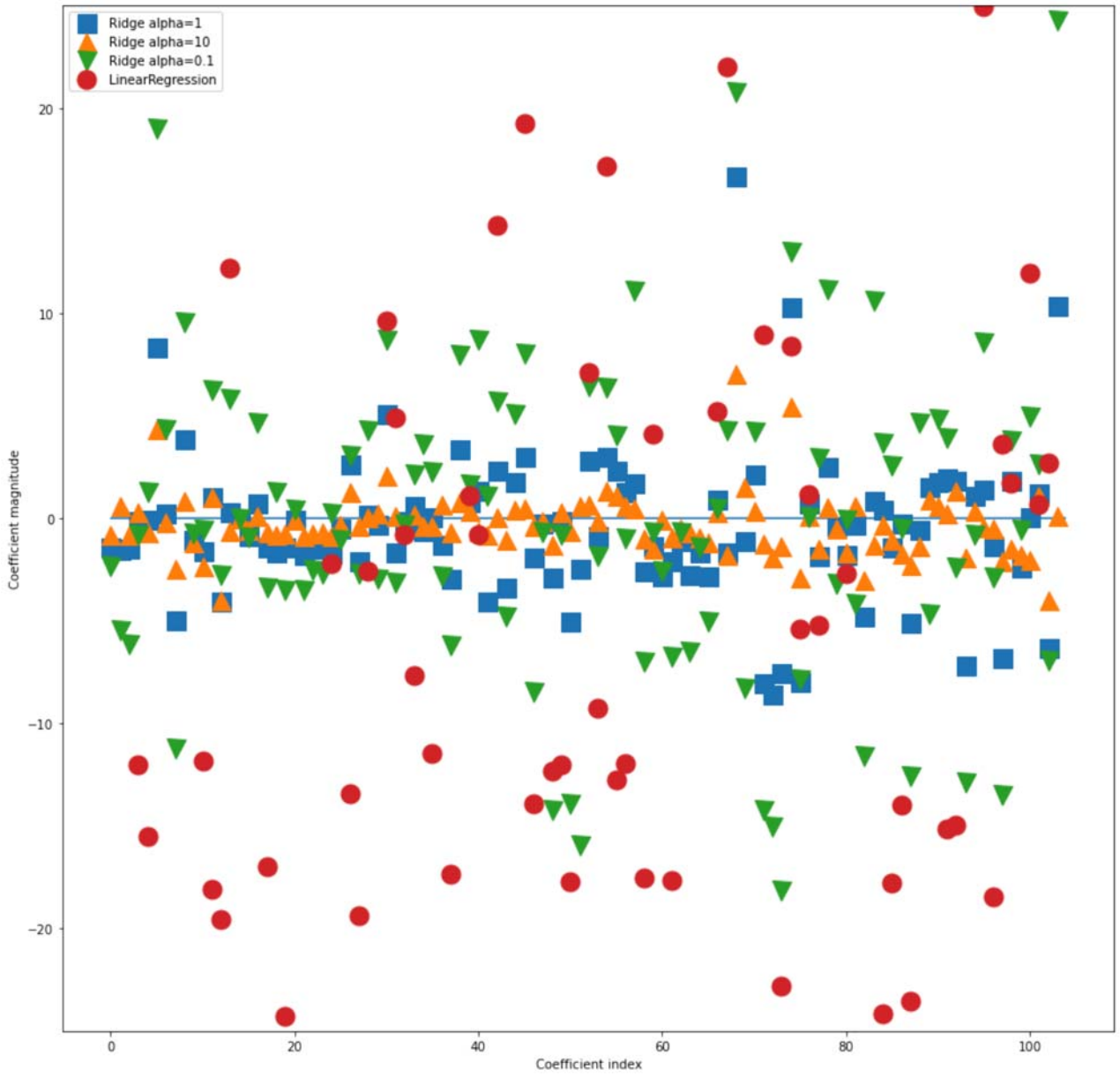
In [9]:
```
ridge01 = Ridge(alpha=0.1).fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge01.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge01.score(X_test, y_test)))
```

```
Training set score: 0.93
Test set score: 0.77
```

