Penalized Linear Regression in Python

Luis Pedro Coelho

Homepage: luispedro.org
Email: luis@luispedro.org
@luispedrocoelho
github/luispedro

Notes on the presentation

This presentation is an Ipython Notebook, which you get at github.com/luispedro/PenalizedRegression or view using the NBViewer.

Dependencies

To run this, you need

- ipython, version 2
- numpy
- matplotlib (for the plots)
- scikit-learn

Preliminary imports

We will import `numpy` using the `np` abbreviation and `matplplotlib.pyplot` using the `plt` abbreviation:

Preliminary imports

We will import `numpy` using the `np` abbreviation and `matplplotlib.pyplot` using the `plt` abbreviation:

In [1]:
```
import numpy as np
from matplotlib import pyplot as plt
```

Preliminary imports

We will import `numpy` using the `np` abbreviation and `matplplotlib.pyplot` using the `plt` abbreviation:

In [1]:
```
import numpy as np
from matplotlib import pyplot as plt
```

We also need to perform some magic to get plots inline:

In [2]:
```
%matplotlib inline
```

Regression

Regression can be used generically to mean "any type of learning from data."

More commonly, though, it is used to mean *learn to predict a **numeric** output from variables*. As opposed to **classification** which learns to predict a categorical output.

Examples

- predicting prices
- predicting blood sugar levels
- predicting product ratings (collaborative filtering)

Example: Boston House prices

The goal is to predict house prices in Boston based on variables such as

1. number of rooms
2. crime rate in area
3. pupil teacher ratio
4. ...

Loading data

The `boston` dataset comes built in with scikit-learn:

In [3]:
```
from sklearn.datasets import load_boston
boston = load_boston()
```

You can use `print(boston.DESCR)` to see more information on the dataset.

Split into testing & training

For all our analyses, it will be important to have split training & testing data.

Split into testing & training

For all our analyses, it will be important to have split training & testing data.

Scikit-learn makes this easy:

In [4]:
```
from sklearn.cross_validation import train_test_split
train_data, test_data, train_target, test_target = \
        train_test_split(boston.data, boston.target, train_size=.8)
```

Split into testing & training

For all our analyses, it will be important to have split training & testing data.

Scikit-learn makes this easy:

In [4]:
```
from sklearn.cross_validation import train_test_split
train_data, test_data, train_target, test_target = \
        train_test_split(boston.data, boston.target, train_size=.8)
```

- `train_data` is our training data with corresponding target variable `train_target`.
- `test_data` and `test_target` are the equivalent testing variables.

Having a peak at the data

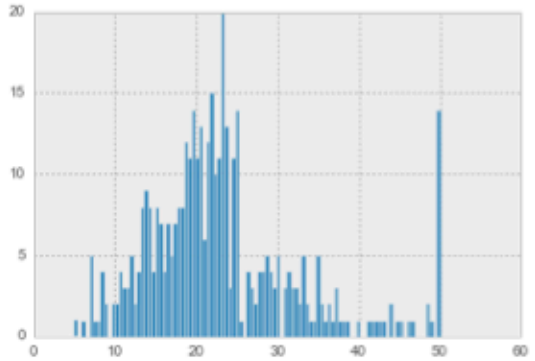
It is always a good idea to peak around, get a feeling for the data.

Extremely important to look out for anomalies (real data is rarely clean).

In [5]:
```
print(train_data.shape)
```

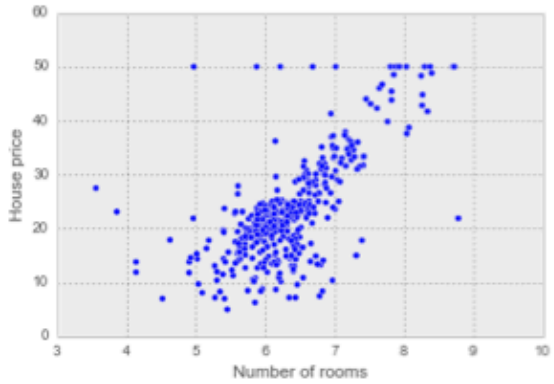
```
(404, 13)
```

In [6]:
```
_=plt.hist(train_target, bins=100)
```



We can look at specific elements in the input data, such as the number of rooms:

In [7]:
```
RoomNr_Index = 5

fig,ax = plt.subplots()
ax.scatter(train_data[:,RoomNr_Index], train_target)
ax.set_xlabel("Number of rooms")
ax.set_ylabel("House price")
pass
```



Using IPython Notebook for exploration

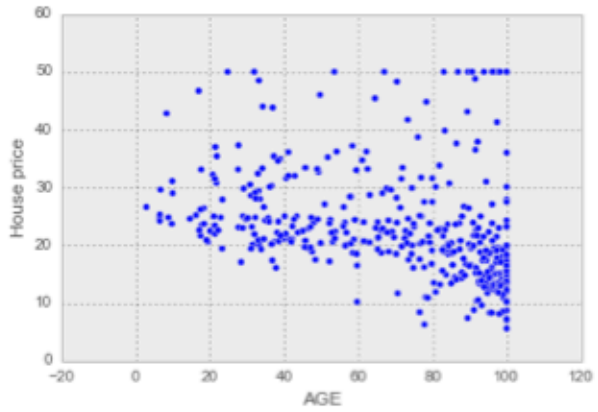
In [8]:
```
from IPython.html.widgets import interact
from scipy import stats
```

We will use the same plotting code, just generalized to take an `index` argument and wrapped with `interact`:

```
@interact(index=(0, train_data.shape[1]))
def plot_scatter(index):
f   ig,ax= p lt.subplots()
x   ,y = t rain_data[:,index],t rain_target
a   x.scatter(x,y )
a   x.set_xlabel(boston.feature_names[index])
a   x.set_ylabel( "House price")
    print("Correlation: {0[0]:.1} (p-value: {0[1]:.1})".format(stats.pearsonr(x, y)))
    return fig
```

