A generic coordinate descent solver for nonsmooth convex optimization

Olivier Fercoq

LTCI, Télécom ParisTech, Université Paris-Saclay, 46 rue Barrault, 75634 Paris Cedex 13, France

ARTICLE HISTORY
Compiled April 9, 2019

ABSTRACT
We present a generic coordinate descent solver for the minimization of a nonsmooth convex objective with structure. The method can deal in particular with problems with linear constraints. The implementation makes use of efficient residual updates and automatically determines which dual variables should be duplicated. A list of basic functional atoms is pre-compiled for efficiency and a modelling language in Python allows the user to combine them at run time. So, the algorithm can be used to solve a large variety of problems including Lasso, sparse multinomial logistic regression, linear and quadratic programs.

KEYWORDS
Coordinate descent; convex optimization; generic solver; efficient implementation

1. Introduction

Coordinate descent methods decompose a large optimization problem into a sequence of one-dimensional optimization problems. The algorithm was first described for the minimization of quadratic functions by Gauss and Seidel in [12]. Coordinate descent methods have become unavoidable in machine learning because they are very efficient for key problems, namely Lasso [7], logistic regression [14] and support vector machines [11, 13]. Moreover, the decomposition into small subproblems means that only a small part of the data is processed at each iteration and this makes coordinate descent easily scalable to high dimensions.

One of the main ingredients of an efficient coordinate descent solver is its ability to compute efficiently partial derivatives of the objective function [9]. In the case of least squares for instance, this involves the definition of a vector of residuals that will be updated during the run of the algorithm. As this operation needs to be performed at each iteration, and millions of iterations are usually needed, the residual update and directional derivative computation must be coded in a compiled programming language.

Many coordinate descent solvers have been written in order to solve a large variety of problems. However, most of the existing solvers can only solve problems of the type

$$\min_{x \in \mathbb{R}^N} \sum_{j=1}^{J} f(A_j x - b_j) + \sum_{i=1}^{I} g(x^{(i)})$$

where $x^{(i)} \in \mathbb{R}^{N_i}$ is the ith block of x, $\sum_{i=1}^{I} N_i = N$, $A_j \in \mathbb{R}^{M_j \times N}$ is a matrix and $b_j \in \mathbb{R}^{M_j}$ is a vector, and where f is a convex differentiable function and g is a convex lower-semicontinuous function whose proximal operator is easy to compute (a.k.a. a proximal-friendly

Email: olivier.fercoq@telecom-paristech.fr

convex function). Each piece of code usually covers only one type of function [3, 10]. Moreover, even when the user has a choice of objective function, the same function is used for every block [2].

In this work, we propose a generic coordinate descent method for the resolution of the convex optimization problem

$$\min_{x\in\mathbb{R}^N} \frac{1}{2}x^\top Qx + \sum_{j=1}^{J} c_j^f f_j(A_j^f x - b_j^f) + \sum_{i=1}^{I} c_i^g g_i(D_i^g x^{(i)} - b_i^g) + \sum_{l=1}^{L} c_l^h h_l(A_l^h x - b_l^h)\ . \qquad (1)$$

We shall call f_j, g_i and h_l atom functions. Each of them may be different. We will assume that f_j's are differentiable and convex, g_i's and h_l's are proximal-friendly convex functions. As before $A_j^f \in \mathbb{R}^{M_j^f\times N}$ and $A_l^h \in \mathbb{R}^{M_l^h\times N}$ are matrices, D_i^g is a multiple of the identity matrix of size N_i, $b_j^f \in \mathbb{R}^{M_j^f}$, $b_i^g \in \mathbb{R}^{N_i}$ and $b_l^h \in \mathbb{R}^{M_l^h}$ are vectors, c_j^f, c_i^g and c_l^h are positive real numbers, Q is a $N\times N$ positive semi-definite matrix.

The algorithm we implemented is described in [4] and can be downloaded on `https://bitbucket.org/ofercoq/cd_solver`. It deals with residual updates and dual variable duplication in a generic fashion and includes a modelling interface in Python for the definition of the optimization problem. Note that unlike most coordinate descent implementations, it can deal with nonseparable nonsmooth objectives and linear constraints.

