Ridge regression in Python

Cristina Aguilera, Jesús M. Antoñanzas
Mathematical Optimization, Data Science and Engineering
UPC

May 2019

Contents

1 Objectives **2**

2 Preliminary algebra **3**
2.1 Objective function 3
2.2 Constraint function 3

3 Results **5**
3.1 Basic analysis 5
3.2 KKT conditions 6
3.3 Full results 7

4 Code **8**

1 Objectives

We have studied and used ridge - or Tikhonov regularized - regression in various instances during the course and using different programming environments (i.e. MATLAB, AMPL and R). In those cases, it was similar to a black box: just a couple of lines of code and all was done. This time, we have written a more transparent Ridge regression implementation, which uses functions in order to compute the derivatives and Hessians of both the objective function and its constraints. In spite of this, the optimization algorithm used is still unknown to us as we have not studied it quite yet. Then, we have studied the optimality of the found point and compared it to what other implementations give as a solution (namely, the AMPL implementation).

2 Preliminary algebra

Before implementing a solution, we first need to know what the analytical expression of both the derivatives and the Hessians associated to our objective function and its constraints (in this case, just one) are.

2.1 Objective function

The objective function is

$$f(w,\gamma) = \frac{1}{2}(Aw + \gamma e - y)^T(Aw + \gamma e - y)$$

where $A \in \mathcal{R}^{m\times n}$, $w \in \mathcal{R}^{n\times 1}$, $\gamma \in \mathcal{R}$, $e \in \mathcal{R}^{m\times 1}$, $y \in \mathcal{R}^{m\times 1}$, e being a vector of ones. The gradient of $f(w,\gamma) \in \mathcal{R}^{(n+1)\times 1}$ is given by

$$\nabla f(w,\gamma) = \begin{bmatrix} \frac{\partial f(w,\gamma)}{\partial w} \\ \frac{\partial f(w,\gamma)}{\partial \gamma} \end{bmatrix}$$

where

$$\frac{\partial f(w,\gamma)}{\partial w} = A^T(Aw + \gamma e - y)$$

and

$$\frac{\partial f(w,\gamma)}{\partial \gamma} = w^T A^T e + n\gamma - e^T y$$

Its Hessian $\nabla^2 f(w,\gamma) \in \mathcal{R}^{(n+1)\times(n+1)}$ is given by the expression

$$\nabla^2 f(w,\gamma) = \begin{bmatrix} \frac{\partial f(w,\gamma)}{\partial^2 w} & \frac{\partial f(w,\gamma)}{\partial w \partial \gamma} \\ \frac{\partial f(w,\gamma)}{\partial \gamma \partial w} & \frac{\partial f(w,\gamma)}{\partial^2 \gamma} \end{bmatrix}$$

where

$$\frac{\partial f(w,\gamma)}{\partial^2 w} = A^T A, \quad \frac{\partial f(w,\gamma)}{\partial w \partial \gamma} = A^T e, \quad \frac{\partial f(w,\gamma)}{\partial \gamma \partial w} = (A^T e)^T, \quad \frac{\partial f(w,\gamma)}{\partial^2 \gamma} = n \tag{1}$$

2.2 Constraint function

The constraint function is

$$h(w,\gamma) = \|w\|_2^2 \leq t$$

Its gradient $\nabla h(w,\gamma) \in \mathcal{R}^{(n+1)\times 1}$ is

$$\nabla h(w,\gamma) = \begin{bmatrix} \frac{\partial h(w,\gamma)}{\partial w} \\ \frac{\partial h(w,\gamma)}{\partial \gamma} \end{bmatrix}$$

where

$$\frac{\partial h(w,\gamma)}{\partial w} = 2w, \quad \frac{\partial h(w,\gamma)}{\partial \gamma} = 0 \tag{2}$$

and its Hessian $\nabla^2 h(w,\gamma) \in \mathcal{R}^{(n+1)\times(n+1)}$ is

$$\nabla^2 h(w,\gamma) = \begin{bmatrix} \frac{\partial h(w,\gamma)}{\partial^2 w} & \frac{\partial h(w,\gamma)}{\partial w \partial \gamma} \\ \frac{\partial h(w,\gamma)}{\partial \gamma \partial w} & \frac{\partial h(w,\gamma)}{\partial^2 \gamma} \end{bmatrix}$$

