Machine Learning Project_1

Shun Zhang, student ID:15300180012

October 9, 2017

1 Linear Regression and Nonlinear Bases

1.1 Simple Linear Regression

Now we have some observations, which we call attributes, $\mathbf{x} = (x_1, x_2, x_3, ..., x_k)^T$ and we want to predict a certain value y, which we call target. Simple linear regression is such a thing that we assume there is a linear relationship between $\mathbf{x}$ and y.

$$y = w_1x_1 + w_2x_2 + ... + w_kx_k = \mathbf{w}^T\mathbf{x} \tag{1}$$

$\mathbf{w}$ is the coefficient column-vector and our goal is to find a 'good enough' $\mathbf{w}$. So we should find a way to define how good the $\mathbf{w}$ is. In this article, all we talk about is the **Least Squares**. Euqtion(1) is about a single sample and we extend it to the multi-sample case.

$$\mathbf{y} = \mathbf{X}\mathbf{w} \tag{2}$$

Here we denote $\mathbf{X}$ as the attributes' matrix($\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n)^T$), of which the row is a sample and the column is an attribute. Also we denote $\mathbf{y}^*$ as our prediction, and what we are interested in is the **error function** of (2).

$$\begin{aligned} E(w) &= \frac{1}{2}(\mathbf{y}^* - \mathbf{y})^T(\mathbf{y}^* - \mathbf{y}) \\ &= \frac{1}{2}\sum_{i=1}^{n}(y_i - \mathbf{w}^T\mathbf{x}_i)^2 \\ E'(w) &= \sum_{i=1}^{n}\mathbf{x}_i(y_i - \mathbf{w}^T\mathbf{x}_i) = 0 \end{aligned}$$

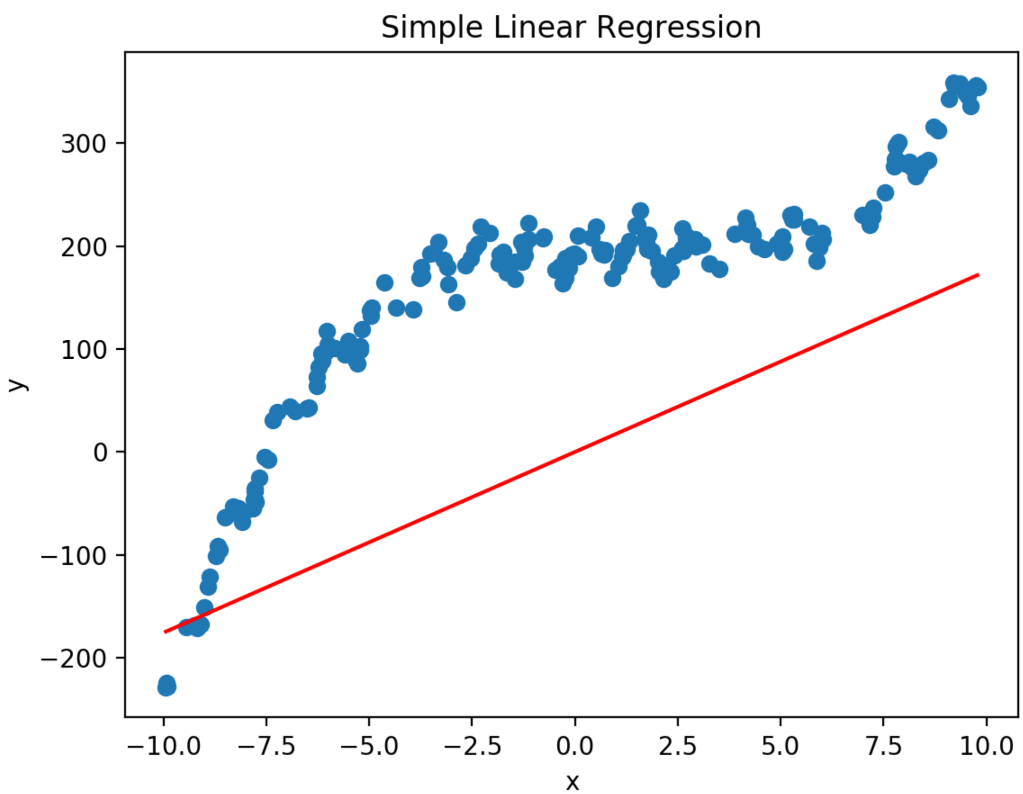
Then we have

$$\begin{aligned} \mathbf{w} &= (\sum_{i=1}^{n}\mathbf{x}_i\mathbf{x}_i^T)^{-1}\sum_{i=1}^{n}\mathbf{x}_iy_i \\ &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \end{aligned} \tag{3}$$

