Computation in Data Science (Third Part) Homework 4

January 5, 2021

- Arthor: HO Ching Ru (R09946006, NTU Data Science Degree Program)
- Course: Computation in Data Science (MATH 5080), NTU 2020 Fall Semester
- Instructor: Tso-Jung Yen (Institute of Statistical Science, Academia Sinica)

1 The proximal operator of the l_0-norm

1.1 Answer

The first statement, **a.** $[\mathrm{HT}_\alpha(\mathbf{x})]_j = x_j \mathbb{I}\left\{ |x_j| \geq \sqrt{2\alpha} \right\}$, is *true*.

1.2 Explanation

By the definition of indicator function $\mathbb{I}(\cdot)$ and l_0 norm:

$$\begin{aligned}\mathrm{HT}_\alpha(\mathbf{x}) &= \arg\min_\beta \left\{ \alpha\|\beta\|_0 + \frac{1}{2}\|\beta - \mathbf{x}\|_2^2 \right\} \\ &= \arg\min_\beta \left(\sum_{i=1}^{p} \alpha\mathbb{I}\{\beta_i \neq 0\} + \frac{1}{2}\left(\beta_i - x_i\right)^2 \right)\end{aligned}$$

Assume that in the j-th element, β_j^* is the optimal β_j:

$$\beta_j^* = \arg\min_{\beta_j} \left(\alpha\mathbb{I}\{\beta_j \neq 0\} + \frac{1}{2}\left(\beta_j - x_j\right)^2 \right)$$

First, consider $x_j \neq 0$:

$$\begin{cases} \beta_j^* = 0,\ \alpha\mathbb{I}\{\beta_j^* \neq 0\} + \frac{1}{2}\left(\beta_j^* - x_j\right)^2 = \frac{1}{2}x_j^2 \\ \beta_j^* \neq 0,\ \alpha\mathbb{I}\{\beta_j^* \neq 0\} + \frac{1}{2}\left(\beta_j^* - x_j\right)^2 = \alpha + \frac{1}{2}\left(\beta_j^* - x_j\right)^2,\ \beta_j^* = x_j \end{cases}$$

Or we can rewrite both cases into:

$$\beta_j^* = \begin{cases} x_j, \text{ if } |x_j| \geq \sqrt{2\alpha} \\ 0, \text{ otherwise} \end{cases}$$

Second, consider $x_j = 0$ and $\beta_j^* = 0$:

$$\alpha\mathbb{I}\{\beta_j \neq 0\} + \frac{1}{2}\left(\beta_j^* - x_j\right)^2 = 0$$

Thus we get:

$$[\mathrm{HT}_\alpha(\mathbf{x})]_j = \beta_j^* = x_j\mathbb{I}\left\{|x_j| \geq \sqrt{2\alpha}\right\}$$

---

2 Programming work

From the definition of proximal gradient, the β update function is:

$$\begin{aligned}
\beta^{r+1} =& \mathrm{prox}_{c_r,g}\left(\beta^r - c_r\nabla l(\beta^r)\right) \\
=& \mathrm{prox}_{c_r,g}\left(\beta^r - c_r(-\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta)\right) \\
=& \mathrm{HT}_{\sqrt{2\alpha}}\left(\beta^r - c_r(-\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta)\right)
\end{aligned}$$

From the definition of fast proximal gradient, the β update function is:

$$\begin{aligned}
\gamma^r &= \beta^r + \left(\frac{b_{r-1} - 1}{b_r}\right)(\beta^r - \beta^{r-1}) \\
\beta^{r+1} &= \mathrm{HT}_{\sqrt{2\alpha}}\left(\gamma^r - c_r(-\mathbf{X}^T(\mathbf{y} - \mathbf{X}\gamma)\right)
\end{aligned}$$

where $\mathrm{HT}(\cdot)$ denotes a hard thresholding function as above.

2.1 Data generation

[1]:
```python
import math
import numpy as np
import time
np.random.seed(5566)

# dimension
p = 500

# number
n = 200

# generate random x_i: from N(0,1)
```

```
X = np.random.normal(size=(n, p))

# generate beta: beta[100], ..., beta[500] given, others assume be zeros
beta_true = np.random.normal(size=(p,1))
for i in range(p):
    if (i==99) or (i==199) or (i==499):
        beta_true[i] = [-2]
    elif (i==299) or (i==399):
        beta_true[i] = [2]
beta_true = np.array(beta_true).reshape(-1, )

# generate random y_i: from X and beta
y = np.array(np.dot(X, beta_true) + (np.random.normal(0, 0.5))).reshape(-1, )
```

2.2 Check each dimension of each variable and parameters

Set four different $\alpha = [5, 10, 15, 20]$, stepsize $c_r = \frac{1}{2\lambda_1(\mathbf{X}^T\mathbf{X})}$, and $\tilde{\alpha} = c_r \alpha$. The stopping criterion is reaching the maximal iteration number 10000.

