Supervised Learning: Regression

1 Outline

- Bias – Variance trade off
- Regression with Regularization
- Advanced technique in regression

2 Bias – Variance Tradeoff

The bias–variance tradeoff is the problem of simultaneously minimizing two sources of error that prevent supervised learning algorithms from generalizing: The bias is error from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting). The variance is error from sensitivity to small fluctuations in the training set. High variance can cause overfitting: which makes the model consider the
random noise in the training data.

Hidden mechanism: $\mathrm{y} = \mathrm{f(x)} + \varepsilon \;\; \varepsilon \sim \mathrm{N}(0, \sigma)$

Data: $(x_0, y_0), (x_1, y_1)$ …

We want to: get $\hat{f}(x)$ or for any x_0, get $\widehat{y_0}$

2.1 Bias – Variance Tradeoff (Details)

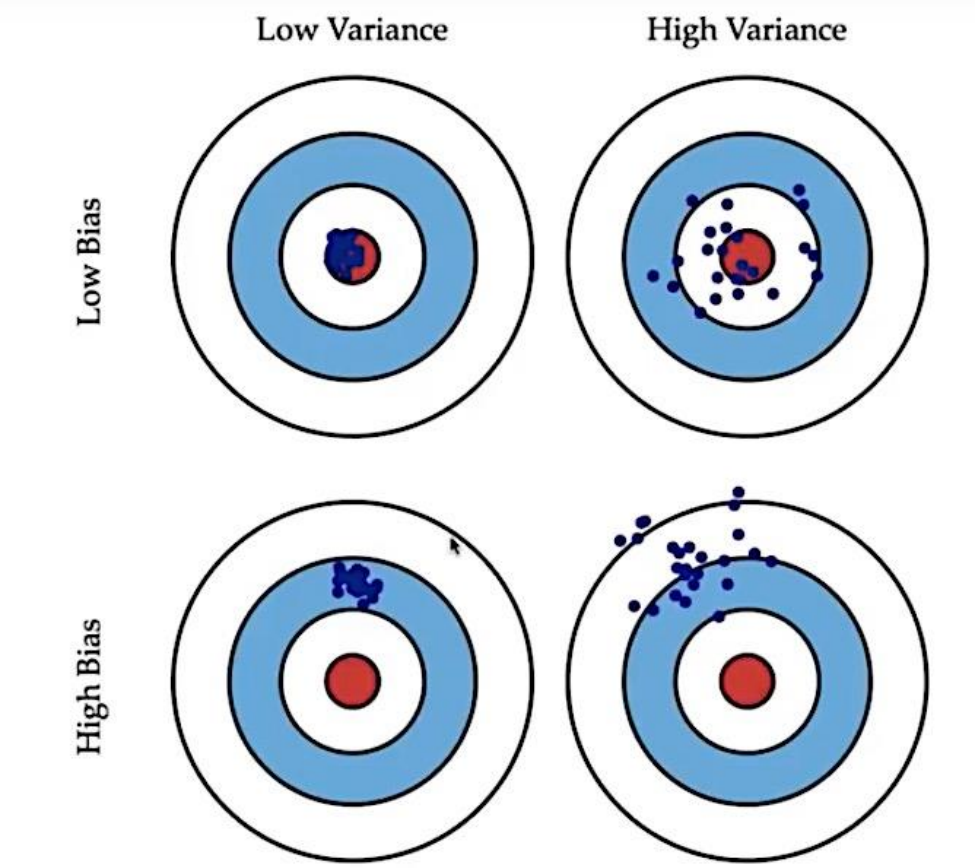
The derivation of the bias–variance decomposition for squared error proceeds as follows.

$$\begin{aligned}
\mathrm{MSE} = \mathrm{E}[(\mathrm{f(x)} - \hat{y})^2] &= E[(f(x) - E(\hat{y}) + E(\hat{y}) - \hat{y})^2] \\
&= E[(f(x) - E(\hat{y}) + E(\hat{y}) - \hat{y})(f(x) - E(\hat{y}) + E(\hat{y}) - \hat{y})] \\
&= E[f(x)^2 - f(x)E(\hat{y}) + f(x)E(\hat{y}) - f(x)\hat{y} - E(\hat{y})f(x) + E(\hat{y})^2 - E(\hat{y})^2 + E(\hat{y})\hat{y} \\
&+ E(\hat{y})f(x) - E(\hat{y})^2 + E(\hat{y})^2 - E(\hat{y})\hat{y} - \hat{y}f(x) + \hat{y}E(\hat{y}) - \hat{y}E(\hat{y}) + \hat{y}^2 \\
&= E[\hat{y}^2 - 2E(\hat{y})\hat{y} + E(\hat{y})^2] + E[E(\hat{y})^2 - 2E(\hat{y})f(x) + f(x)^2] \\
&+ E[f(x)E(\hat{y}) - f(x)\hat{y} - E(\hat{y})^2 + E(\hat{y})\hat{y} + E(\hat{y})f(x) - E(\hat{y})^2 - \hat{y}f(x) + \hat{y}E(\hat{y})] \\
&= E\left[\left(\hat{y} - E(\hat{y})\right)^2\right] \ldots\ldots Varience \\
&+ E\left[\left(E(\hat{y}) - f(x)\right)^2 \ldots\ldots Bias + f(x)E(\hat{y}) - f(x)E(\hat{y}) - E(\hat{y})^2 + E(\hat{y})E(\hat{y}) + E(\hat{y})f(x)\right. \\
&\left. - \mathrm{E}(\hat{y})^2 - E(\hat{y})f(x) + E(\hat{y})E(\hat{y})\right] \\
&= Varience + Bias
\end{aligned}$$

If we make $\mathrm{f(x)} \rightarrow \mathrm{f(x)} + \varepsilon$

We will get: $\mathrm{E}[(\mathrm{E}(y) - \mathrm{f(x)} - \varepsilon)^2] = E[\left(E(\hat{y}) - f(x)\right)^2 - 2\varepsilon(E(\hat{y}) - f(x) + \varepsilon^2]$

$$= E[(E(\hat{y}) - f(x)^2] + 0 + E(\varepsilon^2)$$



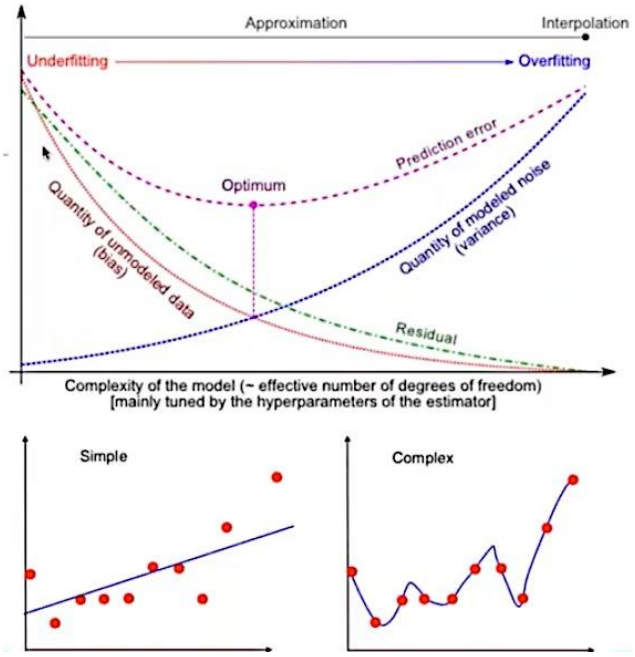
- Error due to bias: How far off a model is on average
- Error due to variance: How inconsistent models are if you repeat the regression

Above is a nice visual example that helps to explain this concept. For low bias and low variance, this is generally when you have a highly predictable system and good observations. High bias but low variance means you are getting consistent results and there is a significant error between the true target. For high variance but low bias, see the low bias if you average all these you get in the neighborhood of the true solution. But high variance means every time you calculate a model, you are getting radically different answers.

