Lasso, Ridge 이해하기 (with python)

캐글 rank 1위의 data scientist가 NYC academy에서 다음과 같은 말을 했다고 한다. 'regularization term 없이 적합을 하려면, 당신은 아주 특별한 사람이어야 되요!'('if you are using regression without regularization, you have to be very special!') 사실 ridge lasso 이후 L1 regular term, L2 regular term은 연구가 아주 많이 되어서 지금은 ols외에도 많이 사용되는 regularization term이다.

1. Brief Overview

변수가 너무 많아 OLS의 해가 존재하지 않는 경우($(X^TX)^{-1}$이 존재하지 않는 경우)나 OLS 값이 너무 불안정할 경우 우리는 shrink model을 사용한다. 이 경우 (선형가정이 맞을 경우)unbiased의 특성을 갖는 OLS에 비해 약간의 bias를 안게 되지만 효과적으로 variance를 줄이게 되어 모델의 성능을 높일 수 있게 된다.

Ridge와 Lasso의 최적화식은 다음과 같다.

< RIdge>

$$\sum_{i=1}^{n}(y_i-\beta_0-\beta_1x_{i1}-\ldots-\beta_px_{ip})^2+\lambda\sum_{j=1}^{p}\beta_j^2=RSS+\lambda\sum_{j=1}^{p}\beta_j^2$$

< Lasso >

$$\sum_{i=1}^{n}(y_i-\beta_0-\beta_1x_{i1}-\ldots-\beta_px_{ip})^2+\lambda\sum_{j=1}^{p}|\beta_j|=RSS+\lambda\sum_{j=1}^{p}|\beta_j|$$

각각 beta의 제곱, 혹은 절대값의 합만큼을 최적화 식에 포함하여 **계수들의 크기를 줄여주는(shrink)** 모델들이다. 그런데 계수의 크기를 줄여주는게 왜 의미가 있을까? y=4x^2+2x+3보다 y=2x^2+x+1.5가 더 좋은 모델인건 아닐텐데.

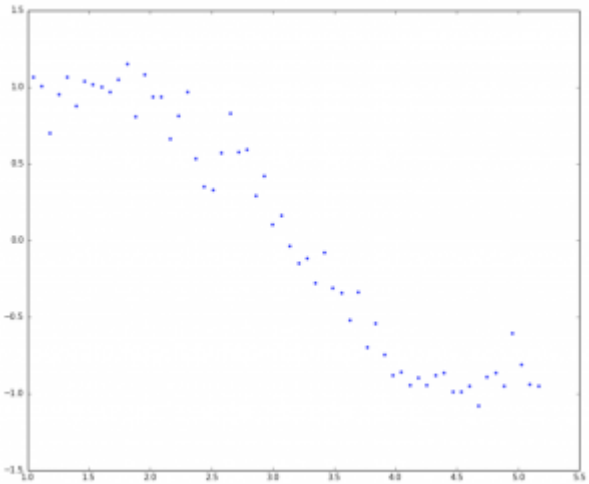
2. Why Penalize the Magnitude of Coefficients?

이에 대한 간단한 실험. (실습코드는 이 사이트를 그대로 도용햇습니다ㅠㅠ 제대로 못준비해 죄송합니다ㅠ)

60~300도의 범위에서 sign곡선에 random noise를 더해서 noisy data를 만들고, 여기에 **다항적합**을 하였다. 차수가 올라갈때마다(2차,3차,..) 1) R^2는 증가할테지만 2) 차수가 증가할수록 (혹은 변수가 많아질 수록) noise에 fitting하여 overfitting의 여지도 심해진다는 것은 명백한 사실일 것이다.

```
#Importing libraries. The same will be used throughout the article.
import numpy as np
import pandas as pd
import random
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib.pylab import rcParams

rcParams['figure.figsize'] = 12, 10
```

```
#Define input array with angles from 60deg to 300deg converted to radians
x = np.array([i*np.pi/180 for i in range(60,300,4)])
np.random.seed(10)  #Setting seed for reproducability
y = np.sin(x) + np.random.normal(0,0.15,len(x))
#sign곡선에 N(0,0.15^2)의 random noise를 추가
data = pd.DataFrame(np.column_stack([x,y]),columns=['x','y'])
plt.plot(data['x'],data['y'],'.')
```



그리고 이에 대해 1~15차 다항회귀 적합을 한다.