index 6

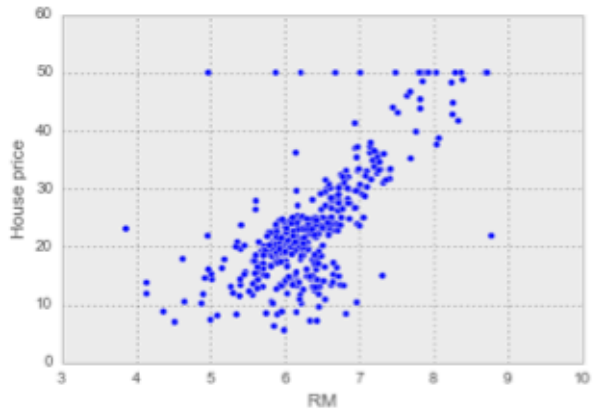
```
Correlation: -0.3 (p-value: 4e-12)
```



```
@interact(index=(0, train_data.shape[1]))
def plot_scatter(index):
f   ig,ax= p lt.subplots()
x   ,y = t rain_data[:,index],t rain_target
a   x.scatter(x,y )
a   x.set_xlabel(boston.feature_names[index])
a   x.set_ylabel( "House price")
    print("Correlation: {0[0]:.1} (p-value: {0[1]:.1})".format(stats.pearsonr(x, y)))
    return fig
```

index 5

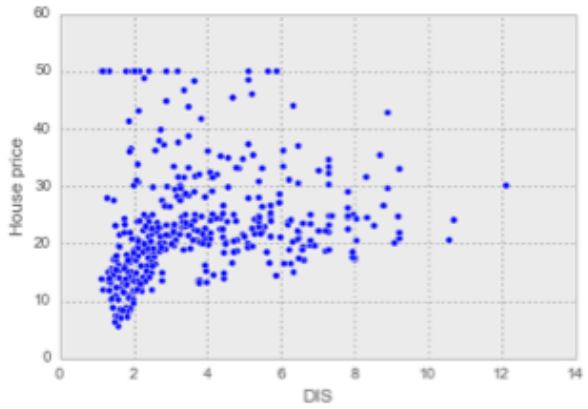
```
Correlation: 0.7 (p-value: 1e-63)
```



```
@interact(index=(0, train_data.shape[1]))
def plot_scatter(index):
f   ig,ax= p lt.subplots()
x   ,y = t rain_data[:,index],t rain_target
a   x.scatter(x,y )
a   x.set_xlabel(boston.feature_names[index])
a   x.set_ylabel( "House price")
    print("Correlation: {0[0]:.1} (p-value: {0[1]:.1})".format(stats.pearsonr(x, y)))
    return fig
```

index 7

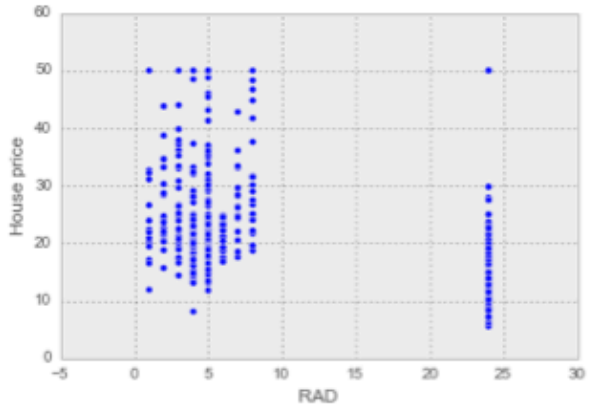
```
Correlation: 0.2 (p-value: 1e-05)
```



```
@interact(index=(0, train_data.shape[1]))
def plot_scatter(index):
f    ig,ax= p lt.subplots()
x    ,y = t rain_data[:,index],t rain_target
a    x.scatter(x,y )
a    x.set_xlabel(boston.feature_names[index])
a    x.set_ylabel( "House price")
     print("Correlation: {0[0]:.1} (p-value: {0[1]:.1})".format(stats.pearsonr(x, y)))
     return fig
```

index 8

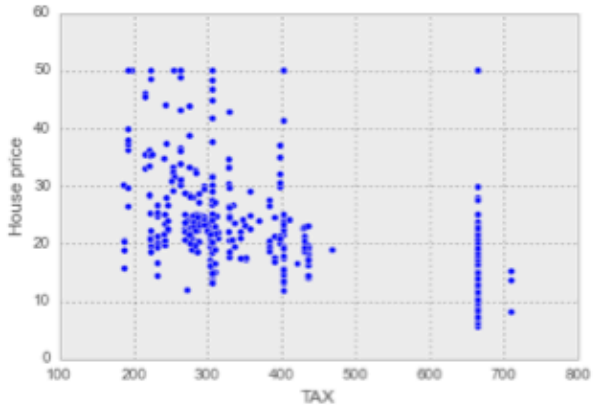
```
Correlation: -0.4 (p-value: 9e-14)
```



```
@interact(index=(0, train_data.shape[1]))
def plot_scatter(index):
f   ig,ax= p lt.subplots()
x   ,y = t rain_data[:,index],t rain_target
a   x.scatter(x,y )
a   x.set_xlabel(boston.feature_names[index])
a   x.set_ylabel( "House price")
    print("Correlation: {0[0]:.1} (p-value: {0[1]:.1})".format(stats.pearsonr(x, y)))
    return fig
```

index 9

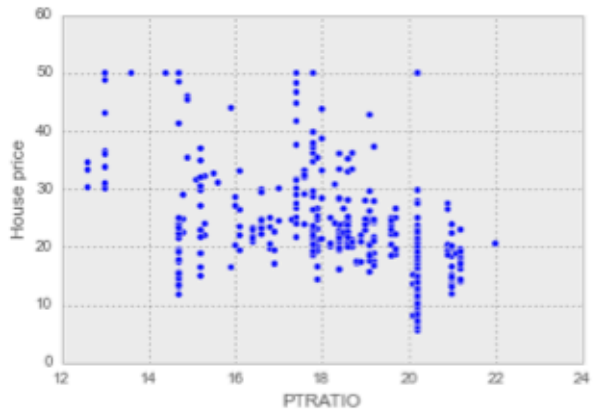
```
Correlation: -0.5 (p-value: 7e-22)
```



```
@interact(index=(0, train_data.shape[1]))
def plot_scatter(index):
f    ig,ax= p lt.subplots()
x    ,y = t rain_data[:,index],t rain_target
a    x.scatter(x,y )
a    x.set_xlabel(boston.feature_names[index])
a    x.set_ylabel( "House price")
     print("Correlation: {0[0]:.1} (p-value: {0[1]:.1})".format(stats.pearsonr(x, y)))
     return fig
```

index 10

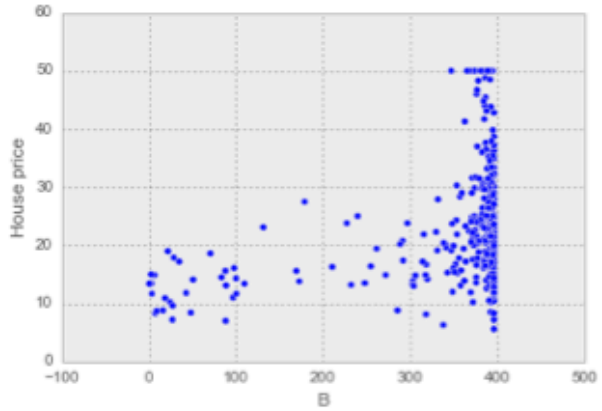
```
Correlation: -0.5 (p-value: 8e-25)
```



```
@interact(index=(0, train_data.shape[1]))
def plot_scatter(index):
f    ig,ax= p lt.subplots()
x    ,y = t rain_data[:,index],t rain_target
a    x.scatter(x,y )
a    x.set_xlabel(boston.feature_names[index])
a    x.set_ylabel( "House price")
     print("Correlation: {0[0]:.1} (p-value: {0[1]:.1})".format(stats.pearsonr(x, y)))
     return fig
```

index 11

```
Correlation: 0.3 (p-value: 7e-12)
```



Can we build a model, then?

