Margarita Geleta – Oriol Narváez Serrano

Mathematical Optimization | Data Science

RIDGE REGRESSION

*In Python*

**About the project**

The aim of the project is to implement and solve in *Python* the constrained ridge regression model:

*subject to* .

The data set we are going to use as an example is the *death rate* data set with 60 observations, where the *death rate* is represented as a function of 15 features as the *nitric oxide* pollution index, the *hydrocarbon* pollution index and so on.

In order to implement the constrained ridge regression model, we first need to find the formulae that define the objective function , the gradient of the objective function , the hessian of the objective function , the inequality constraint , its gradient (the *Jacobian*) and the hessian of the constraint . So, we are going to work step by step upon these functions.

**The objective function**

It is clear that the objective function is the one we want to minimize, that is:

To add some ease to the implementation, it is better to group in just one variable and thus, we add a column of ones in the matrix :

And we can simplify the previous expression of as follows:

In *Python*:

|  |
| --- |
| **def loss\_func** (x):  w = np.asarray(x).T    cost = (1/2) \* ((A @ w) - y).T @  ((A @ w) - y)  **return** np.squeeze(cost) |

**The gradient**

Using the original expression, we have two variables, so we have to take derivatives from with respect to both . Arranging:

An easier expression of the gradient can be obtained using the simplified version :

We define it in *Python* as follows:

|  |
| --- |
| **def loss\_grad** (x):  w = np.asarray(x).T    **return** A.T @ (A @ w - y) |

**The hessian**

Using the original expression we get a more complex expression:

We obtain a matrix by blocks. Using the simplified version of we obtain a much nicer expression:

In *Python* it is as simple as:

|  |
| --- |
| **def loss\_hess** (x):  w = np.asarray(x).T    **return** A.T @ A |

**The inequality constraint**

The *squared Euclidean norm* can be expressed as:

Since we are using , we are going to ignore the last component setting it to zero:

So, in *Python*:

|  |
| --- |
| **def cons\_func** (x):  w = np.asarray(x).T  w = w[:-1]  **return** np.sum(w\*\*2) |

**The *J*acobian**

We are going to use the same trick; we are going to set . Thus, we get:

In *Python*:

|  |
| --- |
| **def cons\_jacobian** (x):  x[-1] = 0  w = np.asarray(x).T  **return** (2\*w).tolist() |

**The hessian of the constraint**

We obtain a diagonal matrix with 2’s on the diagonal except on the position because of being , it is zero:

The way the minimization algorithm is implemented, it does not need a function that returns the raw hessian of the constraint(s), it needs a linear combination of the hessians. Since we just have one constraint, and therefore, just one hessian, we should return the hessian multiplied by an arbitrary value :

|  |
| --- |
| **def cons\_hess** (x, v):  x[-1] = 0  w = np.asarray(x).T    H = np.zeros((w.shape[0], w.shape[0]))  np.fill\_diagonal(H,2)  H[-1,-1] = 0  **return** v \* H |

**Minimization: *death rate* data set**

Now that all the ingredients are set up, we need to plug them into the minimization function from the scipy.optimize library:

|  |
| --- |
| # Define hyperparamater t:  t = 1  ## OPTIMIZATION ##  # Assign initial weights:  w = np.ones(16)  # Define non linear constraint:  nonlinear\_constraint =  **NonlinearConstraint**(cons\_func, -np.inf, t,  jac = cons\_jacobian, hess = cons\_hess)  # Solve:  sol = **minimize**(loss\_func, w, method='trust-  constr', jac=loss\_grad, hess=loss\_hess,  constraints=[nonlinear\_constraint],  options={'verbose': 1})  # Output:  print(sol) |

*Notes*:

* We must define beforehand, since it is a hyperparameter.
* We used a vector with zero components as the initial weights vector , since random initial weights with a seed did not lead to a good solution.

The complete output, using the *death rate* dataset, can be examined at the end of this paper (*Output 1*). Here, we are just going to show the found solution and the gradient:

|  |
| --- |
| # Solution ():  ([ 0.4882, -0.0464, 0.1029, -0.0377,  0.0049, -0.0339, -0,2557, 0.0056,  0.6497, -0.1262, 0.2134, -0.2078,  0.1098, 0.3766, 0.0099, 895.141])  # gamma is the last component!  # Gradient norm: 0.21534461815763473 |

Note that minimizing without the constraint, we obtain the coefficients of the least squares linear regression (*unconstrained*), (*Output 2*):

|  |
| --- |
| # Solution ():  ([ 2.072, -2.177, -2.8319, -14.033,  -115.329, -24.24, -1.145, 0.01,  3.533, 0.5232, 0.268, -0.8889,  1.866, -0.034, 0.533, 1862.39])  # gamma is the last component!  # Gradient norm: 0.08729604430708358 |

**Conclusions**

We have been able to implement the constrained ridge model in *Python* and solve successfully the optimization problem with the *death rate* data set.