If the attribute is only one-dimension(as in the project), then equation (3) will be simplified as

$$w = \frac{\sum_{i=1}^{n} x_i y_i}{\sum_{i=1}^{n} x_i^2} \tag{4}$$

So, with the training data $\mathbf{X}$ and $\mathbf{y}$, we can fit the linear model and get $\mathbf{w}$. Then we can predict with equation (2). As in this project, we can see that simple linear regression shows a poor result intuitively.



1.2 Adding a Bias Variable

Since the simple linear regression shows a poor result. We can conclude from the picture above that the y-intercept of this data is not zero, so we should improve the model's performance by adding a bias variable, also called the dumb variable and the model will be modified as

$$y_i = \mathbf{w}^T \mathbf{x}_i + \beta \tag{5}$$

So the new error function is

$$E(w) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \beta - \mathbf{w}^T\mathbf{x}_i)^2 \tag{6}$$

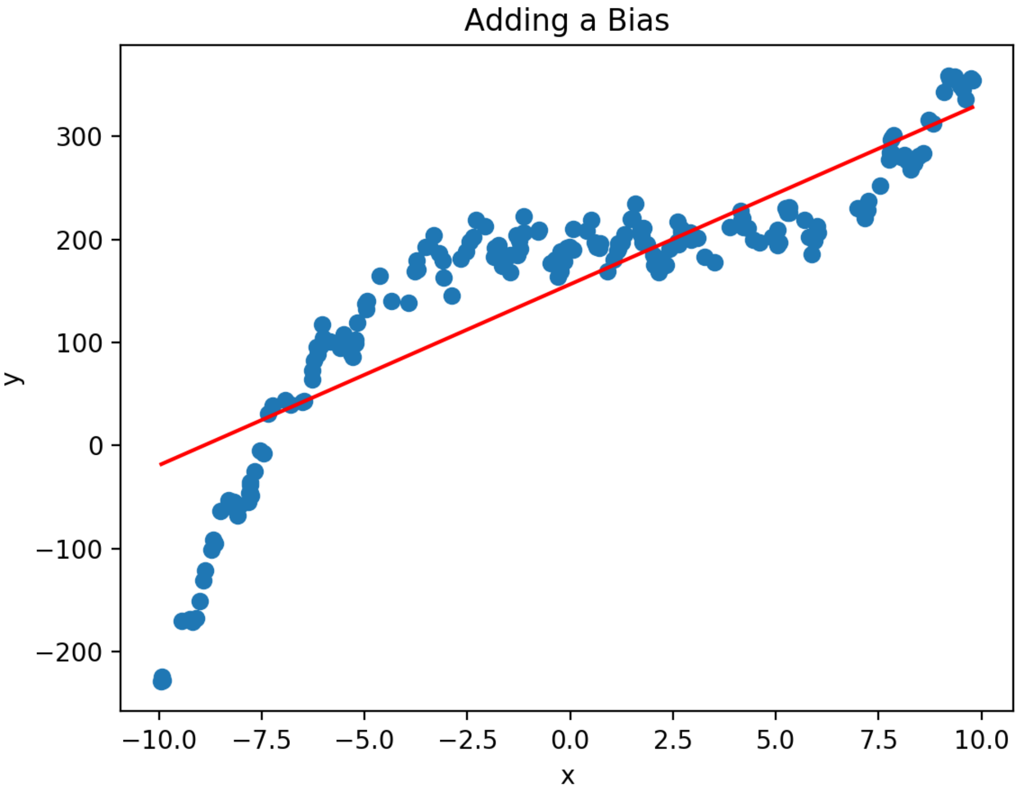
$$E'(w)|_\beta = \sum_{i=1}^{n}(y_i - \beta - \mathbf{w}^T\mathbf{x}_i) = 0$$

$$\beta = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{w}^T\mathbf{x}_i) \tag{7}$$

With equation (4) and (7), we can now fit the model by training data.

```
    def fit(self, train_set_x, train_set_y):
        """
        This is a least_square_bias regression model.
        :param train_set_x:pd.Series x
        :param train_set_y:pd.Series y as targets
        :return:a vector w including bias w_0
        """
        self.w = (sum(train_set_x*train_set_y))/(sum(train_set_x*train_set_x)
            )

        self.bias = (sum(train_set_y)-self.w*sum(train_set_x))/len(
            train_set_x)
```

Then we can have the new fit line as follow



And here we use **average squared** training and test error, which turns out to be 3657.6 and 3338.1 respectively.