We have now poked around the data, can we build a predictive model?

Ordinary Least Squares Linear Regression

Let x be our inputs and y our outputs.

Simple 1-D version

Our *generative* model of the data is simply that given an input x, we obtain y by

1. multiply by β,
2. adding a constant (c)
3. adding some noise (ϵ)

$$y = \beta x + c + \epsilon$$

Ordinary Least Squares Linear Regression

Let x be our inputs and y our outputs.

Simple 1-D version

Our *generative* model of the data is simply that given an input x, we obtain y by

1. multiply by β,
2. adding a constant (c)
3. adding some noise (ϵ)

$$y = \beta x + c + \epsilon$$

Multidimensional linear regression

Instead of a single input variable, we can have multiple inputs, $x_1, x_2, \ldots x_n$:

$$y = \sum_j \beta_j x_j + c + \epsilon$$

We can write the same using vector notation as:

$$y = \boldsymbol{\beta}^T \mathbf{x} + c + \epsilon$$

Least squares regression

- In real life, we have observations and want to fit a model.
- When we have more observations than datapoints, there is very little chance that a model will fit perfectly.
- Therefore, we **minimise the fitting error**.

$$y_i = \boldsymbol{\beta}^T \mathbf{x}_i + c + \epsilon_i$$

We can measure the error as the sum of squared errors:

$$E = \sum_i \epsilon_i^2$$

Least squares regression

- In real life, we have observations and want to fit a model.
- When we have more observations than datapoints, there is very little chance that a model will fit perfectly.
- Therefore, we **minimise the fitting error**.

$$y_i = \boldsymbol{\beta}^T \mathbf{x}_i + c + \epsilon_i$$

We can measure the error as the sum of squared errors:

$$E = \sum_i \epsilon_i^2$$

The criterion is thus

$$\beta^* = \arg\min_{\boldsymbol{\beta}} \sum_i (\boldsymbol{\beta}^T \mathbf{x}_i + c - y_i)^2$$

which we can write in short form as:

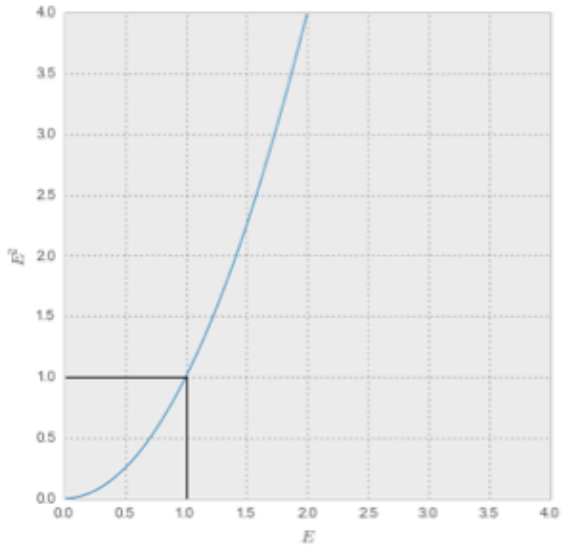
$$\beta^* = \arg\min_{\boldsymbol{\beta}} \sum_i \epsilon_i^2$$

A few notes on the square function

Pretty obvious but it'll come in handy later too:

- larger error are penalized much more heavily than small error
- doubling the error quadruples the penalty

In [11]:
```
display(squared_error_figure)
```



Historical note

This method of least squares was first developed to estimate the motion of cellular bodies by Gauss who was officially an astronomer. He also proved that least squares is optimal under *Gaussian* noise.

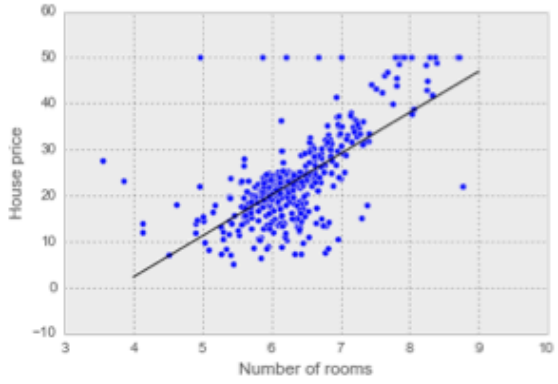
Using OLS on the boston data with scikit-learn

In [12]:
```
from sklearn import linear_model
linreg = linear_model.LinearRegression()
```

1D version

In [13]:
```
_=linreg.fit(train_data[:,RoomNr_Index:RoomNr_Index+1], train_target)
```

In [14]:
```
fig,ax = plt.subplots()
ax.scatter(train_data[:,RoomNr_Index], train_target)
ax.plot([4, 9], linreg.predict([[4],[9]]), 'k-')
ax.set_xlabel("Number of rooms")
_=ax.set_ylabel("House price")
```



Multi-dimensional version

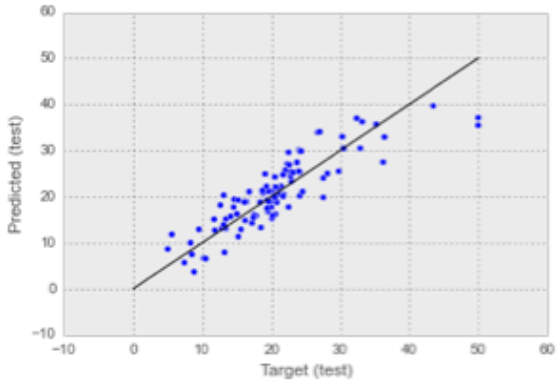
We can use the same `linreg.fit` method:

In [15]:
```
linreg.fit(train_data, train_target)
prediction = linreg.predict(test_data)
```

Visualizing the fit

We cannot easily see the fit in high dimensions, but we can plot *measured vs prediction*:

In [21]:
```
linreg.fit(train_data, train_target)
fig,ax = plt.subplots()
ax.scatter(test_target, linreg.predict(test_data))
ax.plot([0,50], [0,50], 'k-')
ax.set_xlabel('Target (test)')
ax.set_ylabel('Predicted (test)')
pass
```



Evaluating the results

Mean squared error

This is a very natural way to measure the error:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

where y_i is the actual value for example i, and $\hat{y}_i$ is the prediction for the same example.

