1_Introduction

February 13, 2017

1 1.1 Dataset Construction

```
In [1]: import numpy as np
        import pandas as pd
        m = 150 #number of examples
        d = 75 # dimension of features
```

1) Construct a random design matrix

```
In [2]: X = np.random.rand(m*d).reshape(m,d) # Construct random design matrix X
```

2) Constructa true weight vector

```
In [3]: #Initialize zeros theta with dimension d+1
        theta = np.zeros(d)
        #Arbituary replace the first 10 elements to 10 or -10
        theta[:10] = np.random.choice([-10,10],10)
```

3) Construct response variable y

```
In [4]: # Episolon: normal noice
        epsilon = np.random.normal(0,0.1,m)
        # Compute y
        y = np.dot(X,theta)+epsilon
```

4) Split Training validation and testing set.

```
In [5]: train_X = X[0:80,:]
        train_y = y[0:80]
        validation_X = X[80:100,:]
        validation_y = y[80:100]
        test_X = X[100:150,:]
        test_y = y[100:150]

In [6]: # Save data to local folder name p1Data
        np.savetxt('../p1Data/test_X.csv',test_X,delimiter=',')
        np.savetxt('../p1Data/test_y.csv',test_y,delimiter=',')
        np.savetxt('../p1Data/train_X.csv',train_X,delimiter=',')
        np.savetxt('../p1Data/train_y.csv',train_y,delimiter=',')
        np.savetxt('../p1Data/validation_X.csv',validation_X,delimiter=',')
        np.savetxt('../p1Data/validation_y.csv',validation_y,delimiter=',')
```

2 Ridge_Regression

February 13, 2017

```
In [1]: # Import dataset
        import numpy as np
        #import pandas as pd
        from scipy.optimize import minimize
        # train_X = pd.read_csv('../p1Data/train_X.csv',header=None)
        # train_y = pd.read_csv('../p1Data/train_y.csv',header=None)
        # validation_X = pd.read_csv('../p1Data/validation_X.csv',header=None)
        # validation_y = pd.read_csv('../p1Data/validation_y.csv',header=None)
        train_X = np.genfromtxt('../p1Data/train_X.csv',delimiter=',')
        train_y = np.genfromtxt('../p1Data/train_y.csv',delimiter=',')
        validation_X = np.genfromtxt('../p1Data/validation_X.csv',delimiter=',')
        validation_y = np.genfromtxt('../p1Data/validation_y.csv',delimiter=',')
        test_X = np.genfromtxt('../p1Data/test_X.csv',delimiter=',')
        test_y = np.genfromtxt('../p1Data/test_y.csv',delimiter=',')
```

I use both sklearn.optimize.minimized() function and the sklearn.linear_model.Ridge to run ridge regression.

0.1 sklear.optimize.minimize() function

```
In [2]: from scipy.optimize import minimize

        X = np.loadtxt('../data/X_train.txt')
        y = np.loadtxt('../data/y_train.txt')
        X_val= np.loadtxt('../data/X_valid.txt')
        y_val= np.loadtxt('../data/y_valid.txt')

        (N,D) = X.shape

        w = np.random.rand(D,1)

        def ridge(Lambda):
            def ridge_obj(theta):
                return ((np.linalg.norm(np.dot(X,theta) - y))**2)/(2*N) + Lambda*(n
            return ridge_obj

        def compute_loss(Lambda, theta):
```

```
        return ((np.linalg.norm(np.dot(X_val,theta) - y_val))**2)/(2*N)

    for i in range(-5,6):
        Lambda = 10**i;
        w_opt = minimize(ridge(Lambda), w)
        print( Lambda, compute_loss(Lambda, w_opt.x))
```

```
1e-05 0.0173270678551
0.0001 0.0534227742847
0.001 0.384204784306
0.01 1.6785955378
0.1 4.99419973261
1 7.78280178024
10 49.2569463265
100 154.88721917
1000 182.896572549
10000 186.111513059
100000 186.437683461
```

