RIDGE REGRESSION IN PYTHON

2nd *Lab Assignment of MO*

David Bergés Lladó, Roser Cantenys Sabà

Table of Contents

INTRODUCTION

Ridge Regression is a technique for analyzing multiple regression data that suffer from multicollinearity. It is very similar to least squares, but the coefficients are estimated by minimizing a slightly different equation:

$$RSS = \sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2$$

$$subject\ to\colon \|\beta\|_2^2 < t$$

The majority of programming languages solve this similar expression instead:

$$RSS + \lambda\sum_{j=1}^{p}\beta_j^2$$

Minimizing the constrained form or the unconstrained one doesn't give the same solution, but the idea behind is the same. The objective is to introduce a shrinkage penalty. 't' and λ are kind off inversely proportional, in the sense that when t is small, a greater penalty to the coefficients is applied, contrary to λ, where the penalty is greater when it is bigger.

These parameters, also called the tuning parameters serve to control the relative impact of these two terms on the regression coefficient estimates. For example, when $\lambda = 0$, the penalty term has no effect, and ridge regression will produce the least squares estimates. However, as $\lambda \rightarrow \infty$, the impact of the shrinkage penalty grows, and the ridge regression coefficient estimates will approach zero.

Ridge Regression's advantage over least squares is rooted in the bias-variance trade-off. As λ increases or t decreases, the flexibility of the ridge regression fit decreases, leading to decreased variance but increased bias. In general, in situations where the relationship between the response and the predictors is close to linear, the least squares estimates will have low bias but may have high variance. This means that a small change in the training data can cause a large change in the least square's coefficient estimates. In particular, when the number of variables is almost as large as the number of observations the least squares estimates will be extremely variable, and if $p > n$, then the least squares estimates do not even have a unique solution, whereas ridge regression can still perform well by trading off a small increase in bias for a large decrease in variance.

CODE

Our goal in this project is to implement and solve, in Python, the constrained ridge regression model:

$$\min_{\omega,\gamma} \frac{1}{2}(A\omega + \gamma - y)^T (A\omega + \gamma - y)$$
$$s.to \; \|\omega\|_2^2 \leq t$$

We have to implement code for the evaluation of: $f(x)$, $\nabla f(x)$, $\nabla^2 f(x)$, $h_i(x)$, $\nabla h_i(x)$, $\nabla^2 h_i(x)$, i=1,…,m. First of all, we find these derivatives by hand:

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

$$\frac{\partial f(\omega,\gamma)}{\partial\omega} = A^T A\omega + A^T e\gamma - A^T y; \quad \frac{\partial f(\omega,\gamma)}{\partial\gamma} = e^T A\omega + \gamma e^T e - e^T y$$

$$\nabla^2 f(\omega,\gamma) = \begin{bmatrix} \frac{\partial^2 f(\omega,\gamma)}{\partial^2\omega\omega} & \frac{\partial^2 f(\omega,\gamma)}{\partial^2\omega\gamma} \\ \frac{\partial^2 f(\omega,\gamma)}{\partial^2\gamma\omega} & \frac{\partial^2 f(\omega,\gamma)}{\partial^2\gamma\gamma} \end{bmatrix} = \begin{bmatrix} A^T A & A^T e \\ e^T A & e^T e \end{bmatrix}$$

$$h(\omega,\gamma) = \|\omega\|_2^2$$

$$\frac{\partial h(\omega,\gamma)}{\partial\omega} = 2\omega; \quad \frac{\partial h(\omega,\gamma)}{\partial\gamma} = 0$$

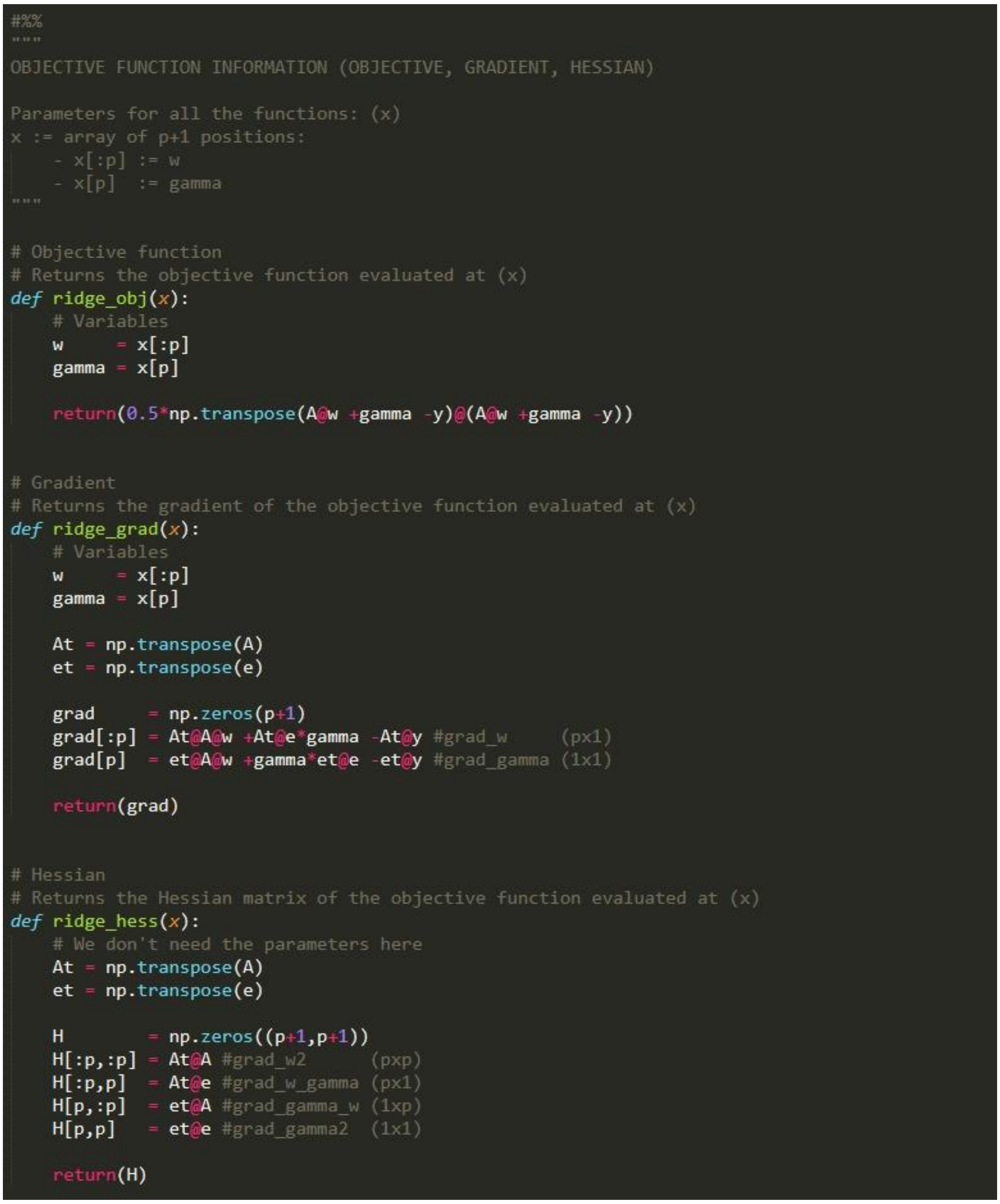
$$\nabla^2 h(\omega,\gamma) = \begin{bmatrix} \frac{\partial^2 h(\omega,\gamma)}{\partial^2\omega\omega} & \frac{\partial^2 h(\omega,\gamma)}{\partial^2\omega\gamma} \\ \frac{\partial^2 h(\omega,\gamma)}{\partial^2\gamma\omega} & \frac{\partial^2 h(\omega,\gamma)}{\partial^2\gamma\gamma} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} 2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 2 \end{bmatrix} & 0 \\ 0 & 0 \end{bmatrix}$$