In [16]:
```
from sklearn import metrics
mse = metrics.mean_squared_error(test_target, linreg.predict(test_data))
print("MSE is {}".format(mse))
```

```
MSE is 15.8112271367
```

Evaluating the results

Mean squared error

This is a very natural way to measure the error:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

where y_i is the actual value for example i, and $\hat{y}_i$ is the prediction for the same example.

In [16]:
```
from sklearn import metrics
mse = metrics.mean_squared_error(test_target, linreg.predict(test_data))
print("MSE is {}".format(mse))
```

```
MSE is 15.8112271367
```

Problem: it is not trivial to interpret the value obtained.

Root mean squared error

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1} N(y_i - \hat{y}_i)^2}$$

1. Same units as original target variable
2. If errors really are normal, then just double this value for an estimate of 95% confidence intervals.

In [17]:
```
rmse = np.sqrt(mse)
print("RMSE is {}".format(rmse))
```

```
RMSE is 3.97633337846
```

Coefficient of determination

Sometimes, there is another value that is very useful, the *coefficient of determination*.

Coefficient of determination

Sometimes, there is another value that is very useful, the *coefficient of determination*.

Idea: compare the RSME with a baseline.

Coefficient of determination

Sometimes, there is another value that is very useful, the *coefficient of determination*.

Idea: compare the RSME with a baseline.

Consider the null predictor, which ignores the input and always returns a contant:

```
def predict(features):
    return constant
```

The *best constant is the mean*:

```
def predict(features):
    return mean_of_training_target
```

Coefficient of determination

Sometimes, there is another value that is very useful, the *coefficient of determination*.

Idea: compare the RSME with a baseline.

Consider the null predictor, which ignores the input and always returns a contant:

```
def predict(features):
    return constant
```

The *best constant is the mean*:

```
def predict(features):
    return mean_of_training_target
```

Now, compute the ratio of error this predictor with the error of your predictor.

Coefficient of determination

Sometimes, there is another value that is very useful, the *coefficient of determination*.

Idea: compare the RSME with a baseline.

Consider the null predictor, which ignores the input and always returns a contant:

```
def predict(features):
    return constant
```

The *best constant is the mean*:

```
def predict(features):
    return mean_of_training_target
```

Now, compute the ratio of error this predictor with the error of your predictor.

$$\text{COD} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} \approx 1 - \frac{\text{MSE}}{\text{VAR}(y)}$$

Where $\bar{y}_i$ is the *average value of* y, i.e.:

$$\bar{y} = \frac{\sum_i y_i}{N}$$

Evaluating COD in Boston dataset

In [18]:
```
cod = metrics.r2_score(test_target, linreg.predict(test_data))
print("COD is {}".format(cod))
```

```
COD is 0.750477222334
```

This measure is also called R^2, which is confusing because it is not the square of anything.

It is also called Q^2 sometimes, which is also confusing.

This measure is the default measure for regression in scikit-learn and we can just use the `score` method:

In [19]:
```
print(linreg.score(test_data, test_target))
```

```
0.750477222334
```

Training vs. Testing Data

Naturally, the results on the training data are better than those obtained on the testing data

In [20]:
```
linreg.fit(train_data, train_target)
r2_train = metrics.r2_score(train_target, linreg.predict(train_data))
r2_test = metrics.r2_score(test_target, linreg.predict(test_data))

print("R2 on training: {:.1%}".format(r2_train))
print("R2 on testing: {:.1%}".format(r2_test))
```

```
R2 on training: 73.3%
R2 on testing: 75.0%
```

Penalized (or Regularized) regression

The criterion for least squared regression was:

$$\beta = \arg\min \sum_j \epsilon_j^2$$

That is, *we minimize the fit on the training data.*

Penalized (or Regularized) regression

The criterion for least squared regression was:

$$\beta = \arg\min \sum_j \epsilon_j^2$$

That is, *we minimize the fit on the training data.*

The general expresssion for penalized regression is

$$\beta = \arg\min \frac{1}{2N} \sum_j \epsilon_j^2 + \alpha R(\boldsymbol{\beta}),$$

where R is the penalty (regularization) term and α is a positive weight.

That is, we minimize the **sum of the fit plus a regularization term.**

Penalized (or Regularized) regression

The criterion for least squared regression was:

$$\beta = \arg\min \sum_j \epsilon_j^2$$

That is, *we minimize the fit on the training data.*

The general expresssion for penalized regression is

$$\beta = \arg\min \frac{1}{2N} \sum_j \epsilon_j^2 + \alpha R(\boldsymbol{\beta}),$$

where R is the penalty (regularization) term and α is a positive weight.

That is, we minimize the **sum of the fit plus a regularization term.**

What about the intercept?

- The intercept is typically not penalized.
- One can simply pre-center the data.

L1 and L2 penalties

L_1 penalty means that we use the *sum of* ***absolute values***:

$$P_1(\boldsymbol{\beta}) = \sum_j |\beta_j|$$

L_2 penalty means that we use the *sum of* ***squares***:

$$P_2(\boldsymbol{\beta}) = \sum_j \beta_j^2$$

(The names come from the generic concept of an L_p norm, see Wikipedia for details)

L1 penalty: the lasso

Using the L1 norm, leads us to the *Lasso*!

Lasso stands for *least absolute shrinkage and selection operator*, but nobody uses that long name.

In scikit-learn:

In [22]:
```
lasso = linear_model.Lasso()
lasso.fit(train_data, train_target)
pass
```

Setting the α parameter:

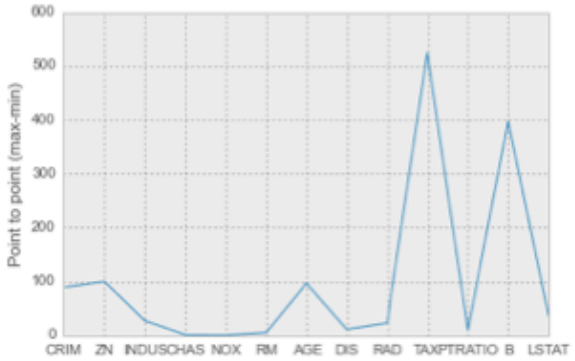
In [23]:
```
lasso = linear_model.Lasso(alpha=0.1)
lasso.fit(train_data, train_target)
pass
```

(We will discuss how to set the α parameter in a smarter later in the talk)

Normalizing the inputs

Some features span much higher values. Thus, when our penalty adds all the weights together, it implicitly weighs the low varying features more.