Note that in the Hessian of the constraint we have taken γ into account, and that is because in the implementation we need all the parameters (w and γ) in a single vector. This procedure results in a matrix with the only elements different from 0 are 2's in the diagonal, except for the element corresponding to the position [i, i], i being the position of γ in the aforementioned parameter vector. For us, γ was the last in the vector (i = n+1).

$$\nabla^2 h(w,\gamma) = \begin{bmatrix} 2 & 0 & 0 & \dots & 0 \\ 0 & 2 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 2 & 0 \\ 0 & 0 & \dots & 0 & 0 \end{bmatrix}$$

3 Results

3.1 Basic analysis

For this data set, we have a total of 15 explanatory variables and 60 observations. The response variable is the death rate of a certain country. We first load this data set and its corresponding ridge regression model in AMPL and solve it, returning the following solution.

```
MINOS 5.51: optimal solution found.
133 iterations, objective 61484.65464
Nonlin evals: obj = 361, grad = 360, constrs = 361, Jac = 360.
w [*] :=
 1    0.48824
 2   -0.0463881
 3    0.102978
 4   -0.0377061
 5    0.00490736
 6   -0.0339172
 7   -0.255786
 8    0.00564845
 9    0.649708
10   -0.12622
11    0.213407
12   -0.207831
13    0.109842
14    0.376641
15    0.00995978
;

gamma = 895.141
norm2_w = -10785.1
```

Listing 1: AMPL solution of the "Death rate" regression.

As we can see, 133 iterations were needed for the routines to find a minimizer. Executing our code, the exact same solution is found in the same number of iterations.

```
In [0]: solution.nfev
Out[0]: 133

In [1]: solution.x
Out[1]:
array([ 4.88239571e-01, -4.63880870e-02,  1.02977785e-01, -3.77061350e-02,
        4.90736137e-03, -3.39172227e-02, -2.55785705e-01,  5.64845151e-03,
        6.49707817e-01, -1.26220227e-01,  2.13407411e-01, -2.07831309e-01,
        1.09842480e-01,  3.76641023e-01,  9.95977558e-03,  8.95141390e+02])
```

Listing 2: Python solution of the "Death rate" regression.

Where, in our case, γ is the last value in the x vector and ω is the rest of the vector. As we can see, the value of the objective function at our solution is the same as in the AMPL one.

```
In[2]: solution.fun
Out[2]: 61484.65464046297
```

Listing 3: Value of the objective function at the found minimizer.

On the other hand, execution time with Python is not faster, as we can see taking a look at the "_solve_time" environmental variable, on which AMPL saves the "system CPU seconds used by most recent solve command" plus the "user CPU seconds used by most recent solve command".

```
ampl: display _solve_time;
_solve_time = 0.007453
```

Listing 4: AMPL solve time for the "Death rate"data set.

While in Python:

```
In [3]: solution.execution_time
Out[4]: 0.2936720848083496
```

Listing 5: Python execution time for the "Death rate" data set.

AMPL is much faster. At the end of the section we explain some of the reasons why this could happen.

3.2 KKT conditions

Knowing that our solution is the same as the AMPL one is a good indication that our point is optimal, but we can also deduce that from the output of the Python routines. Specifically, as Ridge regression is a strictly convex problem, if our minimizer meets the KKT conditions, then that it is a sufficient condition so that it is optimal, as it will always satisfy the sufficient second-order conditions. The first KKT condition is **feasibility**. Let's check that $h(\omega, \gamma) \leq t$.

```
In [4]: cons(solution.x)
Out[4]: 0.9999999999979048
```

Listing 6: Feasibility KKT condition.

The first condition is met. Now, the **second condition**: $\nabla L(\omega^*, \gamma^*, \lambda^*) = 0$.

```
In [5]: solution.lagrangian_grad
Out[5]:
array([-6.64695108e-08,  5.67613370e-08,  4.70404302e-08,  6.35918695e-09,
        5.68779939e-08, -8.96648089e-10,  3.83806764e-10,  6.66005917e-09,
        4.45743353e-08, -6.24581844e-08,  4.29736247e-09, -4.31409717e-08,
       -1.83293196e-07,  1.04246283e-08, -2.83398265e-08, -9.45874490e-11])
```

Listing 7: 2^{nd} KKT condition.