Where A is the model matrix with dimension NxP, y is the response value, ω is the vector of coefficients of the Rigde regression, and γ is the intercept or the offset of the model.

e is an auxiliary parameter defined as a vector of ones with length N.

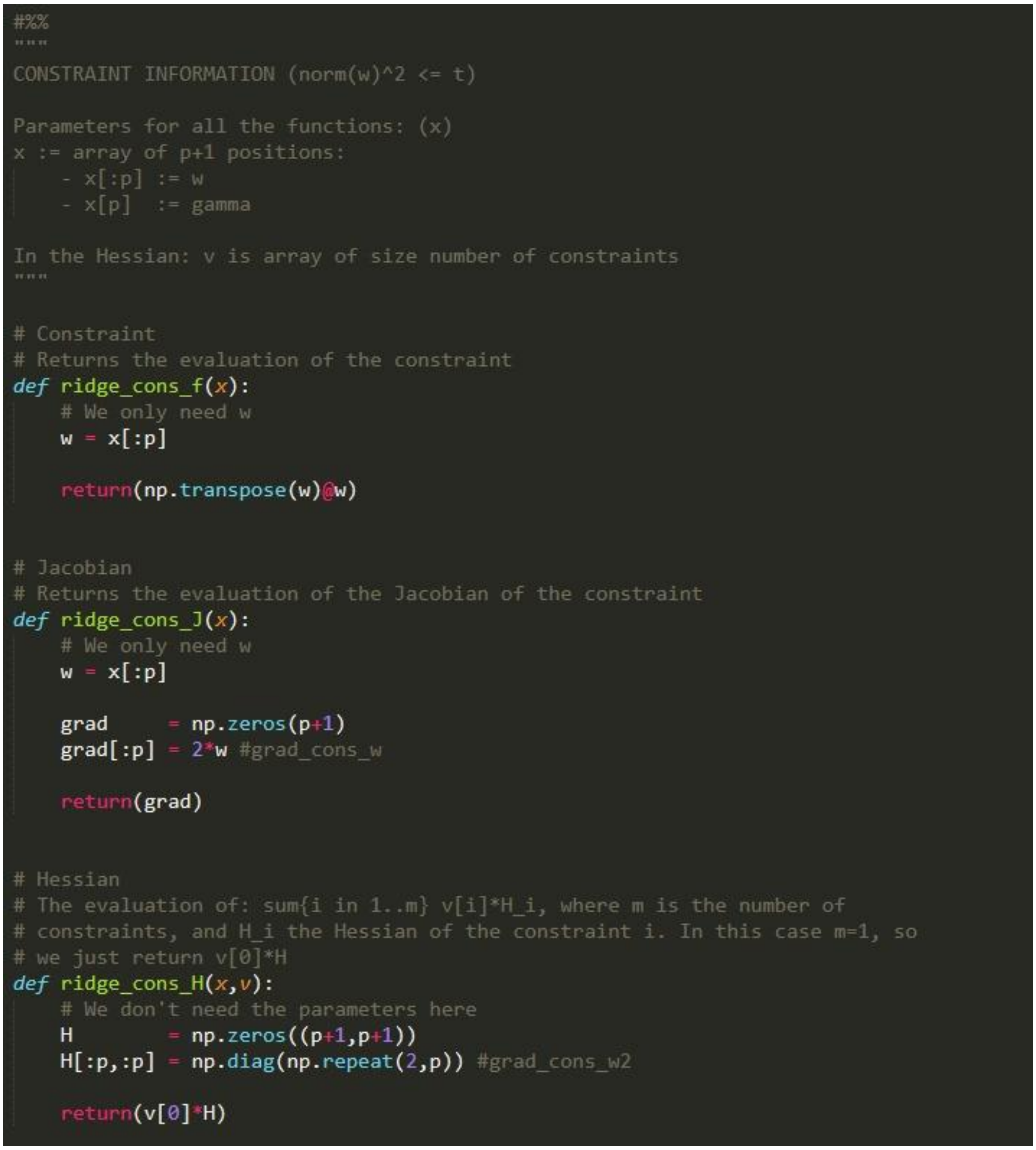
Now, we are going to implement the previous expressions with Python.