Choose $\lambda = 1e - 5$ since it achieves best valdiation square loss.

```
In [7]: w_opt = minimize(ridge(1e-5), w)

In [9]: w_opt.x

Out[9]: array([ -9.94244415e+00,   9.83567726e+00,  -9.76393854e+00,
                -9.84368059e+00,  -9.97545839e+00,  -1.00054636e+01,
                -1.00492982e+01,  -1.00654249e+01,  -9.93792709e+00,
                -9.90725056e+00,   7.86432619e-02,   2.64012171e-01,
                 1.73014901e-01,  -3.87271967e-02,   1.46982996e-02,
                 1.05442100e-01,  -3.10636870e-01,  -1.15626912e-02,
                -2.56517590e-01,  -9.16715855e-02,   2.36415117e-02,
                -5.12369093e-02,   1.63186951e-01,  -6.20184015e-02,
                -2.74729567e-01,   2.08807502e-01,  -7.89390125e-02,
                 3.42368186e-01,   2.19937280e-01,  -1.28661134e-01,
                -1.60172396e-01,  -1.53584379e-03,  -2.94891748e-02,
                -1.24655657e-01,   6.63393209e-02,   3.13957765e-02,
                -3.05021636e-01,   9.77729269e-02,   1.67797289e-01,
                -2.91773672e-01,   1.53056090e-01,  -5.22732532e-02,
                 6.94282310e-02,   2.41839636e-03,   1.06592354e-01,
                 6.15879303e-02,   4.28718822e-02,   1.02156906e-01,
                 8.72525019e-03,   1.62550982e-02,   6.15201468e-02,
                -1.13013923e-01,  -1.20882726e-01,  -1.45966092e-01,
                -1.90685907e-02,   5.98411919e-02,  -1.26758097e-01,
                 2.22842478e-02,  -5.16096128e-03,   1.56190729e-01,
                -7.56512875e-02,  -9.32535914e-02,  -1.60697825e-01,
                 6.02370476e-02,   4.70462944e-02,  -2.51040561e-01,
                -2.21299013e-01,   8.22241106e-02,   1.17879284e-02,
```

```
                  1.16062244e-01,   3.56050316e-02,  -1.36535883e-01,
                 -1.82210193e-02,   2.25831124e-01,   8.76657098e-02])
```

Observe that although our optimized w does gives out close-to-zero estimation to those components whose true values being zero. But in the threshold of 0.001, we do not have sparsity.

0.2 Compare Results with sklearn.linaer_model.Ridge

```
In [3]: from sklearn.linear_model import Ridge
```

```
In [4]: l2model = Ridge(alpha=0.1)
        l2model.fit(train_X,train_y)
```

```
Out[4]: Ridge(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=None,
           normalize=False, random_state=None, solver='auto', tol=0.001)
```

```
In [5]: validation_y_predict = l2model.predict(validation_X)
```

```
In [6]: #test set square loss
        diff = validation_y_predict-validation_y
        0.5/diff.shape[0]*np.dot(diff,diff.T)
```

```
Out[6]: 5.2318688977252545
```

The built in ridge function gives out similar validation set loss when alpha/lambda_reg = 0.1.

3_1

February 13, 2017

In [29]:
```
# Import dataset
import numpy as np
#import pandas as pd
from scipy.optimize import minimize
# train_X = pd.read_csv('../p1Data/train_X.csv',header=None)
# train_y = pd.read_csv('../p1Data/train_y.csv',header=None)
# validation_X = pd.read_csv('../p1Data/validation_X.csv',header=None)
# validation_y = pd.read_csv('../p1Data/validation_y.csv',header=None)
train_X = np.genfromtxt('../p1Data/train_X.csv',delimiter=',')
train_y = np.genfromtxt('../p1Data/train_y.csv',delimiter=',')
validation_X = np.genfromtxt('../p1Data/validation_X.csv',delimiter=',')
validation_y = np.genfromtxt('../p1Data/validation_y.csv',delimiter=',')
test_X = np.genfromtxt('../p1Data/test_X.csv',delimiter=',')
test_y = np.genfromtxt('../p1Data/test_y.csv',delimiter=',')
```