2.2 Under & Over-fitting

In machine learning, Overfitting refers to a model that describes the training data too well. Underfitting refers to a model that can neither model the training data nor generalize to new data.

- Under fitting = high bias
- Over fitting = high variance
- Address over fitting:
    1. Reduce number of features
    2. Regularization



In the real world, we neither like bias or variance, but we can' t live in a perfect world where we have an extremely low bias and variance. So we are going to set an optimum point there which can really balance the trade-off between bias and variance, and the optimum point is shown in above picture. There are two diagrams at the bottom we gave on the last section. Comparing these two models, we think the simple model is probably better. Simple model tends to be better predictive of outcome, because there is always a balance. Complex model gets more risk that we have an overfitting.

2.3 Linear Regression (Analytical Solution)

$$\begin{aligned} \text{error} &= (y - x\omega)'(y - x\omega) \\ &= y'y - (x\omega)'y - y'(x\omega) + (x\omega)'(x\omega) \\ &= \omega'(x'x)\omega - 2y'x\omega \end{aligned}$$

derivative ω should be zero, ignore y′ y (not related with ω.)

$$[(x\omega)'y = \omega'x'y' = y'x\omega]$$
$$A = x'x, b = -2x'y,$$
$$\frac{\partial error}{\partial \omega} = (A + A')\omega + b = 0,$$
$$(x'x + (x'x)')\omega - 2x'y = 0,$$
$$x'x\omega = x'y,$$
$$\omega = (x'x)^{-1}x'y$$

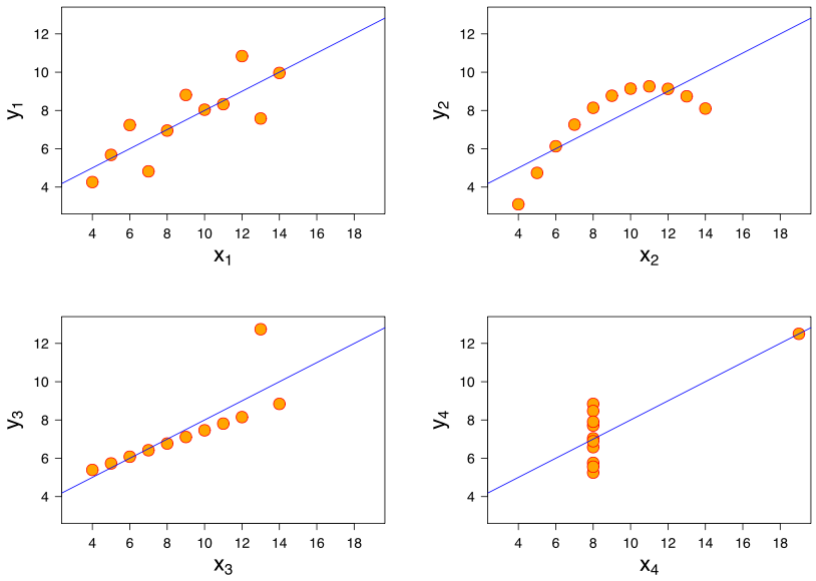
2.4 Linear Regression (Derivative only)

$$\text{error} = \sum_j \left(y_j - \sum_i x_{ij}\omega_i \right)^2, (\, i: all\ features, j: all\ points.\,),$$

$$Derivative = \frac{\partial error}{\partial \omega_i} = \sum_j 2\left(y_j - \sum_i x_{ij}\omega_i \right)(-x_{ij})$$
$$= -2\sum_j (\Delta y_j) x_{ij},$$
$$\Delta y_i = y_i - \sum_i x_{ij}\omega_i \,(different\ between\ actual\ \&\ prediction)$$

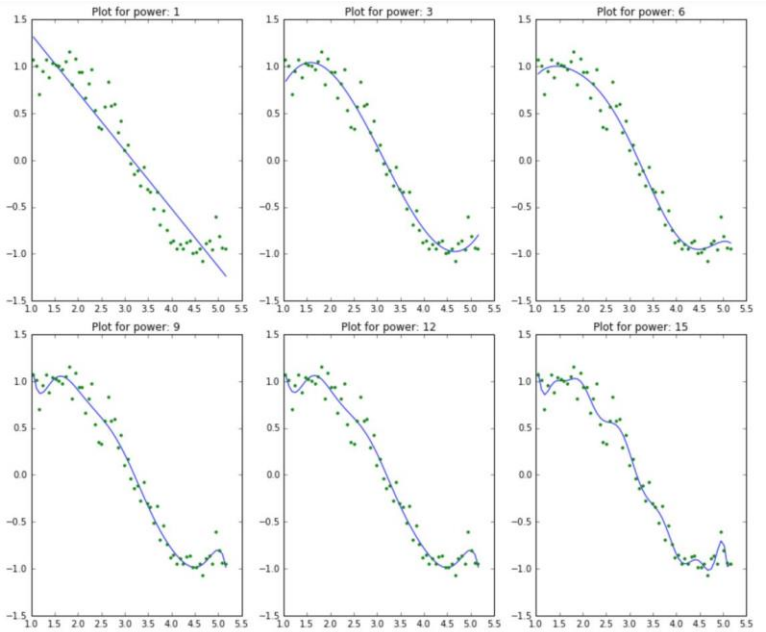
2.5 Linear Regression (Possible Problem)

These datasets have exactly the same main statistical properties (mean of x = 9, variance = 11, mean of y = 7.5, correlation = 0.816 and same linear regression line. However, these datasets are VERY different if we visualize them in a two dimensional space because of outliers. Although the grading of linear regression equals o, and the ω is optimal, the result of your model may not optimal.


y1
x1
y2
x2
y3
x3
y4
x4
4
6
8
10
12
14
16
18

2.6 Linear Regression (Overfitting)

This clearly aligns with our initial understanding. As the model complexity increases, the models tends to fit even smaller deviations in the training data set. Though this leads to overfitting, lets keep this issue aside for some time and come to our main objective, i.e. the impact on the magnitude of coefficients. This can be analysed by looking at the data frame created above.