Objective function with its gradient and its Hessian



```python
#%%
"""
OBJECTIVE FUNCTION INFORMATION (OBJECTIVE, GRADIENT, HESSIAN)

Parameters for all the functions: (x)
x := array of p+1 positions:
    - x[:p] := w
    - x[p]  := gamma
"""

# Objective function
# Returns the objective function evaluated at (x)
def ridge_obj(x):
    # Variables
    w     = x[:p]
    gamma = x[p]

    return(0.5*np.transpose(A@w +gamma -y)@(A@w +gamma -y))


# Gradient
# Returns the gradient of the objective function evaluated at (x)
def ridge_grad(x):
    # Variables
    w     = x[:p]
    gamma = x[p]

    At = np.transpose(A)
    et = np.transpose(e)

    grad      = np.zeros(p+1)
    grad[:p] = At@A@w +At@e*gamma -At@y #grad_w     (px1)
    grad[p]  = et@A@w +gamma*et@e -et@y #grad_gamma (1x1)

    return(grad)


# Hessian
# Returns the Hessian matrix of the objective function evaluated at (x)
def ridge_hess(x):
    # We don't need the parameters here
    At = np.transpose(A)
    et = np.transpose(e)

    H         = np.zeros((p+1,p+1))
    H[:p,:p] = At@A #grad_w2       (pxp)
    H[:p,p]  = At@e #grad_w_gamma  (px1)
    H[p,:p]  = et@A #grad_gamma_w  (1xp)
    H[p,p]   = et@e #grad_gamma2   (1x1)

    return(H)
```

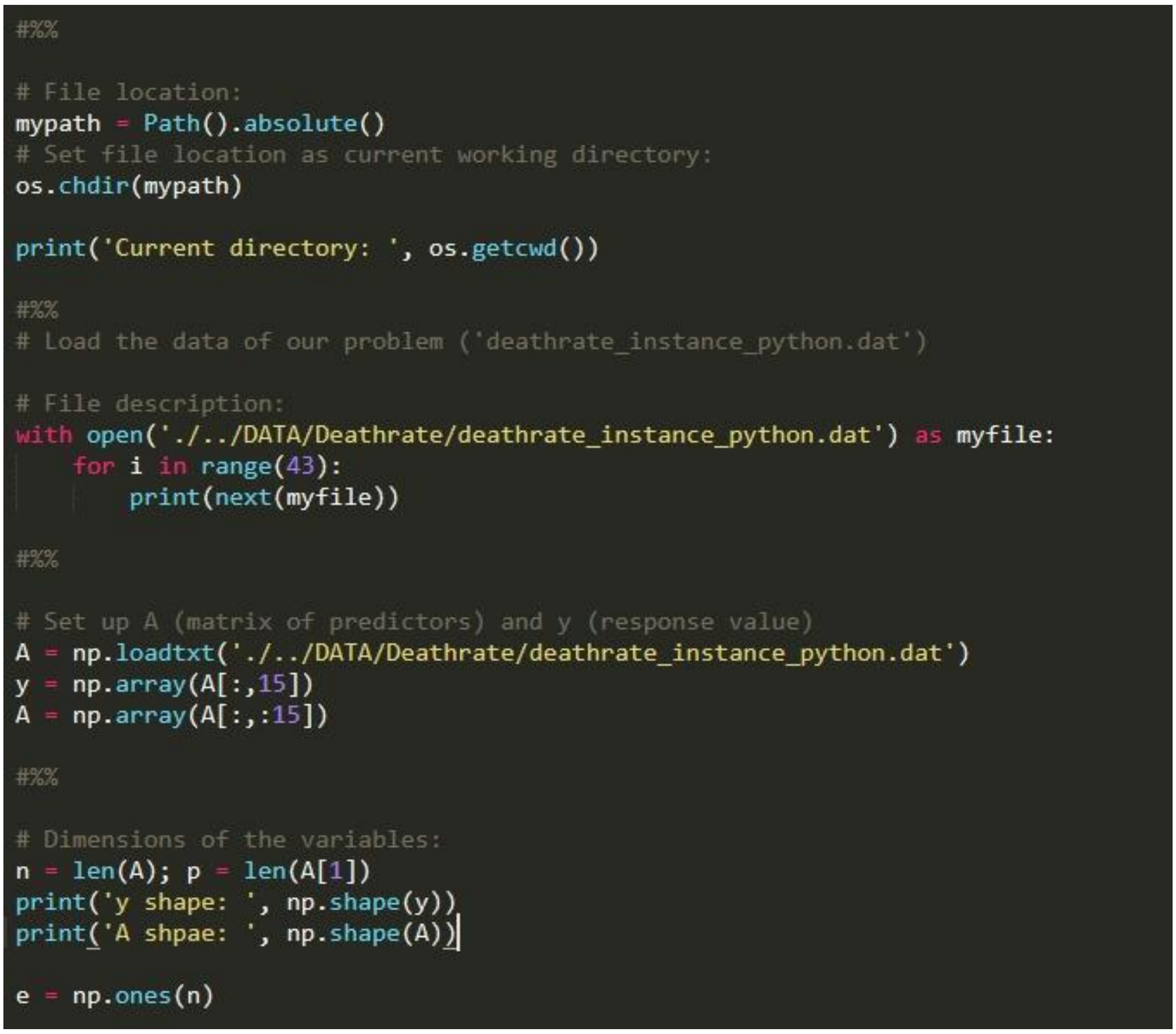
Constraint function with its Jacobian and its Hessian



```python
#%%
"""
CONSTRAINT INFORMATION (norm(w)^2 <= t)

Parameters for all the functions: (x)
x := array of p+1 positions:
    - x[:p] := w
    - x[p]  := gamma

In the Hessian: v is array of size number of constraints
"""

# Constraint
# Returns the evaluation of the constraint
def ridge_cons_f(x):
    # We only need w
    w = x[:p]

    return(np.transpose(w)@w)


# Jacobian
# Returns the evaluation of the Jacobian of the constraint
def ridge_cons_J(x):
    # We only need w
    w = x[:p]

    grad      = np.zeros(p+1)
    grad[:p] = 2*w #grad_cons_w

    return(grad)


# Hessian
# The evaluation of: sum{i in 1..m} v[i]*H_i, where m is the number of
# constraints, and H_i the Hessian of the constraint i. In this case m=1, so
# we just return v[0]*H
def ridge_cons_H(x,v):
    # We don't need the parameters here
    H         = np.zeros((p+1,p+1))
    H[:p,:p] = np.diag(np.repeat(2,p)) #grad_cons_w2

    return(v[0]*H)
```