2. Description of the Algorithm

2.1. General scheme

The algorithm we implemented is a coordinate descent primal-dual method developed in [4]. Let us denote $F(x) = \frac{1}{2}x^\top Qx + \sum_{j=1}^{J} c_j^f f_j(A_j^f x - b_j^f)$, $G(x) = \sum_{i=1}^{I} c_i^g g_i(D_i^g x^{(i)} - b_i^g)$, $H(z) = \sum_{l=1}^{L} c_l^h h_l(z^{(l)} - b_l^h)$, $\mathcal{J}(i) = \{j \,:\, A_{j,i}^h \neq 0\}$, $\mathcal{I}(j) = \{i \,:\, A_{j,i}^h \neq 0\}$, $m_j = |\mathcal{I}(j)|$ and $\rho(A)$ the spectral radius of matrix A. The algorithm writes then as Algorithm 1.

We will denote $U_1, \dots, U_I$ the columns of the identity matrix corresponding to the blocks of $x = (x^{(1)}, \dots, x^{(I)})$, so that $U_i x^{(i)} \in \mathbb{R}^N$ and $V_1, \dots, V_J$ the columns of the identity matrix corresponding to the blocks of $A^f x - b^f = (A_1^f x - b_1^f, \dots, A_J^f x - b_J^f)$.

2.2. Computation of partial derivatives

For simplicity of implementation, we are assuming that G is separable and the blocks of variable will follow the block structure of G. This implies in particular that at each iteration, only $\nabla_i F(x_k)$ needs to be computed. This partial derivative needs to be calculated efficiently because it needs to be performed at each iteration of the algorithm. We now describe the efficient residual update method, which is classically used in coordinate descent implementations [9].

Denote $r_k^{f,x} = A^f x_k - b^f$. By the chain rule, we have

$$\nabla_i F(x_k) = \sum_{j=1}^{J} c_j^f (A^f)_{j,i}^\top \nabla f_j(A_j^f x_k - b_j^f) = \sum_{j\in\mathcal{J}^f(i)} c_j^f (A^f)_{j,i}^\top \nabla f_j((r_k^{f,x})_j)$$

If $r_k^{f,x}$ is pre-computed, only $O(|\mathcal{J}^f(i)|)$ operations are needed.

For an efficient implementation, we will update the residuals $r_k^{f,x}$ as follows, using the fact

Algorithm 1 Coordinate-descent primal-dual algorithm with duplicated variables

Input: Differentiable function $F : \mathbb{R}^N \to \mathbb{R}$, matrix $A^h \in \mathbb{R}^{M^h \times N}$, functions G and H whose proximal operators are available.

Initialization: Choose $x_0 \in \mathbb{R}^N$, $\boldsymbol{y}_0 \in \mathbb{R}^{\mathrm{nnz}(A^h)}$. Denote $\mathcal{J}(i) = \{j \ : \ A^h_{j,i} \neq 0\}$, $\mathcal{I}(j) = \{i \ : \ A^h_{j,i} \neq 0\}$, $m_j = |\mathcal{I}(j)|$ and $\rho(A)$ the spectral radius of matrix A. Choose step sizes $\tau \in \mathbb{R}^I_+$ and $\sigma \in \mathbb{R}^L_+$ such that $\forall i \in \{1, \dots I\}$,

$$\tau_i < \frac{1}{\beta_i + \rho\left(\sum_{j \in \mathcal{J}(i)} m_j \sigma_j (A^h)^\top_{j,i} A^h_{j,i}\right)}. \tag{2}$$

For all $i \in \{1, \dots, I\}$, set $w_0^{(i)} = \sum_{j \in \mathcal{J}(i)} (A^h)^\top_{j,i} \boldsymbol{y}_0^{(j)}(i)$.

For all $j \in \{1, \dots, J\}$, set $z_0^{(j)} = \frac{1}{m_j} \sum_{i \in \mathcal{I}(j)} \boldsymbol{y}_0^{(j)}(i)$.