2.7 Linear regression (why need regularization)

It is clearly evident that the size of coefficients increase exponentially with increase in model complexity. I hope this gives some intuition into why putting a constraint on the magnitude of coefficients can be a good idea to reduce model complexity.

	rss	intercept	coef_x_1	coef_x_2	coef_x_3	coef_x_4	coef_x_5	coef_x_6	coef_x_7	coef_x_8	coef_x_9	coef_x_10	coef_x_11	c
model_pow_1	3.3	2	-0.62	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
model_pow_2	3.3	1.9	-0.58	-0.006	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
model_pow_3	1.1	-1.1	3	-1.3	0.14	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
model_pow_4	1.1	-0.27	1.7	-0.53	-0.036	0.014	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
model_pow_5	1	3	-5.1	4.7	-1.9	0.33	-0.021	NaN	NaN	NaN	NaN	NaN	NaN	N
model_pow_6	0.99	-2.8	9.5	-9.7	5.2	-1.6	0.23	-0.014	NaN	NaN	NaN	NaN	NaN	N
model_pow_7	0.93	19	-56	69	-45	17	-3.5	0.4	-0.019	NaN	NaN	NaN	NaN	N
model_pow_8	0.92	43	-1.4e+02	1.8e+02	-1.3e+02	58	-15	2.4	-0.21	0.0077	NaN	NaN	NaN	N
model_pow_9	0.87	1.7e+02	-6.1e+02	9.6e+02	-8.5e+02	4.6e+02	-1.6e+02	37	-5.2	0.42	-0.015	NaN	NaN	N
model_pow_10	0.87	1.4e+02	-4.9e+02	7.3e+02	-6e+02	2.9e+02	-87	15	-0.81	-0.14	0.026	-0.0013	NaN	N
model_pow_11	0.87	-75	5.1e+02	-1.3e+03	1.9e+03	-1.6e+03	9.1e+02	-3.5e+02	91	-16	1.8	-0.12	0.0034	N
model_pow_12	0.87	-3.4e+02	1.9e+03	-4.4e+03	6e+03	-5.2e+03	3.1e+03	-1.3e+03	3.8e+02	-80	12	-1.1	0.062	-(
model_pow_13	0.86	3.2e+03	-1.8e+04	4.5e+04	-6.7e+04	6.6e+04	-4.6e+04	2.3e+04	-8.5e+03	2.3e+03	-4.5e+02	62	-5.7	0
model_pow_14	0.79	2.4e+04	-1.4e+05	3.8e+05	-6.1e+05	6.6e+05	-5e+05	2.8e+05	-1.2e+05	3.7e+04	-8.5e+03	1.5e+03	-1.8e+02	1
model_pow_15	0.7	-3.6e+04	2.4e+05	-7.5e+05	1.4e+06	-1.7e+06	1.5e+06	-1e+06	5e+05	-1.9e+05	5.4e+04	-1.2e+04	1.9e+03	-:

3 Regression with Regularization

When we deal with data in machine learning, we always combine the regression with regularization. Or we can say that regularization is an extension to regression. Such as extend the cost function from regular RSS to RSS + extra terms:
Total cost = measure of fit + measure of magnitude of coefficients

3.1 Ridge Regression

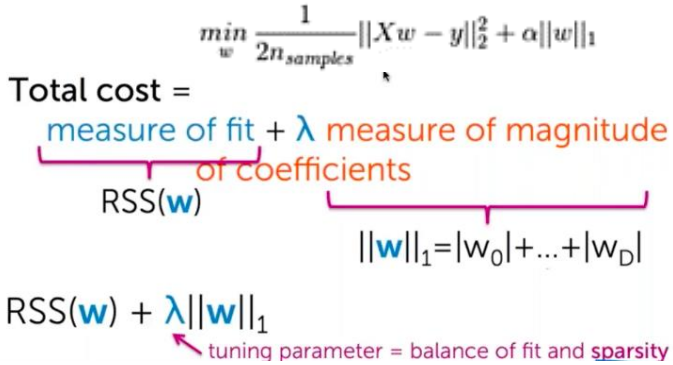
we may get overfitting when we solve problems, so we need remedy it with regularization. Ridge regression is one of the common regularization method. Ridge regression is like least squares but shrinks the estimated coefficients towards zero. The ridge regression has the same old measure of fit and it has this constraint on the magnitude of the coefficients. Here is the formal minimization definition.

The equation of ridge regression is:

$$\min_{w} ||Xw - y||_2^2 + \alpha||w||_2^2$$

3.2 Lasso Regression

There is another regularization method called Lasso Regression. The formal minimization definition of Lasso can be:

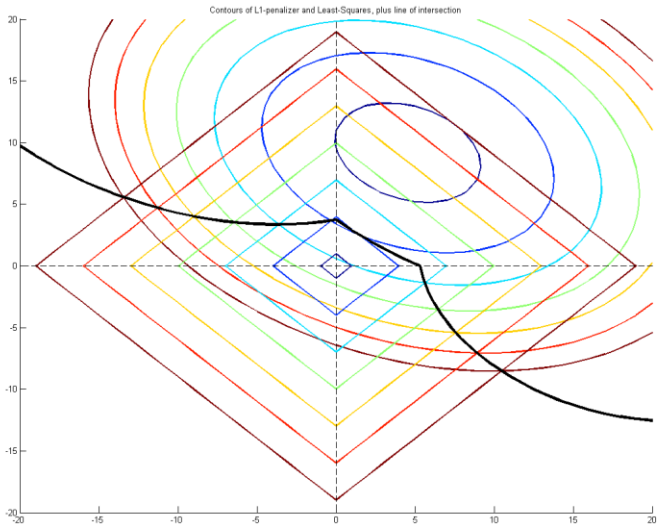


In statistics and machine learning, lasso (least absolute shrinkage and selection operator) (also Lasso or LASSO) is a regression analysis method that performs both variable selection and

regularization in order to enhance the prediction accuracy and interpretability of the statistical model it produces. The regularization term is the absolute value of coefficients.

3.3 Why zero for Lasso?

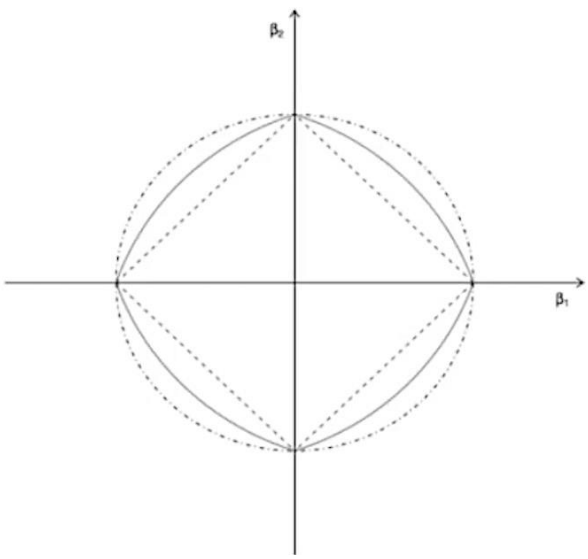
We can see that as we increased the value of alpha, coefficients were approaching towards zero, but if you see in case of lasso, even at smaller alpha' s, our coefficients are reducing to absolute zeroes. Therefore, lasso selects the only some feature while reduces the coefficients of others to zero.