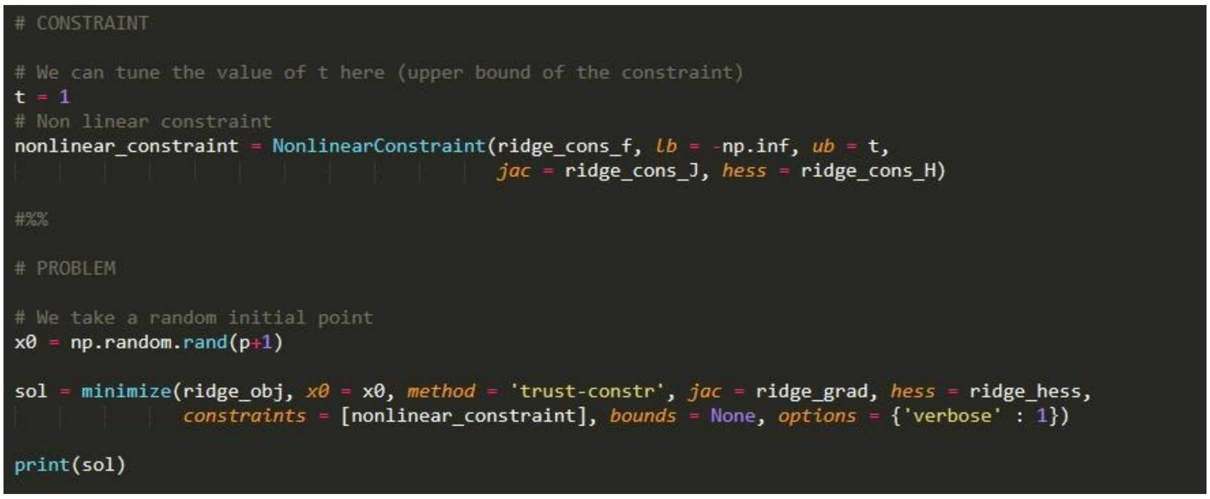
IMPLEMENTATION WITH “DEATH RATE” DATA SET

First, we load the data:



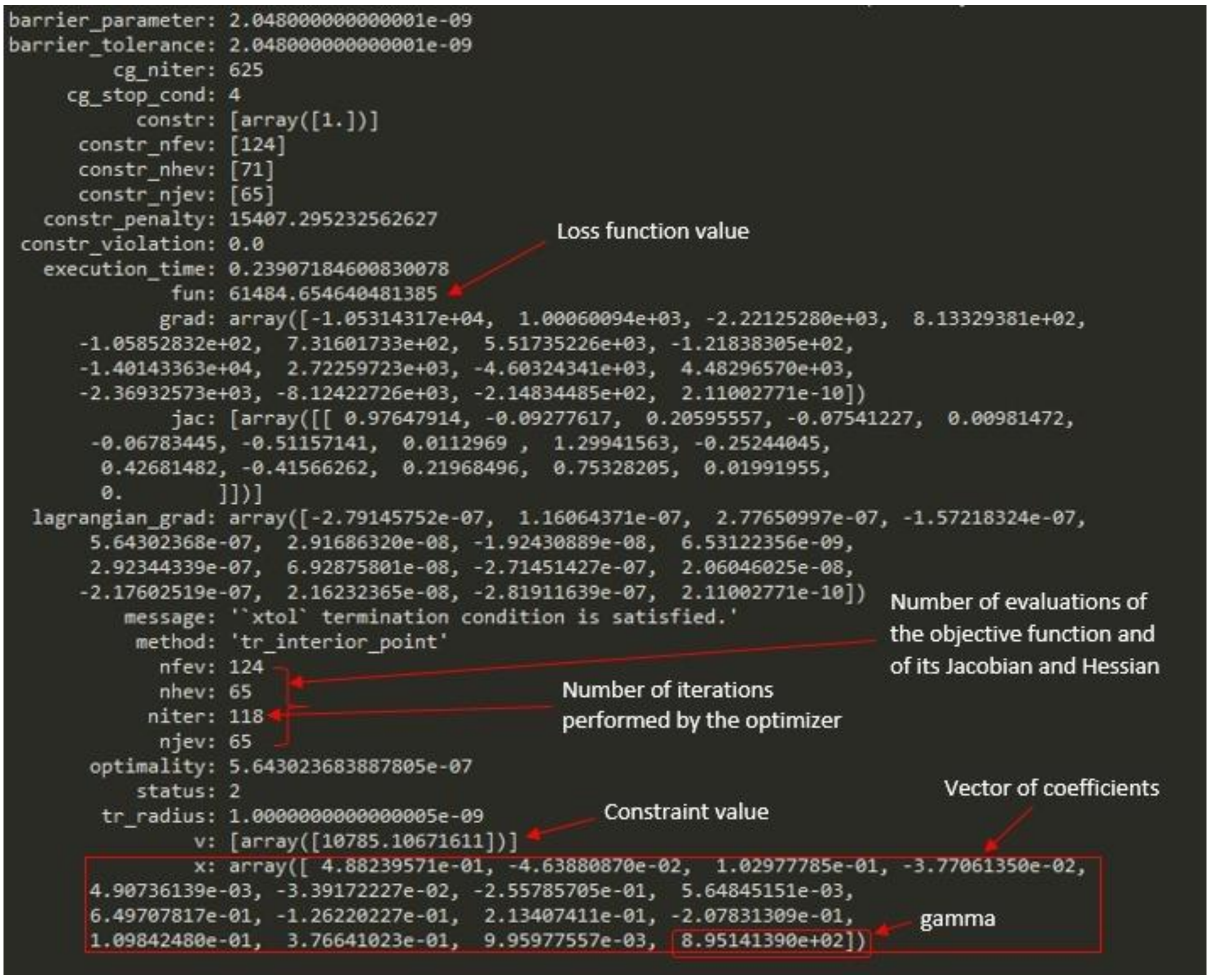
```
#%%

# File location:
mypath = Path().absolute()
# Set file location as current working directory:
os.chdir(mypath)

print('Current directory: ', os.getcwd())

#%%
# Load the data of our problem ('deathrate_instance_python.dat')

# File description:
with open('./../DATA/Deathrate/deathrate_instance_python.dat') as myfile:
    for i in range(43):
        print(next(myfile))

#%%

# Set up A (matrix of predictors) and y (response value)
A = np.loadtxt('./../DATA/Deathrate/deathrate_instance_python.dat')
y = np.array(A[:,15])
A = np.array(A[:,:15])

#%%

# Dimensions of the variables:
n = len(A); p = len(A[1])
print('y shape: ', np.shape(y))
print('A shpae: ', np.shape(A))

e = np.ones(n)
```