Iteration k: Define:

$$\begin{aligned}
\overline{y}_{k+1} &= \mathrm{prox}_{\sigma, H^\star}\left(z_k + D(\sigma) A^h x_k\right) \\
\overline{x}_{k+1} &= \mathrm{prox}_{\tau, G}\left(x_k - D(\tau)\left(\nabla F(x_k) + 2(A^h)^\top \overline{y}_{k+1} - w_k\right)\right).
\end{aligned}$$

For $i = i_{k+1}$ and for each $j \in \mathcal{J}(i_{k+1})$, update:

$$\begin{aligned}
x_{k+1}^{(i)} &= \overline{x}_{k+1}^{(i)} \\
\boldsymbol{y}_{k+1}^{(j)}(i) &= \overline{y}_{k+1}^{(j)} \\
w_{k+1}^{(i)} &= w_k^{(i)} + \sum_{j \in J(i)} (A^h)^\top_{j,i} (\boldsymbol{y}_{k+1}^{(j)}(i) - \boldsymbol{y}_k^{(j)}(i)) \\
z_{k+1}^{(j)} &= z_k^{(j)} + \frac{1}{m_j} (\boldsymbol{y}_{k+1}^{(j)}(i) - \boldsymbol{y}_k^{(j)}(i)).
\end{aligned}$$

Otherwise, set $x_{k+1}^{(i')} = x_k^{(i')}$, $w_{k+1}^{(i')} = w_k^{(i')}$, $z_{k+1}^{(j')} = z_k^{(j')}$ and $\boldsymbol{y}_{k+1}^{(j')}(i') = \boldsymbol{y}_k^{(j')}(i')$.

---

that only the coordinate block i_{k+1} is updated:

$$\begin{aligned}
r_{k+1}^{f,x} = A^f x_{k+1} - b^f = A^f\left(x_k + U_{i_{k+1}}(x_{k+1}^{(i_{k+1})} - x_k^{(i_{k+1})})\right) - b^f &= r_k^{f,x} + A^f U_{i_{k+1}}(x_{k+1}^{(i_{k+1})} - x_k^{(i_{k+1})}) \\
&= r_k^{f,x} + \sum_{j \in \mathcal{J}^f(i_{k+1})} V_j A^f_{j,i_{k+1}} (x_{k+1}^{(i_{k+1})} - x_k^{(i_{k+1})})
\end{aligned}$$

Hence, updating $r_{k+1}^{f,x}$ also requires only $O(|\mathcal{J}^f(i_{k+1})|)$ iterations.

Similarly, updating the residuals $r_k^{h,x} = A^h x_k - b^h$, w_k and z_k can be done in $O(|\mathcal{J}(i_{k+1})|)$ operations.

2.3. Computation of proximal operators using atom functions

Another major step in the method is the computation of the i^{th} coordinate of $\mathrm{prox}_{\tau,G}(x)$ for a given $x \in \mathbb{R}^N$.

As D^g is assumed to be diagonal, G is separable. Hence, by the change of variable $\bar{z} = D^g_i \bar{x} - b^g_i$,

$$\begin{aligned}
(\mathrm{prox}_{\tau,G}(x))_i &= \arg\min_{\bar{x}\in\mathbb{R}^{N_i}} c^g_i g_i(D^g_i \bar{x} - b^g_i) + \frac{1}{2\tau_i}\|\bar{x} - x^{(i)}\|^2 \\
&= (D^g_i)^{-1}\Big(b^g_i + \arg\min_{\bar{z}\in\mathbb{R}^{N_i}} c^g_i g_i(\bar{z}) + \frac{1}{2\tau_i}\|(D^g_i)^{-1}(b^g_i + \bar{z}) - x^{(i)}\|^2\Big) \\
&= (D^g_i)^{-1}\Big(b^g_i + \arg\min_{\bar{z}\in\mathbb{R}^{N_i}} g_i(\bar{z}) + \frac{1}{2c^g_i (D^g_i)^2 \tau_i}\|\bar{z} - (D^g_i x^{(i)} - b^g_i)\|^2\Big) \\
&= (D^g_i)^{-1}\Big(b^g_i + \mathrm{prox}_{c^g_i (D^g_i)^2 \tau_i g}(D^g_i x^{(i)} - b^g_i)\Big)
\end{aligned}$$

where we used the abuse of notation that D^g_i is either the scaled identity matrix or any of its diagonal elements. This derivation shows that to compute $(\mathrm{prox}_{\tau,G}(x))_i$ we only need linear algebra and the proximal operator of the atom function g_i.