3.4 Elastic net

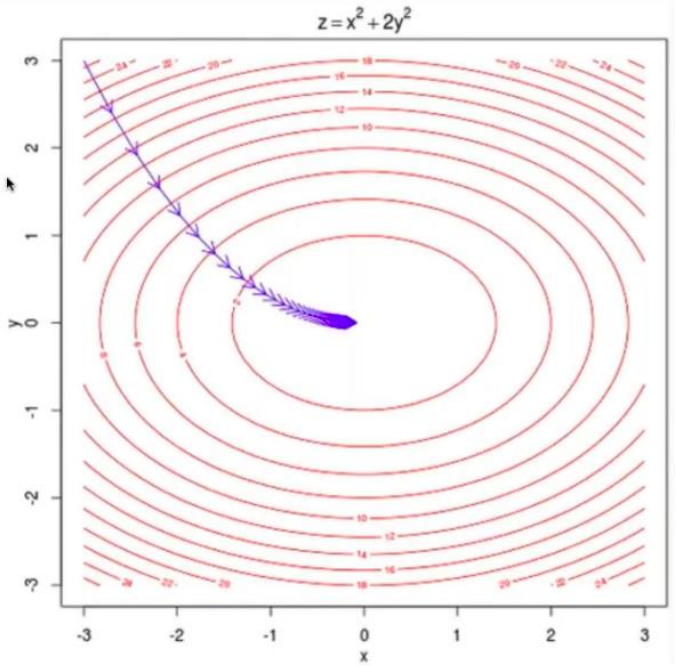
Eleatic Net is a regularized regression method that linearly combines the L1 and L2 penalties of the lasso and ridge methods. The elastic net method overcomes the limitations of the LASSO method which uses a penalty function based on:

$$\bar{\beta} = \underset{\beta}{argmin}\left(||y - X\beta||^2 + \lambda_2||\beta||^2 + \lambda_1||\beta||_1\right)$$



Above figure is a two-dimensional contour plot which shows the contour of the ridge penalty, contour of the LASSO penalty and contour of the elastic net penalty with α=0.5. We can see that singularities at the vertices and the edges are strictly convex, and the strength of convexity varies with α.

3.5 Gradient Descendent



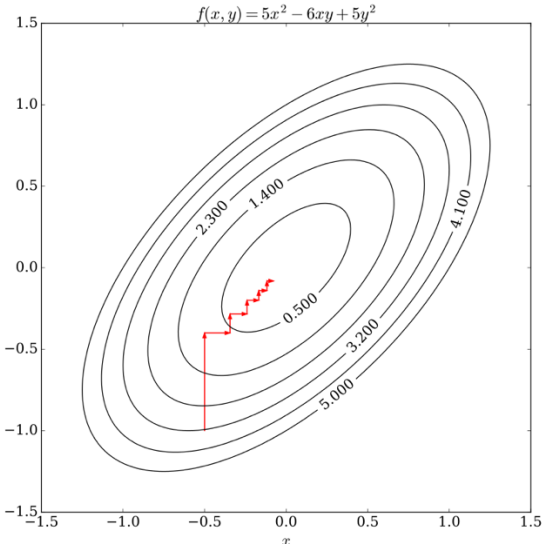
Gradient descent is a first-order iterative optimization algorithm. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or of the approximate gradient) of the function at the current point. If one takes step

proportional to the positive of the gradient instead, he will approach a local maximum of that function. This procedure is then known as gradient ascent.

3.6 Coordinated Descendent

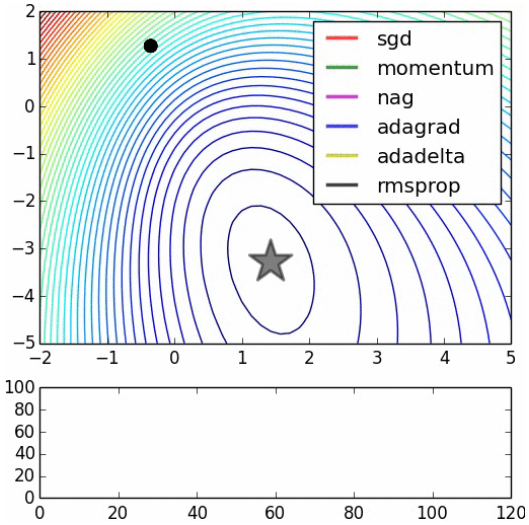
Coordinate descent is a derivative-free optimization algorithm. To find a local minimum of a function, one does line search along one coordinate direction at the current point in each iteration. One uses different coordinate directions cyclically throughout the procedure.

- Choose an initial parameter vector X.
- Until convergence is reached, or for some fixed number of iterations:
- Choose an index I from 1 to n.
- Choose a step size α.
- Update x_i to $x_i - \alpha \frac{\partial F}{\partial x_i}(X)$



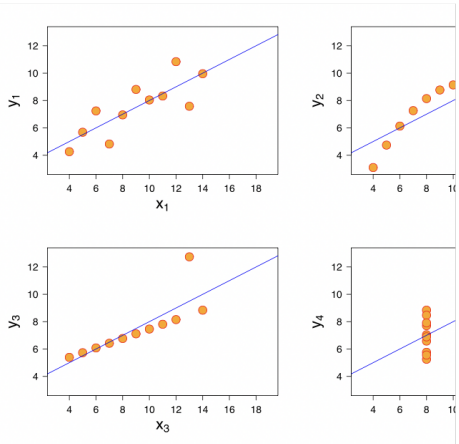
3.7 Stochastic Gradient Descendent

Stochastic gradient descent (often shortened in SGD), also known as incremental gradient descent, is a stochastic approximation of the gradient descent optimization method for minimizing an objective function that is written as a sum of differentiable functions.



3.8 Random Sample Consensus (RANSAC)

```
iterations = 0
bestfit = nul
besterr = something really large
while iterations < k {
    maybeinliers = n randomly selected values from data
    maybemodel = model parameters fitted to maybeinliers
    alsoinliers = empty set
    for every point in data not in maybeinliers {
        if point fits maybemodel with an error smaller than t
             add point to alsoinliers
    }
    if the number of elements in alsoinliers is > d {
        % this implies that we may have found a good model
        % now test how good it is
        bettermodel = model parameters fitted to all points in maybeinliers and alsoinliers
        thiserr = a measure of how well model fits these points
        if thiserr < besterr {
            bestfit = bettermodel
            besterr = thiserr
        }
    }
    increment iterations
}
return bestfit
```



It is robust in the sense of good tolerance to outliers in the experimental data. It is capable of interpreting and smoothing data containing a significant percentage of gross errors. The estimate is only correct with a certain probability, since RANSAC is a randomized estimator. The algorithm has been applied to a wide range of model parameters estimation problems in computer vision, such as feature matching, registration or detection of geometric primitives.