Now, we solve the problem:



```
# CONSTRAINT

# We can tune the value of t here (upper bound of the constraint)
t = 1
# Non linear constraint
nonlinear_constraint = NonlinearConstraint(ridge_cons_f, lb = -np.inf, ub = t,
                                           jac = ridge_cons_J, hess = ridge_cons_H)

#%%

# PROBLEM

# We take a random initial point
x0 = np.random.rand(p+1)

sol = minimize(ridge_obj, x0 = x0, method = 'trust-constr', jac = ridge_grad, hess = ridge_hess,
               constraints = [nonlinear_constraint], bounds = None, options = {'verbose' : 1})

print(sol)
```

RESULTS



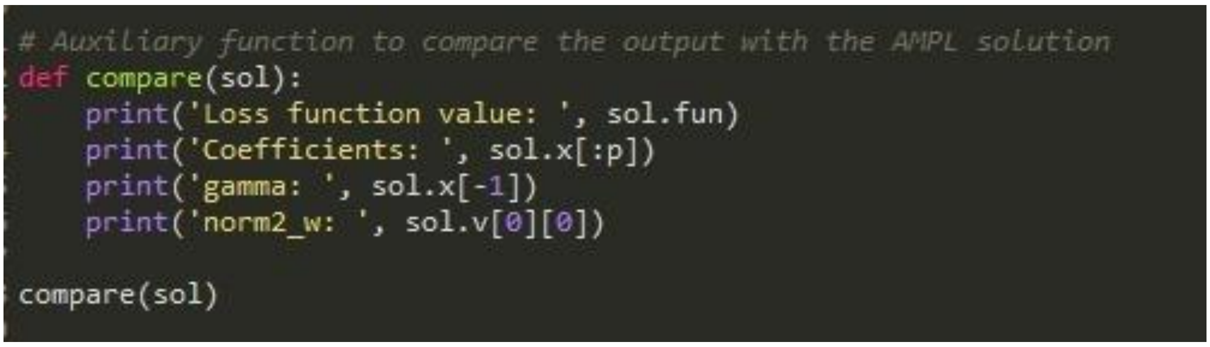
```
barrier_parameter: 2.048000000000001e-09
barrier_tolerance: 2.048000000000001e-09
         cg_niter: 625
     cg_stop_cond: 4
           constr: [array([1.])]
      constr_nfev: [124]
      constr_nhev: [71]
      constr_njev: [65]
   constr_penalty: 15407.295232562627
 constr_violation: 0.0
   execution_time: 0.23907184600830078
              fun: 61484.654640481385
             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,  2.11002771e-10])
              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([-2.79145752e-07,  1.16064371e-07,  2.77650997e-07, -1.57218324e-07,
       5.64302368e-07,  2.91686320e-08, -1.92430889e-08,  6.53122356e-09,
       2.92344339e-07,  6.92875801e-08, -2.71451427e-07,  2.06046025e-08,
      -2.17602519e-07,  2.16232365e-08, -2.81911639e-07,  2.11002771e-10])
          message: '`xtol` termination condition is satisfied.'
           method: 'tr_interior_point'
             nfev: 124
             nhev: 65
            niter: 118
             njev: 65
       optimality: 5.643023683887805e-07
           status: 2
        tr_radius: 1.0000000000000005e-09
                v: [array([10785.10671611])]
                x: array([ 4.88239571e-01, -4.63880870e-02,  1.02977785e-01, -3.77061350e-02,
      4.90736139e-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,  9.95977557e-03,  8.95141390e+02])
```

Loss function value
Number of evaluations of the objective function and of its Jacobian and Hessian
Number of iterations performed by the optimizer
Vector of coefficients
Constraint value
gamma

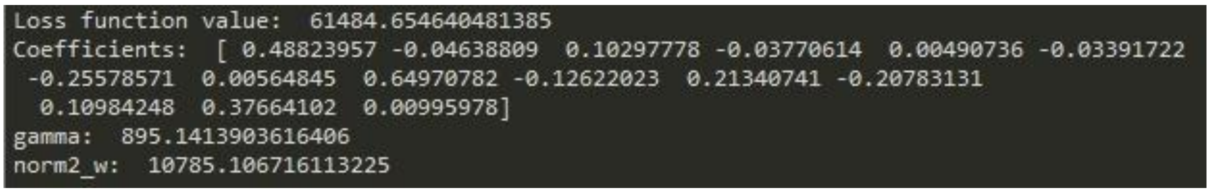
As we can see in the output, the loss function value is 61484.65, the number of iterations performed by the optimizer is 118, the number of evaluations of the objective function is 124 and the one of its Jacobian and Hessian 65. The value of gamma is 895.14 and the value of $||w||_2^2$ is 10785.10.

We observe that the lagrangian gradient is 0, so it verifies Karush-Kuhn-Tucker conditions (KKT) which means that our nonlinear programming solution is optimal.

