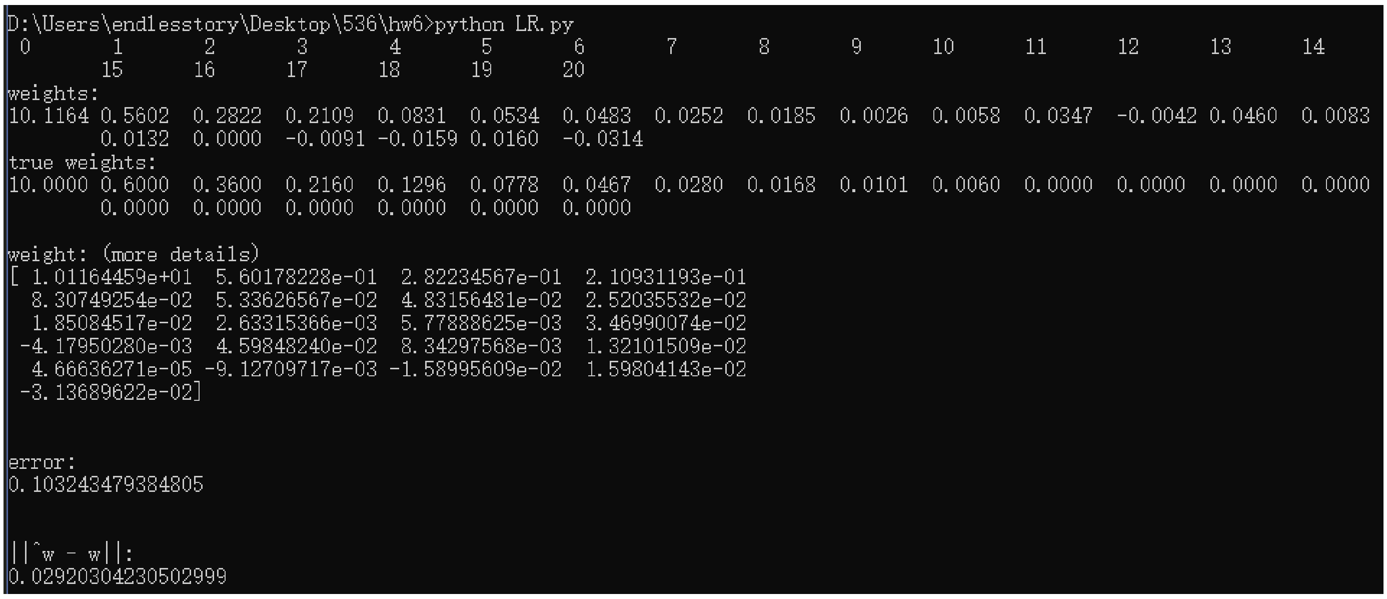
Linear Regression

1. naive regression

Here is the result: (Fig.1)



```
D:\Users\endlesstory\Desktop\536\hw6>python LR.py
 0       1       2       3       4       5       6       7       8       9       10      11      12      13      14
        15      16      17      18      19      20
weights:
10.1164 0.5602  0.2822  0.2109  0.0831  0.0534  0.0483  0.0252  0.0185  0.0026  0.0058  0.0347  -0.0042 0.0460  0.0083
        0.0132  0.0000  -0.0091 -0.0159 0.0160  -0.0314
true weights:
10.0000 0.6000  0.3600  0.2160  0.1296  0.0778  0.0467  0.0280  0.0168  0.0101  0.0060  0.0000  0.0000  0.0000  0.0000
        0.0000  0.0000  0.0000  0.0000  0.0000  0.0000

weight: (more details)
[ 1.01164459e+01  5.60178228e-01  2.82234567e-01  2.10931193e-01
  8.30749254e-02  5.33626567e-02  4.83156481e-02  2.52035532e-02
  1.85084517e-02  2.63315366e-03  5.77888625e-03  3.46990074e-02
 -4.17950280e-03  4.59848240e-02  8.34297568e-03  1.32101509e-02
  4.66636271e-05 -9.12709717e-03 -1.58995609e-02  1.59804143e-02
 -3.13689622e-02]


error:
0.103243479384805


||^w - w||:
0.02920304230502999
```

Fig.1 The result of naive regression. w_0 refers to b.

Notice that the first 3 weights and bias are similar to the true values, but the others are kind of different from the true ones. Especially it involves nearly all irrelevent features.

According to the weights, the first feature is the most significant one, which is correct, and the 16th one is the least significant.

However, it still cannot prune any features directly, even if the 16th features' weight is relatively small.

2. ridge regression

Here is the result: (Fig.2 and 3)