1.3 Polynomial Basis

From the result just mentioned above we could see that the data set apparently doesn't go after a linear basis(a line in the case of 2-dimension). So here we apply a non-linear basis, which is the polynomial basis in this case, hoping to get a better result with lower average squared error. The model is

$$y_i = \sum_{m=1}^{k} w_m x_i^m + \beta$$

If we denote β as w_0 then we have

$$\begin{aligned} y_i &= \sum_{m=0}^{k} w_m x_i^m \\ &= (1, x_i, x_i^2, ..., x_i^k)\mathbf{w} \end{aligned} \tag{8}$$

The **design matrix** for polynomial basis is

$$\mathbf{X}_{poly-k} = \begin{pmatrix} 1 & x_1 & \dots & (x_1)^k \\ 1 & x_2 & \dots & (x_2)^k \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \dots & (x_n)^k \end{pmatrix} \tag{9}$$

where k is the degree of equation (8). Then we apply it to the *normal equations* for the least squares and we have

$$\mathbf{w} = (\mathbf{X}_{poly-k}^T \mathbf{X}_{poly-k})^{-1} \mathbf{X}_{poly-k}^T \mathbf{y} \tag{10}$$

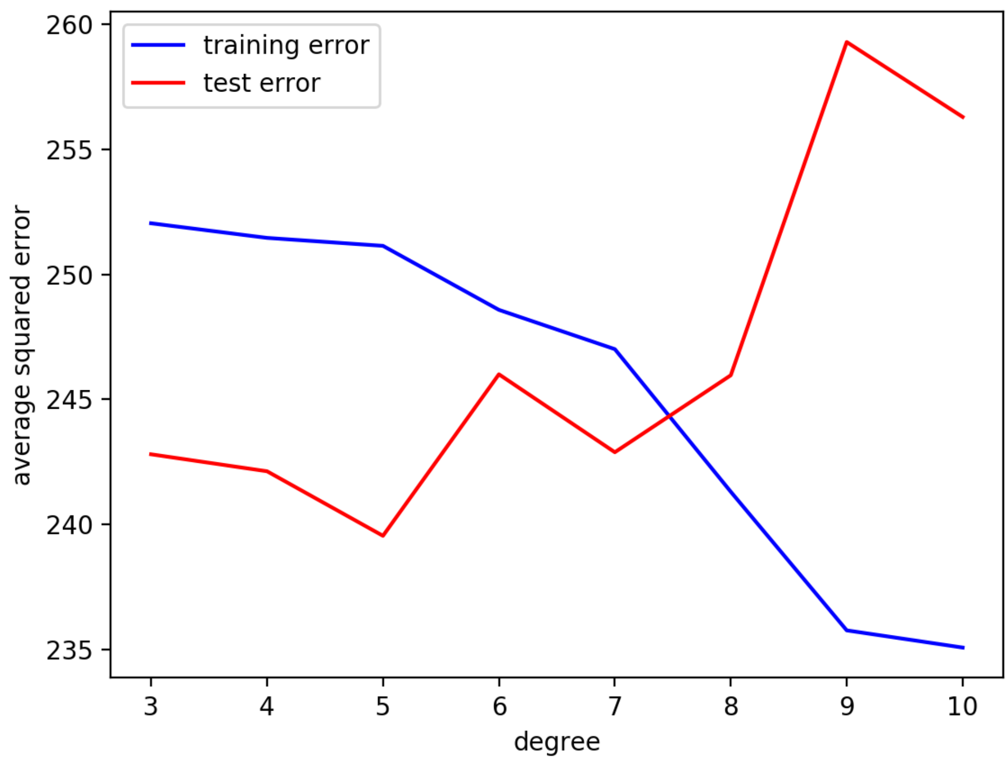
which has the similar format as (3), where $\mathbf{X}$ is the design matrix of simple linear regression. We can now calculate $\mathbf{w}$ with equation (10).

```
def least_squares_basis(x, y, deg):
    """
    This is a polynomial regression.
    :param x: training set x (pd.Series)
    :param y: target set y (pd.Series)
    :param deg: the degree of the polynomial (int)
    """
    y = y.values.reshape(-1, 1)

    x_mat = []
    for d in range(deg+1):
        add_ = x ** d
        x_mat.append(add_)

    x_mat = np.matrix(x_mat).T

    w = (x_mat.T * x_mat)**-1 * (x_mat.T * y)
```