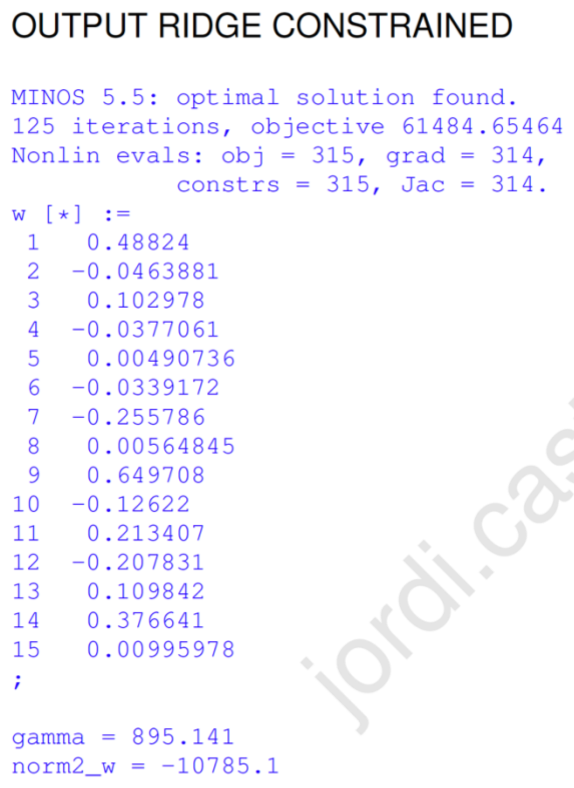
COMPARISION AMPL VS PYTHON IMPLEMENTATION OF RIDGE REGRESSION



```python
# Auxiliary function to compare the output with the AMPL solution
def compare(sol):
    print('Loss function value: ', sol.fun)
    print('Coefficients: ', sol.x[:p])
    print('gamma: ', sol.x[-1])
    print('norm2_w: ', sol.v[0][0])

compare(sol)
```



```
Loss function value:  61484.654640481385
Coefficients:  [ 0.48823957 -0.04638809  0.10297778 -0.03770614  0.00490736 -0.03391722
 -0.25578571  0.00564845  0.64970782 -0.12622023  0.21340741 -0.20783131
  0.10984248  0.37664102  0.00995978]
gamma:  895.1413903616406
norm2_w:  10785.106716113225
```



OUTPUT RIDGE CONSTRAINED

```
MINOS 5.5: optimal solution found.
125 iterations, objective 61484.65464
Nonlin evals: obj = 315, grad = 314,
           constrs = 315, Jac = 314.
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
```

If we compare both solutions we conclude that our Python implementation gives the same output as the AMPL implementation given in class.

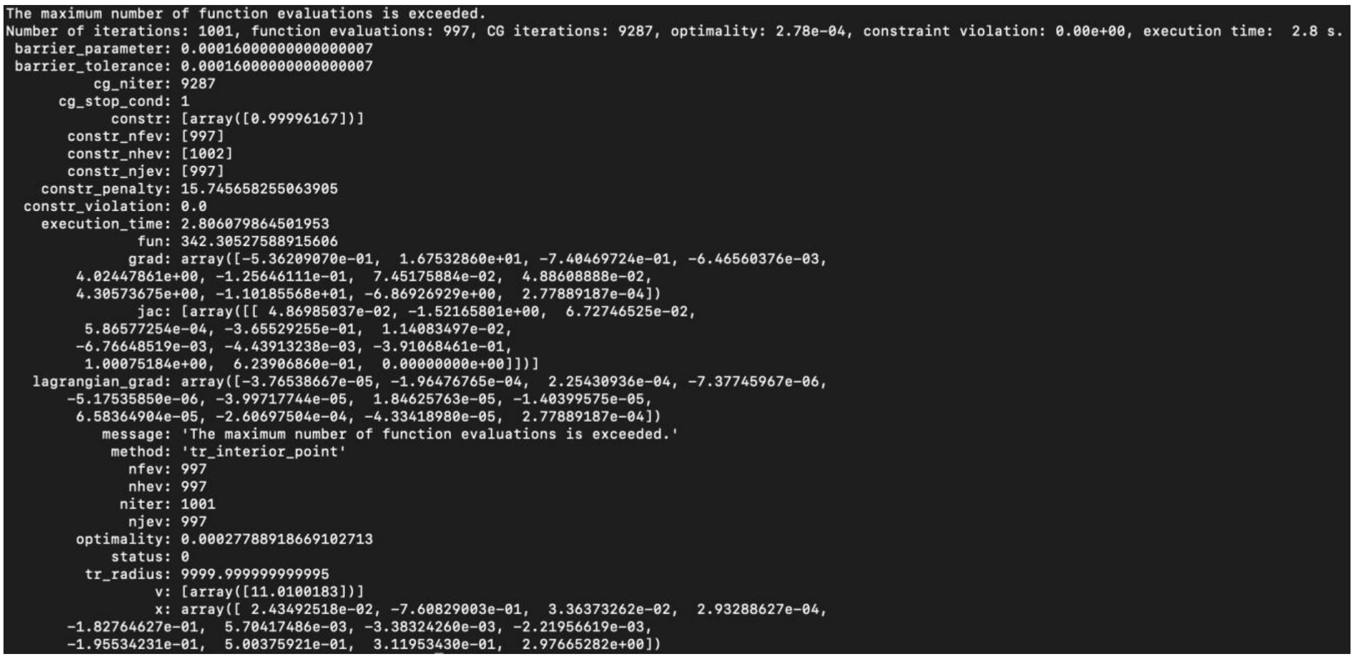
Notice that our Python implementation is faster, performs less iterations that AMPL does and it computes very few evaluations of the function, its Jacobian and its Hessian compared to AMPL.

A possible reason for that, could be that AMPL calculates the Jacobian and the Hessian, both the objective function and the restrictions by itself. Contrary to that, in our implementation, we have previously calculated the Jacobian and the Hessian and we have given it as parameters to our function. So, it doesn't have to calculate it.

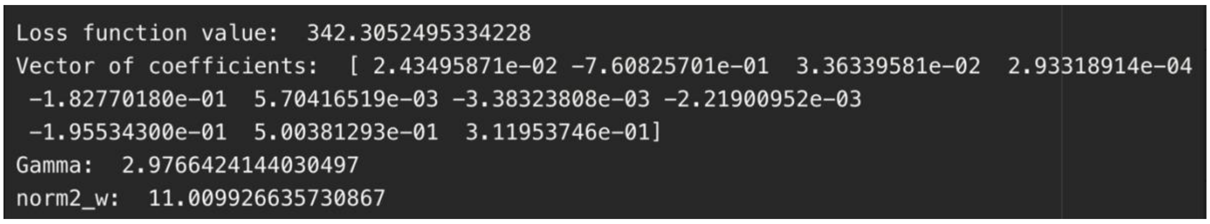
IMPLEMENTATION WITH "WINES" DATA SET

Now, we are going to study another data set which is about red wines from north of Portugal. This data set is taken from *Machine Learning Repository* web of UCI (University of Califòrnia in Irvine). It is called and we can find it in Wine Quality Data Set.