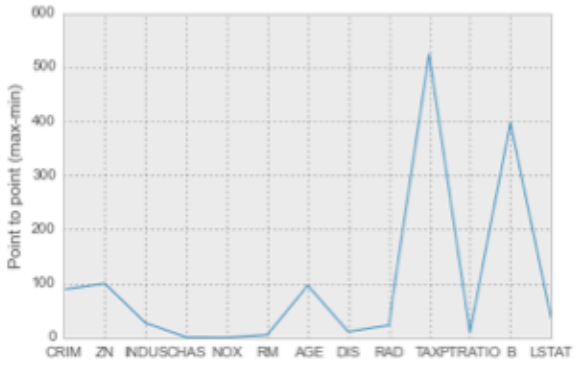
In [24]:
```
fig,ax = plt.subplots()
ax.plot(train_data.ptp(0))
ax.set_xticklabels(boston.feature_names)
ax.set_xticks(np.arange(train_data.shape[1]))
ax.set_ylabel('Point to point (max-min)')
pass
```



Normalizing the inputs

Some features span much higher values. Thus, when our penalty adds all the weights together, it implicitly weighs the low varying features more.

In [24]:
```
fig,ax = plt.subplots()
ax.plot(train_data.ptp(0))
ax.set_xticklabels(boston.feature_names)
ax.set_xticks(np.arange(train_data.shape[1]))
ax.set_ylabel('Point to point (max-min)')
pass
```

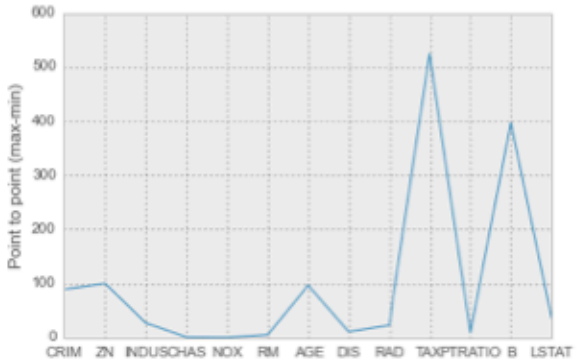


Normalization ensures that all the inputs are on the same scale.

Normalizing the inputs

Some features span much higher values. Thus, when our penalty adds all the weights together, it implicitly weighs the low varying features more.

In [24]:
```
fig,ax = plt.subplots()
ax.plot(train_data.ptp(0))
ax.set_xticklabels(boston.feature_names)
ax.set_xticks(np.arange(train_data.shape[1]))
ax.set_ylabel('Point to point (max-min)')
pass
```



Normalization ensures that all the inputs are on the same scale.

In [25]:
```
lasso = linear_model.Lasso(normalize=True, alpha=.1)
lasso.fit(train_data, train_target)
pass
```

Evaluating Lasso vs. OLS

In [26]:
```
linreg.fit(train_data, train_target)
r2_ols_train = linreg.score(train_data, train_target)
r2_ols = linreg.score(test_data, test_target)

lasso.fit(train_data, train_target)
r2_lasso_train = lasso.score(train_data, train_target)
r2_lasso = lasso.score(test_data, test_target)
```

In [28]:
```
print(results)
```

```
      | TRAINING | TESTING
------+----------+---------
OLS   | 73.34%   | 75.05%
------+----------+---------
Lasso | 59.13%   | 65.88%
--------------------------
```

In a bit, we will see another dataset, where Lasso shines brighter.

L2 penalty: Ridge regression

Using the L2 norm, leads us to *Ridge* regression (a.k.a. Tikhonov regularization).

Unlike Lasso, which was invented only recently (1996), Ridge goes back a few decades.

In [29]:
```
ridge = linear_model.Ridge(normalize=True, alpha=.1)

ridge.fit(train_data, train_target)
r2_ridge_train = ridge.score(train_data, train_target)
r2_ridge = ridge.score(test_data, test_target)
```

In [31]:
```
print(results)
```

```
      | TRAINING | TESTING
------+----------+---------
OLS   | 73.34%   | 75.05%
------+----------+---------
Lasso | 59.13%   | 65.88%
------+----------+---------
Ridge | 72.30%   | 77.23%
--------------------------
```

Lasso returns a sparse solution

This means that **some coefficients are exactly zero**:

In [32]:
```
print(lasso.coef_)
```

```
[-0.          0.         -0.          0.         -0.          2.92816138
 -0.          0.         -0.         -0.         -0.37909045  0.
 -0.45839082]
```

We can use Lasso to select important features!

Lasso returns a sparse solution

This means that **some coefficients are exactly zero**:

In [32]:
```
print(lasso.coef_)
```

```
[-0.          0.         -0.          0.         -0.          2.92816138
 -0.          0.         -0.         -0.         -0.37909045  0.
 -0.45839082]
```

We can use Lasso to select important features!

Higher penalties (higher α) means more sparsity:

In [33]:
```
lasso.alpha = .2
lasso.fit(train_data, train_target)
print(lasso.coef_)
```

```
[-0.          0.         -0.          0.         -0.          1.47538364
 -0.          0.         -0.         -0.         -0.          0.
 -0.30706838]
```

Lasso returns a sparse solution

This means that **some coefficients are exactly zero**:

In [32]:
```
print(lasso.coef_)
```

```
[-0.          0.         -0.          0.         -0.          2.92816138
 -0.          0.         -0.         -0.         -0.37909045  0.
 -0.45839082]
```

We can use Lasso to select important features!

Higher penalties (higher α) means more sparsity:

In [33]:
```
lasso.alpha = .2
lasso.fit(train_data, train_target)
print(lasso.coef_)
```

```
[-0.          0.         -0.          0.         -0.          1.47538364
 -0.          0.         -0.         -0.         -0.          0.
 -0.30706838]
```

Too high value for α results in zeros everywhere:

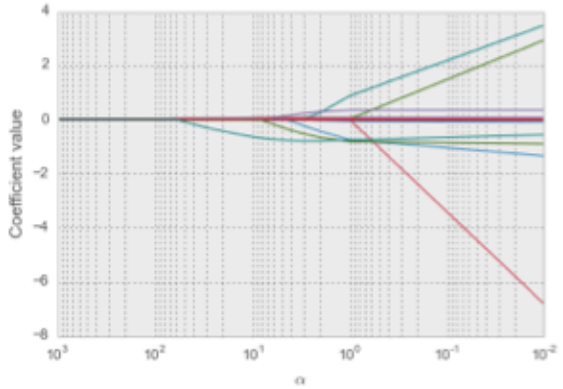
In [34]:
```
lasso.alpha = 1.
lasso.fit(train_data, train_target)
print(lasso.coef_)
```

```
[-0.  0. -0.  0. -0.  0. -0.  0. -0. -0. -0.  0. -0.]
```

Looking at the Lasso Path

This is a way to visualize what happens when we increase/decrease regularization.