Here is the table of average squared training and test error from degree 0 to 10.

degree	0	1	2	3	4	5	6	7	8	9	10
training error	15480.5	3551.1	2168.0	252.0	251.5	251.1	248.6	247.0	241.3	235.8	235.0
test error	14390.8	3393.9	2480.7	242.8	242.1	239.5	246.0	242.8	246.0	259.3	256.3

From this table we can see that the training error goes down straightly as the degree increases. However the test error first decreases and then increases as the degree becomes much larger(as we can see intuitively as follow), which can be interpreted as the phenomenon of overfitting.



2 Regularization

Though the overfitting above is slight, it can be very serious sometimes. In order to overcome such phenomenon, we now introduce the idea of regularization.

2.1 Data Standardization

Now that we have more than one attribute, eight actually, we should modify our input data to make different attribute comparable.

$$x_{ij} = \frac{x_{ij} - \bar{x}_j}{\sigma_j} \tag{11}$$

where x_{ij} is denoted as the j-th attribute of the i-th sample and $\bar{x}_j$ and σ_j are denoted as the j-th attribute's mean and standard deviation respectively.
Then we ramdonly shuffle the input data and choose the first 50 samples as training set and the rest as test set.

```
slide_ = list(range(len(target_data)))
np.random.shuffle(slide_)
train_slide = slide_[:50]
test_slide = slide_[50:]
```

2.2 Ridge Regression

We will now construct a model using ridge regression to predict the 9th variable as a linear combination of the other 8. The ridge method is a regularized version of least squares, with error function:

$$E(\theta) = ||\mathbf{y} - \mathbf{X}\theta||_2^2 + \delta^2||\theta||_2^2 \tag{12}$$

$$E'(\theta) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\theta) + 2\delta^2\theta = 0$$

$$\theta = (\delta^2\mathbf{I}_d + \mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{13}$$

where d is the number of attributes(eight in this case).
So when δ^2 is given, we can calculate θ according to (13).

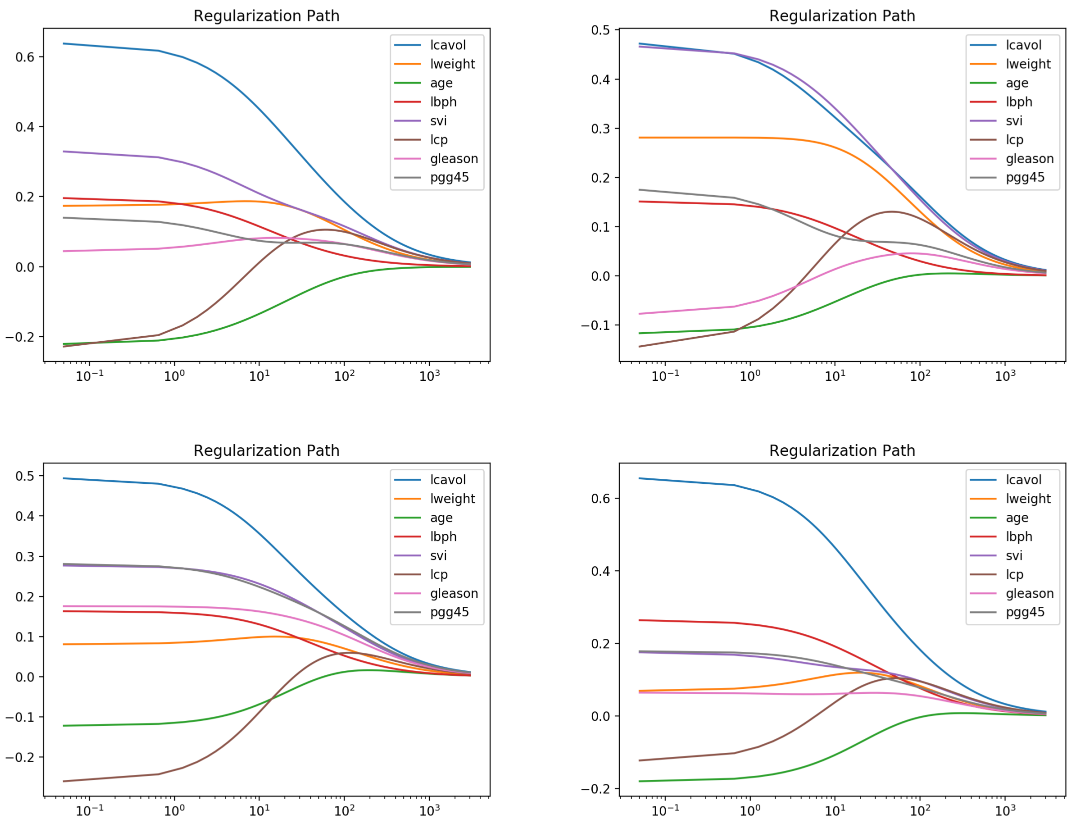
```
def ridge(x, y, d2):
    """
    This is a ridge regression function given delta^2 as d2
    :param x: attributes (matrix)
    :param y: target (array)
    :param d2: the ridge regression hypo-parameter
    :return:a 8-dim theta list
    """
    theta = (d2 * np.identity(len(x.T)) + x.T * x)**-1 * x.T
    theta = np.dot(theta, y)

    return theta
```