RESULTS FROM PYTHON

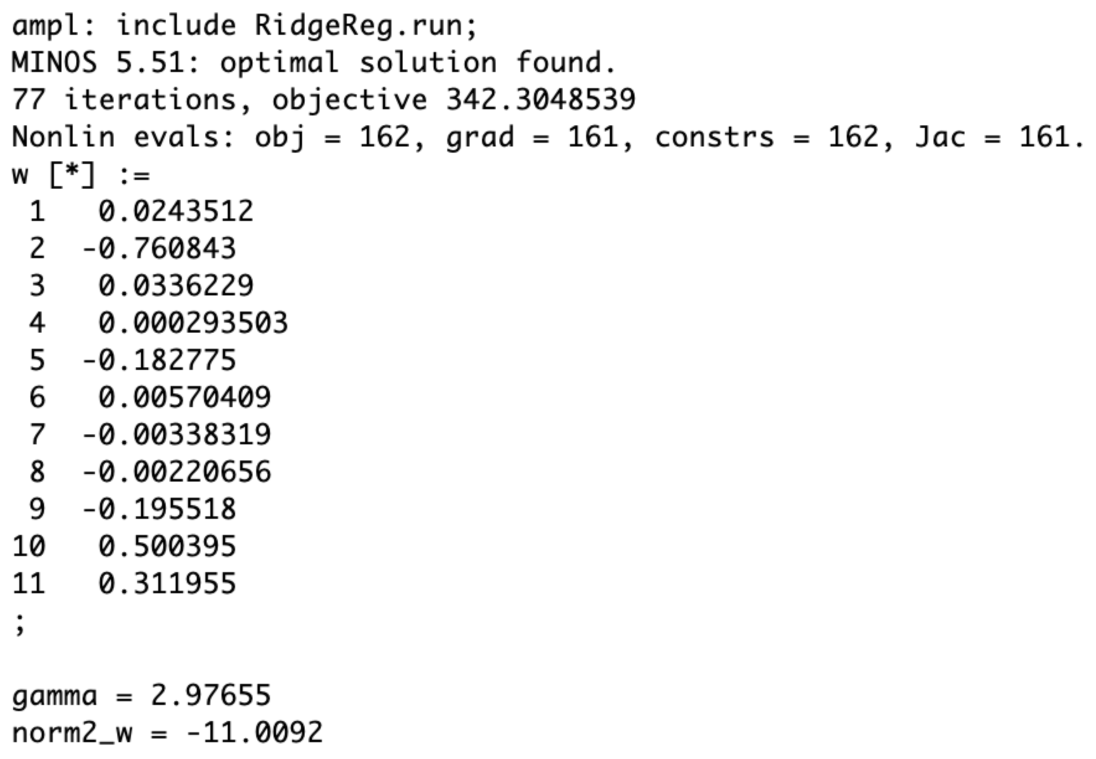


```
The maximum number of function evaluations is exceeded.
Number of iterations: 1001, function evaluations: 997, CG iterations: 9287, optimality: 2.78e-04, constraint violation: 0.00e+00, execution time:  2.8 s.
 barrier_parameter: 0.00016000000000000007
 barrier_tolerance: 0.00016000000000000007
          cg_niter: 9287
      cg_stop_cond: 1
            constr: [array([0.99996167])]
       constr_nfev: [997]
       constr_nhev: [1002]
       constr_njev: [997]
    constr_penalty: 15.745658255063905
  constr_violation: 0.0
    execution_time: 2.806079864501953
               fun: 342.30527588915606
              grad: array([-5.36209070e-01,  1.67532860e+01, -7.40469724e-01, -6.46560376e-03,
        4.02447861e+00, -1.25646111e-01,  7.45175884e-02,  4.88608888e-02,
        4.30573675e+00, -1.10185568e+01, -6.86926929e+00,  2.77889187e-04])
               jac: [array([[ 4.86985037e-02, -1.52165801e+00,  6.72746525e-02,
         5.86577254e-04, -3.65529255e-01,  1.14083497e-02,
        -6.76648519e-03, -4.43913238e-03, -3.91068461e-01,
         1.00075184e+00,  6.23906860e-01,  0.00000000e+00]])]
   lagrangian_grad: array([-3.76538667e-05, -1.96476765e-04,  2.25430936e-04, -7.37745967e-06,
       -5.17535850e-06, -3.99717744e-05,  1.84625763e-05, -1.40399575e-05,
        6.58364904e-05, -2.60697504e-04, -4.33418980e-05,  2.77889187e-04])
           message: 'The maximum number of function evaluations is exceeded.'
            method: 'tr_interior_point'
              nfev: 997
              nhev: 997
             niter: 1001
              njev: 997
        optimality: 0.00027788918669102713
            status: 0
         tr_radius: 9999.999999999995
                 v: [array([11.0100183])]
                 x: array([ 2.43492518e-02, -7.60829003e-01,  3.36373262e-02,  2.93288627e-04,
       -1.82764627e-01,  5.70417486e-03, -3.38324260e-03, -2.21956619e-03,
       -1.95534231e-01,  5.00375921e-01,  3.11953430e-01,  2.97665282e+00])
```



```
Loss function value:  342.3052495334228
Vector of coefficients:  [ 2.43495871e-02 -7.60825701e-01  3.36339581e-02  2.93318914e-04
 -1.82770180e-01  5.70416519e-03 -3.38323808e-03 -2.21900952e-03
 -1.95534300e-01  5.00381293e-01  3.11953746e-01]
Gamma:  2.9766424144030497
norm2_w:  11.009926635730867
```