4 Examples in Python

4.1 Bias and Variance

In [3]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

%matplotlib inline
```

In [ ]:
```
# the physical law says: S = 1/2 * F / M * t*2
# S: displacement, M: mass, F: force
# Let's say we know nothing about physics
# Given this data measure points:
# (t, S) for N points, we want to predict for (t_new) what is S_new
```

In [5]:
```
ground_truth_coef = 1./2 * 10 / 2
t = np.random.random(10) * 3
ground_truth_result = ground_truth_coef * t**2

# measure error
measure_result = ground_truth_result + np.random.randn(10)*2

t_new = np.random.random(5) * 3
# s_new
```

In [6]:
```
from sklearn import linear_model, metrics
lm_lr = linear_model.LinearRegression()
```

In [7]:
```
# Under fitting, what is the error? (using 1 order)
# we need to run it over multple times to see the difference
N = 10000
error = []
error_train = []
for i in range(N):
    t = np.random.random(10) * 3
    ground_truth_result = ground_truth_coef * t**2
    measure_result = ground_truth_result + np.random.randn(10)*2
    t.resize([len(t), 1])

    lm_lr.fit(t, measure_result)

    t_new = (np.random.random(5) * 3).reshape(-1, 1)
    t_truth = ground_truth_coef * t_new[:,0]**2
    t_pred = lm_lr.predict(t_new)

    error.append(metrics.mean_squared_error(t_pred, t_truth))
    #error_train.append(metrics.mean_squared_error(t, measure_result))

error = np.array(error)
#error_train = np.array(error_train)
```

In [8]:
```
#print error_train.mean(), error_train.std()
print (error.mean(), error.std())
```

```
5.08799581363 6.31522601554
```

In [9]:
```
# Over fitting, what is the error? (using 3 order)
N = 10000
error = []
for i in range(N):
    t = (np.random.random(10) * 3).reshape([-1, 1])
    ground_truth_result = ground_truth_coef * t**2
    measure_result = ground_truth_result +
    (np.random.randn(10)*2).ipynb_checkpoints.reshape([-1,1])
    t = np.hstack([t, t**2, t**3])

    lm_lr.fit(t, measure_result)

    t_new = (np.random.random(5)).reshape([-1, 1])
    t_new = np.hstack([t_new, t_new**2, t_new**3])
    t_truth = ground_truth_coef * t_new[:,0]**2
    t_pred = lm_lr.predict(t_new)

    error.append(metrics.mean_squared_error(t_pred, t_truth))
    #error_train.append(metrics.mean_squared_error(t[:,0], measure_result)


error = np.array(error)
#error_train = np.array(error_train)
```

In [10]:
```
print (error.mean(), error.std())
```

```
104.390906118 6909.96024482
```

In [11]:
```
# Over fitting, what is the error? (using 2 order)
N = 10000
error = []
for i in range(N):
    t = (np.random.random(10) * 3).reshape([-1, 1])
    ground_truth_result = ground_truth_coef * t**2
    measure_result = ground_truth_result +
    (np.random.randn(10)*2).reshape([-1,1])
    t = np.hstack([t, t**2])

    lm_lr.fit(t, measure_result)

    t_new = (np.random.random(5)).reshape([-1, 1])
    t_new = np.hstack([t_new, t_new**2])
    t_truth = ground_truth_coef * t_new[:,0]**2
    t_pred = lm_lr.predict(t_new)

    error.append(metrics.mean_squared_error(t_pred, t_truth))

error = np.array(error)
```

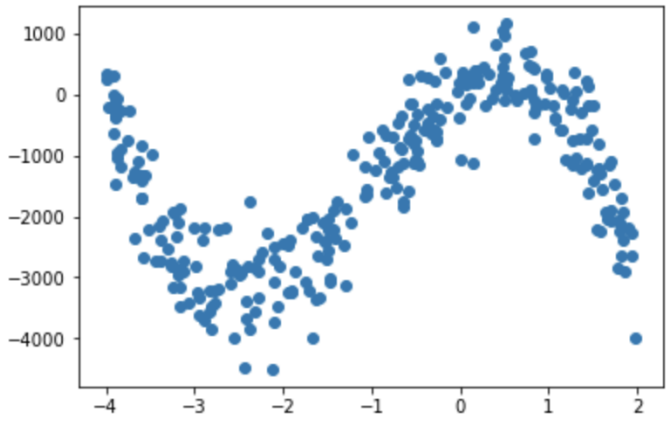
In [12]:
```
print (error.mean(), error.std())
```

```
3.08755144418 16.7023638307
```

In [ ]:
```
# Balance Varience & Bias
np.random.seed(1)
x = (np.random.random([30, 1]) * 6 - 4).ravel()
y = -2*(5*x)**3 - 30 * (5*x) **2 + 100*(10*x) + 500 *
np.random.randn(len(x))
plt.scatter(x,y)
plt.show()
```

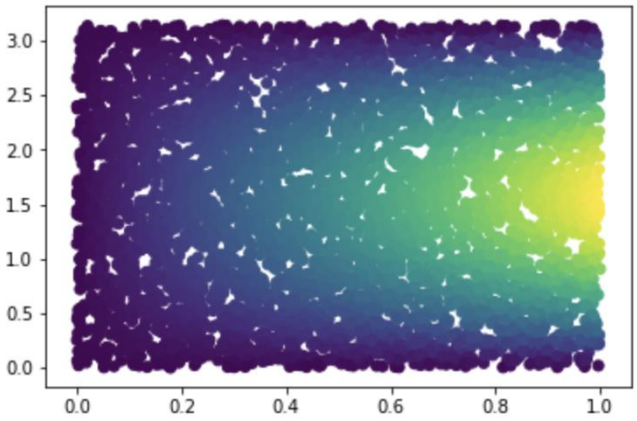
4.2 Bias & Variance: Model Complexity

In [15]:
```
# Data set 1.
N = 300
np.random.seed(1)
x = (np.random.random([N, 1]) * 6 - 4).ravel()
y = -2*(5*x)**3 - 30 * (5*x) **2 + 100*(10*x) + 500 *
np.random.randn(len(x))
plt.scatter(x,y)
plt.show()
```



In [16]:
```
# Data set 2.
N = 5000
r = np.random.rand(N)
theta = np.random.rand(N) * 1.0 * np.pi
y = np.sin(theta) * r
plt.scatter(r, theta, c=y)
```

Out[16]:
```
<matplotlib.collections.PathCollection at 0x1114ac630>
```



4.3 Bias & Variance: Data Size

In [21]:
```
from sklearn import datasets, cross_validation
data = datasets.make_regression(n_samples=1000,
                                n_features=100, n_informative=100,
                                noise=100)
X = data[0]
Y = data[1]
print (X.shape, Y.shape)
```