We can similarly compute $\mathrm{prox}\, H$. To compute $\mathrm{prox}_{\sigma,H^\star}$, we use Moreau's formula:

$$\mathrm{prox}_{\sigma,H^\star}(z) = z - D(\sigma)\,\mathrm{prox}_{\sigma^{-1},H}(D(\sigma)^{-1}z)$$

2.4. Duplication of dual variables

Algorithm 1 maintains duplicated dual variables $\boldsymbol{y}_k \in \mathbb{R}^{\mathrm{nnz}(A^h)}$ as well as averaged dual variables $z_k \in \mathbb{R}^M$ where $M = \sum_{l=1}^L M_l$ and $A^h_{l,i}$ is of size $M_l \times N_i$. The sets $\mathcal{J}(i)$ for all i are given by the sparse column format representation of A^h. Yet, for all i, we need to construct the set of indices of $\boldsymbol{y}_{k+1}$ that need to be updated. This is the table `dual_vars_to_update` in the code. Moreover, as H is not separable in general, in order to compute $\bar{y}^j_{k+1}$, for $j \in \mathcal{J}(i_{k+1})$, we need to determine the set of dual indices j' that belong to the same block as j with respect to the block decomposition of H. This is the purpose of the tables `inv_blocks_h` and `blocks_h`.

3. Code structure

The code is organized in seven files. The main file is `cd_solver.pyx`. It contains the Python callable and the data structure for the problem definition. The other files are `atoms.pyx/pxd`, `algorithm.pyx/pxd`, and `helpers.pyx/pxd`. They contain the definition of the atom functions, the algorithms and functions for computing objective value respectively. In Figure 1, we show in which function each subfunction is used. The user needs to call the Python class `Problem` and the Python function `coordinate_descent`. Atom functions can be added by the user without modifying the main algorithm.

All tables are defined using Numpy's array constructor in the `coordinate_descent` function. The main loop of coordinate descent and the atom functions are pre-compiled for efficiency.

4. Atom functions

The code allows us to define atom functions independently of the coordinate descent algorithm. As an example, we provide in Figure 2 the code for the square function atom.

As inputs, it gets `x` (an array of numbers which is the point where the operation takes place), `buff` (the buffer for vectorial outputs), `nb_coord` (is the size of `x`), `mode`, `prox_param` and `prox_param2` (numbers which is needed when computing the proximal operator). The input `mode` can be:

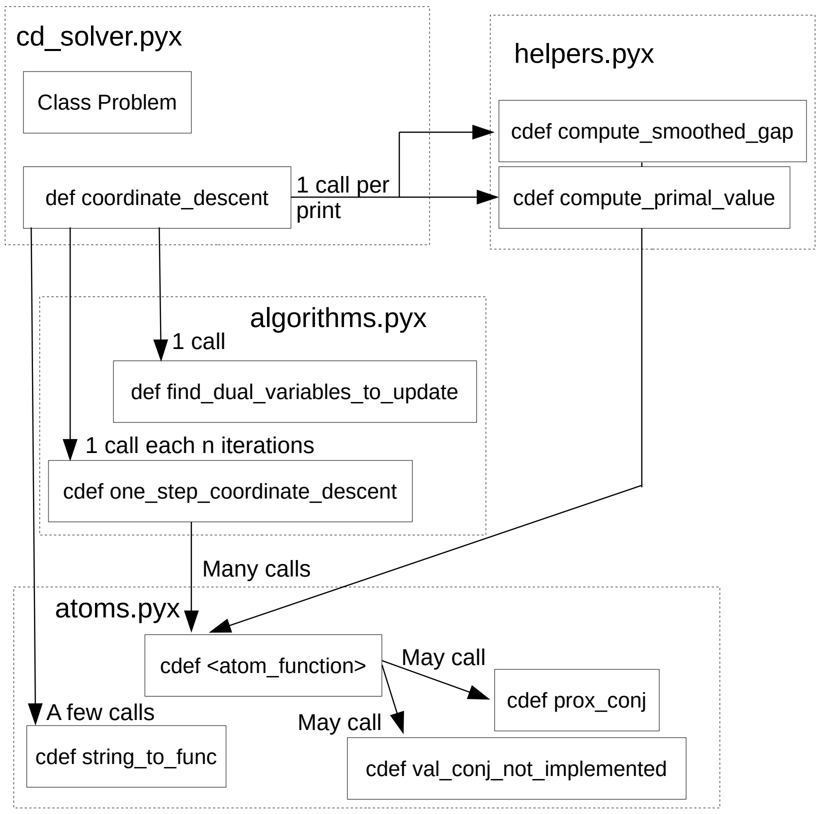

Figure 1. Code structure

- `GRAD` in order to compute the gradient.
- `PROX` to compute the proximal operator.
- `PROX_CONJ` uses Moreau's formula to compute the proximal operator of the conjugate function.
- `LIPSCHITZ` to return the Lipschitz constant of the gradient.
- `VAL_CONJ` to return the value of the conjugate function. As this mode is used only by `compute_smoothed_gap` for printing purposes, its implementation is optional and can be approximated using the helper function `val_conj_not_implemented`. Indeed, for a small $\epsilon > 0$, $h^*(y) = \sup_z \langle z, y\rangle - h(z) \approx \sup_z \langle z, y\rangle - h(z) - \frac{\epsilon}{2}\|z\|^2 = \langle p, y\rangle - h(p) - \frac{\epsilon}{2}\|p\|^2$, where $p = \mathrm{prox}_{h/\epsilon}(y/\epsilon)$.
- `VAL` to return the value of the function.

Some functions naturally require multi-dimensional inputs, like $\|\cdot\|_2$ or the log-sum-exp function. For consistency, we define all the atoms with multi-dimensional inputs: for an atom function $f_0 : \mathbb{R} \to \mathbb{R}$, we extend it to an atom function $f : \mathbb{R}^{N_i} \to \mathbb{R}$ by $f(x) = \sum_{l=1}^{N_i} f_0(x_l)$.

For efficiency purposes, we are bypassing the square atom function when computing a gradient and implemented it directly in the algorithm.

5. Modelling language

In order to use the code in all its generality, we defined a modelling language that can be used to define the optimization problem we want to solve (1).

The user defines a problem using the class `Problem`. Its arguments can be:

```
cdef DOUBLE square(DOUBLE[:] x, DOUBLE[:] buff, int nb_coord, MODE mode,
        DOUBLE prox_param, DOUBLE prox_param2) nogil:
    # Function x -> x**2
    cdef int i
    cdef DOUBLE val = 0.
    if mode == GRAD:
        for i in range(nb_coord):
            buff[i] = 2. * x[i]
        return buff[0]
    elif mode == PROX:
        for i in range(nb_coord):
            buff[i] = x[i] / (1. + 2. * prox_param)
        return buff[0]
    elif mode == PROX_CONJ:
        return prox_conj(square, x, buff, nb_coord, prox_param, prox_param2)
    elif mode == LIPSCHITZ:
        buff[0] = 2.
        return buff[0]
    elif mode == VAL_CONJ:
        return val_conj_not_implemented(square, x, buff, nb_coord)
    else:  # mode == VAL
        for i in range(nb_coord):
            val += x[i] * x[i]
        return val
```

Figure 2. Code for the square function atom

- `N` the number of variables, `blocks` the blocks of coordinates coded in the same fashion as the indptr index of sparse matrices (default [0, 1, . . . , `N`]), `x_init` the initial primal point (default 0) and `y_init` the initial duplicated dual variable (default 0)
- Lists of strings `f`, `g` and `h` that code for the atom functions used. The function `string_to_func` is responsible for linking the atom function that corresponds to the string. Our convention is that the string code is exactly the name of the function in `atoms.pyx`. The size of the input of each atom function is defined in `blocks_f`, `blocks` and `blocks_h`. The function strings `f`, `g` or `h` may be absent, which means that the function does not appear in the problem to solve.
- Arrays and matrices `cf`, `Af`, `bf`, `cg`, `Dg`, `bg`, `ch`, `Ah`, `bh`. The class initiator transforms matrices into the sparse column format and checks whether `Dg` is diagonal.

6. Extensions

6.1. Non-uniform probabilities

We added the following feature for an improved efficiency. Under the argument `sampling='kink_half'`, the algorithms periodically detects the set of blocks I_{kink} such that $i \in I_{\text{kink}}$ if $x^{(i)}$ is at a kink of g_i. Then, block i is selected with probability law

$$\mathbb{P}(i_{k+1} = i) = \begin{cases} \frac{1}{n} & \text{if } |I_{\text{kink}}| = n \\ \frac{1}{2n} & \text{if } |I_{\text{kink}}| < n \text{ and } i \in I_{\text{kink}} \\ \frac{1}{2n} + \frac{1}{2(n-|I_{\text{kink}}|)} & \text{if } |I_{\text{kink}}| < n \text{ and } i \notin I_{\text{kink}} \end{cases}$$

The rationale for this probability law is that blocks at kinks are likely to incur no move when we try to update them. We thus put more computational power for non-kinks. On the other hand, we still keep an update probability weight of at least $\frac{1}{2n}$ for each block, so even in unfavourable cases, we should not observe too much degradation in the performance as compared to the uniform law.