RESULTS FROM AMPL



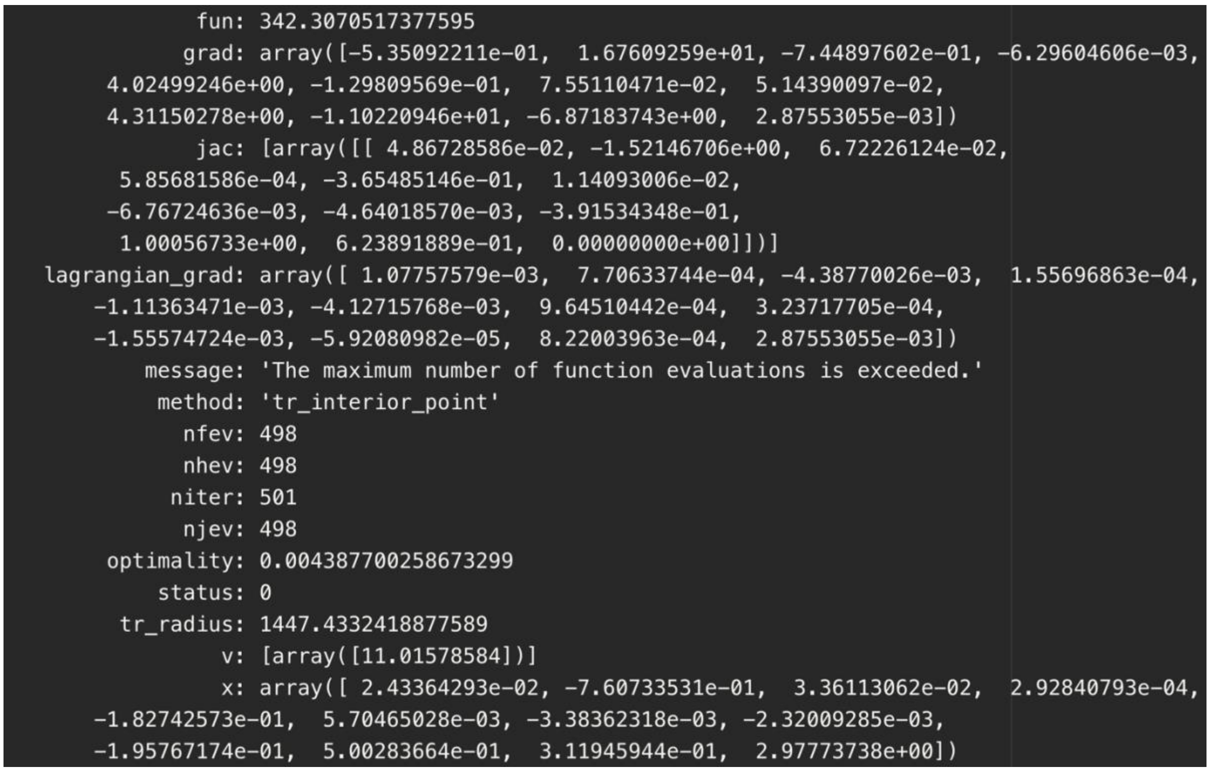
```
ampl: include RidgeReg.run;
MINOS 5.51: optimal solution found.
77 iterations, objective 342.3048539
Nonlin evals: obj = 162, grad = 161, constrs = 162, Jac = 161.
w [*] :=
 1   0.0243512
 2  -0.760843
 3   0.0336229
 4   0.000293503
 5  -0.182775
 6   0.00570409
 7  -0.00338319
 8  -0.00220656
 9  -0.195518
10   0.500395
11   0.311955
;

gamma = 2.97655
norm2_w = -11.0092
```

We get the same value of: the objective function, 342.30, gamma, 2.97 and norm2_w, 11.01, with both implementations. In addition, it verifies KKT conditions as we observe that lagrangian_grad equals zero.

However, we can see that AMPL found the optimal solution and our implementation in Python doesn't, we exceeded the maximum number of function evaluation.

Why does it happen?

The truth is that we don't know exactly the reason of that. It could be because the number of allowed iterations is too short. However, we think it is not the key point since the data set we are exploring is not that big. It only has 11 variables and 1599 entries. In addition, if we proceed to implement our routine with only 500 iterations, we obtain the same result as we can see in the next image:



```
              fun: 342.3070517377595
             grad: array([-5.35092211e-01,  1.67609259e+01, -7.44897602e-01, -6.29604606e-03,
        4.02499246e+00, -1.29809569e-01,  7.55110471e-02,  5.14390097e-02,
        4.31150278e+00, -1.10220946e+01, -6.87183743e+00,  2.87553055e-03])
              jac: [array([[ 4.86728586e-02, -1.52146706e+00,  6.72226124e-02,
         5.85681586e-04, -3.65485146e-01,  1.14093006e-02,
        -6.76724636e-03, -4.64018570e-03, -3.91534348e-01,
         1.00056733e+00,  6.23891889e-01,  0.00000000e+00]])]
  lagrangian_grad: array([ 1.07757579e-03,  7.70633744e-04, -4.38770026e-03,  1.55696863e-04,
       -1.11363471e-03, -4.12715768e-03,  9.64510442e-04,  3.23717705e-04,
       -1.55574724e-03, -5.92080982e-05,  8.22003963e-04,  2.87553055e-03])
          message: 'The maximum number of function evaluations is exceeded.'
           method: 'tr_interior_point'
             nfev: 498
             nhev: 498
            niter: 501
             njev: 498
       optimality: 0.004387700258673299
           status: 0
        tr_radius: 1447.4332418877589
                v: [array([11.01578584])]
                x: array([ 2.43364293e-02, -7.60733531e-01,  3.36113062e-02,  2.92840793e-04,
       -1.82742573e-01,  5.70465028e-03, -3.38362318e-03, -2.32009285e-03,
       -1.95767174e-01,  5.00283664e-01,  3.11945944e-01,  2.97773738e+00])
```

As we obtain the same results in less iterations we conclude that our method become stagnant or we have numeric problems produced by the *trust-constr* method used which we don't know exactly how it internally works.

Although our Python implementation does not converge properly in this second case, we assume that it is because of numeric problems as we obtain the same results in AMPL.

BIBLIOGRAPHY

- Slides of the Mathematical Optimization course of GCED of UPC
- DEATH RATE data set: Reference: Richard Gunst, Robert Mason, Regression Analysis and Its Applications: a data-oriented approach, Dekker, 1980, pages 370-371. ISBN: 0824769937. Gary McDonald, Richard Schwing, Instabilities of regression estimates relating air pollution to mortality, Technometrics, Volume 15, Number 3, pages 463-482, 1973. Helmut Spaeth, Mathematical Algorithms for Linear Regression, Academic Press, 1991,ISBN 0-12-656460-4.
- WINES data set, P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 47(4):547-553, 2009.