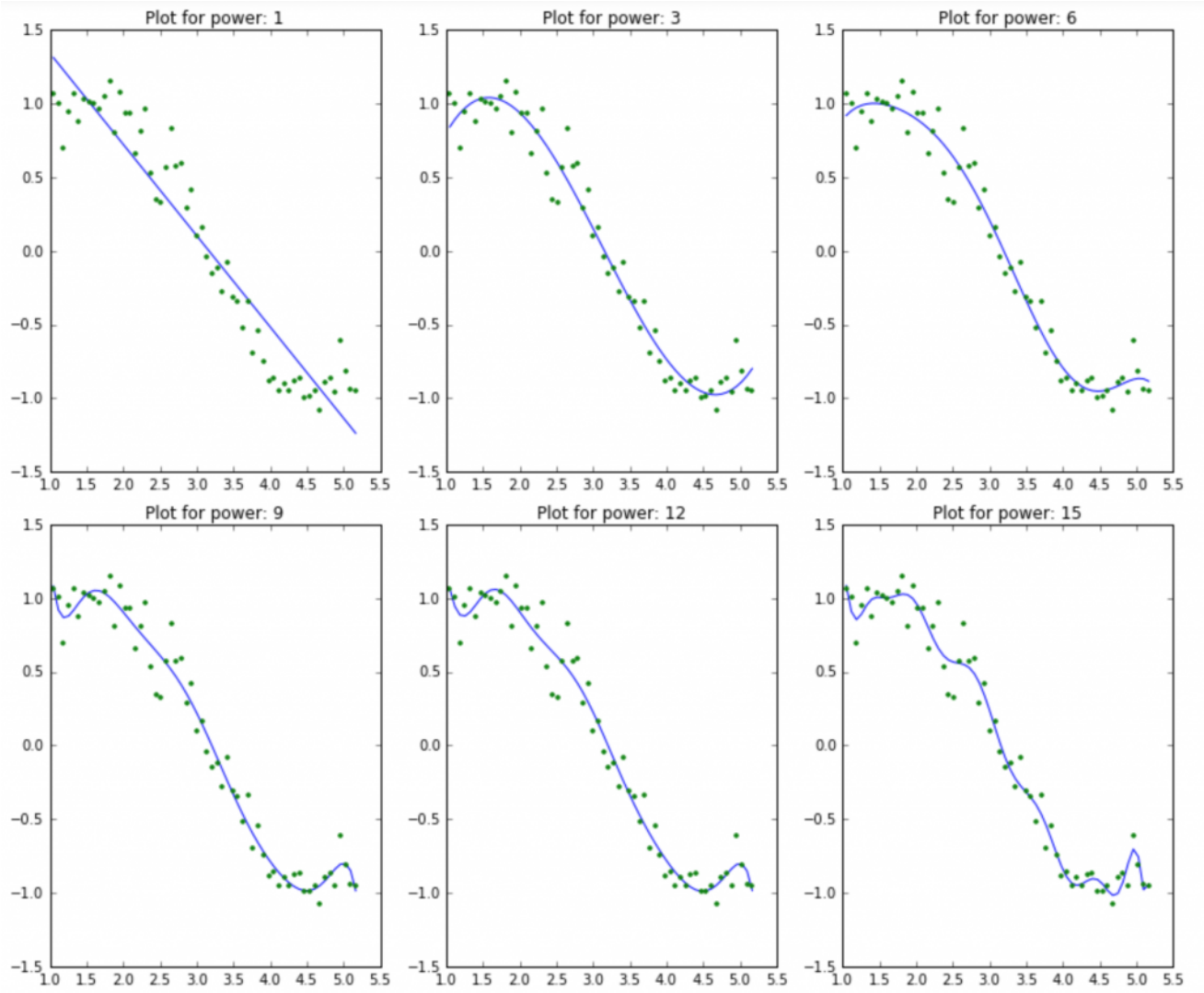
```
#1~15차를 가지고 있는 dataframe생성
for i in range(2,16):  #power of 1 is already there
    colname = 'x_%d'%i      #new var will be x_power
    data[colname] = data['x']**i
print data.head()
```

```
          x         y       x_2       x_3       x_4       x_5       x_6  \
0  1.047198  1.065763  1.096623  1.148381  1.202581  1.259340  1.318778
1  1.117011  1.006086  1.247713  1.393709  1.556788  1.738948  1.942424
2  1.186824  0.695374  1.408551  1.671702  1.984016  2.354677  2.794587
3  1.256637  0.949799  1.579137  1.984402  2.493673  3.133642  3.937850
4  1.326450  1.063496  1.759470  2.333850  3.095735  4.106339  5.446854

        x_7       x_8       x_9      x_10      x_11      x_12      x_13  \
0  1.381021  1.446202  1.514459  1.585938  1.660790  1.739176  1.821260
1  2.169709  2.423588  2.707173  3.023942  3.377775  3.773011  4.214494
2  3.316683  3.936319  4.671717  5.544505  6.580351  7.809718  9.268760
3  4.948448  6.218404  7.814277  9.819710 12.339811 15.506664 19.486248
4  7.224981  9.583578 12.712139 16.862020 22.366630 29.668222 39.353420

       x_14       x_15
0  1.907219   1.997235
1  4.707635   5.258479
2 11.000386  13.055521
3 24.487142  30.771450
4 52.200353  69.241170
```

```
#Import Linear Regression model from scikit-learn.
```

```
from sklearn.linear_model import LinearRegression
def linear_regression(data, power):
    #initialize predictors:
    predictors=['x']
    if power>=2:
        predictors.extend(['x_%d'%i for i in range(2,power+1)])
        #해당 차수만큼을 불러와서

    #Fit the model
    linreg = LinearRegression(normalize=True)
    linreg.fit(data[predictors],data['y'])#다항적합
    y_pred = linreg.predict(data[predictors])

    #Check if a plot is to be made for the entered power
    if power in models_to_plot:
        plt.subplot(models_to_plot[power])
        plt.tight_layout()
        plt.plot(data['x'],y_pred)
        plt.plot(data['x'],data['y'],'.')
        plt.title('Plot for power: %d'%power)

    #Return the result in pre-defined format
    rss = sum((y_pred-data['y'])**2)
    ret = [rss]
    ret.extend([linreg.intercept_])
    ret.extend(linreg.coef_)#rss와 각 계수들을 순서대로 저장
    return ret

#Initialize a dataframe to store the results:
#계수 저장할 빈 df생성
col = ['rss','intercept'] + ['coef_x_%d'%i for i in range(1,16)]
ind = ['model_pow_%d'%i for i in range(1,16)]
coef_matrix_simple = pd.DataFrame(index=ind, columns=col)


#Iterate through all powers and assimilate results
for i in range(1,16):#각 power에 대해 rss와 계수들을 저장
    coef_matrix_simple.iloc[i-1,0:i+2] = linear_regression(data, power=i)
```

