Proiect Tehnici de Optimizare

Paper 13 - Accelerating Greedy Coordinate Descent Methods

Dincă Vlad-Andrei
grupa 344

Gustin Samuel
grupa 344

25 mai 2020

Abstract

Analizăm algoritmii de coborâre pe coordonată, rezultatele lui Lu et al. despre accelerarea algoritmilor Greedy: AGCD (Accelerated Greedy Coordinate Descent) și ASCD (Accelerated Semi-Greedy Coordinate Descent), reluăm comparația cu rezultatele lui Nesterov, al algoritmului ARCD (Accelerated Randomized Coordinate Descent) și exemplificăm prin optimizarea funcțiilor de cost aplicate regresiei liniare și regresiei logistice.

1 Introducere

Algoritmii de coborâre pe coordonată prezintă interes deoarece tind să fie mai rapizi în practică decât algoritmii de coborâre pe gradient pentru problemele ce prezintă sparsitate. Actualizările pentru o epocă a algoritmului sunt mai eficient de realizat deoarece, în general, presupun calcularea derivatei pentru o singură coordonată.

Algoritmii Coordinate Descent sunt în mare de trei tipuri, în funcție de regula de alegere a coordonatei pe care se realizează actualizările la un anumit pas: GCD (Greedy Coordinate Descent) cu regula Gauss-Southwell, sau Gauss-Southwell-Lipschitz (GSL), studiat de către Nutini et al. în lucrarea sa, CCD (Cyclic Coordinate Descent) ce alege coordonatele în ordine, circular și algoritmul RCD (Randomized Coordinate Descent), studiat de către

Nesterov în lucrările sale. Randomized Coordinate Descent s-a bucurat de cea mai mare atenție de la lucrarea lui Nesterov despre accelerarea sa.

Lu et al. vine în susținerea algoritmului Greedy, studiind metodele accelerate bazate pe regula GSL: Accelerated Semi-Greedy Coordinate Descent (ASCD), ce îmbină regulile folosite de algoritmii GCD și RCD în alegerea coordonatelor, alegând coordonata pentru x-actualizări în manieră Greedy și coordonata pentru z-actualizări în mod aleator. De asemenea, chiar dacă reușește să susțină teoretic doar algoritmul ASCD, introduce și algoritmul Accelerated Greedy Coordinate Descent (AGCD), ce alege atât coordonata pentru x-actualizări, cât și coordonata pentru z-actualizări conform regulii Greedy. Chiar dacă AGCD nu se bucură încă de susținerea teoretică pentru a demonstra superioritatea față de ARCD, precum se bucură algoritmul ASCD, autorul îl susține empiric, iar în practică AGCD tinde să depășească performanțele ARCD și ASCD. [1]

Problema de interes este minimizarea funcției convexe f:

$$f^* = \min_x f(x)$$

Nutini et al. merge mai departe și împarte aceste funcții în două familii principale de funcții ce prezintă interes. Relevant pentru prezenta lucrare este familia de funcții:

$$h_1(x) := \sum_{i=1}^{n} g_i(x_i) + f(Ax)$$

unde $h_1, (g_i)_i$ și f sunt funcții convexe. Această familie include funcțiile de cost ale regresiilor liniare și logistice: least squares și logistic loss.

Nesterov arată că alegerea aleatoare a coordonatei pe care se face coborârea la un pas al algoritmului duce la o rată de convergență asemănătoare cu cea garantată de alegerea coordonatei în manieră Greedy.

Acest rezultat pare să transforme folosirea algoritmului Greedy într-un lucru indezirabil. Totuși, Nutini et al. demonstrează empiric că în pofida rezultatelor teoretice, regula Gauss-Southwell tinde să depășească performanțele algoritmului de coborâre randomizată pe gradient.

Regula Gauss-Southwell ar trebui ocolită în practică atunci când gradientul complet este costisitor de calculat din punct de vedere computațional.

La fiecare pas al algoritmului, coordonata aleasă este:

$$i_k = \underset{i}{\operatorname{argmax}} \mid \nabla_i f(x^k) \mid \qquad (1)$$

Deoarece Gauss-Southwell alege la un pas în mod "lacom" coordonata cu magnitudinea derivatei parțiale cea mai mare, este nevoie de calculul complet al gradientului, pe când, pentru algoritmul randomizat, coordonata i_k este aleasă uniform aleator și la pasul k este suficient calculul derivatei parțiale $\nabla_{i_k} f(x^k)$.