We can say this one is also met. Lastly, **complementary slackness**. We need for $\mu^* \geq 0$ (in this case, $\mu^* = \nu^*$) and $\nu^* g(\omega^*, \gamma^*) = 0$, where $g(\omega, \gamma) = \|\omega\|_2^2 - t$. Running a quick check:

```
In [6]: solution.v[0]
Out[6]: array([10785.1067161])

In [7]: solution.v[0]*(cons(solution.x)-t)
Out[7]: array([-2.25970946e-08])
```

Listing 8: Complementary slackness KKT condition.

So, KKT conditions are met at the found minimizer, meaning that, in fact, the found point is a global minimum.

A comment on speed. Our implementation is not faster than the AMPL one in spite of ours not needing to compute the gradients and Hessians symbolically. Of course, AMPL is a professional software, many times tested and improved, but we can take a guess at the reasons for this being the case are. First, because of the way dot products and matrix multiplications are performed. They can be improved with more proficient knowledge of the data structures and, specially, of the problem, as there are lots of so called "tricks" that we don't know of that help speed up the optimization and use less resources (for example, sparse matrices manipulation). Lastly, while on our implementations there is no need for the routines to compute the first and second derivatives symbolically, there are not so many in this problem so that it makes a difference.

3.3 Full results

The following is the full Python output of the optimization routine.

```
In [8]: print(solution)
`xtol` termination condition is satisfied.
Number of iterations: 104, function evaluations: 133, CG iterations: 599,
    optimality: 1.83e-07, constraint violation: 0.00e+00, execution time: 0.29 s.
 barrier_parameter: 2.0480000000000001e-09
 barrier_tolerance: 2.0480000000000001e-09
          cg_niter: 599
      cg_stop_cond: 4
            constr: [array([1.])]
       constr_nfev: [133]
       constr_nhev: [69]
       constr_njev: [63]
    constr_penalty: 15407.287523808163
  constr_violation: 0.0
    execution_time: 0.2936720848083496
               fun: 61484.65464046297
              grad:
array([-1.05314317e+04,  1.00060094e+03, -2.22125280e+03,  8.13329381e+02,
       -1.05852832e+02,  7.31601733e+02,  5.51735226e+03, -1.21838305e+02,
       -1.40143363e+04,  2.72259723e+03, -4.60324341e+03,  4.48296570e+03,
       -2.36932573e+03, -8.12422726e+03, -2.14834485e+02, -9.45874490e-11])
               jac:
[array([[0.97647914, -0.09277617,  0.20595557, -0.07541227,  0.00981472,
        -0.06783445, -0.51157141,  0.0112969 ,  1.29941563, -0.25244045,
         0.42681482, -0.41566262,  0.21968496,  0.75328205,  0.01991955,
         0.        ]])]
   lagrangian_grad:
array([-6.64695108e-08,  5.67613370e-08,  4.70404302e-08,  6.35918695e-09,
        5.68779939e-08, -8.96648089e-10,  3.83806764e-10,  6.66005917e-09,
        4.45743353e-08, -6.24581844e-08,  4.29736247e-09, -4.31409717e-08,
       -1.83293196e-07,  1.04246283e-08, -2.83398265e-08, -9.45874490e-11])
          message: 'xtol termination condition is satisfied.'
           method: 'tr_interior_point'
             nfev: 133
             nhev: 63
            niter: 104
             njev: 63
       optimality: 1.8329319573240355e-07
           status: 2
        tr_radius: 1.3176355152026543e-09
                v: [array([10785.1067161])]
                x:
array([ 4.88239571e-01, -4.63880870e-02,  1.02977785e-01, -3.77061350e-02,
        4.90736137e-03, -3.39172227e-02, -2.55785705e-01,  5.64845151e-03,
        6.49707817e-01, -1.26220227e-01,  2.13407411e-01, -2.07831309e-01,
        1.09842480e-01,  3.76641023e-01,  9.95977558e-03,  8.95141390e+02])
```

Listing 9: Full Python output.