6.2. *Acceleration*

We also coded accelerated coordinate descent [6], as well as its restarted [5] and primal-dual [1] variants.

The accelerated algorithms improve the worst case guarantee as follows:

	$h = 0$	$h \neq 0$
Alg. 1	$O(1/k)$	$O(1/\sqrt{k})$
APPROX / SMART-CD	$O(1/k^2)$	$O(1/k)$

Table 1. Convergence speed of the algorithms implemented

However, accelerated algorithms do not take profit of regularity properties of the objective like strong convexity. Hence, they are not guaranteed to be faster, even though restart may help.

6.3. *Variable screening*

The code includes the Gap Safe screening method presented in [8]. Note that the method has been studied only for the case where $h = 0$. Given a non-differentiability point $x_\star^{(i)}$ of the function g_i where the subdifferential $\partial g_i(x_\star^{(i)})$ has a non-empty interior, a test is derived to check whether $x_\star^{(i)}$ is the i^{th} variable of an optimal solution. If this is the case, one can set $x^{(i)} = x_\star^{(i)}$ and stop updating this variable. This may lead to a huge speed up in some cases. As the test relies on the computation of the duality gap, which has a nonnegligible cost, it is only performed from time to time.

7. Numerical validation

7.1. *Performance*

In order to evaluate the performance of the implementation, we compare our implementation with a pure Python coordinate descent solver and code written for specific problems: Scikit learn's Lasso solver and Liblinear's SVM solver. We run the code on an Intel Xeon CPU at 3.07GHz.

We can see on Table 2 that our code is hundreds of times faster than the pure Python code. This is due to the compiled nature of our code, that does not suffer from the huge number of

Lasso	
Pure Python	308.76s
`cd_solver`	0.43s
Scikit learn Lasso	0.11s

SVM	
Pure Python	126.24s
`cd_solver`	0.31s
Liblinear SVM	0.13s

Table 2. Comparison of our code with a pure Python code and reference implementations for performing $100n$ coordinate descent iterations for the Lasso problem on the Leukemia dataset with regularization parameter $\lambda = 0.1\|(A^f)^\top b^f\|_\infty$, and for $10n$ coordinate descent iterations for the dual SVM problem on the RCV1 dataset with penalty parameter $C = 10$.

iterations required by coordinate descent. On the other hand, our code is about 4 times slower than state-of-the-art coordinate descent implementations designed for a specific problem. We can see it in both examples we chose. This overhead is the price of genericity.

We believe that, except for critical applications like Lasso or SVM, a 4 times speed-up does not justify writing a new code from scratch, since a separate piece of code for each problem makes it difficult to maintain and to improve with future algorithmic advances.

7.2. Genericity

We tested our algorithm on the following problems:

- Lasso problem

$$\min_{x\in\mathbb{R}^n} \frac{1}{2}\|Ax-b\|_2^2 + \lambda\|x\|_1$$

- Binomial logistic regression

$$\min_{x\in\mathbb{R}^n} \sum_{i=1}^m \log(1+\exp(b_i(Ax)_i)) + \frac{\lambda}{2}\|x\|_2^2$$

  where $b_i \in \{-1, 1\}$ for all i.
- Sparse binomial logistic regression

$$\min_{x\in\mathbb{R}^n} \sum_{i=1}^m \log(1+\exp(b_i(Ax)_i)) + \lambda\|x\|_1$$

- Dual SVM without intercept

$$\min_{x\in\mathbb{R}^n} \frac{1}{2\alpha}\|A^\top D(b)x\|_2^2 - e^\top x + \iota_{[0,1]^n}(x)$$

  where $\iota_{[0,1]^n}$ is the convex indicator function of the set $[0,1]^n$ and encodes the constraint $x \in [0,1]^n$.
- Dual SVM with intercept

$$\min_{x\in\mathbb{R}^n} \frac{1}{2\alpha}\|A^\top D(b)x\|_2^2 - e^\top x + \iota_{[0,1]}(x) + \iota_{\{0\}}(b^\top x)$$