```
(1000, 100) (1000,)
```

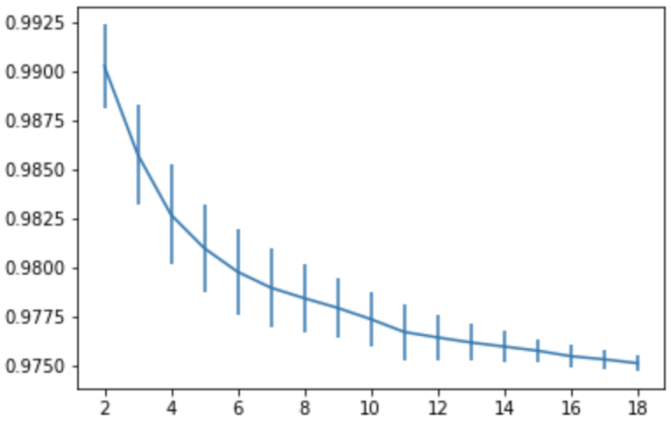
```
/Users/peter/anaconda/lib/python3.5/site-packages/sklearn/cross_validatio
n.py:44: DeprecationWarning: This module was deprecated in version 0.18 i
n favor of the model_selection module into which all the refactored class
es and functions are moved. Also note that the interface of the new CV it
erators are different from that of this module. This module will be remov
ed in 0.20.
  "This module will be removed in 0.20.", DeprecationWarning)
```

In [22]:
```
ts_range = np.arange(0.05, 1, 0.05)
N_run = 50
t_score = []
v_score = []
for train_s in ts_range:
    t_score.append([])
    v_score.append([])
    for iseed in range(N_run):
        X1, X2, Y1, Y2 = cross_validation.train_test_split(X, Y,
                        train_size=train_s, random_state=iseed)
        rlm = linear_model.LinearRegression()
        rlm.fit(X1, Y1)
        t_score[-1].append(metrics.r2_score(Y1, rlm.predict(X1)))
        v_score[-1].append(metrics.r2_score(Y2, rlm.predict(X2)))
    # print train_s
```

In [23]:
```
t_score = np.array(t_score)
v_score = np.array(v_score)
```

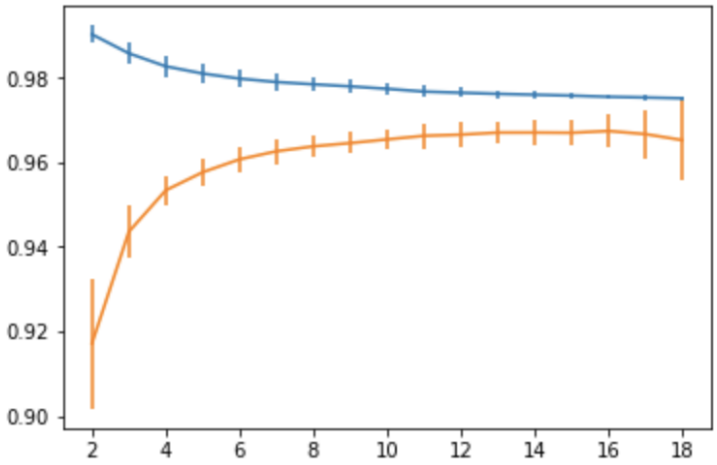
In [24]:
```
# errorbar plot (training and validation)
fig = plt.figure()
ax = fig.add_subplot(1,1,1)

x = range(t_score.shape[0])[2:]
y = t_score.mean(axis=1)[2:]
s = t_score.std(axis=1)[2:]
ax.errorbar(x, y, yerr=s)
fig.show()
```

```
/Users/peter/anaconda/lib/python3.5/site-packages/matplotlib/figure.py:402: UserWarning: matplotlib is currently using a non-GUI backend, so cannot show the figure
  "matplotlib is currently using a non-GUI backend, "
```



In [25]:
```
# errorbar plot (training and validation)
fig = plt.figure()
ax = fig.add_subplot(1,1,1)

x = range(t_score.shape[0])[2:]
y = t_score.mean(axis=1)[2:]
s = t_score.std(axis=1)[2:]
ax.errorbar(x, y, yerr=s)
x = range(v_score.shape[0])[2:]
y = v_score.mean(axis=1)[2:]
s = v_score.std(axis=1)[2:]
ax.errorbar(x, y, yerr=s)
fig.show()
```

```
/Users/peter/anaconda/lib/python3.5/site-packages/matplotlib/figure.py:402: UserWarning: matplotlib is currently using a non-GUI backend, so cannot show the figure
  "matplotlib is currently using a non-GUI backend, "
```

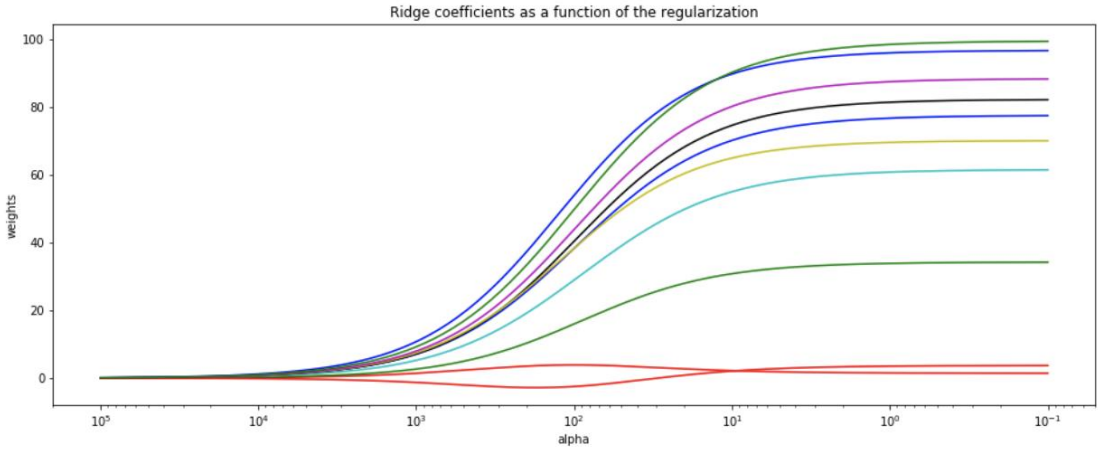


4.4 Regularization

In [16]:
```
from sklearn import datasets
# X is the 10x10 Hilbert matrix
X = 1. / (np.arange(1, 11) + np.arange(0, 10)[:, np.newaxis])
y = np.ones(10)
data = datasets.make_regression(n_samples=100,
                                n_features=10, n_informative=10,
                                random_state=0)

X = data[0]
y = data[1]

##############################################################################
# Compute paths

n_alphas = 200
alphas = np.logspace(-1, 5, n_alphas)
clf = linear_model.Ridge()
```

```
coefs = []
for a in alphas:
    clf.set_params(alpha=a)
    clf.fit(X, y)
    coefs.append(clf.coef_)

##############################################################################
# Display results
plt.figure(figsize=[16, 6])

ax = plt.gca()
ax.set_color_cycle(['b', 'r', 'g', 'c', 'k', 'y', 'm'])

ax.plot(alphas, coefs)
ax.set_xscale('log')
ax.set_xlim(ax.get_xlim()[::-1])  # reverse axis
plt.xlabel('alpha')
plt.ylabel('weights')
plt.title('Ridge coefficients as a function of the regularization')
plt.axis('tight')
plt.show()
```

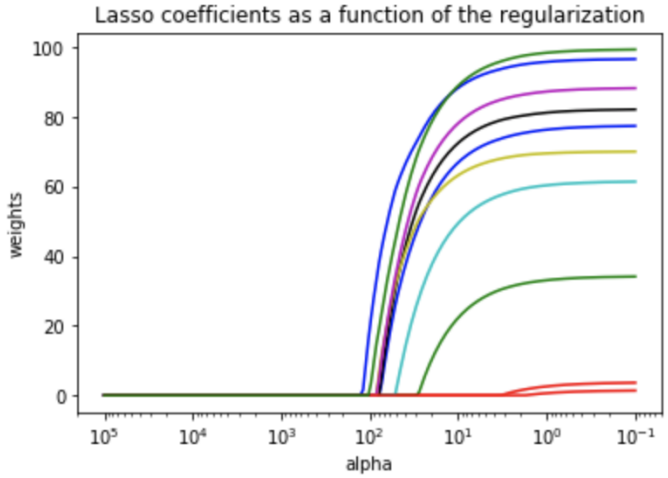
```
/Users/peter/anaconda/lib/python3.6/site-packages/matplotlib/cbook.py:136
: MatplotlibDeprecationWarning: The set_color_cycle attribute was depreca
ted in version 1.5. Use set_prop_cycle instead.
  warnings.warn(message, mplDeprecation, stacklevel=1)
```



In [17]:
```
##############################################################################
# Compute paths

n_alphas = 200
alphas = np.logspace(-1, 5, n_alphas)
clf = linear_model.Lasso()

coefs = []
for a in alphas:
    clf.set_params(alpha=a)
    clf.fit(X, y)
    coefs.append(clf.coef_)
```