On a final note, notice that the constrain is not violated: *constraint violation* : $0.00e+00$.

4 Code

Here is the full code, which has also been attached to this document in conjunction with the "Death rate" data set in case it needs to be tried.

```
# Authors: Cristina Aguilera, Jesus Antonanzas
#          GCED, OM - grup 11.
#
# Python implementation of ridge - or Tikhonov - regularization:
#
# min           1/2(A*w + gamma - y)'*(A*w + gamma - y)
# subject to    w'*w <= t
#
# Where 'A' is a set of observations, 'gamma' a vector of real numbers
# and 'y' the target variable.

import numpy as np
from scipy.optimize import minimize
from scipy.optimize import NonlinearConstraint

# ========== INPUT DATA ==========
# Read dataset.
# Observations by rows and variables by columns.
# Target variable needs to be the last column.

# A = np.loadtxt('octane_rating.txt', usecols =(1,2,3,4,5))
A = np.loadtxt('deathrate_instance_python.dat')

# Copy target variable.
y = A[:,len(A[0])-1]

# Delete target variable from dataset.
A = A[:,0:len(A[0])-1]

# Declare dimensions:
#   m = # of observations.
#   n = # of variables.
m = A.shape[0]
n = A.shape[1]

# ========== PRELIMINARY COMPUTATIONS ==========

# First, declare variables that remain constant
# and come in handy during calculations.

At = A.transpose()
AtA = np.matmul(At,A)
Aty = np.matmul(At,y)
ones_m = np.ones(m)
ones_dot_y = np.dot(ones_m,y)

# Gradient of the restriction:
upperleft = np.matmul(At,A)
upperright = np.matmul(At,np.ones((m,1)))
tophalf = np.append(upperleft, upperright, axis = 1)
bottomhalf = np.append(upperright,m)
const_grad_var = np.append(tophalf, [bottomhalf], axis = 0)

# Hessian of the constraint
cons_hess_var = np.diag(np.append(np.full((n),2),0))


# ========== EVALUATION FUNCTIONS ==========
```

```
59
60 def obj_f(w):
61     # Pos 15 of w contains gamma, all other positions are w_i, i = 0...n-1 (num.
       var).
62     gamma = w[n]
63     w = w[0:n]
64     # gamma = gamma * np.ones((m,1)) veure si funciona sense
65     p1 = np.dot(A, w) + gamma * ones_m - y
66     return 1/2 * np.dot(p1, p1)
67
68 def obj_grad(w):
69     # Pos 16 of w contains gamma, all other positions are w_i, i = 0...n-1 (num.
       var).
70     gamma = w[n]
71     w = w[0:n]
72     grad = np.zeros(n+1)
73     AtAw = np.matmul(AtA,w)
74     grad[:-1] = AtAw + np.matmul(At, gamma*ones_m) - Aty
75     grad[-1] = np.dot(np.matmul(w.transpose(),At),ones_m) + m*gamma - ones_dot_y
76     return grad
77
78 def obj_hess(w):
79     # Returned value remains constant, so just calculate once at the initialisation
       .
80     return const_grad_var
81
82 def cons(w):
83     w = w[0:n]
84     return np.dot(w.transpose(),w)
85
86 def cons_grad(w):
87     return np.append(2*w[0:n],0)
88
89 def cons_hess(w,v):
90     # Returns sum of all constraint Hessians times their respective Lagrange
       multipliers.
91     return v[0]*cons_hess_var
92
93 # ========== MINIMIZATION ==========
94
95 # Declare initial values.
96 w0 = np.zeros(n+1)
97 t = 1
98
99 # Declare constraint.
100 nonlinear_constraint = NonlinearConstraint(cons, lb = -np.inf, ub = t,
101                                            jac=cons_grad, hess=cons_hess)
102
103 # Variables are not bounded. We could write
104
105 # bounds = scipy.optimize.Bounds(-np.inf, np.inf)
106
107 # but increases computation time.
108
109 # Minimization.
110 solution = minimize(obj_f, w0, method='trust-constr', jac=obj_grad,
111                     hess=obj_hess, constraints=[nonlinear_constraint],
112                     options={'verbose': 1})
113
114 # Display the solution.
115 print(solution)
```

Listing 10: Full Python code.