Pentru atunci când gradientul complet poate fi calculat rapid, câștigurile sunt semnificative, iar rata de convergență a regulii Gauss-Southwell poate fi adițional îmbunătățită pentru funcțiile convexe ale căror gradienți sunt Lipschitz continui pe coordonate, i.e. $\exists (L_i)_i, i \in \{1, 2, ..., p\}$ a.î.:

$$\mid \nabla_i f(x + \alpha e_i) - \nabla_i f(x) \mid \leq L_i \mid \alpha \mid, \quad \forall \alpha \in IR, \forall x \in IR^p$$

În sensul acesta, Nutini et al. îmbină abordarea regulii Gauss-Southwell și cea a coborârii pe coordonată cu pași Lipschitz, unde pasul este constant pentru fiecare coordonată în parte (pentru fiecare coordonată i, mărimea pasului este egală cu L_i^{-1}) :

$$x^{k+1} = x^k - \frac{1}{L_{i_k}} \nabla_{i_k} f(x^k) e_{i_k} \qquad (2)$$

astfel îmbinând (1) și (2) se obține regula Gauss-Southwell-Lipschitz (GSL) folosită în alegerea coordonatei din algoritmul Greedy Coordinate Descent:

$$i_k = \underset{i}{\operatorname{argmax}} \frac{\mid \nabla_i f(x^k) \mid}{\sqrt{L_i}} \qquad (3)$$

Regula GSL reușește să obțină o rată de convergență mai bună decât ambele reguli [2].

2 Arhitectura algoritmului Coordinate Descent pentru funcții tari convexe

Folosim următorul Framework pentru algoritmii AGCD, ASCD și pentru a compara rezultatele cu algoritmul ARCD. Această construcție se pretează pentru o funcție f ce este μ-tare convexă, i.e.:

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle + \frac{\mu}{2}||y - x||_L^2, \quad unde ||x||_L^2 = \sum_{i=1}^{p} x_i^2 L_i$$

```
# gradient = the function that computes the full gradient for
    a parameter beta => gradient(beta)

# partial_gradient = computes a partial derivative for coord
   j and evaluates it in beta => partial_gradient(beta, j)

# coord1 and coord2 are functions that take in (L,
   full_gradient) as params and return the chosen coords
for k in range(iters):
    yk = (1 - a) * xprev + a * zprev
    full_grad = gradient(yk)

    jk1 = coord1(L, full_grad)
    # the basis vector with a 1 value on jk1 position
    ejk1 = np.zeros((n, 1))
    ejk1[jk1] = 1

    # the full gradient is not needed for ARCD
    if is_arcd:
        Djk1 = partial_gradient(yk, jk1)
    else:
        Djk1 = full_grad[jk1]

    xnext = yk - (1 / L[jk1]) * Djk1  * ejk1
    uk = (a * a) / (a * a + b) * zprev + b / (a * a + b) * yk

    # don't compute the argmin again for agcd
    if is_agcd:
        jk2 = jk1
    else:
        jk2 = coord2(L, full_grad)

```

```
    # the basis vector with a 1 value on jk2 position
    ejk2 = np.zeros((n, 1))
    ejk2[jk2] = 1

    # the full gradient is not needed for ARCD
    if is_arcd:
        Djk2 = partial_gradient(yk, jk2)
    else:
        Djk2 = full_grad[jk2]

    znext =
        uk - a / (a * a + b) * 1 / (n * L[jk2]) * Djk2 * ejk2

    zprev = znext
    xprev = xnext
```

Listing 1: Partea iterativă a Framework-ului Accelerated Coordinate Descent

Algoritmul se folosește de funcțiile `coord1` și `coord2` pentru a alege coordonatele j_{k_1} și j_{k_2} atunci când face x-actualizările, respectiv z-actualizările. Aceste funcții sunt primite ca parametri, în funcție de algoritmul dorit.

Pentru ARCD `coord1` și `coord2` vor fi `uniform_choice`:

```
# choose a coordinate uniformly for the randomized cd / semi-
    greedy cd
def uniform_choice(L, full_gradient=None):
    n, _ = L.shape

    return randint(0, n - 1)
```

Listing 2: Alegerea coordonatei în mod uniform aleator

Iar pentru AGCD, ambele funcții primite ca parametru vor fi `greedy_choice`:

```
# choose the coordinate according to the Gauss-Southwell-
    Lipschitz rule
def greedy_choice(L, full_gradient):
    gsl = (1 / np.sqrt(L)) * np.abs(full_gradient)

    return np.argmax(gsl)
```

Listing 3: Alegerea coordonatei în manieră Greedy

Iar algoritmul ASCD va primi ca parametru `greedy_choice` pentru `coord1` și `uniform_choice` pentru `coord2`.