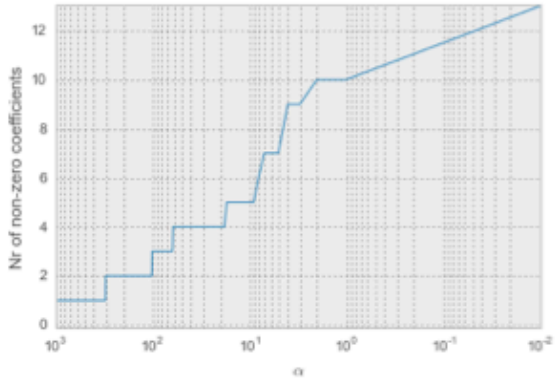
In [35]:
```
alphas = np.linspace(.01, 1000., 1000)
alphas, coefs, _= lasso.path(train_data, train_target, alphas=alphas)

fig,ax = plt.subplots()
ax.plot(alphas, coefs.T)
ax.set_xscale('log')
ax.set_xlim(alphas.max(), alphas.min())
ax.set_xlabel(r'$\alpha$')
ax.set_ylabel('Coefficient value')
pass
```



Another look at the path

Instead of the values, we can also just count how many are non zero:

In [36]:
```
fig,ax = plt.subplots()
ax.plot(alphas, np.sum(coefs != 0.0, axis=0))
ax.set_xscale('log')
ax.set_xlim(alphas.max(), alphas.min())
ax.set_ylim(-.1, 13.1)
ax.set_xlabel(r'$\alpha$')
ax.set_ylabel('Nr of non-zero coefficients')
pass
```



Ridge solutions are not sparse

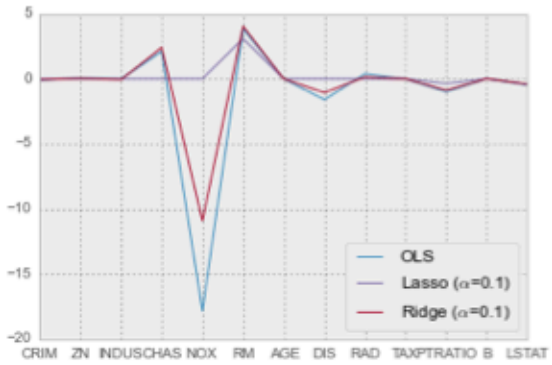
Unlike the Lasso, ridge does not set any coefficients to zero!

In [39]:
```
ridge.fit(train_data, train_target)
print(ridge.coef_)
```

```
[ -7.16516257e-02   2.61968547e-02  -6.48444922e-02   2.39713486e+00
  -1.09193751e+01   4.03200954e+00  -9.44894418e-03  -1.06942801e+00
   1.31698631e-01  -6.45883648e-03  -8.87770177e-01   7.47187438e-03
  -4.17026739e-01]
```

Comparing OLS, Lasso, and Ridge

In [41]:
```
fig,ax = plt.subplots()
ax.plot(linreg.coef_, label='OLS')
ax.plot(lasso.coef_, label=r'Lasso ($\alpha$=0.1)')
ax.plot(ridge.coef_, label=r'Ridge ($\alpha$=0.1)')
ax.set_xticklabels(boston.feature_names)
ax.set_xticks(np.arange(train_data.shape[1]))
ax.legend(loc='best')
pass
```



- OLS is always larger than either Lasso or Ridge
- Lasso sets some weights to zero; Ridge does not.

Why does the Lasso return a sparse solution while Ridge does not?

Argument 1: algorithmic argument

Consider the following pseudo-algorithm:

1. Start with $\boldsymbol{\beta} = 0$.
2. Find the best index to increment j and spend a bit of your budget to move β_j in the right direction.
3. If this is better than before, then **goto 2**; else, stop.

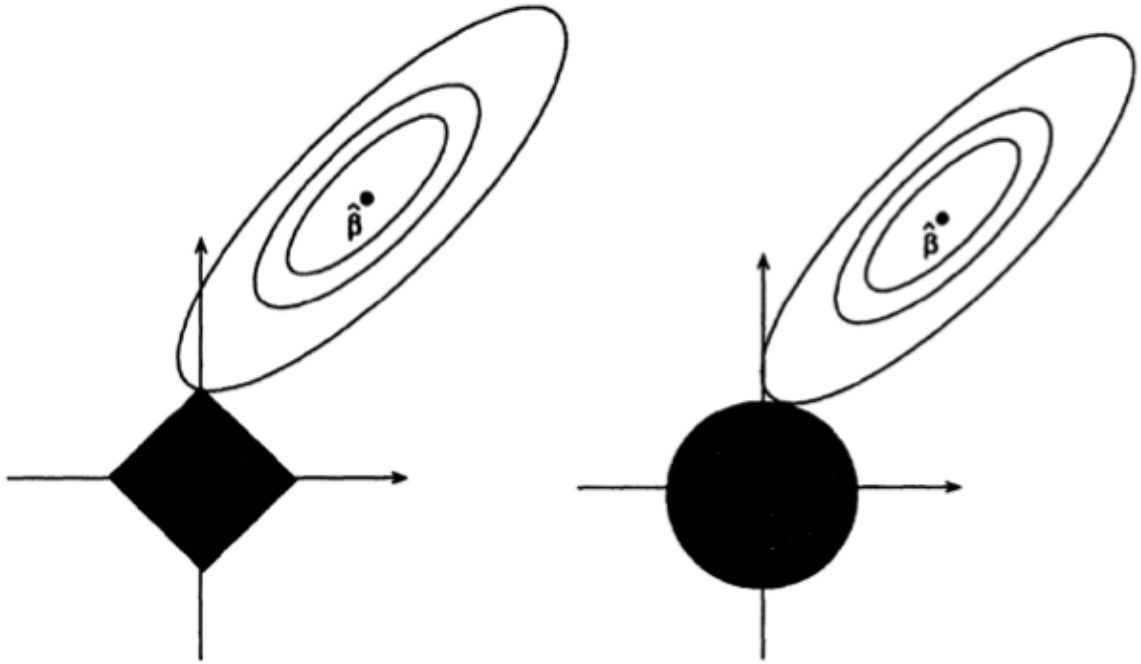
Because Ridge uses squared penalties, βs that are still stuck at zero are very cheap in step 2, so we might choose them even if they are not very good.