1 3_1_1

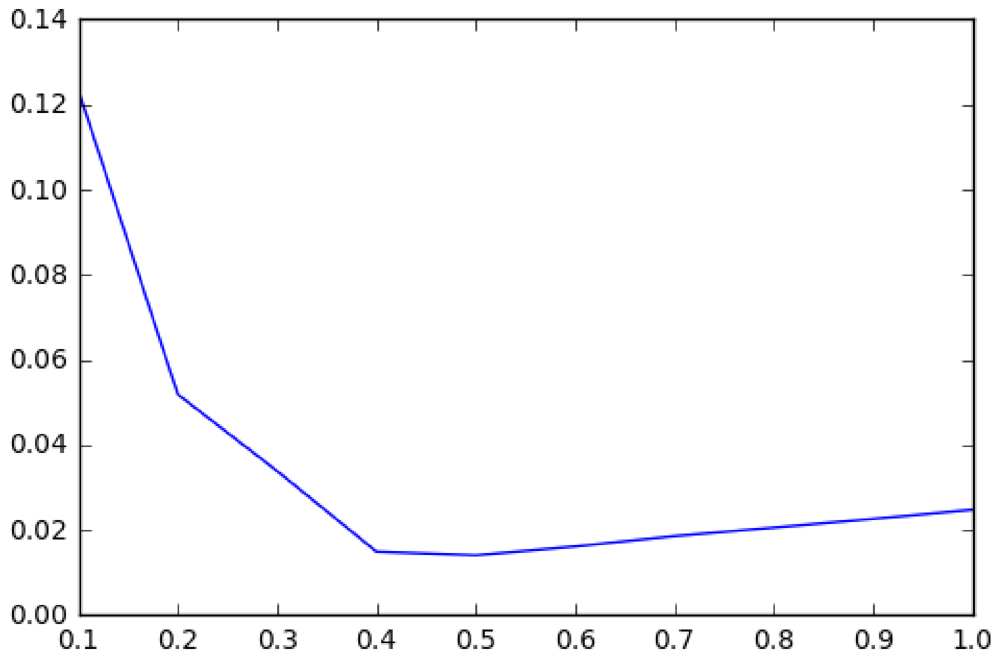
In [24]:
```
import time
```

In [52]:
```
# Calculate closed form solution for lasso regression using Shooting Algor
def lasso_shooting(X,y,lambda_reg=0.1,max_steps = 1000,tolerence = 1e-5):
    start_time = time.time()
    converge = False
    steps = 0
    #Get dimension info
    n = X.shape[0]
    d = X.shape[1]
    #initializing theta
    w = np.linalg.inv(X.T.dot(X)+lambda_reg*np.identity(d)).dot(X.T).dot(y
    def soft(a,delta):
        sign_a = np.sign(a)
        if np.abs(a)-delta <0:
            return 0
        else:
            return sign_a*(abs(a)-delta)
    while converge==False and steps<max_steps:
        a = []
        c = []
        old_w = w
    ####For loop for computing aj cj w
        for j in range(d):
            aj = 0
            cj = 0
            for i in range(n):
                xij = X[i,j]
                aj += 2*xij*xij
                cj += 2*xij*(y[i]-w.T.dot(X[i,:])+w[j]*xij)
            w[j] = soft(cj/aj,lambda_reg/aj)
            convergence = np.sum(np.abs(w-old_w))<tolerence
            a.append(aj)
            c.append(cj)
        steps +=1
        a = np.array(a)
        c = np.array(c)
    run_time = time.time()-start_time
    print('lambda:',lambda_reg,'run_time:',run_time,'steps_taken:',steps)
    return w,a,c
```

