Optimization for Data and Decision Sciences

Project 2021-2022

Thomas Teulery

Professor: Clément Royer

I. Newton's method

As part of this assignment, we want to apply standard Newton's Method in order to solve the following optimization problem:

$$\min_{w \in \mathbb{R}^3} q(w) = (w_1 + w_2 + w_3 - 5)^2 + 3(w_2 - w_3)^2 + 2(w_2 - 2w_3)^2$$

As needed in Newton's Method, we compute the first and second-order derivates of $q(w)$:

$$\nabla q(w) = \begin{bmatrix} 8w_1 - 4w_2 + 2w_3 - 10 \\ -4w_1 + 12w_2 - 6w_3 - 10 \\ 2w_1 - 6w_2 + 18w_3 - 10 \end{bmatrix}$$

$$\nabla^2 q(w) = \begin{bmatrix} 8 & -4 & 2 \\ -4 & 12 & -6 \\ 2 & -6 & 18 \end{bmatrix}$$

The method will follow the given iteration:

1. $Compute\ \nabla q(w_k)\ and\ \nabla^2 q(w_k)$
2. $If\ \nabla^2 q(w_k)\ is\ invertible,\ compute\ d_k = -[\nabla^2 q(w_k)]^{-1} \cdot \nabla q(w),\ otherwise\ stop$
3. $Compute\ w_{k+1} = w_k + d_k$

The process should be reiterated as long as the difference between two successive values of w, denoted as $d_k$remains greater than a certain threshold ε, or the algorithms reaches a limited number of loops. These parameters should be chosen by the user.

An implementation of the method, made with Python:

```python
def solve(self, max_iter, eps, w0):
  wk = w0
  iter = 0
  solved = False
  while iter < max_iter:
    f_gradient_wk = self.f_gradient(wk)
    f_hessian_wk = self.f_hessian(wk)
    if self.is_inversible(f_hessian_wk):
      dk = - np.linalg.inv(f_hessian_wk).dot(f_gradient_wk)
      wk = wk + dk
      iter += 1
      if dk.T.dot(dk) < eps: # convergence condition test
        solved = True
        break
    else:
      wk = None
      break
  return {'w': wk, 'iteration': iter, 'solved': solved}
```

Fig. 1: Standard Newton's Method

We can see through the execution, starting with different values for w_0, that the algorithm converges with this function in one iteration. Geometrically, Newton's Method search for the minimum (in case of minimization) of the current $w_k$parabola.

```
Starting point [1. 1. 1.]:
[2. 2. 1.]

Starting point [0. 0. 0.]:
[2. 2. 1.]

Starting point (random):
[2. 2. 1.]

q(w*) = 0
```

Fig. 2: Execution with different starting points

We will now consider minimizing the Rosenbrock function. This gives us the following optimization problem:

$$\min_{w \in \mathbb{R}^2} 100(w_2 - {w_2}^2)^2 + (1 - w_1)^2$$

First and second order Rosenbrock function derivatives:

$$\nabla f(w) = \begin{bmatrix} 2(200{w_1}^3 - 200w_1w_2 + w_1 - 1) \\ 200(w_2 - {w_1}^2) \end{bmatrix}$$

$$\nabla^2 f(w) = \begin{bmatrix} 1200{w_1}^2 - 400w_1 + 2 & -400w_1 \\ -400w_1 & 200 \end{bmatrix}$$

We will be running the standard Newton's Method implementation with the two following points:

$$w_{01} \coloneqq \begin{bmatrix} -1.2 \\ 1 \end{bmatrix} \qquad \text{and} \qquad w_{02} \coloneqq \begin{bmatrix} 0 \\ \frac{1}{400} + 10^{-12} \end{bmatrix}$$

```
Starting point [-1.2  1. ]:
{'w': array([1., 1.]), 'iteration': 6, 'solved': True}

Starting point [0.     0.0025]:
{'w': array([1.       , 0.9999999]), 'iteration': 5, 'solved': True}
```

Fig. 3: Execution starting from $w_{01}$and w_{02}

The parameters chosen for the execution in *Fig.3* are $MAX_{ITER} = 100$ and $\varepsilon = 10^{-3}$. We can see that for both starting points the solution vector is found quickly. We can now confirm the function minimum:

$$f(w^*) = 0\,, with\ w^* = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

However, the problem is not solvable from any starting point. In fact, if $\nabla^2 f(w)$ is not invertible, we cannot proceed with this algorithm. It is the case for the point $w_{03} = \begin{bmatrix} 0 \\ 0.005 \end{bmatrix}$ for which $\det\left(f_{rosenbrock}(w_{03})\right) = 0$, therefore $\nabla^2 f(w)$ is not invertible, thus Newton's Method is not applicable.