With Lasso, however, they always cost the same. Therefore, the algorithm will typically keep choosing the same index repeatedly.

Note: An improved version of this idea forms the basis of the famous Least-Angle Regression method to solve the Lasso problem.

Argument 2: geometric argument

- Consider Lasso (left) vs Ridge (right)



(a) (b)

- The filled areas are those were the penalty is below a certain limit, while the ellipsis show the isocontours.
- With Lasso, they intersect at edges, with Ridge, in intermediate values.

(This is Figure 2 from the paper *Regression shrinkage and selection via the lasso*, by R. Tibshirani)

Combining L1 and L2 penalties: elastic nets

We can combine the two types of penalty to obtain *elastic nets*:

$$P(\boldsymbol{\beta}) = \alpha_1 P_1(\boldsymbol{\beta}) + \alpha_2 P_2(\boldsymbol{\beta}) = \alpha_1 \sum_j |\beta_j| + \alpha_2 \sum_j \beta_j^2$$

We now have two penalties, with different weights, α_1 and α_2.

Combining L1 and L2 penalties: elastic nets

We can combine the two types of penalty to obtain *elastic nets*:

$$P(\boldsymbol{\beta}) = \alpha_1 P_1(\boldsymbol{\beta}) + \alpha_2 P_2(\boldsymbol{\beta}) = \alpha_1 \sum_j |\beta_j| + \alpha_2 \sum_j \beta_j^2$$

We now have two penalties, with different weights, α_1 and α_2.

In [40]:
```
en = linear_model.ElasticNet(normalize=True)
```

Combining L1 and L2 penalties: elastic nets

We can combine the two types of penalty to obtain *elastic nets*:

$$P(\boldsymbol{\beta}) = \alpha_1 P_1(\boldsymbol{\beta}) + \alpha_2 P_2(\boldsymbol{\beta}) = \alpha_1 \sum_j |\beta_j| + \alpha_2 \sum_j \beta_j^2$$

We now have two penalties, with different weights, α_1 and α_2.

In [40]:
```
en = linear_model.ElasticNet(normalize=True)
```

Instead of α_1, and α_2, in scikit-learn one specifies two parameters, α and ρ (a.k.a. `l1_ratio`)

$$\alpha_1 = \rho\alpha$$

$$\alpha_2 = \frac{1}{2}(1-\rho)\alpha$$

In [41]:
```
en = linear_model.ElasticNet(normalize=True, alpha=0.1, l1_ratio=.5)
```

Combining L1 and L2 penalties: elastic nets

We can combine the two types of penalty to obtain *elastic nets*:

$$P(\boldsymbol{\beta}) = \alpha_1 P_1(\boldsymbol{\beta}) + \alpha_2 P_2(\boldsymbol{\beta}) = \alpha_1 \sum_j |\beta_j| + \alpha_2 \sum_j \beta_j^2$$

We now have two penalties, with different weights, α_1 and α_2.

In [40]:
```
en = linear_model.ElasticNet(normalize=True)
```

Instead of α_1, and α_2, in scikit-learn one specifies two parameters, α and ρ (a.k.a. `l1_ratio`)

$$\alpha_1 = \rho\alpha$$

$$\alpha_2 = \frac{1}{2}(1-\rho)\alpha$$

In [41]:
```
en = linear_model.ElasticNet(normalize=True, alpha=0.1, l1_ratio=.5)
```

This way, `alpha` defines *amount of penalty* and `l1_ratio` how the penalty is split between L1 and L2.

An Elastic net can interpolate between Lasso and Ridge

In [42]:
```
lasso = linear_model.ElasticNet(normalize=True, alpha=0.1, l1_ratio=1.)
ridge = linear_model.ElasticNet(normalize=True, alpha=0.1, l1_ratio=0.)
half_way =linear_model.ElasticNet(normalize=True, alpha=0.1, l1_ratio=.5)
```

One issue with pure Lasso is that it is greedy and unstable

- Between very similar features, it will choose the one that is best
- Even if the differences are very slight
- An elastic net with a small L2 penalty will make the learner "split the difference"

In [43]:
```
almost_lasso = linear_model.ElasticNet(normalize=True, alpha=0.1, l1_ratio=.95)
almost_lasso.fit(train_data, train_target)
lasso.fit(train_data, train_target)

print(almost_lasso.coef_)
print(lasso.coef_)
```

```
[ -1.13393598e-02   2.60068231e-03  -4.33640642e-02   0.00000000e+00
  -1.07617129e+00   1.51737956e+00  -0.00000000e+00   0.00000000e+00
  -0.00000000e+00  -1.42137277e-03  -2.63138038e-01   8.77320092e-04
  -1.67198243e-01]
[-0.          0.         -0.          0.         -0.          2.92816138
 -0.          0.         -0.         -0.         -0.37909045  0.
 -0.45839082]
```

Tackling a large problem

The boston dataset is small, so it does not fully do justice to the power of penalized regression.

The 10-K Corpus

Quoting from the online page for this dataset:

> *The corpus contains 10-K reports from many US companies during years 1996-2006, as well as measured volatility of stock returns for the twelve-month periods preceding and following each report*

This was generated and used in this paper:

> *Predicting Risk from Financial Reports with Regression Shimon Kogan, Dimitry Levin, Bryan R. Routledge, Jacob S. Sagi, and Noah A. Smith NAACL-HLT 2009, Boulder, CO, May–June 2009 Online version of paper*

Loading and Splitting the Dataset

In [45]:
```
from sklearn.datasets import load_svmlight_file
E2006_data, E2006_target = load_svmlight_file('E2006.train.bz2')
print(E2006_data.shape)
```

```
(16087, 150360)
```

We have 16k examples and 150k features!

In [46]:
```
print(type(E2006_data))
```

```
<class 'scipy.sparse.csr.csr_matrix'>
```

The dataset is staored in memory as a sparse matrix as most entries are zero.

Split into train/test

In [47]:
```
E2_train_data, E2_test_data, E2_train_target, E2_test_target = \
        train_test_split(E2006_data, E2006_target, train_size=.8)
```

Trying OLS on E2006 dataset

In [48]:
```
linreg.fit(E2_train_data, E2_train_target)
r2_ols_train = linreg.score(E2_train_data, E2_train_target)
r2_ols = linreg.score(E2_test_data, E2_test_target)
```

- Note that we just used the same code as before even though we are using a sparse matrix!
- Underneath the hood, scikit-learn selects the right implementation.