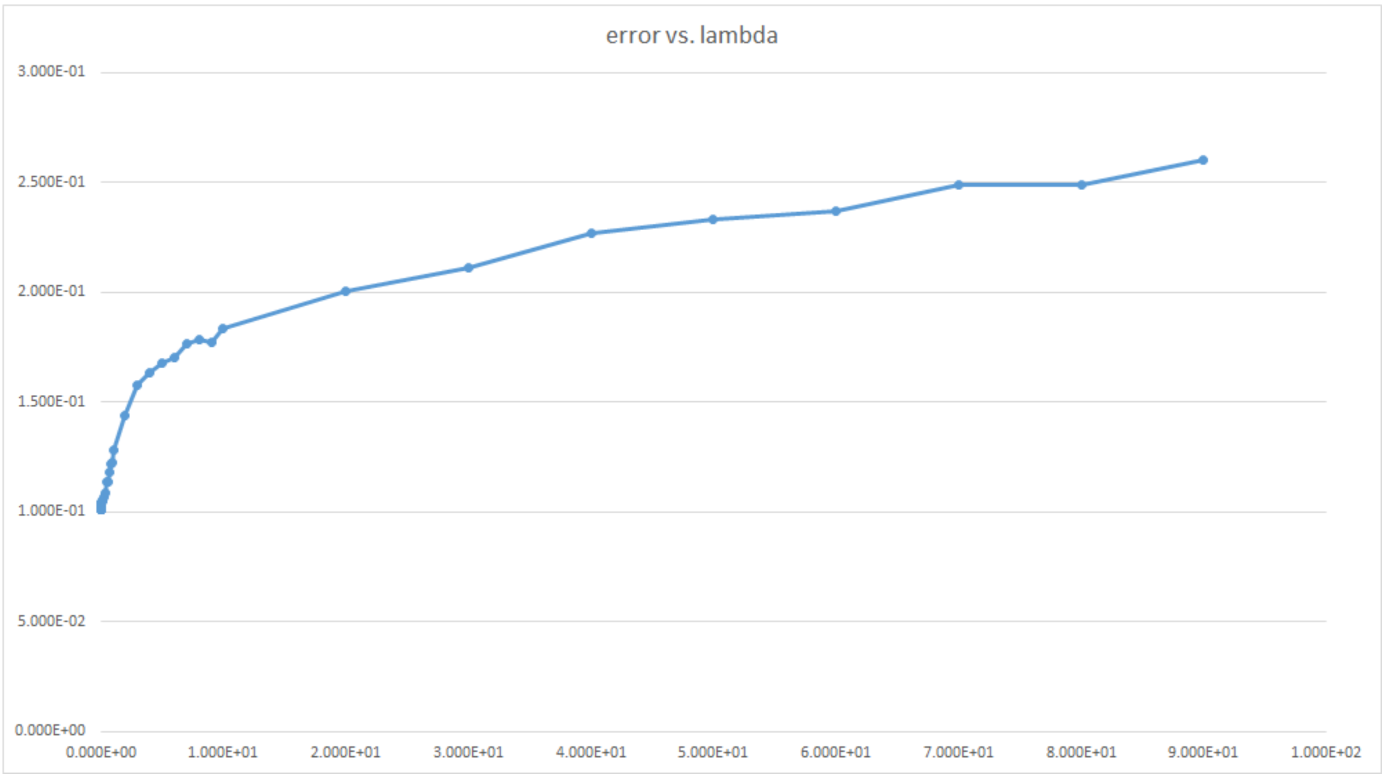


Fig.2 estimated true error of ridge regression vs. λ

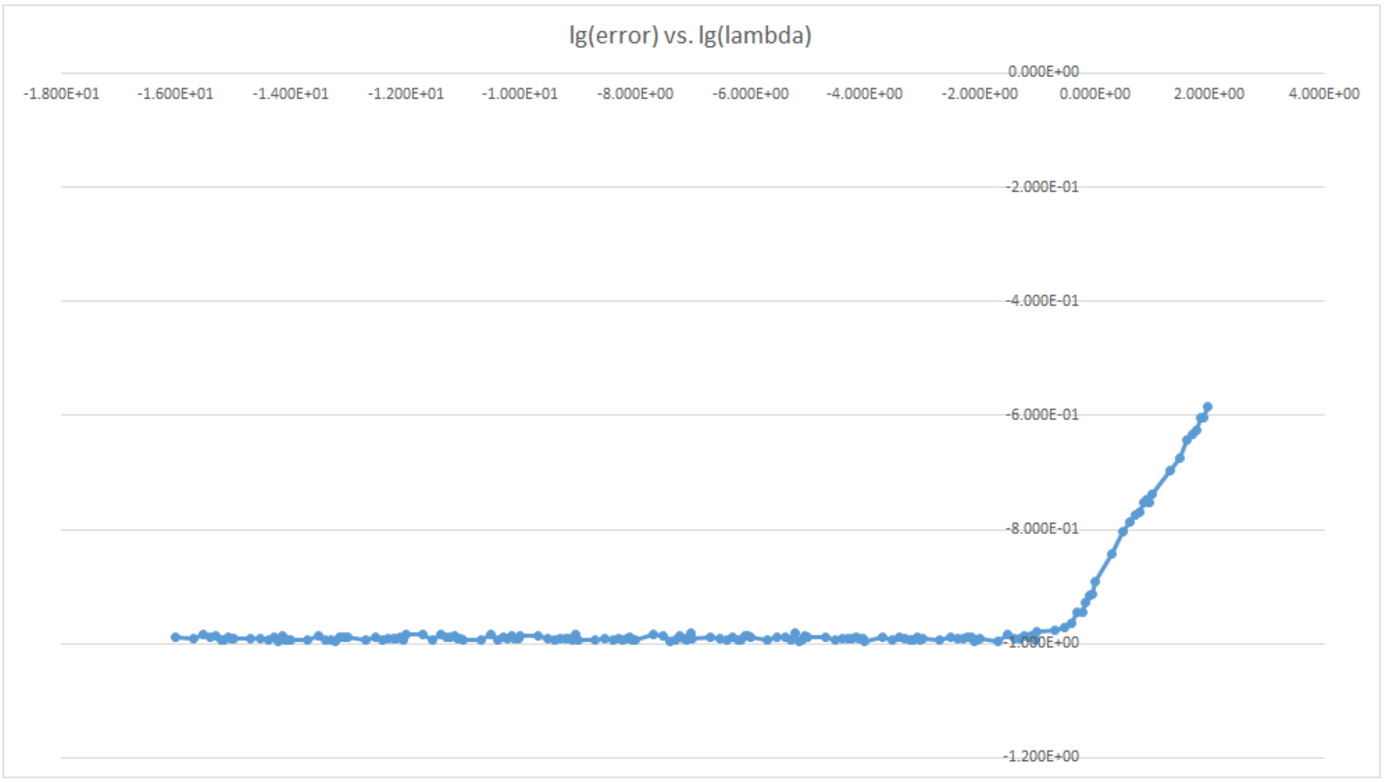


Fig.3 estimated true error of ridge regression vs λ (logarithmic scaled)

Therefore, we can find that when $\lambda = 1 \times 10^{-4}$, we have the minmum true error 0.1009. Here is the detail outputs of $\lambda = 1 \times 10^{-4}$:

```
lambda:
0.00010000000000000003


weight: (more details)
[ 9.98618685e+00  6.09589418e-01  3.64863281e-01  2.60964012e-01
  1.61785663e-01  6.27955681e-02  3.93113639e-02  2.11599308e-02
  1.82946863e-02  1.60586015e-02  1.24375580e-02 -6.39626116e-03
 -4.49011848e-02  9.33927831e-03 -5.05524411e-03 -1.81015595e-03