This property makes the method local and not global, since a w_k vector with a non-invertible hessian can be picked during the method iterations, the algorithm returning no solution. It is also possible for the method to converge towards a saddle point, making it inefficient.

II. A globally convergent version of Newton's method

Nevertheless, there is a possibility in modifying the method to make it efficient, with no regards neither to the function convexity nor to the starting points. We will be using a mix of the *Levenberg-Marquardt* regularization and the *Line Search* method to avoid returning no solution.

An implementation of the modified method, made with Python:

```python
def solve_with_regularization(self, max_iter, eps, c, mu, w0):
  d = w0.shape[0]
  wk = w0
  iter = 0
  solved = False
  while iter < max_iter:
    f_gradient_wk = self.f_gradient(wk)
    f_hessian_wk = self.f_hessian(wk)
    gamma_k = self.get_gamma_k(
        mu,
        f_hessian_wk
    )
    aux = f_hessian_wk + gamma_k*np.identity(d)
    if self.is_inversible(aux):
      dk = self.dk_formula(
          f_gradient_wk,
          aux
      )
      while self.f(wk + dk) >= self.f(wk) + c * dk.T.dot(f_gradient_wk):
        gamma_k = mu * gamma_k
        aux = f_hessian_wk + gamma_k*np.identity(d)
        if self.is_inversible(aux):
          dk = self.dk_formula(
              f_gradient_wk,
              aux
          )
        else:
          # not inversible
          return {'w': None, 'iteration': iter, 'solved': solved}
      wk = wk + dk
      if dk.T.dot(dk) < eps: # convergence condition test
        solved = True
        break
      iter += 1
    else:
      # not inversible
      wk = None
      break
  return {'w': wk, 'iteration': iter, 'solved': solved}
```

Fig. 4: Implementation of a globally convergent version of Newton's Method

```python
@staticmethod
def is_inversible(matrix):
  return np.linalg.det(matrix) != 0

@staticmethod
def get_gamma_k(mu, f_hessian_wk):
  lambda_min = np.nanmin(np.linalg.eigvalsh(f_hessian_wk)).item()
  return mu * np.nanmax(np.append(lambda_min, 10^-10)).item()

@staticmethod
def dk_formula(f_grad_w, f_hess_w):
  return - np.linalg.inv(f_hess_w).dot(f_grad_w)
```

Fig. 5: Additional tooling functions

After execution and with the same parameters as before (MAX_{ITER} and ε) and with the new parameters $c = 0.0001$ and $\mu = 2$, we get the following results:

```
Starting point [-1.2  1. ] (with regularization):
{'w': array([0.96971296, 0.94005116]), 'iteration': 25, 'solved': True}

Starting point [0.     0.0025] (with regularization):
{'w': array([0.970239  , 0.94107877]), 'iteration': 16, 'solved': True}
```

Fig. 6: Execution starting from w_{01} and w_{02}

With this version, we add regularization to make the method global, in order to avoid saddle points and to always choose the right search direction. We add to this a feature, originating from *Line Search* methods, to choose the right factor γ_k so that the identity matrix added to the computed hessian at w_k before inversion is enough to see progress. As a direct consequence, it is possible that this method progresses slower compared to the standard version, especially for points that didn't need this modified version to be efficient i.e., w_{01} and w_{02}. It is due to the algorithm choosing γ_k just large enough to make the iteration proceed in the right direction, even though it could be larger without jeopardizing the result.

We can observe the influence of c and μ parameters on the number of iterations to solve the problem:

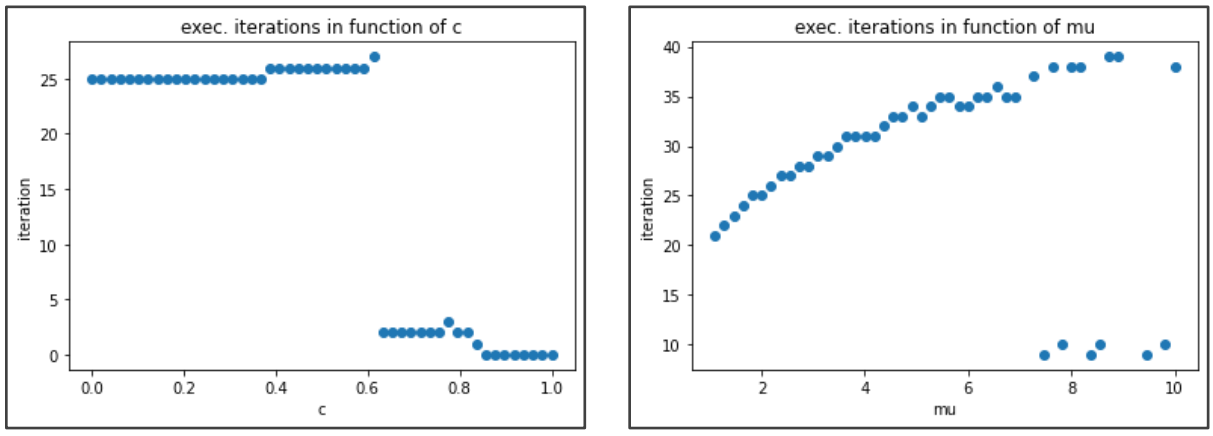

Fig. 7 & 8: Iterations in function of c and μ parameters

On one hand, the c parameter influence regularization size through γ_k. The larger c is, the larger will be iterations progress. It is confirmed in the *Fig. 7*, in which the iteration needed to find the solution starting from w_{01} is diminishing as more progress is made with a larger c.

On another hand, μ impact is on the size of each step in the *"Line Search"* of γ_k. The smaller it would be, the longer will be each iteration whilst giving more accuracy. In *Fig. 8* we can see that by increasing this parameter, it takes longer and longer for the solver to find an adequate solution (fitting the ε condition). With μ being too big, the algorithm will tend to circle around

the solution before attaining a close enough point. In the worst-case scenario, it might not even find a suiting point, reaching the maximum number of iterations.

III. Subsampling Newton-type methods

In this part, we will be tackling a machine learning problem, with the help of Newton's Method. Given a dataset of n entities, we need to solve the following problem:

$$\min_{w \epsilon \mathbb{R}^d} f(w) := \frac{1}{n} \sum_{i=1}^{n} f_i(w)$$

The workstream will be focused on the implementation of a subsampling method, similar to batch methods in stochastic gradient.

We define partial functions as such:

$$f_i(w) = \ln(1 + \exp(-y_i {x_i}^T w)) + \frac{\lambda}{2} \|w\|^2$$

$$\nabla f_i(w) = -\frac{y_i}{1 + \exp(y_i {x_i}^T w)} x_i + \lambda \mathrm{w}$$

$$\nabla^2 f_i(w) := \frac{\exp(y_i w^T x_i)}{(1 + \exp(y_i w^T x_i))^2} x_i x_i^T + \lambda \boldsymbol{I}_d$$

Given the partial function, we can compute subsampling functions:

$$f_{S_k}(w_k) = \frac{1}{|S_k|} \sum_{i \, \epsilon \, S_k} f_i(w_k)$$

$$\nabla f_{S_k}(w_k) = \frac{1}{|S_k|} \sum_{i \, \epsilon \, S_k} \nabla f_i(w_k)$$

$$\nabla^2 f_{{S_k}^H}(w_k) = \frac{1}{|{S_k}^H|} \sum_{i \, \epsilon \, {S_k}^H} \nabla^2 f_i(w_k)$$

With S_k and ${S_k}^H$ being a batch of entity indexes from $S = \{1, 2, \dots, n\}$

An implementation of the subsampling version, made with Python:

```python
def solve_sub_sampling(self, max_iter, eps, c, mu, batch_size_grad, batch_size_hess, w0):
  wk = w0
  i = 0
  flag = False
  while i < max_iter:
    Sk = self.generator.choice(self.S_full, batch_size_grad, replace=False)
    Sk_h = self.generator.choice(self.S_full, batch_size_hess, replace=False)
    f_gradient_wk = self.batch_f_gradient(Sk, wk)
    f_hessian_wk = self.batch_f_hessian(Sk_h, wk)
    gamma_k = self.get_gamma_k(mu, f_hessian_wk)
    aux = f_hessian_wk - gamma_k*np.identity(self.d)
    if self.is_inversible(aux):
      dk = self.dk_formula(
          f_gradient_wk,
          aux
      )
      while self.batch_f(Sk, wk + dk) >= self.batch_f(Sk, wk) + c * dk.T.dot(f_gradient_wk):
        gamma_k = mu * gamma_k
        aux = f_hessian_wk - gamma_k*np.identity(self.d)
        if self.is_inversible(aux):
          dk = self.dk_formula(
            f_gradient_wk,
            aux
          )
        else:
          return {'w': None, 'epoch': self.get_epoch(i, batch_size_grad, batch_size_hess), 'solved': False}
      wk = wk + dk
      if dk.T.dot(dk) < eps:
        flag = True
        break
      i += 1
    else:
      wk = None
      break
  return {'w': wk, 'epoch': self.get_epoch(i, batch_size_grad, batch_size_hess), 'solved': flag}
```