대충 3~6차적합이 적당하고 그 이후로 갈수록 overfit같아 보인다.



그리고 저장된 계수들을 확인해보면

```
#Set the display format to be scientific for ease of analysis
pd.options.display.float_format = '{:,.2g}'.format
coef_matrix_simple
```

	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	-

계수들을 보면, 차수가 증가할때마다 계수의 크기가 exponentially increase하는것을 볼 수 있다.

변수가 많아질수록 noise에 fitting하게 된다. 우리가 원하지 않는 상황. 그를 위해 변수선택을하기도. 변수의 coeff를 중요도에 따른 weight라고 rough하게 볼때, (계수의 크기==weight는 다소 무리가 잇지만 multicolinearity가 없다는 가정하에서 변수들의 중요성으로 볼 수 있다) 기존의 선형회귀는 각 변수를 아무런 제약이 없이 바라보고(표현이 이상한데, 뒤에 specify한다), 우리의 train data를 잘 설명할 수 있는 변수에 weight를 주게 된다. 그리고 차수가 높아질수록(혹은 변수가 많아질수록) 모델의 자유도가 높아지고, 우리가 원하지 않는 작은noise를 잘 설명한다면 그 변수에도 큰 weight(=coefficient)를 주게 될것이다. 이런것을 바라지 않기에 우리가 shirnk모델을 쓴다. 즉 기본적으로 변수선택과 같은 것이다.

> best subset도 L0 regularization로 생각할"수도"잇다. 참고

> 물론 데이터가 무수히 많다면 noise는 noise로써 mitigate할 수 있을것이다. y=2x+1를 따르는 무한의 데이터에 4차 다항회귀를 적합하면 2차항 이상의 계수의 크기는 매우 줄어들 것이다. 그러나 우리는 무한의 데이터가 없다. mitigate시켜줄만한 충분한 데이터가 없기에 계수들이 불안정하게 날뛰는거라고 보면 된다.

> 또다른 설명으로는 선형대수적으로, X^TX의 inverse matrix를 구하는게 X가 singluar거나 nearly singular일때 문제가 된다.(n<p일때) 즉 mse의 conour가 분산이큰 형태로 약간의 데이터 변화에도(noise로 인해 자주 그러겟지) 모델이 역변한다. 모델의 분산은 가정이 얼마나 strict한지, 모델에게 자유를 얼마나 줫는지에 따라 달라진다. 이런 관점에서 X^TX에 diagonal에 람다를 더해주면, 분산이 줄어들게 되고 invertable하게 된다(고한다. 참고). 기하하적으로 보면 우리가 많이 보앗던 그림속에서 beta들의 존재가능한 space를 제한함으로써 variance를 낮춘다고도 볼 수 있다.(이러한 수학적인 해석은 기본적으로 linear independent를 가정한다. multicolinearity를 VIF가 분석에 큰문제를 주는 경우가 많지 않다는 점에서 중요하지 않으나 모델의 해석에서 가정상의 한계를 알기 위해서는 주요하다)

shrink term을 다시 살펴보면 다음과 같다.