[2]:
```
# set parameters
alpha_ls = [5, 10, 15, 20]
alpha_tilde = [float(0.5 / max(np.linalg.eigvals(np.dot(X.T, X))))*ele for ele␣
 ↪in alpha_ls]  # stepsize
num_iterations = 10000
```

```
<ipython-input-2-5ec1e8cb20ac>:3: ComplexWarning: Casting complex values to real
discards the imaginary part
  alpha_tilde = [float(0.5 / max(np.linalg.eigvals(np.dot(X.T, X))))*ele for ele
in alpha_ls]  # stepsize
```

[3]:
```
print("dimension of x:", np.shape(X))
print("dimension of y:", np.shape(y))
print("dimension of true beta:", np.shape(beta_true))
print("l2 Norm of true beta:", np.linalg.norm(beta_true, ord = 2))
```

```
dimension of x: (200, 500)
dimension of y: (200,)
dimension of true beta: (500,)
l2 Norm of true beta: 23.097943613621123
```

[4]:
```
def b_sequence(maxiter):
    b_list = list()
    b = 1
    for r in range(maxiter):
        b_list.append(b)
        b = (1 + math.sqrt(1 + 4* (b**2)))/2
    return b_list
```

```python
def objective_function(X ,y, theta_r, lamb):
    return 0.5 * np.linalg.norm(y - np.dot(X, beta), ord = 2)**2 + alpha*np.
↪linalg.norm(beta, ord = 0)

def gradient_function(X, y, beta):
    return np.dot(-X.T, y) +  np.dot(np.dot(X.T, X), beta).reshape(-1, )

def l0_prox(X, alpha):
    # Hard thresholding
    return np.array([ele if abs(ele) >= math.sqrt(2*alpha) else 0 for ele in X])

def proximal_gradient(X, y, n, p, alpha, maxiter):
    start = time.time()
    beta_iter = np.random.normal(size=(p))
    px_beta_history = list()
    px_beta_history.append(beta_iter)
    c = 0.5 / max(np.linalg.eigvals(np.dot(X.T, X)))  # stepsize
    for iterindex in range(maxiter):
        beta_tmp = l0_prox(beta_iter - c*gradient_function(X, y, beta_iter),␣
↪alpha)
        beta_iter = beta_tmp
        px_beta_history.append(beta_iter)
    end = time.time()
    print("Beta norm by Proximal Gradient:", np.linalg.norm(beta_iter), ". Time␣
↪Cost:", end-start)
    return px_beta_history

def fast_proximal_gradient(X, y, n, p, alpha, maxiter, b):
    start = time.time()
    beta_iter = np.random.normal(size=(p))
    fpx_beta_history = list()
    fpx_beta_history.append(beta_iter)
    c = 0.5 / max(np.linalg.eigvals(np.dot(X.T, X)))  # stepsize
    beta_iter_old = beta_iter
    for iterindex in range(1, maxiter):
        gamma = beta_iter + ((b[iterindex-1]-1)/(b[iterindex]))*(beta_iter -␣
↪beta_iter_old)
        beta_tmp = l0_prox(gamma - c*gradient_function(X, y, gamma), alpha)
        beta_iter_old = beta_tmp
        beta_iter = beta_tmp
        fpx_beta_history.append(beta_iter)
    end = time.time()
    print("Beta norm by Fast Proximal Gradient:", np.linalg.norm(beta_iter), ".␣
↪Time Cost:", end-start)
    return fpx_beta_history
```

```
# fast_proximal_gradient(X, y, n, p, beta_new, alpha_ls[0], num_iterations,␣
 ↪b_ls)
```

2.3 Training on different alpha

[5]:
```
b_ls = b_sequence(num_iterations)
proximal_history = list()
fast_proximal_history = list()
for index in alpha_tilde:
    print("alpha =", index)
    X_ = np.copy(X)
    y_ = np.copy(y)
    proximal_history.append(proximal_gradient(X_, y_, n, p, index,␣
 ↪num_iterations))
    fast_proximal_history.append(fast_proximal_gradient(X_, y_, n, p, index,␣
 ↪num_iterations, b_ls))
    print("----------")
```

```
alpha = 0.001825941127809183
Beta norm by Proximal Gradient: 23.09102866088622 . Time Cost: 40.28210496902466
Beta norm by Fast Proximal Gradient: 22.997002112648655 . Time Cost:
40.35296988487244
----------
alpha = 0.003651882255618366
Beta norm by Proximal Gradient: 23.888835440434026 . Time Cost:
41.21158766746521
Beta norm by Fast Proximal Gradient: 23.875075042904587 . Time Cost:
41.81175136566162
----------
alpha = 0.005477823383427549
Beta norm by Proximal Gradient: 23.146794181744188 . Time Cost:
41.51826620101929
Beta norm by Fast Proximal Gradient: 24.225580036823207 . Time Cost:
45.88677668571472
----------
alpha = 0.007303764511236732
Beta norm by Proximal Gradient: 23.804004495221413 . Time Cost:
43.36311435699463
Beta norm by Fast Proximal Gradient: 26.609995800305 . Time Cost:
43.64905381202698
----------
```