**References**

*J. CASTRO & F. J. HEREDIA*, *Mathematical Optimization lecture notes*.

*Output 1*

`xtol` termination condition is satisfied.

Number of iterations: 144, function evaluations: 191, CG iterations: 794, optimality: 3.97e-05, constraint violation: 0.00e+00, execution time: 0.27 s.

barrier\_parameter: 2.048000000000001e-09

barrier\_tolerance: 2.048000000000001e-09

cg\_niter: 794

cg\_stop\_cond: 4

constr: [array([1.])]

constr\_nfev: [191]

constr\_nhev: [91]

constr\_njev: [86]

constr\_penalty: 1.0

constr\_violation: 0.0

execution\_time: 0.2699158191680908

fun: 61484.65464046687

grad: array([-1.05314313e+04, 1.00060115e+03, -2.22125180e+03, 8.13329504e+02,

-1.05852785e+02, 7.31601887e+02, 5.51735331e+03, -1.21838297e+02,

-1.40143362e+04, 2.72259783e+03, -4.60324321e+03, 4.48296568e+03,

-2.36932573e+03, -8.12422725e+03, -2.14833708e+02, 1.38982657e-05])

jac: [array([[ 0.9764791 , -0.09277619, 0.20595548, -0.07541228, 0.00981472,

-0.06783446, -0.51157151, 0.0112969 , 1.29941563, -0.25244051,

0.42681481, -0.41566262, 0.21968496, 0.75328205, 0.01991948,

0. ]])]

lagrangian\_grad: array([-7.95691449e-06, -7.13667055e-06, 2.56749299e-05, -3.02727517e-06,

2.19963935e-05, 1.20500522e-06, -3.96506430e-05, -2.06478006e-06,

-1.83694065e-05, 8.83560779e-06, 1.73796743e-05, -1.15212924e-05,

-4.01049829e-06, -5.80644701e-06, 1.72957752e-05, 1.38982657e-05])

message: '`xtol` termination condition is satisfied.'

method: 'tr\_interior\_point'

nfev: 191

nhev: 86

nit: 144

niter: 144

njev: 86

optimality: 3.965064297517529e-05

status: 2

success: True

tr\_radius: 3.0038937028194658e-09

v: [array([10785.1066819])]

x: array([ 4.88239551e-01, -4.63880973e-02, 1.02977740e-01, -3.77061410e-02,

4.90736021e-03, -3.39172299e-02, -2.55785757e-01, 5.64845110e-03,

6.49707815e-01, -1.26220254e-01, 2.13407404e-01, -2.07831310e-01,

1.09842480e-01, 3.76641024e-01, 9.95974041e-03, 8.95141404e+02])

*Output 2*

`xtol` termination condition is satisfied.

Number of iterations: 437, function evaluations: 426, CG iterations: 6106, optimality: 8.34e-02, constraint violation: 0.00e+00, execution time: 2.3 s.

cg\_niter: 6106

cg\_stop\_cond: 4

constr: []

constr\_nfev: []

constr\_nhev: []

constr\_njev: []

constr\_penalty: 1.0

constr\_violation: 0

execution\_time: 2.34763503074646

fun: 23000.38378504645

grad: array([ 0.00508902, 0.000236 , -0.00427007, -0.02073108, 0.00183165,

0.00412808, 0.00030917, 0.08338374, -0.00225027, 0.01116387,

-0.00093402, -0.00107135, 0.00372709, 0.00236066, 0.00010012,

-0.00470158])

jac: []

lagrangian\_grad: array([ 0.00508902, 0.000236 , -0.00427007, -0.02073108, 0.00183165,

0.00412808, 0.00030917, 0.08338374, -0.00225027, 0.01116387,

-0.00093402, -0.00107135, 0.00372709, 0.00236066, 0.00010012,

-0.00470158])

message: '`xtol` termination condition is satisfied.'

method: 'equality\_constrained\_sqp'

nfev: 426

nhev: 425

nit: 437

niter: 437

njev: 425

optimality: 0.0833837449317798

status: 2

success: True

tr\_radius: 3.499966067021263e-09

v: []

x: array([ 2.07211884e+00, -2.17740576e+00, -2.83192146e+00, -1.40338963e+01,

-1.15329889e+02, -2.42401483e+01, -1.14517625e+00, 1.00428382e-02,

3.53300851e+00, 5.23275992e-01, 2.68007969e-01, -8.88934149e-01,

1.86632165e+00, -3.44450563e-02, 5.33898531e-01, 1.86239381e+03])