< RIdge>

$$\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_{i1} - \ldots - \beta_p x_{ip})^2 + \lambda \sum_{j=1}^{p} \beta_j^2 = RSS + \lambda \sum_{j=1}^{p} \beta_j^2$$

계수가 크기를 가질때, 특히 큰 beta일때 regular term에 의해 제약을 받게 되고, 결국 **큰 beta만큼 sse를 설명할 가치가 잇는 변수들**만이 큰 beta를 유지하며 남게 된다. 그 계수가 커지는 정도만큼 sse를 설명하지 못하면 그 계수를 줄이거나 0으로 만드는것. 참고

3. Ridge Regressio

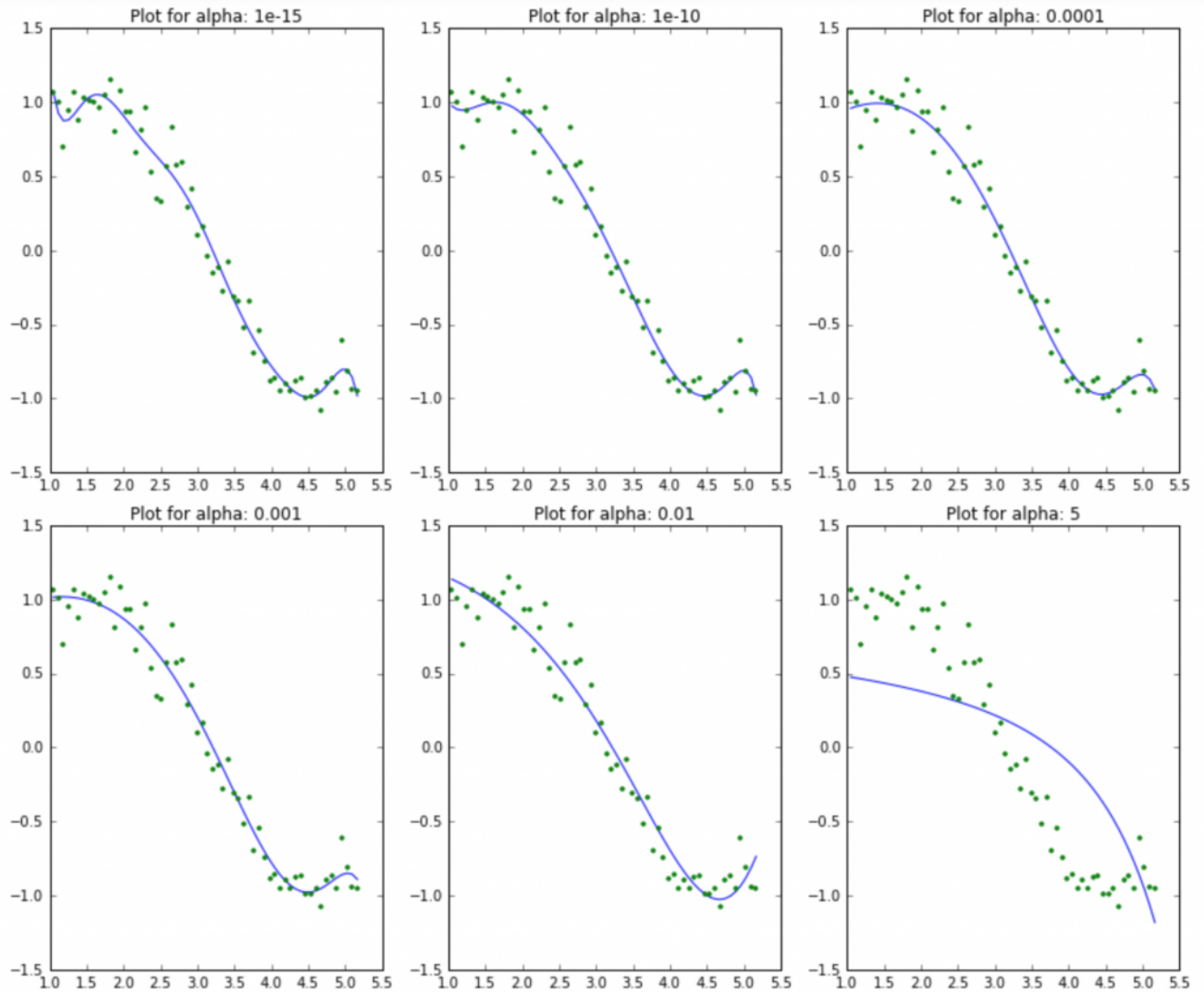
같은 방식으로 ridge 돌려본다. ridge가 L2 regularization을 통해 계수들을 shrink해주는 것은 알겠는데, 얼마나 shrink할까?