Fig. 9: Implementation of subsampling version of Newton's Method

```python
@staticmethod
def get_gamma_k(mu, f_hessian_wk):
  lambda_min = np.nanmin(np.linalg.eigvalsh(f_hessian_wk)).item()
  return mu * np.nanmax(np.append(-lambda_min, 10^-10)).item()

@staticmethod
def dk_formula(f_grad_w, f_hess_w):
  return - np.linalg.inv(f_hess_w).dot(f_grad_w)

@staticmethod
def is_inversible(matrix):
  return np.linalg.det(matrix) != 0
```

Fig. 10: Additional tooling functions

We will be working with simulated data composed of 500 samples of 4 attributes. We can now proceed to the execution, and compare different batch size and their behaviors. For reproductivity purpose, we will be using a seed equal to 0 during executions:

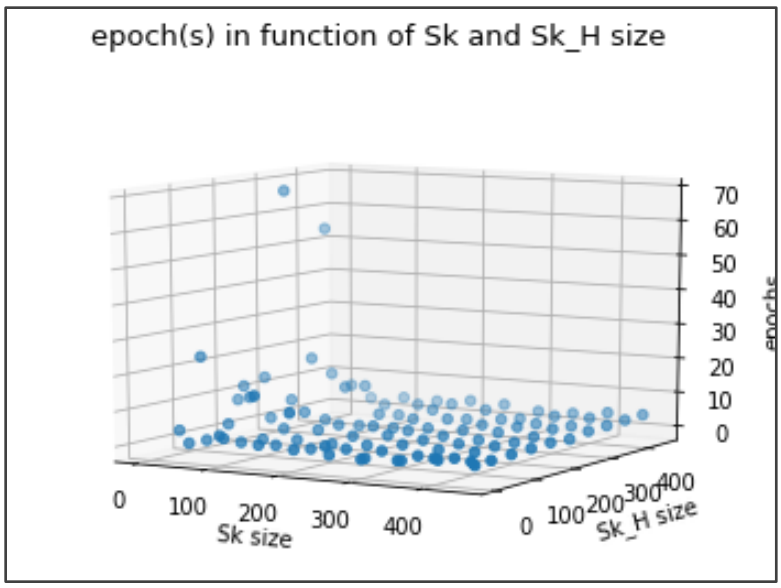

Fig. 11: Performance for regularized version of Newton's Method

As display on *Fig. 15*, it's hard to distinguish which batch size to give credit to between S_k or ${S_k}^H$ as both seem to have impact on efficiency. Nevertheless, we can say extreme size batches are not recommended:

- The smaller ones ($|S_k|$ and $\left|{S_k}^H\right| < 0.1|S|$) tend to not finish, returning no solution, and for the other cases accessing more data and taking drastically more time to solve.
- The higher ones ($|S_k|$ and $\left|{S_k}^H\right| > 0.4|S|$) doesn't provide better performances than the rest of the middle ground batch sizes.

As a matter of fact, with the given seed and the chosen parameters, the best was $|S_k| = 100$ and $|S_k^H| = 100$, with a total of **0.60 epochs**.

```
[REGULAR]
> f(w)=    0.59130 (solved: True)       in     3.00 epoch(s)
```

Fig. 12: Performance for regularized version of Newton's Method

Given the regularized version result, we can conclude that the subsampling version can be more efficient, given the right batch parameters ($\sim 0.2|S|$).

We can now compare our best solution, to the Stochastic Gradient method would find with the same batch size as we used (100):

125	5.97e-01	9.34e-01
130	5.96e-01	9.17e-01
135	5.95e-01	9.00e-01
140	5.95e-01	8.92e-01
145	5.94e-01	8.76e-01
150	5.94e-01	8.74e-01
155	5.94e-01	8.64e-01
160	5.94e-01	8.59e-01
165	5.93e-01	8.44e-01
170	5.93e-01	8.29e-01
175	5.93e-01	8.16e-01
180	5.92e-01	8.13e-01
185	5.92e-01	8.05e-01
190	5.92e-01	7.98e-01

Fig. 13: Stochastic gradient results

In the middle row of *Fig. 13* are displayed the solution found at each iteration. Since each iteration browses through 100 rows, and that the Stochastic Gradient solver seem to attain this result in-between 125 and 190 iterations, our solution seems to be greatly more efficient in this context (0.6 epoch compared to 25-38).