```
##############################################################################
# Display results

ax = plt.gca()
ax.set_color_cycle(['b', 'r', 'g', 'c', 'k', 'y', 'm'])

ax.plot(alphas, coefs)
ax.set_xscale('log')
ax.set_xlim(ax.get_xlim()[::-1])  # reverse axis
plt.xlabel('alpha')
plt.ylabel('weights')
plt.title('Lasso coefficients as a function of the regularization')
plt.axis('tight')
plt.show()
```

```
/Users/peter/anaconda/lib/python3.6/site-packages/matplotlib/cbook.py:136
: MatplotlibDeprecationWarning: The set_color_cycle attribute was depreca
ted in version 1.5. Use set_prop_cycle instead.
  warnings.warn(message, mplDeprecation, stacklevel=1)
```

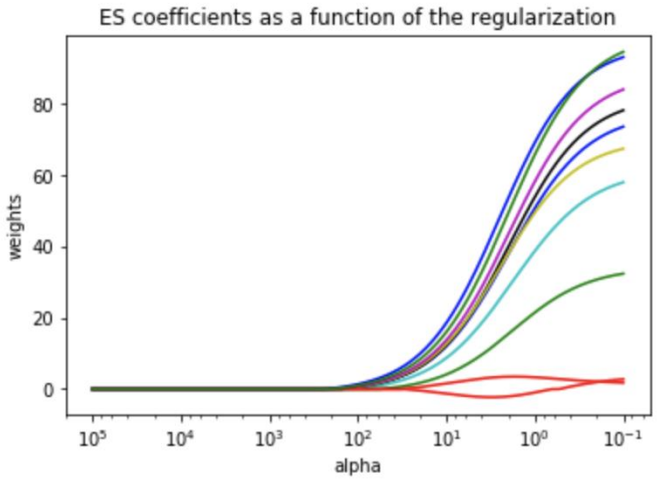


In [28]:

```
##############################################################################
# Compute paths

n_alphas = 200
alphas = np.logspace(-1, 5, n_alphas)
clf = linear_model.ElasticNet(l1_ratio=0.5)

coefs = []
for a in alphas:
    clf.set_params(alpha=a)
    clf.fit(X, y)
    coefs.append(clf.coef_)
```

```
##############################################################################
# Display results

ax = plt.gca()
ax.set_color_cycle(['b', 'r', 'g', 'c', 'k', 'y', 'm'])

ax.plot(alphas, coefs)
ax.set_xscale('log')
ax.set_xlim(ax.get_xlim()[::-1])  # reverse axis
plt.xlabel('alpha')
plt.ylabel('weights')
plt.title('ES coefficients as a function of the regularization')
plt.axis('tight')
plt.show()
```

```
/Users/peter/anaconda/lib/python3.5/site-packages/matplotlib/cbook.py:136
: MatplotlibDeprecationWarning: The set_color_cycle attribute was depreca
ted in version 1.5. Use set_prop_cycle instead.
  warnings.warn(message, mplDeprecation, stacklevel=1)
```



4.5 Stochastic Gradient Descent (SGD)

In [35]:
```
data = datasets.make_regression(n_samples=1000, n_features=100,
                                n_informative=100)
X = data[0]
y = data[1]
```

In [36]:
```
# compare efficiency and accuracy for SGD and tradition LR
```

In [37]:
```
%%time
lrg1 = linear_model.LinearRegression()
lrg1.fit(X, y)
```

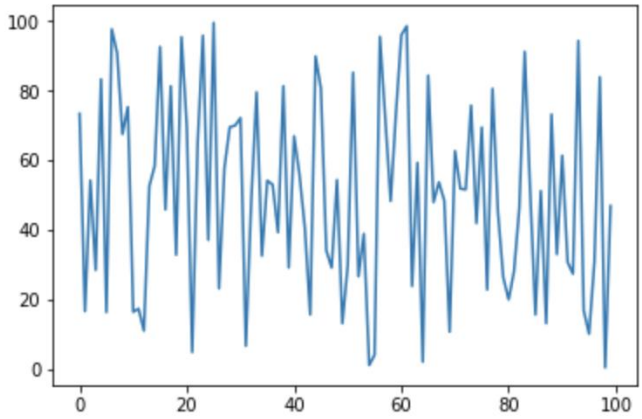
```
CPU times: user 15.9 ms, sys: 2 ms, total: 17.9 ms
Wall time: 6.16 ms
```

In [39]:
```
%%time
lrg2 = linear_model.SGDRegressor()
lrg2.fit(X,y)
```

```
CPU times: user 3.18 ms, sys: 934 µs, total: 4.12 ms
Wall time: 2.95 ms
```

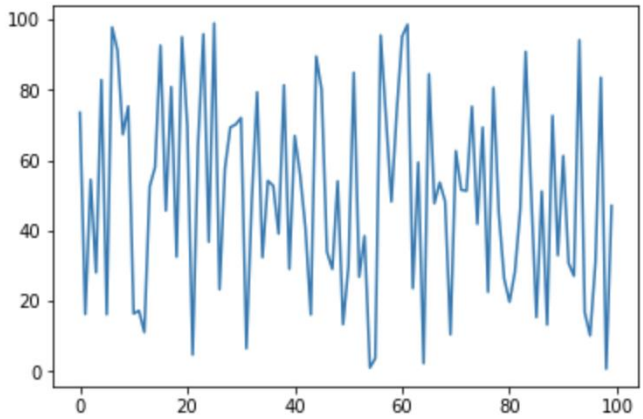
In [41]:
```
%matplotlib inline
plt.plot(lrg1.coef_)
```

Out[41]:
```
[<matplotlib.lines.Line2D at 0x11a5b2828>]
```



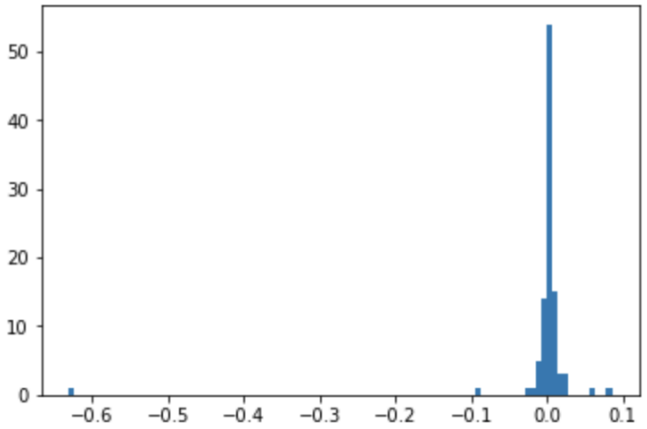
In [42]:
```
plt.plot(lrg2.coef_)
```

Out[42]:
```
[<matplotlib.lines.Line2D at 0x1193fddd8>]
```



In [43]:
```
a = (lrg1.coef_ - lrg2.coef_) / lrg1.coef_