얼마나 shrink할지를 결정하는 parameter α에 따라 적합하여 결과를 본다. 이때 noarmalize=True를 해주는 것은 변수들간의 scaling의 의미도 있고, shrink대상에서 intercept는 빠져야 하는 의미도 있다. 근데 이거말고 preprocessing에 있는 scaling함수쓰라 그런다.

```
from sklearn.linear_model import Ridge
def ridge_regression(data, predictors, alpha, models_to_plot={}):
    #Fit the model
    ridgereg = Ridge(alpha=alpha,normalize=True)
    ridgereg.fit(data[predictors],data['y'])
    y_pred = ridgereg.predict(data[predictors])

    #Check if a plot is to be made for the entered alpha
    if alpha in models_to_plot:
        plt.subplot(models_to_plot[alpha])
        plt.tight_layout()
        plt.plot(data['x'],y_pred)
        plt.plot(data['x'],data['y'],'.')
        plt.title('Plot for alpha: %.3g'%alpha)

    #Return the result in pre-defined format
    rss = sum((y_pred-data['y'])**2)
    ret = [rss]
    ret.extend([ridgereg.intercept_])
    ret.extend(ridgereg.coef_)
    return ret
```

```
#Initialize predictors to be set of 15 powers of x
predictors=['x']
predictors.extend(['x_%d'%i for i in range(2,16)])

#Set the different values of alpha to be tested
alpha_ridge = [1e-15, 1e-10, 1e-8, 1e-4, 1e-3,1e-2, 1, 5, 10, 20]

#Initialize the dataframe for storing coefficients.
col = ['rss','intercept'] + ['coef_x_%d'%i for i in range(1,16)]
ind = ['alpha_%.2g'%alpha_ridge[i] for i in range(0,10)]
coef_matrix_ridge = pd.DataFrame(index=ind, columns=col)

models_to_plot = {1e-15:231, 1e-10:232, 1e-4:233, 1e-3:234, 1e-2:235, 5:236}
for i in range(10):
    coef_matrix_ridge.iloc[i,] = ridge_regression(data, predictors, alpha_ridge[i],
models_to_plot)
```

α가 커질수록 점점 flexibility를 잃다가 나중에는 underfit을 하는 것을 볼 수 있다



역시나 계수들을 봐보면,

```
#Set the display format to be scientific for ease of analysis
pd.options.display.float_format = '{:,.2g}'.format
coef_matrix_ridge
```

	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	coe
alpha_1e-15	0.87	95	-3e+02	3.8e+02	-2.4e+02	66	0.96	-4.8	0.64	0.15	-0.026	-0.0054	0.00086	0.0
alpha_1e-10	0.92	11	-29	31	-15	2.9	0.17	-0.091	-0.011	0.002	0.00064	2.4e-05	-2e-05	-4.2
alpha_1e-08	0.95	1.3	-1.5	1.7	-0.68	0.039	0.016	0.00016	-0.00036	-5.4e-05	-2.9e-07	1.1e-06	1.9e-07	2e-
alpha_0.0001	0.96	0.56	0.55	-0.13	-0.026	-0.0028	-0.00011	4.1e-05	1.5e-05	3.7e-06	7.4e-07	1.3e-07	1.9e-08	1.9
alpha_0.001	1	0.82	0.31	-0.087	-0.02	-0.0028	-0.00022	1.8e-05	1.2e-05	3.4e-06	7.3e-07	1.3e-07	1.9e-08	1.7
alpha_0.01	1.4	1.3	-0.088	-0.052	-0.01	-0.0014	-0.00013	7.2e-07	4.1e-06	1.3e-06	3e-07	5.6e-08	9e-09	1.1
alpha_1	5.6	0.97	-0.14	-0.019	-0.003	-0.00047	-7e-05	-9.9e-06	-1.3e-06	-1.4e-07	-9.3e-09	1.3e-09	7.8e-10	2.4
alpha_5	14	0.55	-0.059	-0.0085	-0.0014	-0.00024	-4.1e-05	-6.9e-06	-1.1e-06	-1.9e-07	-3.1e-08	-5.1e-09	-8.2e-10	-1.3
alpha_10	18	0.4	-0.037	-0.0055	-0.00095	-0.00017	-3e-05	-5.2e-06	-9.2e-07	-1.6e-07	-2.9e-08	-5.1e-09	-9.1e-10	-1.6
alpha_20	23	0.28	-0.022	-0.0034	-0.0006	-0.00011	-2e-05	-3.6e-06	-6.6e-07	-1.2e-07	-2.2e-08	-4e-09	-7.5e-10	-1.4

α가 커질수록 rss는 조금씩 증가한다(train error는 더 커진다). 그리고 $\alpha = 1e-15$에서도 이전 다항회귀의 마지막 행과 비교하여 계수들이 줄어들었음을 볼 수 있다. 그리고 모두가 알듯이, α가 커져도 계수가 정확히 0이 되지 않는다.

4. Lasso Regression

LASSO stands for *Least Absolute Shrinkage and Selection Operator*.