  1.04056969e-02  2.83713117e-03  2.49014415e-03  2.56751320e-03
  6.33691608e-04]


error:
0.10089216153583791
```

Notice that the first 3 weights and bias are still similar to the true weights, but it pushes irrelevant weights to a relatively small value.

The most significant feature is the same as the naive one, the least significant feature is the 20th. But it still cannot prune anything.

The overall true error is less because the irrelevant weights are pushed.

3. lasso regression will eventually eliminate all features

Here is the result: (Fig.4)

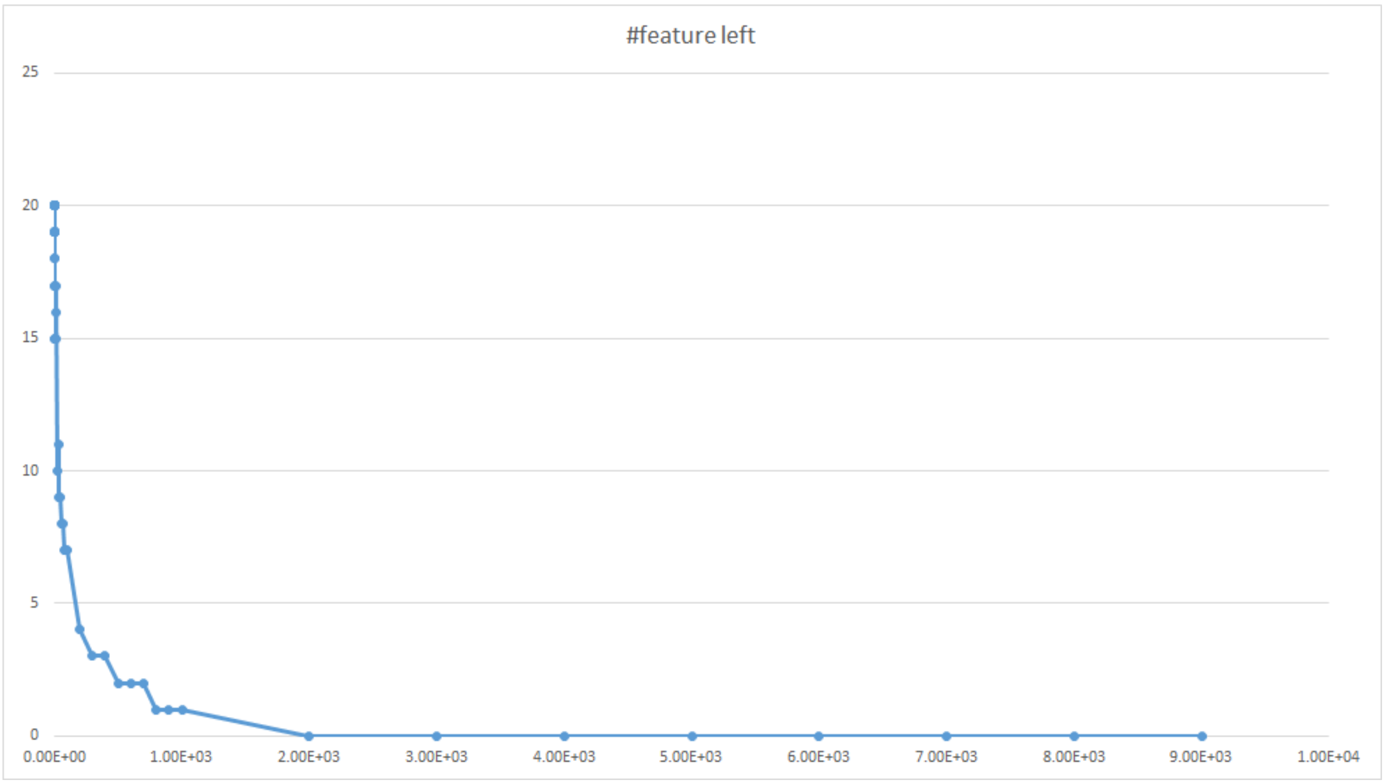


Fig.4 The number of features left of lasso regression vs. λ

Notice that when $\lambda \geq 2000$, all features are eliminated.