In [54]:
```
sqr_loss = []
for lambda_reg in np.arange(0.1,1.1,0.1):
    w,a,c = lasso_shooting(train_X,train_y,lambda_reg)
    validation_predict = validation_X.dot(w)
    diff = validation_predict - validation_y
```

```
    sqr_loss.append(1/validation_y.shape[0]*np.dot(diff,diff.T))
import matplotlib.pyplot as plt
plt.plot(np.arange(0.1,1.1,0.1),sqr_loss)
plt.show()
```

```
lambda: 0.1 run_time: 16.355605602264404 steps_taken: 1000
lambda: 0.2 run_time: 17.021519422531128 steps_taken: 1000
lambda: 0.3 run_time: 17.161158084869385 steps_taken: 1000
lambda: 0.4 run_time: 17.454284191131592 steps_taken: 1000
lambda: 0.5 run_time: 17.41083812713623 steps_taken: 1000
lambda: 0.6 run_time: 17.356815814971924 steps_taken: 1000
lambda: 0.7 run_time: 17.236926794052124 steps_taken: 1000
lambda: 0.8 run_time: 17.2065269947052 steps_taken: 1000
lambda: 0.9 run_time: 17.269866466522217 steps_taken: 1000
lambda: 1.0 run_time: 17.64631462097168 steps_taken: 1000
```



The sqaure loss on validation set reach minimun when $\lambda = 0.5$.

2 3_1_2

```
In [36]: w,a,c = lasso_shooting(train_X,train_y,lambda_reg=0.5)

lambda: 0.5 run_time: 18.43110227584839 steps_taken: 1000


In [37]: threshold = 0.001
         w[(w<threshold)&(w>-threshold)] = 0
         # Measure the sparsity of result
         len(w[10:][w[10:]!=0])

Out[37]: 11
```

11 out of 65 zero values have been estimated to be non-zero. (threshold = 0.001)

3 3_1_3

In [20]:
```
# Warstarting
lambda_max = max(2*np.abs(train_X.T.dot(train_y)))
def warm_start(X,y,lambda_reg=0.1,steps = 1000):
    #Get dimension info
    n = X.shape[0]
    d = X.shape[1]
    #initializing theta
    w = np.zeros(d) # result w dimension: d
    def soft(a,delta):
        sign_a = np.sign(a)
        if np.abs(a)-delta <0:
            return 0
        else:
            return sign_a*(abs(a)-delta)
    for step in range(steps):
        a = []
        c = []
    ####For loop for computing aj cj wj
        for j in range(d):
            aj = 0
            cj = 0
            for i in range(n):
                xij = X[i,j]
                aj += 2*xij*xij
                cj += 2*xij*(y[i]-w.T.dot(X[i,:])+w[j]*xij)
            w[j] = soft(cj/aj,lambda_reg/aj)
            a.append(aj)
            c.append(cj)
        a = np.array(a)
        c = np.array(c)
    return w,a,c
```

In [23]:
```
w_start,_,_ = warm_start(train_X,train_y,lambda_reg=lambda_max)
w_start
```

Out[23]:
```
array([ 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.,  0.])
```

My warmstart meets error.

4 3_1_4

```
In [49]: def lasso_shooting_vectorize(X,y,lambda_reg=0.1,max_steps = 1000,tolerence
             start_time = time.time()
             n = X.shape[0]
             d = X.shape[1]
             #initializing theta
             w = np.linalg.inv(X.T.dot(X)+lambda_reg*np.identity(d)).dot(X.T).dot(y
             steps = 0
             converge = False
             def soft(a,delta):
                 sign_a = np.sign(a)
                 pos_part = np.abs(a)-delta
                 pos_part[pos_part<0] = 0
                 return sign_a*pos_part
             # Instead of loop calculate a c w using matrix operation
             # Store a c w into three d-dimension vector
             # a can be calculated using the diagonal elements of XT.X
             while converge==False and steps<max_steps:
                 steps+=1
                 old_w = w
                 a = 2*X.T.dot(X).diagonal()
                 # steps for calculating c
                 # duplicate y-wx d times
                 y_wx = np.tile(y-X.dot(w),(d,1))
                 # duplicate w n times
                 w_n = np.tile(w,(n,1))
                 # elementwise multiplication of w_n and x
                 wjxij =w_n*X
                 # elementwise addition
                 right = y_wx.T + wjxij
                 # return c
                 c = 2*(X.T.dot(right).diagonal())
                 w = soft(a/c,lambda_reg/a)
                 convergence = np.sum(np.abs(w-old_w))<tolerence
                 run_time = time.time()-start_time
             print('lambda:',lambda_reg,'run_time:',run_time,'steps_taken:',steps)
             return w

In [51]: w = lasso_shooting_vectorize(train_X,train_y)

lambda: 0.1 run_time: 0.3244040012359619 steps_taken: 1000
```