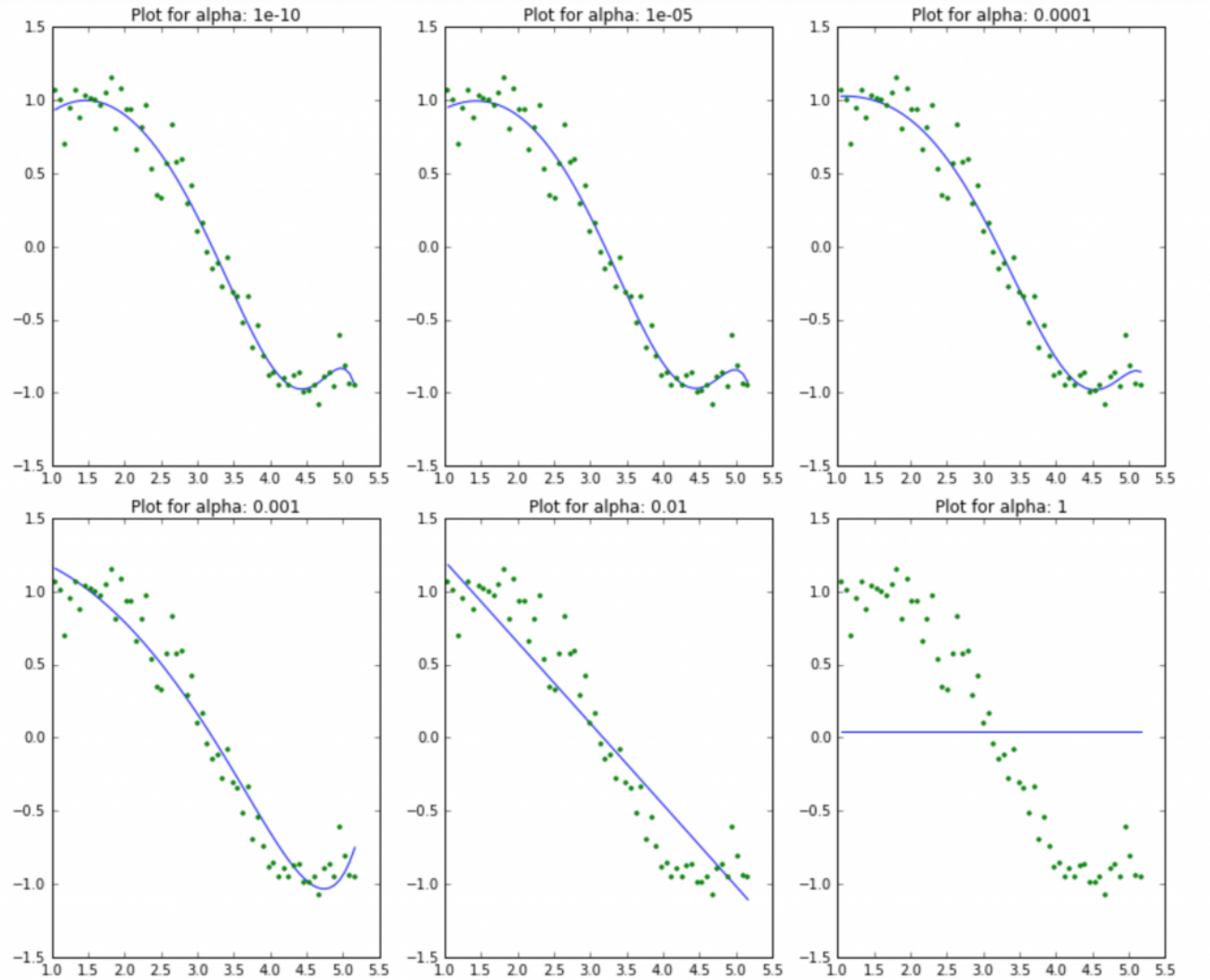
똑같은 방식으로 α에 따라 적합, plot을 띄우고 계수들을 본다. 그런데 이번에는 **max_iter** 파라미터를 설정해준다. 이유는 Lasso의 최적화 방식의 특성때문인데, 뒤에서 설명하고 우선은 계수의 크기부터 비교한다.

```
from sklearn.linear_model import Lasso
def lasso_regression(data, predictors, alpha, models_to_plot={}):
    #Fit the model
    lassoreg = Lasso(alpha=alpha,normalize=True, max_iter=1e5)
    lassoreg.fit(data[predictors],data['y'])
    y_pred = lassoreg.predict(data[predictors])

    #Check if a plot is to be made for the entered alpha
    if alpha in models_to_plot:
        plt.subplot(models_to_plot[alpha])
        plt.tight_layout()
        plt.plot(data['x'],y_pred)
        plt.plot(data['x'],data['y'],'.')
        plt.title('Plot for alpha: %.3g'%alpha)

    #Return the result in pre-defined format
    rss = sum((y_pred-data['y'])**2)
    ret = [rss]
    ret.extend([lassoreg.intercept_])
    ret.extend(lassoreg.coef_)
    return ret
```

```
#Initialize predictors to all 15 powers of x
predictors=['x']
predictors.extend(['x_%d'%i for i in range(2,16)])

#Define the alpha values to test
alpha_lasso = [1e-15, 1e-10, 1e-8, 1e-5,1e-4, 1e-3,1e-2, 1, 5, 10]

#Initialize the dataframe to store coefficients
col = ['rss','intercept'] + ['coef_x_%d'%i for i in range(1,16)]
ind = ['alpha_%.2g'%alpha_lasso[i] for i in range(0,10)]
coef_matrix_lasso = pd.DataFrame(index=ind, columns=col)

#Define the models to plot
models_to_plot = {1e-10:231, 1e-5:232,1e-4:233, 1e-3:234, 1e-2:235, 1:236}

#Iterate over the 10 alpha values:
for i in range(10):

    coef_matrix_lasso.iloc[i,] = lasso_regression(data, predictors, alpha_lasso[i],
```

```
models_to_plot)
```