Trying OLS on E2006 dataset

In [48]:
```
linreg.fit(E2_train_data, E2_train_target)
r2_ols_train = linreg.score(E2_train_data, E2_train_target)
r2_ols = linreg.score(E2_test_data, E2_test_target)
```

- Note that we just used the same code as before even though we are using a sparse matrix!
- Underneath the hood, scikit-learn selects the right implementation.

In [49]:
```
print("R2 Training (OLS): {:.1%}".format(r2_ols_train))
print("R2 Testing (OLS): {:.1%}".format(r2_ols))
```

```
R2 Training (OLS): 100.0%
R2 Testing (OLS): 31.3%
```

Ordinary Least Squares fails when "P > N"

- P is the number of dimensions
- N is the number of examples
- when $P > N$, we have more dimensions than examples
- mathematically, it "always" fits training data perfectly!, but generalizes poorly

In [50]:
```
lasso = linear_model.Lasso(normalize=True, alpha=.01)
lasso.fit(E2_train_data, E2_train_target)

r2_lasso_train = lasso.score(E2_train_data, E2_train_target)
r2_lasso = lasso.score(E2_test_data, E2_test_target)
ridge = linear_model.Ridge(normalize=True, alpha=.01)
ridge.fit(E2_train_data, E2_train_target)

r2_ridge_train = ridge.score(E2_train_data, E2_train_target)
r2_ridge = ridge.score(E2_test_data, E2_test_target)
```

In [52]:
```
print(results_p_gt_n)
```

```
      | TRAINING | TESTING
------+----------+---------
OLS   | 100.00%   | 31.29%
------+----------+---------
Lasso | 0.00%   | -0.01%
------+----------+---------
Ridge | 68.59%   | 56.44%
--------------------------
```

Fitting hyperparameters properly

Those α values I set, worked well for the examples, but why did I use them and not others?

So far, we have been **cheating** and should be sent to machine learning re-education camp.

- We need to set parameters (I've heard it argued that *setting hyperparameters is the last big open problem in statistics*)
- So far, we hand waived this problem away
- I gave you parameters that worked because I cheated!
- Remember that parameters depend on scale of inputs (sklearn normalization scales variables, but not target)

Generic parameter setting solution

Validation dataset or cross-validation!

Generic parameter setting solution

Validation dataset or cross-validation!

- These technique work well for this problem, because we use *warm starts*.
- We already took advantage of this fact, when using the `path` method.

Generic parameter setting solution

Validation dataset or cross-validation!

- These technique work well for this problem, because we use *warm starts*.
- We already took advantage of this fact, when using the `path` method.

In [53]:
```
enCV = linear_model.ElasticNetCV(normalize=True,
                                 l1_ratio=[.1, .5, .7, .9, .95, .99, 1],
                                 )
```

Generic parameter setting solution

Validation dataset or cross-validation!

- These technique work well for this problem, because we use *warm starts*.
- We already took advantage of this fact, when using the `path` method.

In [53]:
```
enCV = linear_model.ElasticNetCV(normalize=True,
                                 l1_ratio=[.1, .5, .7, .9, .95, .99, 1],
                                 )
```

Use multiple processors

In [54]:
```
enCV = linear_model.ElasticNetCV(normalize=True,
                                 l1_ratio=[.1, .5, .7, .9, .95, .99, 1],
                                 n_jobs=-1,
                                 )
```

Let me repeat that

In [55]:
```
from sklearn import linear_model
enCV = linear_model.ElasticNetCV(normalize=True,
                                 l1_ratio=[.1, .5, .7, .9, .95, .99, 1],
                                 n_jobs=-1,
                                 )
```

This can be your **default object for regression**

- Generally applicable
- Takes advantage of multiple CPUs
- Automatically sets hyperparameters
- Explores different models

Final solution for E2006 dataset.

The full example shown above takes too long for a demo, so let us run a scaled down version (which takes a few minutes):

In [56]:
```
enCV = linear_model.ElasticNetCV(normalize=True,
                                 l1_ratio=[.1, .5, .9, .99],
                                 alphas=[.001, .01, .05, .1, .25, .5, 1.],
                                 n_jobs=-1,
                                 )
enCV.fit(E2_train_data, E2_train_target)
r2_enCV_train = enCV.score(E2_train_data, E2_train_target)
r2_enCV = enCV.score(E2_test_data, E2_test_target)
```

When you can run analyses offline, then you may gain a bit from exploring more of the parameter space.

In [58]:
```
print(result_en_cv)
```

```
      | TRAINING | TESTING
------+----------+---------
OLS   | 100.00%   | 31.29%
------+----------+---------
Lasso | 0.00%   | -0.01%
------+----------+---------
Ridge | 68.59%   | 56.44%
------+----------+---------
EN-CV | 59.11%   | 59.41%
-------------------------
```

Loose Ends

There are some parameters which can be interesting, which we did not look at:

In [59]:
```
print(enCV)
```

```
ElasticNetCV(alphas=[0.001, 0.01, 0.05, 0.1, 0.25, 0.5, 1.0], copy_X=True,
       cv=None, eps=0.001, fit_intercept=True,
       l1_ratio=[0.1, 0.5, 0.9, 0.99], max_iter=1000, n_alphas=100,
       n_jobs=-1, normalize=True, positive=False, precompute='auto',
       random_state=None, selection='cyclic', tol=0.0001, verbose=0)
```

- `alphas` lets you specify the α values to test (similar to `l1_ratio`). It is fine to leave this unspecified.
- `fit_intercept` can be set to `False` to avoid setting an intercept.
- `n_jobs` sets the number of job (-1 means all CPUs)
- `positive` can be used to force all coefficients to be positive.

All the other ones are algorithm-internal and useful mainly if you are having problems (try updating scikit-learn first, though, newer versions have better implementations for large problems).

Conclusions

- Linear regression is a solid trick
- Lasso gives sparse solutions, Ridge does not.
- Lasso can be used as a feature selection method.
- Elastic nets combines both penalties
- Use `ElasticNetCV` with a wide range of parameters as your "go to regression method"
- scikit-learn is great

Thank you for your time

This presentation was based on Chapter 7 (*Regression*) of Building Machine Learning Systems with Python

Find me at luispedro.org or on twitter at @luispedrocoelho

This whole presentation can be obtained at github.com/luispedro/PenalizedRegression or viewed using the NBViewer.