And we now draw a **regularization path** according to different δ^2 and θ of all eight input attributes.

```
regularization_path = [[] for i in range(len(names))]
for d2 in np.linspace(0.05, 3000, 5000):
    theta_new = ridge(np.matrix(train_attr), np.array(train_target), d2)
    for i in range(len(names)):
        regularization_path[i].append(theta_new[0, i])

plt.axes(xscale='log')
for i in range(len(names)):
    plt.plot(np.linspace(0.05, 3000, 5000), regularization_path[i])
```

```
plt.legend(names)
plt.show()
```

As in the code, we calculate different δ^2 from 0.05 to 3000 with 5000 steps and the regularization path is shown as follow.



Because of the **ramdomly shuffle** during the data processing, we will get different result every time we run the code. However, they share the same tendency that all of the eight coefficient converges to zero when δ^2 tends to infinity.

2.3 Cross-Validation

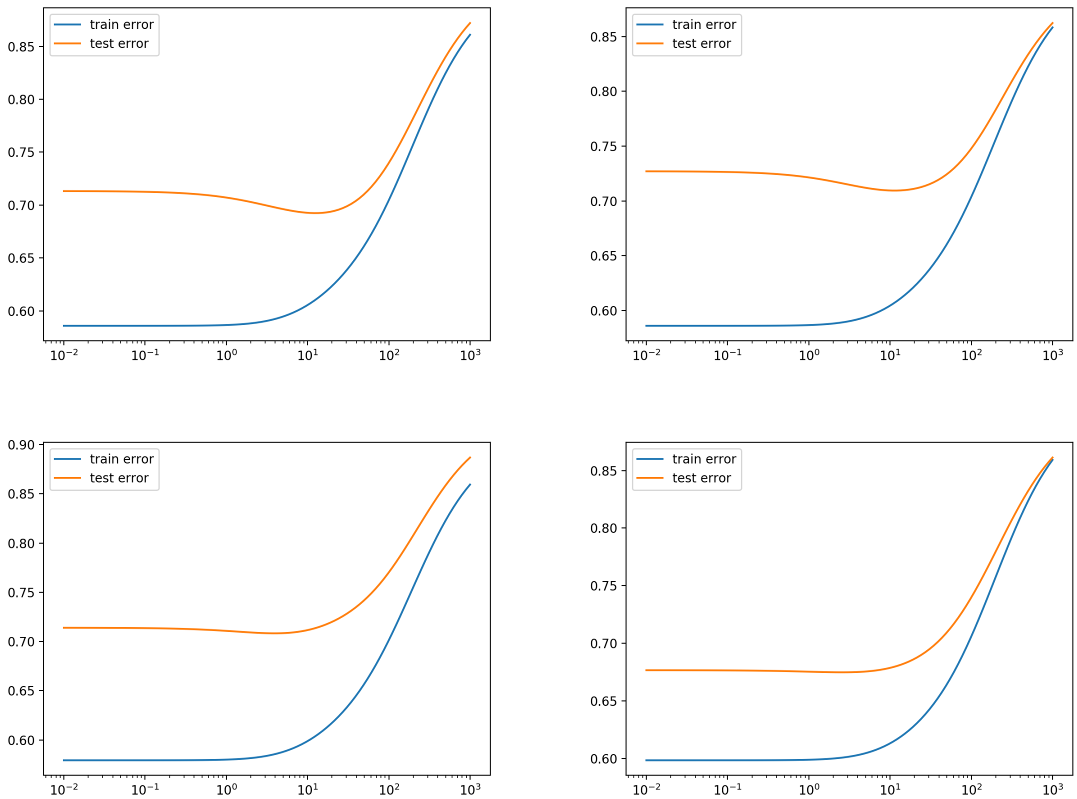
Here we use **five-fold cross-validation** to choose the best value of δ^2 according to the training and test error calculated by (14).