α가 커질수록 underfit하는것은 동일하게 보이는데, 단순히 underfit하는게 아니라 linear모양이 되는것을 확인할 수 있다.

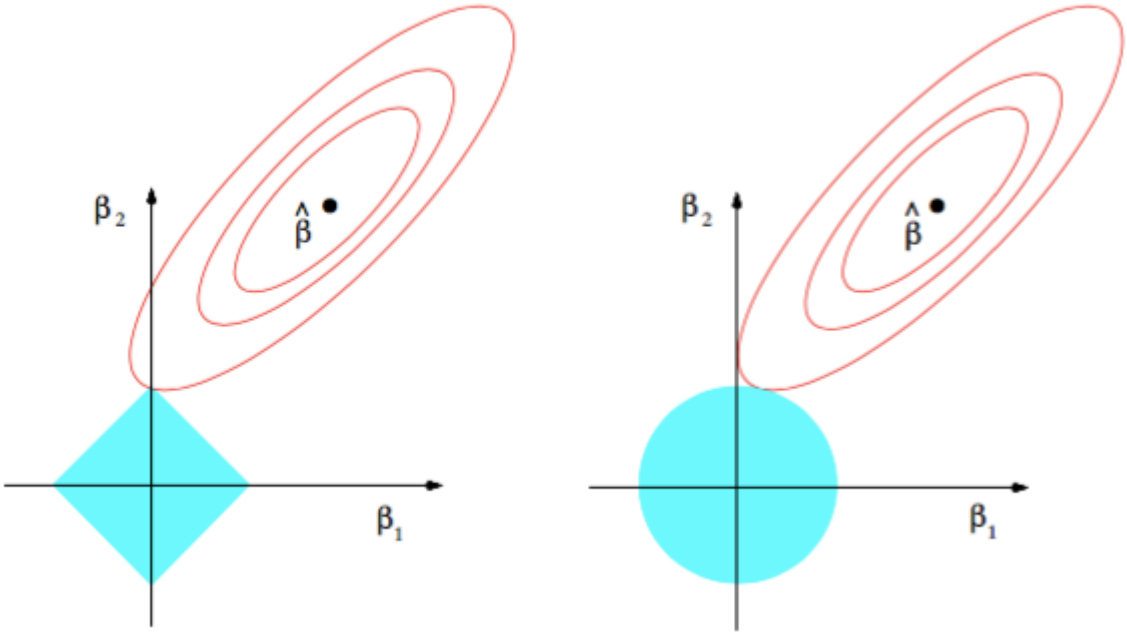


	rss	intercept	coef_x_1	coef_x_2	coef_x_3	coef_x_4	coef_x_5	coef_x_6	coef_x_7	coef_x_8	coef_x_9	coef_x_10	coef_x_11	coe
alpha_1e-15	0.96	0.22	1.1	-0.37	0.00089	0.0016	-0.00012	-6.4e-05	-6.3e-06	1.4e-06	7.8e-07	2.1e-07	4e-08	5.4
alpha_1e-10	0.96	0.22	1.1	-0.37	0.00088	0.0016	-0.00012	-6.4e-05	-6.3e-06	1.4e-06	7.8e-07	2.1e-07	4e-08	5.4
alpha_1e-08	0.96	0.22	1.1	-0.37	0.00077	0.0016	-0.00011	-6.4e-05	-6.3e-06	1.4e-06	7.8e-07	2.1e-07	4e-08	5.3
alpha_1e-05	0.96	0.5	0.6	-0.13	-0.038	-0	0	0	0	7.7e-06	1e-06	7.7e-08	0	0
alpha_0.0001	1	0.9	0.17	-0	-0.048	-0	-0	0	0	9.5e-06	5.1e-07	0	0	0
alpha_0.001	1.7	1.3	-0	-0.13	-0	-0	-0	0	0	0	0	0	1.5e-08	7.5
alpha_0.01	3.6	1.8	-0.55	-0.00056	-0	-0	-0	-0	-0	-0	-0	0	0	0
alpha_1	37	0.038	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0
alpha_5	37	0.038	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0
alpha_10	37	0.038	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0