Observe that the regularization path is significantly faster than using for_loop.

3_2

February 13, 2017

1) If $x_{ij} = 0$ then $f(w_j) = (\sum\limits_{k \neq j} w_k x_{ik} - y_i)^2 + \lambda |w_j| + \lambda \sum\limits_{k \neq j} |w_k|$. The loss function reach minimun when $w_j = 0$

2) we have:

$$
\begin{aligned}
f(w_j) &= \sum_{i=1}^{n}[w_j x_{ij} + \sum_{k\neq j} w_k x_{ik} - y_i]^2 + \lambda w_j + \lambda \sum_{k\neq j}|w_k| \\
\frac{df(w_j)}{w_j} &= 2\sum_{i=1}^{n}[w_j x_{ij} + \sum_{k\neq j} w_k x_{ik} - y_i]x_{ij} + sign(w_j)\lambda \\
&= 2\sum_{i=1}^{n} \hat{w}_j x_{ij}^2 + 2\sum_{i=1}^{n} x_{ij}(\sum_{k\neq j} w_k x_{ik} - y_i) + sign(w_j)\lambda \\
&= a_j w_j - c_j + sign(w_j)\lambda
\end{aligned}
$$

3) If $w_j > 0$, the minimizer $\hat{w}_j$ must satisfy $a_j\hat{w}_j - c_j + \lambda = 0$, then $\hat{w}_j = \frac{c_j - \lambda}{a_j}$. Similarly if $w_j < 0$, the minimizer $\hat{w}_j$ must satisfy $a_j\hat{w}_j - c_j - \lambda = 0$, then $\hat{w}_j = \frac{c_j + \lambda}{a_j}$.

Since we have $aj = 2\sum_{i=1}^{n} x_{ij}^2$ always positive, when $c_j > \lambda$, if $w_j < 0$, $w_j = \frac{c_j - sign(w_j)\lambda}{a_j}$ will result in contradiction, then w_j must be positive. Similarly when $c_j < -\lambda$, w_j must be negative.

4) Let $\alpha = \sum\limits_{k \neq j} w_k x_{ik} - y_i$ (Notice that $-2\alpha x_{ij} = c_j$) we have:

$$\lim_{\epsilon \to 0^+} \frac{f(\epsilon) - f(0)}{\epsilon} = \lim_{\epsilon \to 0^+} \frac{(\epsilon x_{ij} + \alpha)^2 - \alpha^2 + \lambda\epsilon}{\epsilon}$$

Apply L'Hopital rule,

$$\begin{aligned}\lim_{\epsilon \to 0^+} \frac{f(\epsilon) - f(0)}{\epsilon} &= \lim_{\epsilon \to 0^+} 2(\epsilon x_{ij} + \alpha)x_{ij} + \lambda \\ &= \lim_{\epsilon \to 0^+} 2\epsilon x_{ij}^2 + 2\alpha x_{ij} + \lambda \\ &= -c_j + \lambda\end{aligned}$$

Similarly

$$\lim_{\epsilon \to 0^+} \frac{f(-\epsilon) - f(0)}{\epsilon} = c_j + \lambda$$

It shows that when $-\lambda < c_j < \lambda$ the two-side derivative at f(0) is positive and $\hat{w}_j$ comebined with the conclusions of part 3, we shows that when $c_j \in [-\lambda, \lambda]$, $\hat{w}_j = 0$

5) From part 3, we know tha when $c_j > \lambda$, w_j must be positive and $\hat{w_j} = \frac{c_j - \lambda}{a_j}$; when $c_j < -\lambda$, w_j must be negative and $\hat{w_j} = \frac{c_j + \lambda}{a_j}$; From part 4 we know, When $-\lambda < c_j < \lambda$, $\hat{w_j} = 0$.