Framework-ul primește ca parametrii de intrare: vectorul de constante Lipschitz `L`, valoarea μ, cunoscută pentru funcția μ-tare convexă `miu` și valorile inițiale ale parametrului x^0 pentru funcția $f(x)$ `x0`.

Deoarece funcțiile de cost ale regresiilor liniară și logistică au un singur punct de minim și fiind convexe, acest punct de minim va fi atins, algoritmii pot alege ca punct inițial vectorul $x^0 = (0, 0, ..., 0)$.

Pentru inițializarea algoritmului, codul rulat va fi:

```
n, _ = x0.shape

a = np.sqrt(miu) / (n + np.sqrt(miu))
b = (miu * a) / (n * n)

# determine whether the algorithm being run is agcd/arcd/ascd
# both coord selection methods are greedy
is_agcd = coord1 is coord2 and coord1 is greedy_choice
# both coord select meth are random uniform
is_arcd = coord1 is coord2 and coord1 is uniform_choice

# z0 = x0
zprev = x0
xprev = x0
```

Listing 4: Inițializarea parametrilor pentru Coordinate Descent Framework

3 Optimizarea funcției de cost pentru Regresia Logistică: Logistic Loss

Ne uităm peste problema de optimizare a funcției de cost, logistic regression loss cu ajutorul algoritmilor AGCD, ASCD și ARCD:

$$f^* = \min_{\beta \in IR^p} \frac{1}{n} \sum_{i=1}^{n} log(1 + exp(-y_i \beta^T x_i))$$

unde $x_i \in IR^p$ și $y_i \in \{-1, 1\}$, (x_i, y_i) fiind perechea de (features, etichetă) pentru sample-ul al i-lea din datele de învățare. Folosim Framework-ul pentru funcții tari convexe, cu parametrul $\mu = 10^{-7}$.

Pentru pasul de coborâre pe coordonată avem:

$$\frac{\partial f}{\partial \beta_j} = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{1}{1 + exp(-y_i \beta^T x_i)} - 1 \right) (y_i x_{ij})$$

Iar pentru $L = (L_i)$, conform Nutini et al.: "Pentru funcții dublu diferențiabile, acest lucru este echivalent cu presupunerea că elementele diagonale ale Hessianei sunt mărginite în magnitudine de către L" [2]. Astfel, avem:

$$L_j \leq \frac{\partial f}{\partial \beta_j^2} := \frac{1}{n} \sum_{i=1}^{n} \left(\frac{exp(-y_i \beta^T x_i)(y_i x_{ij})^2}{(1 + exp(-y_i \beta^T x_i))^2} \right), \quad \forall j \in \{1, 2, ..., p\}$$

Cum:

$$\frac{exp(z_i)(y_i x_{ij})^2}{(1 + exp(z_i))^2} = \frac{(y_i x_{ij})^2}{exp(z_i) + exp(-z_i) + 2} \qquad (*_1)$$

Și observăm că:

$$\frac{1}{exp(z_i) + exp(-z_i) + 2} \qquad (*_2)$$

își atinge maximul atunci când $z_i = 0$ și $(*_2) = 0.25$, rezultând că $(*_1) \leq 0.25 \cdot (y_i x_{ij})^2$ și deci deducem că $L_j \leq \frac{1}{n} \sum_{i=1}^{n} 0.25 \cdot (y_i x_{ij})^2$, $\forall j$ $(\#)$