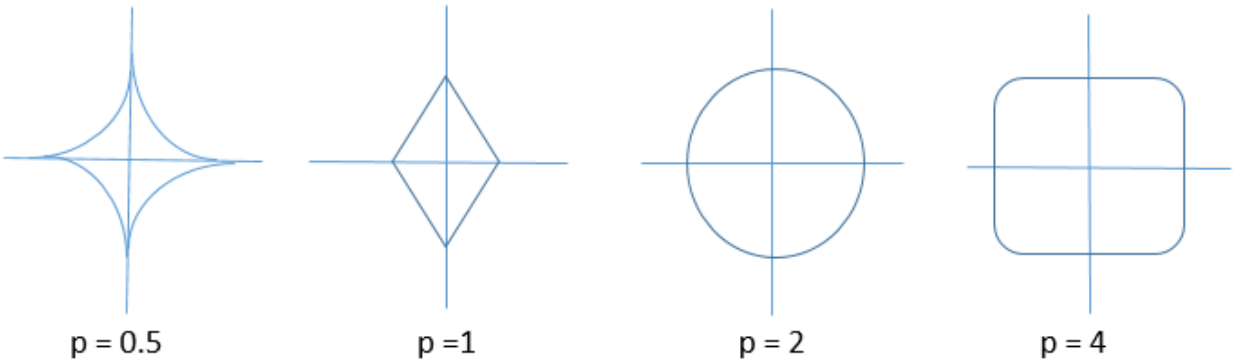
HIGH SPARSITY

1. 같은 alpha에 대해, lasso가 ridge보다 shrink가 크다.
2. 대부분의 경우 lasso의 RSS가 더 크다. 즉, 더 오차가 크다.
3. 상당수의 계수가 0으로 간다. 이렇게 몇몇 계수들을 아예 0으로 만드는 현상을 '**sparsity**'라고 한다.

이유는 ISL에서 나왔듯이 β_j들의 공간을 제한해주는 방식에서 L1 regularliztion의 특성때문.



그럼 L3,L4...L1/2는 어떨까? 대충 요런 그림



그러나 Lp(p<1)은 normed space의 성질을 띄지 못하고 norm은 convex optim이 안돼 어렵다고 한다 참고

암튼 릿지는 0이아니라 shrink하게 하고, 라소는 0으로 보내는 형식으로 업데이트를 한다.

그렇기때문에 변수의 갯수가 엄청나게 많아졌을때 릿지는 취약점을 보인다. 모든 변수를 끌어안고 계속가게 되기 때문에. 그럼 다 0으로 만든느게 무조건 좋지 않은가?

꼭 그렇지는 않은게, multicolinearity상황에서 라쏘는 취약점을 보인다. 릿지는 서로 correlate되어있는 변수들을 모두 가져가며, weight를 조절하여 colinearity를 어느정도 조절하면서도 모든 information을 가져가게 된다. 그러나 라쏘는 correlated된 변수 중 (그것이 noise에 의해 아주 근소하게라도 더) 우리의 train data를 잘 설명하는 변수 하나만을 남기고 나머지는 0으로 보내버려, 데이터의 information을 그만큼 잃는다고 할 수 있다. noise에 의해 correlated중 어떤 변수가 살아남게 될지 상당히 자주 변한다면, 이 특성은 특히나 안좋아 진다. (ridge is more stable than lasso)

mulity colinearity를 조절하기 위해 lasso를 바꾼 여러 방식이 잇다. 그중 가장 알기 위운게 elastic net. 참고 Elastic net은 다음과 같은 cost func을 갖는다. $\min\left(\left|\left|\mathrm{Y}-\mathrm{X}\theta\right|\right|_2^2+\lambda_1||\theta||_1+\lambda_2||\theta||_2^2\right)$

우리가 사이킷런에서 조절해줄 파라미터는 $\boldsymbol{alpha}=\boldsymbol{\lambda_1}+\boldsymbol{\lambda_2}$의 크기와 $\boldsymbol{L_1}-\boldsymbol{ratio}=\frac{\lambda_1}{\lambda_1+\lambda_2}$이다. 이를 잘 조절하면 correlated된 변수들을 한번에 다 지우지 않으면서도 적절히 0으로 보내며 shrink를 하게 한다.

그외의 모델들은...훗날을 기약하며...ㅎㅎ 알고싶다면 참고