plt.hist(a[(a>-10) & (a<10)], bins=100)
```

Out[43]:
```
(array([  1.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   1.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   1.,   1.,   5.,  14.,
         54.,  15.,   3.,   3.,   0.,   0.,   0.,   0.,   1.,   0.,   0.,
          1.]),
 array([-0.6310778 , -0.62391628, -0.61675475, -0.60959322, -0.60243169,
        -0.59527017, -0.58810864, -0.58094711, -0.57378558, -0.56662406,
```



4.6 Preprocessing

In [44]:
```
from sklearn import datasets, preprocessing
```

In [45]:
```
X, Y = datasets.make_regression(n_samples=1000, n_features=2,
                                n_informative=2)
```

In [46]:
```
X1 = X
X2 = X.copy()
X2[:,1] *= 1000
```

In [47]:
```
from sklearn import linear_model
```

In [48]:
```
m1 = linear_model.LinearRegression()
m1.fit(X1, Y)

m2 = linear_model.LinearRegression()
m2.fit(X2, Y)
```

Out[48]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=Fal
se)
```

In [49]:
```
m1.coef_
```

Out[49]:
```
array([ 61.44968143,  15.06919954])
```

In [50]:
```
m2.coef_
```

Out[50]:
```
array([  6.14496814e+01,   1.50691995e-02])
```

In [51]:
```
m1 = linear_model.SGDRegressor()
m1.fit(X1, Y)

m2 = linear_model.SGDRegressor()
m2.fit(X2, Y)
```

Out[51]:
```
SGDRegressor(alpha=0.0001, average=False, epsilon=0.1, eta0=0.01,
       fit_intercept=True, l1_ratio=0.15, learning_rate='invscaling',
       loss='squared_loss', n_iter=5, penalty='l2', power_t=0.25,
       random_state=None, shuffle=True, verbose=0, warm_start=False)
```

In [52]:
```
m1.coef_, m1.intercept_
```

Out[52]:
```
(array([ 61.4189541 ,  15.05734222]), array([ 0.00189971]))
```

In [53]:
```
from sklearn import metrics
```

In [54]:
```
metrics.mean_squared_error(m1.predict(X1), Y)
```

Out[54]:
```
0.001064673883092252
```

In [55]:
```
m2.coef_, m2.intercept_
```

Out[55]:
```
(array([  1.14820626e+09,   1.71603668e+12]), array([ -6.94941825e+10]))
```

In [56]:
```
metrics.mean_squared_error(m2.predict(X2), Y)
```

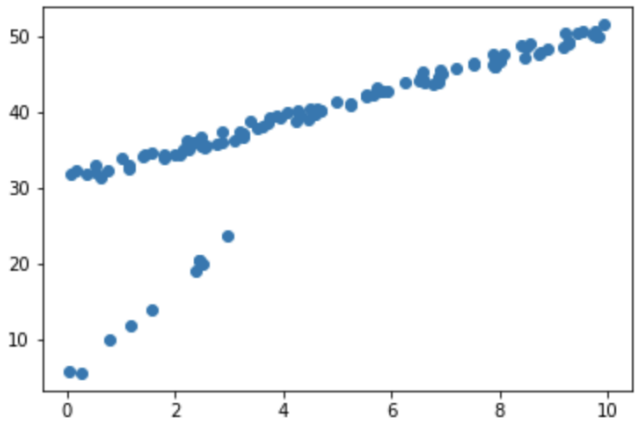
Out[56]:
```
2.7794647006979491e+30
```

4.7 Robustness Regression

In [57]:
```
# Why robustness is needed?

x = np.random.random(100) * 10
y = 2.0 * x + 30 + 2.0 * np.random.rand(len(x))

xn = np.random.random(10) * 3
yn = 6.0 * xn + 4 + 2.0 * np.random.rand(len(xn))

xnew = np.concatenate([x,xn])
ynew = np.concatenate([y,yn])

xnew = xnew.reshape([-1,1])
```

In [59]:
```
plt.scatter(xnew, ynew)
```

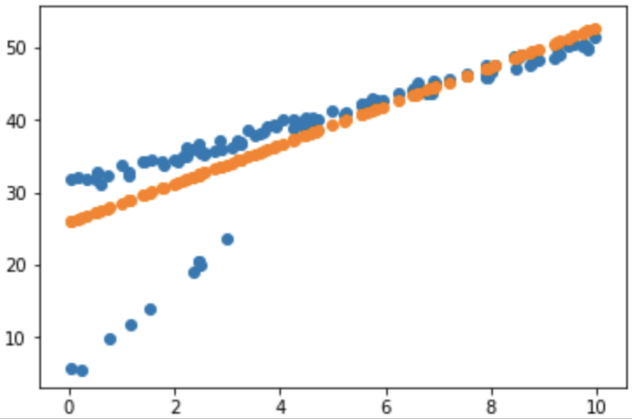
Out[59]:
```
<matplotlib.collections.PathCollection at 0x11afb0a58>
```



In [60]:
```
lm_lr = linear_model.LinearRegression()
```

In [61]:
```
lm_lr.fit(xnew, ynew)
y_pred = lm_lr.predict(xnew)
plt.scatter(xnew, ynew)
plt.scatter(xnew, y_pred)
```

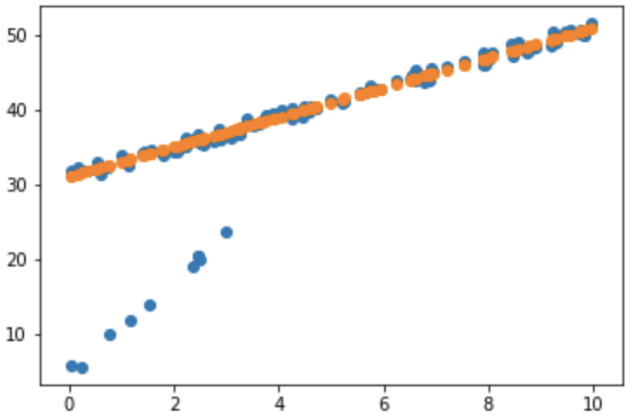
Out[61]:
```
<matplotlib.collections.PathCollection at 0x11afcf080>
```



In [62]:
```
# linear_model.RANSACRegressor?
lm_lr = linear_model.RANSACRegressor()
```

In [63]:
```
lm_lr.fit(xnew, ynew)
y_pred = lm_lr.predict(xnew)
plt.scatter(xnew, ynew)
plt.scatter(xnew, y_pred)
```

Out[63]:
```
<matplotlib.collections.PathCollection at 0x11b3b0748>
```


50
40
30
20
10
0
2
4
6
8
10