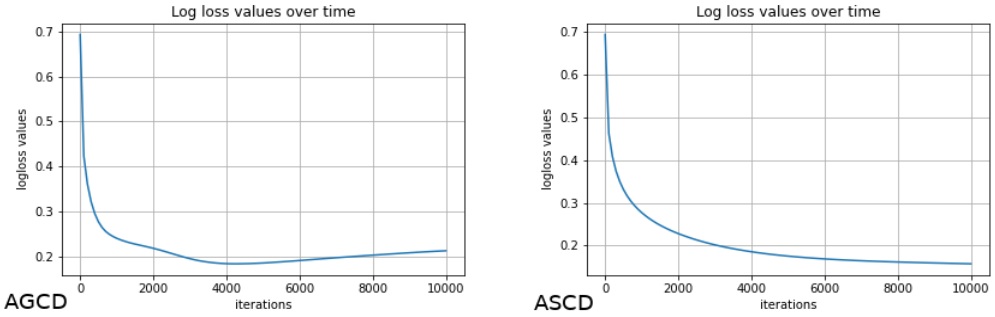

Figure 1: AGCD și ASCD cu L_i-urile inițializate aleator

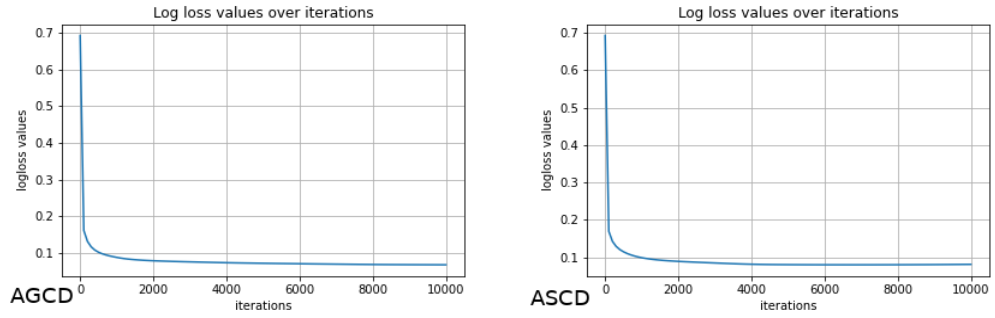

Figure 2: AGCD și ASCD cu L_i-urile aproximate de diagonala Hessianei

Într-adevăr, rulând atât AGCD, cât și ASCD pentru un număr de 10 000 de iterații pentru datasetul w1a din LIBSVM, prima dată inițializând pașii de coborâre (i.e. valorile vectorului L) cu valori aleatoare, iar a doua oară cu valorile aproximate conform (#), observăm că rata de convergență este mai bună în cazul al doilea.

Continuând cu datasetul w1a, rulăm AGCD și ASCD și comparăm cu performanțele obținute de ARCD pentru 1000 de iterații. Confruntăm valorile funcției de cost obținute de cei trei algoritmi pe parcursul iterațiilor realizate, dar ne uităm și la performanță vs. timp.

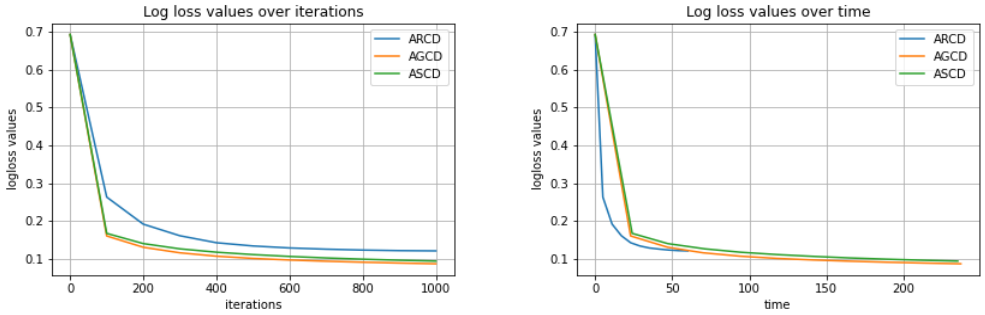

Figure 3: Cei trei algoritmi și valorile obținute pentru logistic loss

Se observă că AGCD și ASCD depășesc performanțele algoritmului ARCD per iterație, dar, ARCD neavând necesitatea de a calcula gradientul complet, este semnificativ mai rapid comparat la unități de timp.

Într-adevăr, AGCD are nevoie de 237 de secunde pentru a realiza cele 1000 de iterații, ASCD de 235 de secunde, pe când ARCD reușește să completeze cele 1000 de iterații în numai 60 de secunde.

4 Arhitectura algoritmului Coordinate Descent pentru funcții slab convexe

În continuare prezentăm Framework-ul folosit pentru algoritmii AGCD și ASCD în cazul în care acesta este utilizat pentru o funcție f despre care nu știm că este tare convexă.

Algoritmul are ca parametri de intrare funcția f (despre care știm că este L-smooth), L, $z^0 = x^0$ și secvența θ_k definită recursiv conform relației

$(1-\theta_k)/\theta_k^2 = 1/\theta_{k-1}^2$.