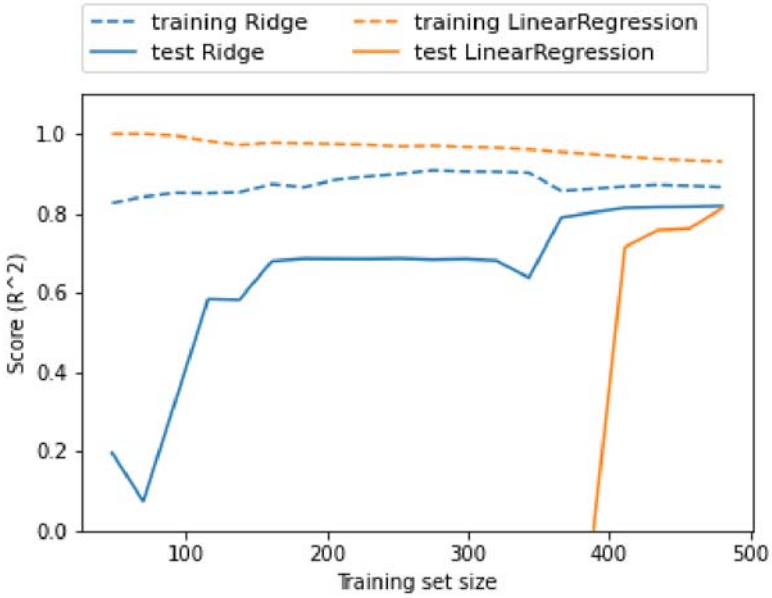
Decreasing alpha allows the coefficients to be less restricted, meaning we move right. For very small values of alpha, coefficients are barely restricted at all, and we end up with a model that resembles LinearRegression.

In [10]:
```
plt.figure(figsize=(15,15))
plt.plot(ridge.coef_, 's', label="Ridge alpha=1", markersize=15)
plt.plot(ridge10.coef_, '^', label="Ridge alpha=10", markersize=15)
plt.plot(ridge01.coef_, 'v', label="Ridge alpha=0.1", markersize=15)
plt.plot(lr.coef_, 'o', label="LinearRegression", markersize=15)
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
plt.hlines(0, 0, len(lr.coef_))
plt.ylim(-25, 25)
plt.legend()
```

Out[10]:
```
<matplotlib.legend.Legend at 0x2781f12ec10>
```



In [11]:
```
mglearn.plots.plot_ridge_n_samples()
```



The test score for ridge is better, particularly for small subsets of the data. For less than 400 data points, linear regression is not able to learn anything. As more and more data becomes available to the model, both models improve, and linear regression catches up with ridge in the end. The lesson here is that with enough training data, regularization becomes less important, and given enough data, ridge and linear regression will have the same performance.

Lasso (L1/ab)

In [12]:
```
from sklearn.linear_model import Lasso

lasso = Lasso().fit(X_train, y_train)

print("Training set score: {:.2f}".format(lasso.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso.coef_ != 0)))
```

```
Training set score: 0.29
Test set score: 0.21
Number of features used: 4
```

This indicates that we are underfitting, and we find that it used only 4 of the 105 features.

In [13]:
```
lasso001 = Lasso(alpha=0.01, max_iter=100000).fit(X_train, y_train)

print("Training set score: {:.2f}".format(lasso001.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso001.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso001.coef_ != 0)))
```

```
Training set score: 0.90
Test set score: 0.77
Number of features used: 33
```

A lower alpha allowed us to fit a more complex model,using only 33 of the 105 features.

In [14]:
```
lasso00001 = Lasso(alpha=0.0001, max_iter=100000).fit(X_train, y_train)
```

```
print("Training set score: {:.2f}".format(lasso00001.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso00001.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso00001.coef_ != 0)))
```

```
Training set score: 0.95
Test set score: 0.64
Number of features used: 96
```