- Linearly constrained quadratic program

$$\min_{x\in\mathbb{R}^n} \frac{1}{2}\|(A^f)^\top x - b^f\|_2^2 + \iota_{\{0\}}(A^h x - b^h)$$

- Linear program

$$\min_{x\in\mathbb{R}^n} c^\top x + \iota_{\mathbb{R}^n_+}(x) + \iota_{\mathbb{R}^m_-}(Ax-b)$$

- TV-regularized regression

$$\min_{x\in\mathbb{R}^{n_1 n_2 n_3}} \frac{1}{2}\|Ax-b\|_2^2 + \alpha\|Dx\|_{2,1}$$

where D is the discrete gradient operator and $\|y\|_{2,1} = \sum_{i,j,k} \sqrt{\sum_{l=1}^{3} y_{i,j,k,l}^2}$.

- Sparse multinomial logistic regression

$$\min_{x \in \mathbb{R}^{n \times q}} \sum_{i=1}^{m} \log \Big(\sum_{j=1}^{q} \exp \big(\sum_{l=1}^{n} A_{i,l} x_{l,j} \big) \Big) + \sum_{i=1}^{n} \sum_{j=1}^{q} x_{i,j} b_{i,j} + \sum_{l=1}^{n} \sqrt{\sum_{j=1}^{q} x_{l,j}^2}$$

where $b_{i,j} \in \{0, 1\}$ for all i, j.

This list demonstrates the ability of the method to deal with differentiable functions, separable or nonseparable nondifferentiable functions, as well as use several types of atom function in a single problem.

References

[1] Ahmet Alacaoglu, Quoc Tran Dinh, Olivier Fercoq, and Volkan Cevher. Smooth primal-dual coordinate descent algorithms for nonsmooth convex optimization. In *Advances in Neural Information Processing Systems*, pages 5852–5861, 2017.

[2] Mathieu Blondel and Fabian Pedregosa. Lightning: large-scale linear classification, regression and ranking in Python, 2016.

[3] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *Journal of machine learning research*, 9(Aug):1871–1874, 2008.

[4] Olivier Fercoq and Pascal Bianchi. A Coordinate Descent Primal-Dual Algorithm with Large Step Size and Possibly Non Separable Functions. *arXiv preprint arXiv:1508.04625*, 2015. Accepted for publication in the SIAM Journal on Optimization.

[5] Olivier Fercoq and Zheng Qu. Restarting the accelerated coordinate descent method with a rough strong convexity estimate. *arXiv preprint arXiv:1803.05771*, 2018.

[6] Olivier Fercoq and Peter Richtárik. Accelerated, parallel and proximal coordinate descent. *SIAM Journal on Optimization*, 25(4):1997–2023, 2015.

[7] Jerome Friedman, Trevor Hastie, Holger Höfling, and Robert Tibshirani. Pathwise coordinate optimization. *Ann. Appl. Stat.*, 1(2):302–332, 2007.

[8] Eugène Ndiaye. *Safe Optimization Algorithms for Variable Selection and Hyperparameter Tuning*. PhD thesis, Université Paris-Saclay, 2018.

[9] Yurii Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization*, 22(2):341–362, 2012.

[10] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[11] John C Platt. Fast Training of Support Vector Machines using Sequential Minimal Optimization. In Bernhard Schölkopf, Christopher J. C. Burges, and Alexander J. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1999. http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets.

[12] Ludwig Seidel. Ueber ein Verfahren, die Gleichungen, auf welche die Methode der kleinsten Quadrate führt, sowie lineäre Gleichungen überhaupt, durch successive Annäherung aufzulösen:(Aus den Abhandl. dk bayer. Akademie II. Cl. XI. Bd. III. Abth. *Verlag der k. Akademie*, 1874.

[13] Shai Shalev-Shwartz and Tong Zhang. Stochastic Dual Coordinate Ascent Methods for Regularized Loss Minimization. *Journal of Machine Learning Research*, 14(1):567–599, 2013.

[14] Hsiang-Fu Yu, Fang-Lan Huang, and Chih-Jen Lin. Dual coordinate descent methods for logistic regression and maximum entropy models. *Machine Learning*, 85(1-2):41–75, 2011.