4. lasso regression

Here is the result: (Fig.5 and 6)

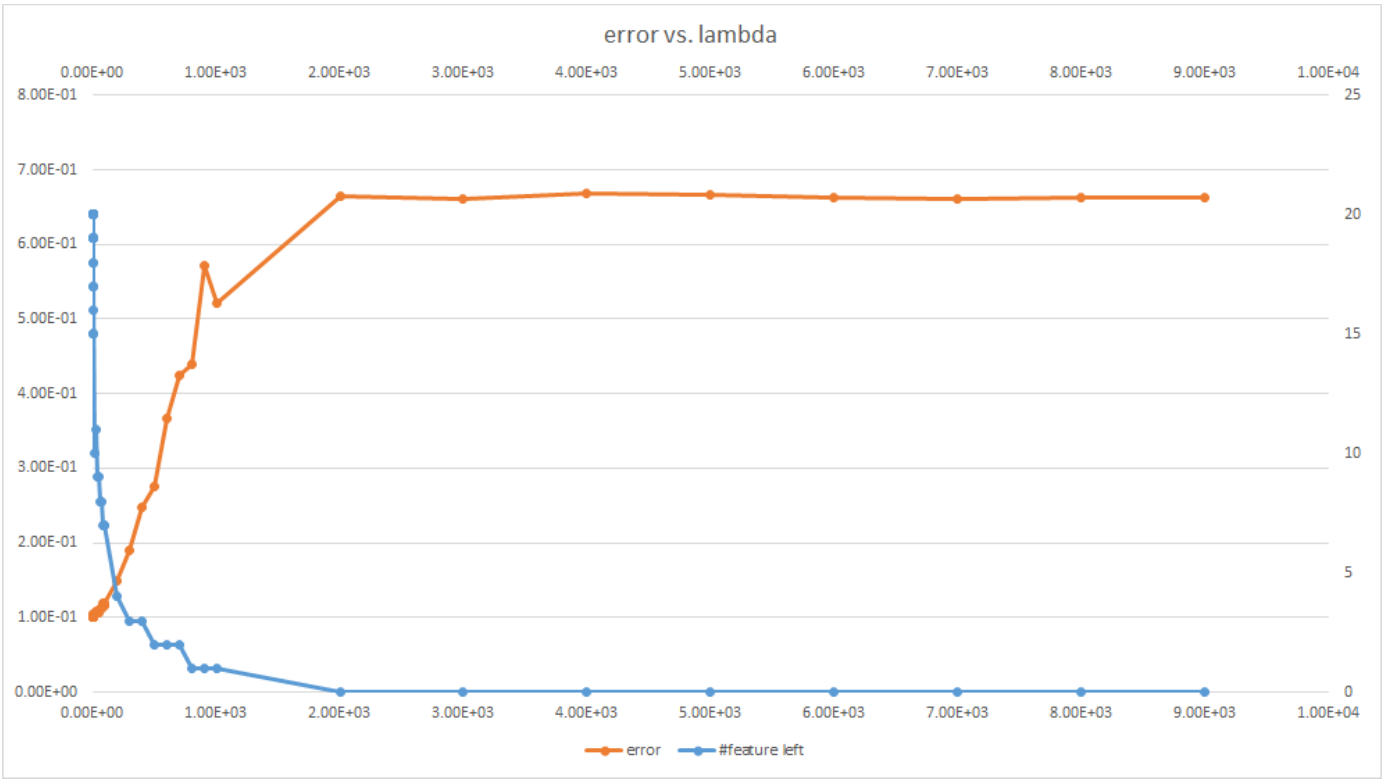


Fig.5 estimated true error of lasso regression vs. λ

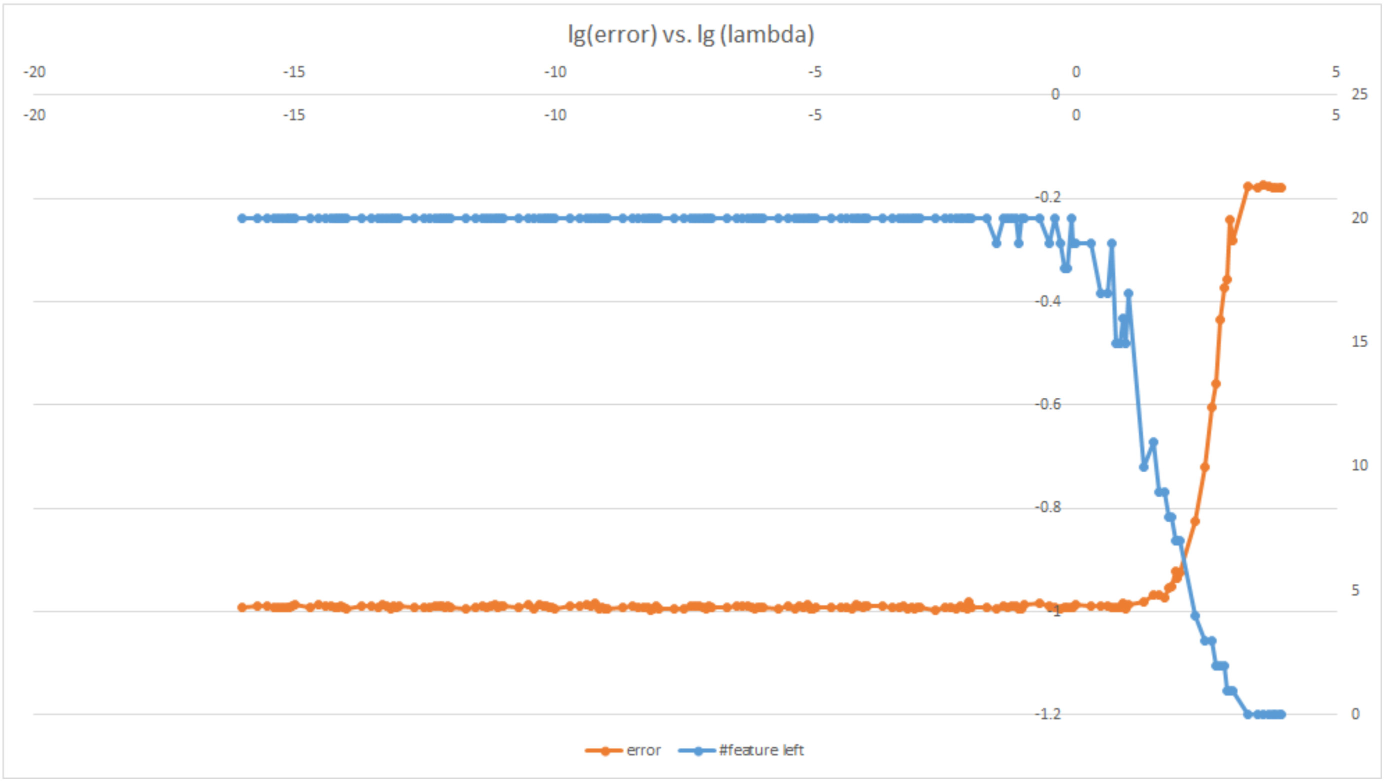


Fig.6 estimated true error of lasso regression vs. λ (logarithmic scaled)

Notice that when $\lambda \approx 1$, the error and the number of features left change sharply. Therefore, we can focus on this range in order to get an optimal λ. (Fig.7)