2.4 Calculating the difference and plotting the graph

[6]:
```
proximal_diff = list()
fast_proximal_diff = list()
for i in range(len(alpha_ls)):
    diff = [np.linalg.norm(proximal_history[i][j+1]-proximal_history[i][j],␣
↪ord=2) for j in range(len(proximal_history[i])-1) ]
    proximal_diff.append(diff)
    diff = [np.linalg.
↪norm(fast_proximal_history[i][j+1]-fast_proximal_history[i][j], ord=2) for j␣
↪in range(len(fast_proximal_history[i])-1) ]
    fast_proximal_diff.append(diff)
```

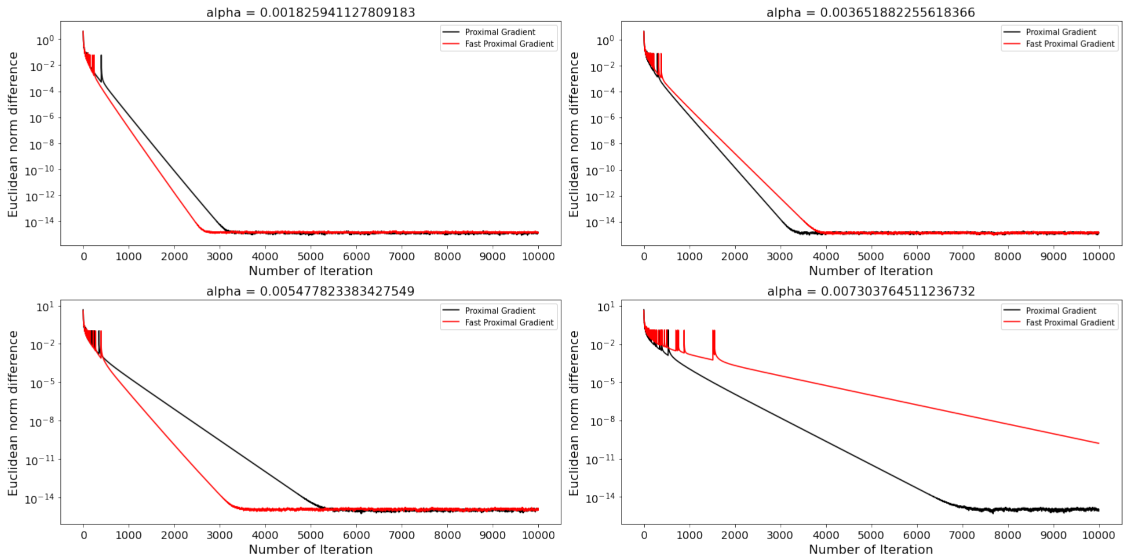
[7]:
```
import matplotlib.pyplot as plt

plt.figure(figsize = (20, 10))

for index in range(len(alpha_ls)):
    plt.subplot(2, 2, index+1)
    plt.plot(proximal_diff[index], label = "Proximal Gradient", color = "black")
    plt.plot(fast_proximal_diff[index], label = "Fast Proximal Gradient", color␣
↪= "red")

    plt.legend()
    plt.title("alpha = "+str(alpha_tilde[index]), fontsize = 16)
    plt.xlabel("Number of Iteration", fontsize = 16)
    plt.ylabel("Euclidean norm difference", fontsize = 16)
    plt.xticks(np.arange(0, num_iterations+1, 1000), fontsize = 14)
    plt.yticks(fontsize = 14)
    plt.yscale("log")

plt.tight_layout()
plt.show()
```



2.5 Apply on test dataset

In training dataset, we set $n = 200$, and find $\tilde{\alpha}^* \sim 0.00365$ has the best fitting result. Thus, out test dataset $n = 2000$ and use $\tilde{\alpha}^*$ to predict. The fitting results is as below.

n	n^{test}	$\tilde{\alpha}^*$	MSE	$\mathrm{Err}^{\mathrm{train}}$	$\mathrm{Err}^{\mathrm{test}}$
200	2000	0.00365	0.054	0.79	0.82

3 Reference

- Yang, Y., & Yu, J. (2020, August). Fast Proximal Gradient Descent for A Class of Non-convex and Non-smooth Sparse Learning Problems. In *Uncertainty in Artificial Intelligence* (pp. 1253-1262). PMLR.
- Antonello, N., Stella, L., Patrinos, P., & van Waterschoot, T. (2018). Proximal gradient algorithms: Applications in signal processing. *arXiv preprint arXiv:1803.01621.*
- Tanaka, M., & Okutomi, M. (2017, May). Unified optimization framework for L2, L1, and/or L0 constrained image reconstruction. In *Computational Imaging II* (Vol. 10222, p. 102220J). International Society for Optics and Photonics.