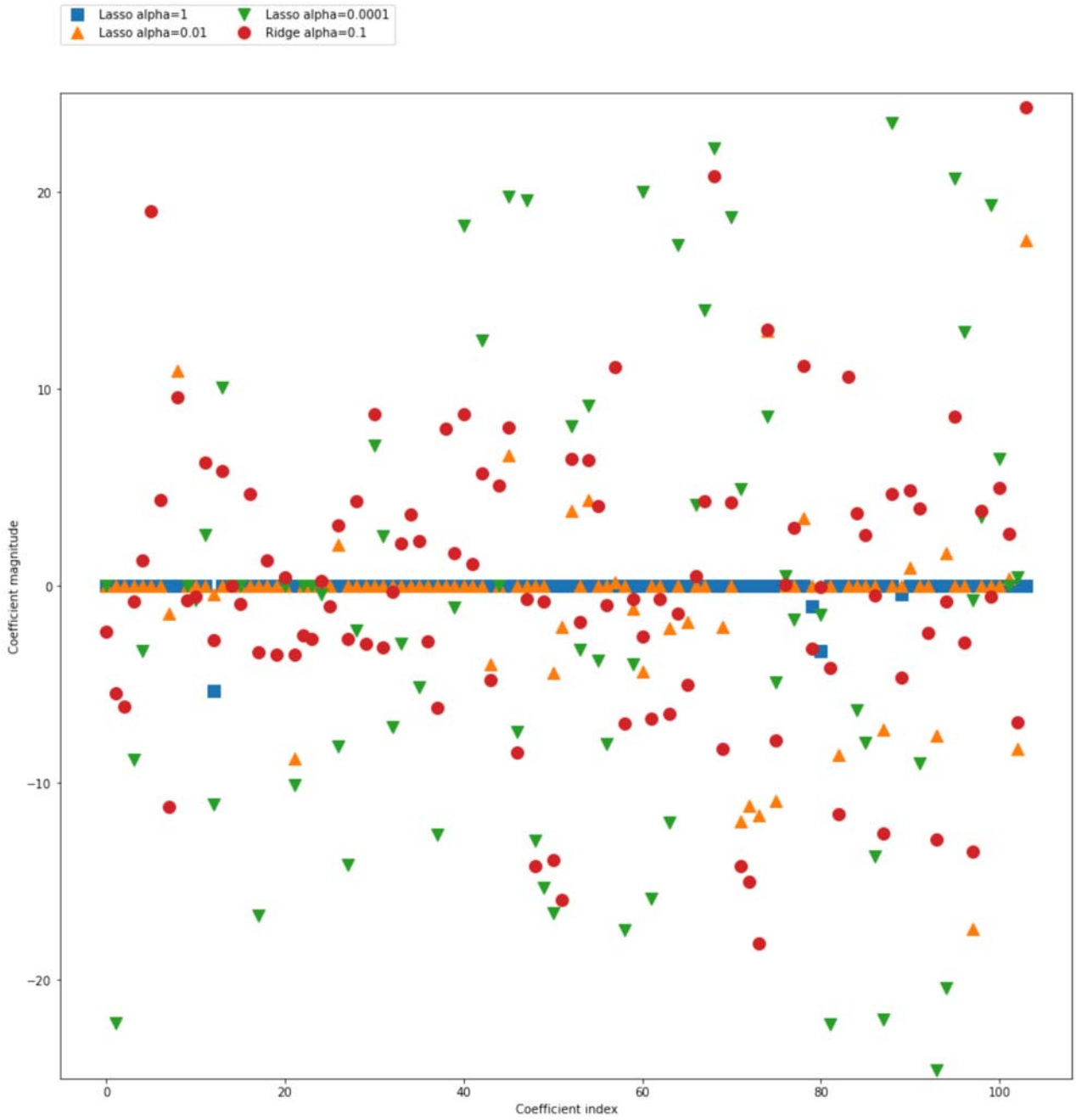
If we set alpha too low, however, we again remove the effect of regularization and end up overfitting, with a result similar to LinearRegression.

In [15]:

```
plt.figure(figsize=(15,15))
plt.plot(lasso.coef_, 's', label="Lasso alpha=1", markersize=10)
plt.plot(lasso001.coef_, '^', label="Lasso alpha=0.01", markersize=10)
plt.plot(lasso00001.coef_, 'v', label="Lasso alpha=0.0001", markersize=10)
plt.plot(ridge01.coef_, 'o', label="Ridge alpha=0.1", markersize=10)
plt.legend(ncol=2, loc=(0, 1.05))
plt.ylim(-25, 25)
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
```

Out[15]:

```
Text(0, 0.5, 'Coefficient magnitude')
```



In practice, ridge regression is usually the first choice between these two models. However, if you have a large amount of features and expect only a few of them to be important, Lasso might be a better choice.