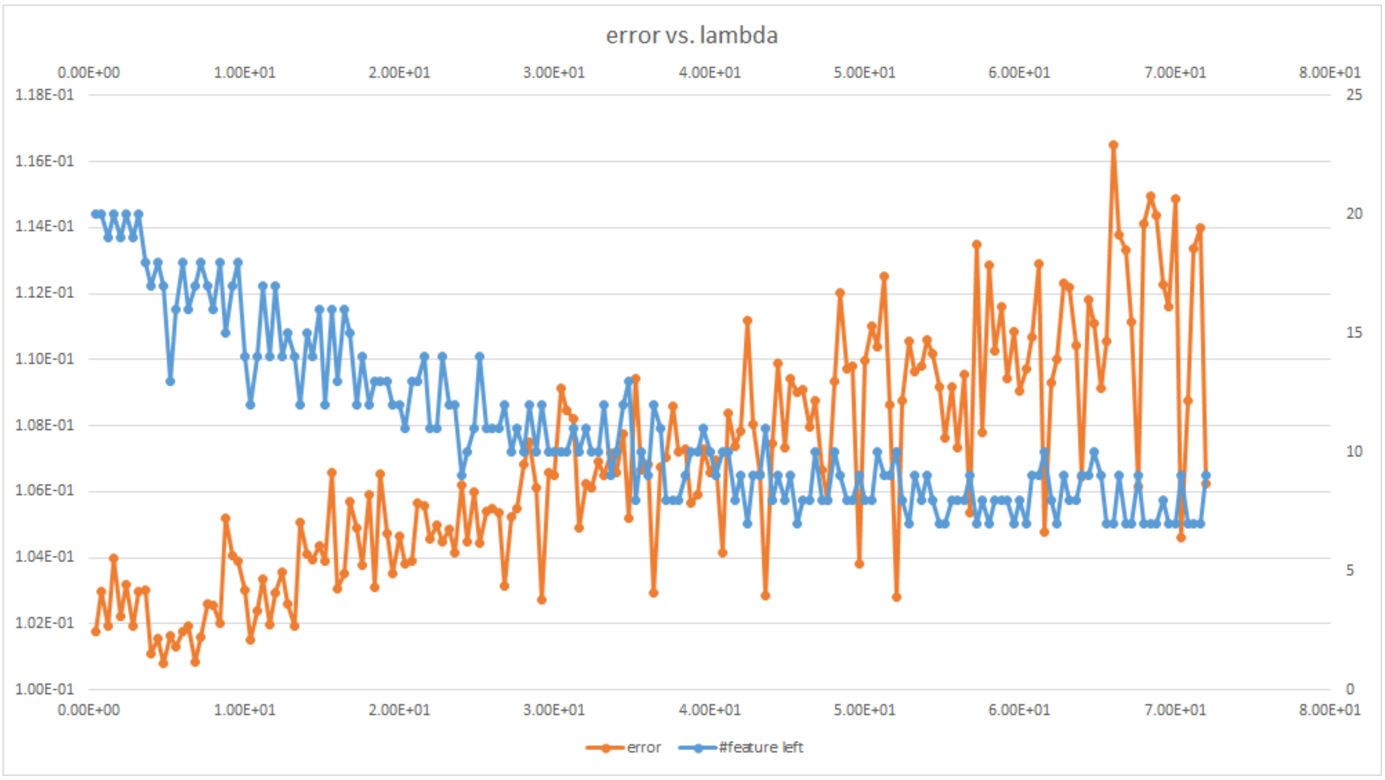


Fig.7 estimated true error of lasso regression vs. λ

Notice that when $\lambda = 4.8$, we have the minmum true error 0.1008. Here is the detail outputs:

```
lambda:
4.8


I: reach max iter.
weight: (more details)
[ 1.04929957e+01  5.86919391e-01  2.47389484e-01  1.83095243e-01
  8.59164811e-02  8.69438224e-02  4.20795412e-02  2.78244401e-02
  1.19496010e-02  4.32951045e-03  4.16288870e-03  8.58513561e-03
  2.78502142e-02  0.00000000e+00  2.00987614e-03  4.99882068e-02
 -3.31481298e-03  0.00000000e+00 -8.91071984e-03 -1.29617562e-03
  0.00000000e+00]


error:
0.10080735610987508
#non-zero weights:
17
```

We can find that it is relatively far from the true weights comparing with ridge regression. But it does prune 3 features: X_{13}, X_{17}, X_{20}.

Notice that lasso regression also return a reletively smaller error compared with the naive one.

5. lasso-ridge combination

A. a good λ

Here is a little trick:
Notice that when $\lambda \geq 0.1$, the error of ridge regression is obviously increasing. (Fig.3) We can conclude that we have inflated the dimension too much. Therefore, a feasible λ cannot be larger than 0.1. It is showed in Fig. 9, and we can find the error of lasso-ridge regression are extremely closed to ridge regression when $\lambda \geq 0.1$.

(Maybe we can even claim the optimal λ of ridge regression is less than the one of lasso-ridge one, because we have eliminated some irrelevent features. But personally I feel it is sort of bold.)

B. results

Here are results: (Fig. 8 and 9)