Pentru fiecare iterație k, algoritmul actualizează y_k, x_k, și z_k.

```
xk1 = [(1 - data['theta'][k]) * x for x in xk]
zk1 = [data['theta'][k] * z for z in zk]
yk = [x + z for x, z in zip(xk1, zk1)]

ej1 = np.zeros(data['p'])
ej1[j1] = gradient_k(data['y'], data['X'], yk, j1) / L[j1]
xk = yk - ej1

ej2 = np.zeros(data['p'])
ej2[j2] = gradient_k(data['y'], data['X'], yk, j2) /
          (data['n'] * L[j2] * data['theta'][k])
zk = zk - ej2
```

Listing 5: Actualizare coordonate

Alegem $j1$, $j2$ diferit în funcție de varianta aleasă a algoritmului, funcția *gradient* returnând $arg\min_i \frac{1}{\sqrt{L_i}}|\nabla_i f(y_k)|$.

```
if alg_type == 'arcd':
  j1 = int(np.random.uniform(0, data['p']))
elif alg_type == 'agcd' or alg_type == 'ascd':
  j1 = gradient(data['y'], data['X'], yk)



if alg_type == 'arcd' or lg_type == 'agcd':
  j2 = j1
elif alg_type == 'ascd':
  j2 = int(np.random.uniform(0, data['p']))
```

Listing 6: Alegere j1 și j2

5 Optimizarea funcției de cost pentru Regresia Liniară: Least Squares

Ne propunem să minimizăm functia

$$f^* := \min_{\beta \in R^p} f(\beta) := ||y - X\beta||_2^2$$

unde perechea (y, X) este generata conform Apendice B.1 din Lu et al.[1]

Întrucât funcția f este convexă și are un singur punct de minim, putem inițializa $x^0 = z^0 = (0, 0, 0, ...)$. În plus, f este derivabilă de doua ori, deci putem aproxima L cu diagonala matricei hessiene, i.e.:

$$L_k = \nabla_k^2 f(\beta) = 2 \sum_{i=1}^{n} x_{ik}^2$$

```
# init L with diag of hessian matrix
def gen_L(X):
    L = []
    for k in range(data['p']):
      sum = 0
      for i in range(data['n']):
        sum += X[i][k] ** 2
      sum = sum * 2
      # offset L from zero values
      L.append(sum + EPSILON)

    return L
```

Listing 7: Generare L

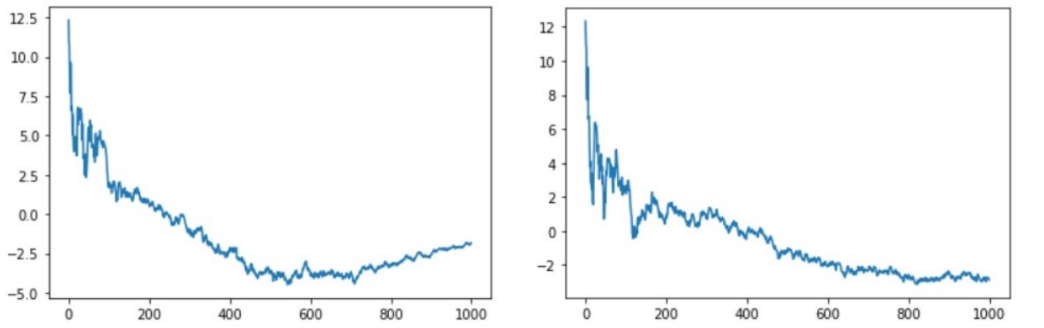

Figure 4: AGCD și ASCD pentru $k = 1000$

În urma testelor, se poate observa că modul de inițializare a lui L cu diagonala hessienei pentru un număr de pași $k = 1000$ aduce o rată de convergență mai bună atât pentru $AGCD$, cât și pentru $ASCD$. Totodată, observăm că algoritmul $AGCD$ converge mai repede la rezultatul dorit.

References

[1] H. Lu, R. M. Freund, and V. Mirrokni. Accelerating greedy coordinate descent methods, 2018.

[2] J. Nutini, M. Schmidt, I. Laradji, M. Friedlander, and H. Koepke. Coordinate descent converges faster with the gauss-southwell rule than random selection. In *International Conference on Machine Learning*, pages 1632–1641, 2015.