4_1

February 13, 2017

1 4_1_1)

$$
\begin{aligned}
L'(0:v) &= \lim_{h \to 0^+} \frac{L(hv) - L(0)}{h} \\
&= \lim_{h \to 0^+} \frac{||hxv - y||^2 + \lambda h ||v||_1 - ||y||^2}{h}
\end{aligned}
$$

Apply L'Hopital Rule:

$$
\begin{aligned}
L'(0:v) &= (\lim_{h \to 0^+} 2(hxv - y)^T xv) + \lambda ||v||_1 \\
&= -2y^T xv + \lambda ||v||_1
\end{aligned}
$$

2 4_1_2)

If $L'(0:v) \geq 0$ we must have $\lambda \geq \frac{2y^T xv}{||v||_1}$. Hence the lower bound of λ is $\frac{2y^T xv}{||v||_1}$

3 4_1_3)

The lower bound is equal to $\frac{\sum\limits_i 2y_i x_i v_i}{\sum\limits_i |v|} \leq \frac{\sum\limits_i 2|y_i x_i||v_i|}{\sum\limits_i |v|}$. This is a weighted sum of $2|y_i x_i|$, which is smaller than the largest component of $2|y_i x_i|$

4_2

February 13, 2017

1 4_2_1)

Claim: a and b must have the same sign.

Proof: Suppose a and b have opposite signs and are optimizer for L. Let's $c = a + b$ and $d = 0$ Since x_1 and x_2 are repeat features, $x_1 = x_2$,We must have $cx_1 + dx_2 = ax_1 + bx_2$. By Cauchy equality we must have $\lambda|a| + \lambda|b| \geq \lambda|a + b| = \lambda|c| + \lambda|d|$. Hence c and d produce smaller L than a and b do. This is a contradiction.

Claim: If $c + d = a + b$ while c and d have the same sign. Then c and d are also optimizer for L.

Proof: We must have $cx_1 + dx_2 = ax_1 + bx_2$. Since c and d have the same sign, we must have $\lambda|c| + \lambda|d| = \lambda|c + d| = \lambda|a + b| = \lambda|a| + \lambda|b|$. Then c and d produce same L as a and b. Since a and b are defined to be minimizer, L must be optimized. Then c and d are also optimizer for L.

2 4_2_2)

Claim: a and b must have the same sign.

Proof: Suppose a and b have opposite signs and are optimizer for L. Let a be the positive one and b be the negative one.

If $a + b > 0$ let $c = a + b$ and $d = 0$

Since x_1 and x_2 are repeat features, $x_1 = x_2$,We must have $cx_1 + dx_2 = ax_1 + bx_2$. If $a + b > 0$, Since b is negative then $a + b < a$, we have $|c| = |a + b| \leq |a|$ and $|d| = 0 \leq |b|$. If $a + b < 0$, Since a is positive then $a + b > b$, we have $|c| = |a + b| \leq |b|$ and $|d| = 0 \leq |a|$. We can always create a new minimizer produce less L, introducing an contradiction.

Claim: $a = b$

Proof: Holding the sum S of a and b constant. The $\lambda|a| + \lambda|b|$ is minimized when a = b. Since if both a and b are positive, $\lambda|a| + \lambda|b| = a^2 + b^2 = (a + b)^2 - 2ab = S^2 - 2a(S - a)$ is minimized when $a = \frac{S}{2}$. The identical statement hold when both a and b are negative.