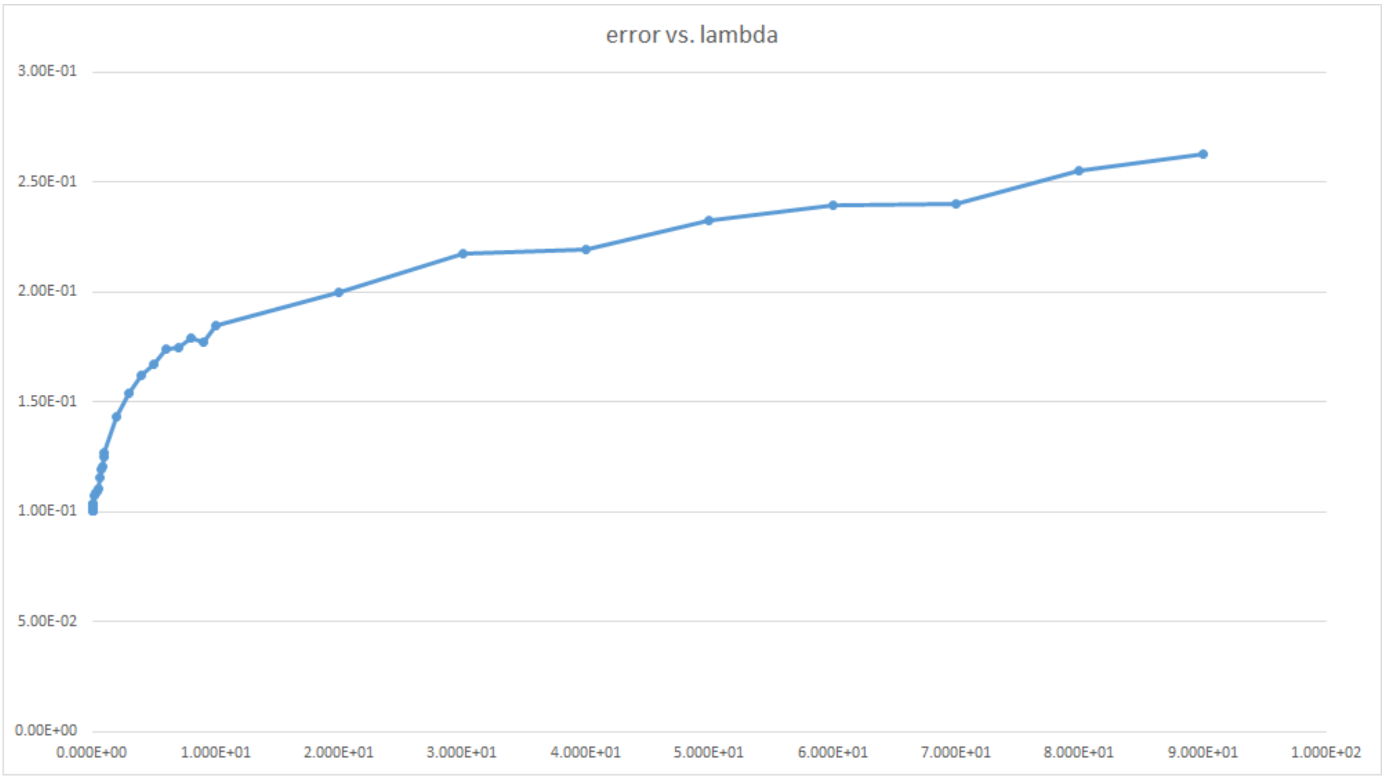


Fig.8 estimated true error of lasso-ridge regression vs. λ

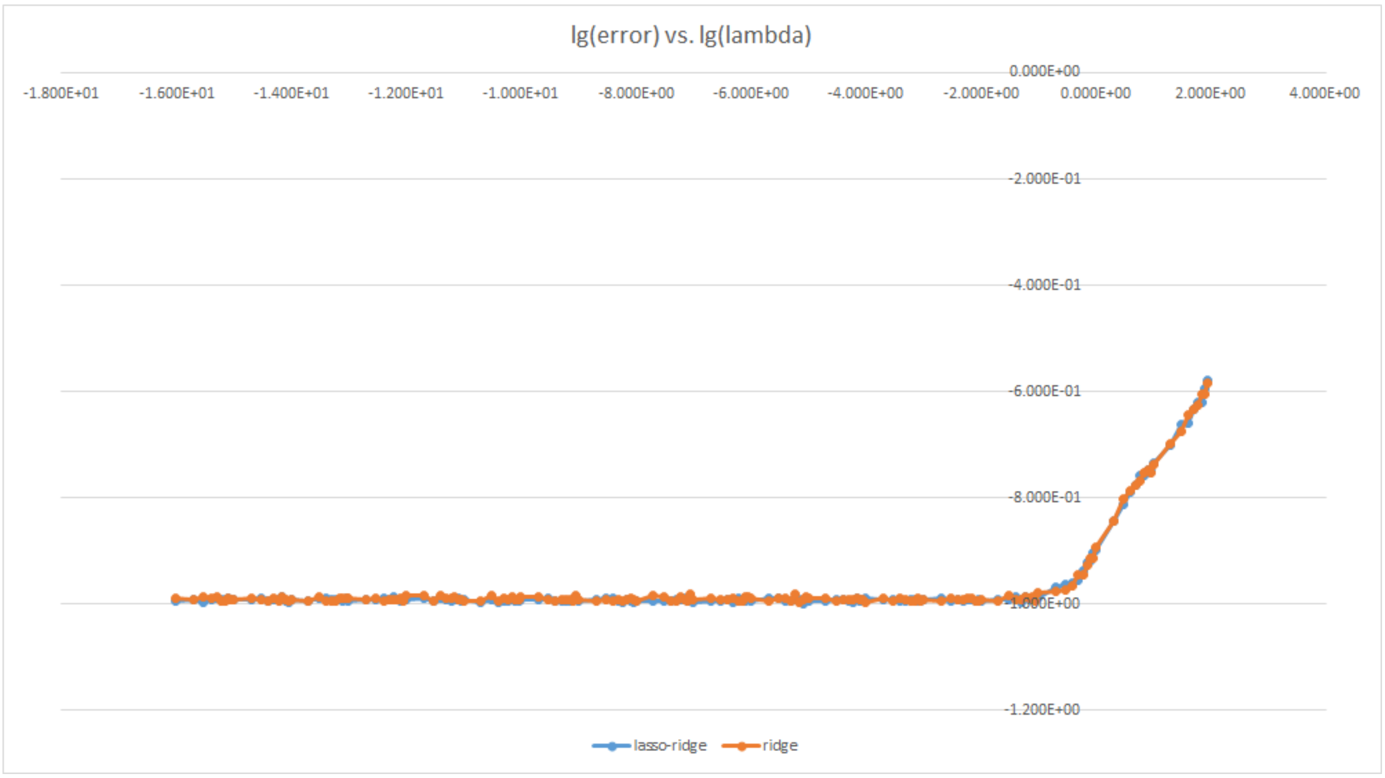


Fig.9 estimated true error of lasso-ridge regression vs. λ comparing with ridge regression (logarithmic scaled)

We can find that when $\lambda = 8 \times 10^{-6}$, we have the minmum true error 0.1004. It is much better than the naive one. Here is the detail outputs of $\lambda = 8 \times 10^{-6}$:

```
lambda:
8.000000000000003e-06


weight: (more details)
[ 9.82562568e+00  5.90335775e-01  4.00875536e-01  2.41450147e-01
  1.35212434e-01  7.49975790e-02  5.33539549e-02  2.85872636e-02
  2.07062778e-02  1.86559768e-02  1.11426457e-02  9.97834609e-04
 -2.07879126e-02 -9.98750555e-03 -1.65992306e-02  2.90335380e-03
 -1.56313455e-04  4.87059138e-03]


error:
0.10037712465059263
```

The result does not contain w_{13}, w_{17}, w_{20}, because they should always be zero according to the lasso regression.

Notice that all the first 7 weights are closed to the true ones, and they are significant. Others features are relatively insignificant, where X_{18} is the least significant. Comparing with 0.1009, the lasso-ridge one is better in terms of the testing errors.

SVMs

1. Implementation

A. feasible initial solution

Say we have $Y^+ = \{i|y_i \in Y, y_i = 1\}$, $Y^- = \{i|y_i \in Y, y_i = -1\}$. We can initialize $\alpha_i = \frac{1}{|Y^+|}, \forall i \in Y^+; \alpha_i = \frac{1}{|Y^-|}, \forall i \in Y^-$. In this case, we have $\forall \alpha_i > 0$, and $\sum\limits_i \alpha_i = 1 - 1 = 0$.

B. no step outside

Notice that since we initiallize α's feasible, the only case we may step out is over shooting. Namely, we step too far away. Therefore, the simplest way to fix it is to shrink our step size.

```
while True:
    tempA = a + rate * step //because we are searching maximizer
    if checkA(tempA):
        a = tempA
        break
    else:
        rate = rate / 2
```

In this case, we can exponentially decrease the step size to get a feasible step enough fast. But we might face to the following case: if we had shrink the step size too tiny, we would take a unacceptably large number of steps to converge.