$$E(\theta) = \frac{||\mathbf{y} - \mathbf{X}\theta||_2}{||\mathbf{y}||_2} \tag{14}$$

```python
# first we will standardize the import data
# then we will fit a ridge regression model
# here we apply ten-fold CV to find the best delta

import numpy as np
import matplotlib.pyplot as plt


def load_data():...

# a matrix of attributes
attr_data = load_data()[0]

# an array of target
target_data = load_data()[1]

# separate the data set into ten parts
slide_ = list(range(len(target_data)))
np.random.shuffle(slide_)
k_fold = 5  # the times of k-fold CV


def get_attr(slide):...


def get_target(slide):...

# data processing
def standardize_data(array, mean, std):
    array = (array - mean)/std
    return array


def ridge(x, y, d2):
    """
    This is a ridge regression function given delta^2 as d2
    :param x: attributes (matrix)
    :param y: target (array)
    :param d2: the ridge regression hypo-parameter
    :return:a 8-dim theta list
    """
    theta = (d2 * np.identity(len(x.T)) + x.T * x)**-1 * x.T
```

```
    theta = np.dot(theta, y)

    return theta


def test(x, y, theta_):...

test_mean_error = []
train_mean_error = []
delta2_set = np.logspace(-2, 3, 100)
for delta2 in delta2_set:
    test_error = []
    train_error = []
    # calculate error using 10-fold CV
    for i in range(k_fold):
        # create sets
        test_slide = slide_[20*i:20*(i+1)]
        train_slide = slide_[:20*i]+slide_[20*(i+1):]
        cv_train_attr = get_attr(train_slide)
        cv_test_attr = get_attr(test_slide)
        cv_train_target = get_target(train_slide)
        cv_test_target = get_target(test_slide)
        # standardize datas
        for j in range(len(cv_test_attr[0])):
            cv_train_attr[j] = standardize_data(cv_train_attr[j], np.mean(cv_train_attr[j]),
                                                np.std(cv_train_attr[j]))
            cv_test_attr[j] = standardize_data(cv_test_attr[j], np.mean(cv_train_attr[j]),
                                               np.std(cv_train_attr[j]))
        cv_train_target = standardize_data(cv_train_target, np.mean(cv_train_target),
                                           np.std(cv_train_target))
        cv_test_target = standardize_data(cv_test_target, np.mean(cv_test_target),
                                          np.std(cv_test_target))
        # fit
        theta = ridge(cv_train_attr, cv_train_target, delta2)
        # test
        train_error.append(test(cv_train_attr, cv_train_target, theta))
        test_error.append(test(cv_test_attr, cv_test_target, theta))
    train_mean_error.append(np.mean(train_error))
    test_mean_error.append(np.mean(test_error))

plt.axes(xscale='log')
plt.plot(delta2_set, train_mean_error)
plt.plot(delta2_set, test_mean_error)
plt.legend(['train error', 'test error'])
plt.show()
```

Also we have a lot of results because of **randomly shuffle**.



Some of them, like the top two, have an apparent extreme point within the test error line, while there are still some other ones, like the bottom two, show a week extreme point. Here we just make our conclusion by those apparent ones that the best value of δ^2 lies in the interval [10, 30].

It may be confusing that we only got an interval instead of a certain value. However, as for me, the order is the most important 'value'. That is to say, we've made great efforts just to achieve the right order of δ^2, which is about one in this case. So it is some kind of meaningless to get 'a certain value' of δ^2, which still requires a lot of work to be done.