Therefore, a little bit better way is let the step size adapt to the barrier punalty. If we step out, shrink it, and if not, increase it.

```
while True:
    tempA = a + rate * step
    if checkA(tempA):
        a = tempA
        rate = min(rate * 4, maxRate)
        break
    else:
        rate = rate / 2
```

Using a larger constant to increase step size will decrease the totle number of steps, but it may cause more “out-steps”.

2. projection methon

Another better method is to project the over-shoot step back to the feasible region. For $\alpha_i,\ i > 1$, it is eaiser. We can just modify its value to a positive number. For α_1, we have $\alpha_1' = \alpha_1 - r \cdot (u.\alpha)u$, where u is the normalized constrain of α, and $r > 1$ is a constant to pull α_1 not just back to 0, but further to a positive value.

```
for i in range(m-1):
    if a[i] <= 0:
        a[i] = some positive value
a1 = -y1 * sum(a * y)
if a1 <= 0:
    a = a - r * np.sum(a * u) * u
a1 = -y1 * sum(a * y)
```

C. ϵ_t

Initialize t_0 as 1, and decrease it by half each time until it reaches 1×10^{-16}.

The initial value of ϵ_0 is not too large in order to avoid pushing α's too far away from the optimal ones. After $\log 10^{16} \approx 54$ times, the penalty will be enough tiny.

Actually a better approach can be following: (But I just skipped it because the prior approach converges fast enough.)

After each time, compute the objective function $F(\alpha)$ and the penalty term $P(\alpha) = \sum_i \ln(\alpha_i)$. If $\frac{F(\alpha)}{P(\alpha)} > \epsilon_t c$, where c is a tiny number to indicate that we can ignore it comparing with 1, we immediately return α.

D. another trick to implement it

Here is an interesting trick to computing the gradient faster using the notion of ***error propagation***. (Or it is just a trick of the chain rule of multivariate calculus.)

2. XOR using $(1 + x.y)^2$

Say the 4 data points are sorted in the order of quadrants. Namely $(X_1, Y_1) = ((1, 1), -1)$, and so on.

A. solver

Here is the result:

```
epsilon 1:
..no more patience at iter 20
..[0.42153504 0.42191289 0.42229074 0.42191289]
epsilon 0.5:
..no more patience at iter 22
..[0.32019389 0.32064135 0.32108881 0.32064135]


......


epsilon 9.313225746154785e-10:
..no more patience at iter 5
..[0.125      0.125      0.12500001 0.125     ]
epsilon 4.656612873077393e-10:
..no more patience at iter 5
..[0.125 0.125 0.125 0.125]


.....


epsilon 1.1102230246251565e-16:
..no more patience at iter 5
..[0.125 0.125 0.125 0.125]
```

Namely $\alpha_i = 0.125, \forall i \in \{1, 2, 3, 4\}$

B. solve by hand

We can simplify the quesion as following[1]:

> *Find the set of values $\{a_i\}$ that solves the following:*
> $\max\limits_a \sum\limits_{i=1}^m a_i - \frac{1}{2}\sum\limits_{i=1}^m \sum\limits_{i=1}^m a_i y^i K(x^i, x^j) y^j a_j,\ s.t. \sum\limits_{i=1}^m a_i y^i = 0, \forall i, a_i \geq 0.$
> *Here we have* $m = 4$:
> $\max\limits_a \sum\limits_{i=1}^4 a_i - \frac{1}{2}\sum\limits_{i=1}^4 \sum\limits_{i=1}^4 (-1)^{i+j} a_i K(x^i, x^j) a_j,\ s.t. a_1 + a_3 = a_2 + a_4, a_i \geq 0$
> *Here we have* $\max\limits_a \sum\limits_{i=1}^4 a_i - \frac{1}{2}\sum\limits_{i=1}^4 \sum\limits_{i=1}^4 (-1)^{i+j} a_i a_j (1 + x^i.x^j)^2$
> *Let* $I = \sum\limits_{i=1}^4 a_i - \frac{1}{2}\sum\limits_{i=1}^4 \sum\limits_{i=1}^4 (-1)^{i+j} a_i a_j (1 + x^i.x^j)^2$
> *We have* $I = \sum\limits_{i=1}^4 (a_i - \frac{9}{2}a_i^2) + a_1a_2 - a_1a_3 + a_1a_4 + a_2a_3 - a_2a_4 + a_3a_4.$

Then using the Lagrange Multiplier Method we can solve it to $\forall i \in \{1, 2, 3, 4\}, \alpha_i = \frac{1}{8}$. Namely, the solver works.

3. reconstruct the primal classifier

Using $\alpha_i = \frac{1}{8}$, we can get the primal classifer[1]:

> *Therefore, we have* $y = \sum\limits_{i=1}^4 a_i y^i (1 + x^i x)^2 = -x_1 x_2$ *as the separator.*

Therefore, the first and third quadrants are classified to negative, and the second and fourth are positive. Namely, the XOR data is classified correctly.

---

1. SVM Problems Homework: https://content.sakai.rutgers.edu/access/content/attachment/13eb977e-0fb8-4ee0-9b17-4edccb798315/Assignments/69a72886-c9c6-4b76-89e9-6b93c6aec939/SVM.pdf ↩ ↩