Dual Relationship of Coordinate Descent and Dykstra's Projection Algorithm, Application to Lasso and Extensions

Kwok Hung Ho

05/26/ 2022

1 Introduction and Problem Description

Consider the problem:

$$\min_{\beta \in \mathbb{R}^p} \frac{1}{2}\|y - X\beta\|_2^2 + \sum_{i=1}^{d} h_i(\beta_i) \tag{P1}$$

And the problem:

$$\begin{aligned} &\min_{z \in \mathbb{R}^n} \|y - z\|_2 \\ &z \in \mathcal{C}_1 \cap \mathcal{C}_2 \dots \mathcal{C}_d \end{aligned} \tag{P2}$$

This paper will show the necessary and sufficient conditions for which coordinate descent will succeed on P1 and Dykstra's projection algorithm will succeed on P2, and examine their relationship via duality.

The primal Lasso problem will also be formulated as P1, and its dual will be derived to take the form of P2.

A code implementation of coordinate descent and Dykstra's algorithm will also be provided and experimental results regarding the complexity and support recovery performance will be shown.

Two problems in literature, (Ridge Regression and Matrix Completion) will also be examined for their suitableness for Coordinate Descent and Dykstra's Projection algorithm to be applied on.

2 Coordinate Descent

2.1 Coordinate Descent Description and Algorithm

Coordinate Descent is an optimization algorithm for finding the minimum of a function. Instead of calculating the gradient and taking a step in that direction (as in gradient descent), coordinate descent determines a coordinate (cyclically or randomly) in which to minimize over while fixing all other coordinates, and minimizes that corresponding hyperplane. [1]

Let's say coordinate k was chose at iteration t, then the update from β^t to β^{t+1} is as follows:

$$\beta_k^{t+1} = \arg\min_{\beta_k} f(\beta_1^t, \beta_1^t, \ldots, \beta_{k-1}^t, \boldsymbol{\beta_k^t}, \beta_{k+1}^t, \ldots, \beta_p^t) \tag{1}$$

and $\beta_j^{t+1} = \beta_j^t$ for $j \neq k$. Where β is the minimizing variable or (weights) as in (P1). When blocks of coordinates are taken instead of individual coordinates, this algorithm is known as Block Coordinate Descent (BCD).

More formally, the BCD algorithm applied on (P1) can be expressed as such: Initialize $\beta^{(0)} = 0$ and repeat for $k = 1, 2, 3, \ldots$

$$\beta_i^{(k)} = \arg\min_{\beta_i \in \mathbb{R}_i^p} \frac{1}{2} \|y - \sum_{j<i} X\beta_j^{(k)} - \sum_{j>i} X\beta_j^{(k-1)} - X_i\beta_i\|_2^2 + h_i(\beta_i), \quad i = 1, \ldots d \tag{2}$$

Where $X_i \in \mathbb{R}^{n \times p_i}$ as a result of block selection and are assumed to have full rank for convenience and uniqueness[4] of updates.

2.2 Coordinatewise Optimality

For non-differentiable objective functions or ones with a non-differentiable term, gradient descent is not an option. However, certain classes of problems, like Lasso and its variants have a separability property which allows for coordinate descent to work.

A sufficient property for coordinate descent to converge to the global minimum is for the objective function f to be continuously differentiable and strictly convex at every coordinate[5]. For a convex function, the global minimum is attained when the gradient is 0 and one can see that

$$\nabla f(x) = (\frac{\partial f}{\partial x_1}(x), \ldots \frac{\partial f}{\partial x_n}(x)) = 0 \tag{3}$$

along each coordinate, the partial derivative is equal to 0 and hence coordinate-wise optimal.

When the function is only convex but not differentiable, coordinate-wise optimality is not guaranteed.

At the highlighted point. Coordinate descent along both $x1$ and $x2$ will be optimal but one can see from the level curves that the global minimum is not attained. Hence, non-differentiable functions can cause issues.

However, the sufficient property is not necessary, as statistical regularizers can allow coordinate descent to succeed on even non-differentiable objective functions.

For an arbitrary objective function of the form:

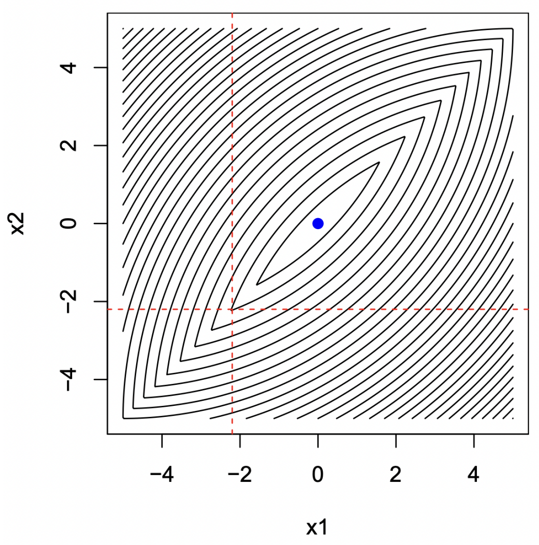

Figure 1: Coordinate Descent (Failure)

$$f(\beta_1, \dots \beta_p) = g(\beta_1, \dots \beta_p) + \sum_{j=1}^{p} h_j(\beta_j) \tag{4}$$

Where $g : \mathbb{R}^p \mapsto \mathbb{R}$ is a convex and differentiable function, and the univariate functions $h_j : \mathbb{R} \mapsto \mathbb{R}$ are convex. As a whole, the second term can be viewed as a non-differentiable but separable term $h(\beta) = \sum_{j=1}^{p} h_j(\beta_j)$, of convex functions.

When this condition is matched, coordinate descent is a suitable algorithm for minimization.

A stronger definition of this requirement is provided. Define a stationary point $z \in domf$:

$$f'(z; d) \geq 0, \forall d \tag{5}$$

z is also a coordinatewise minimum point of f if $z \in domf$ and:

$$f'(z + (0, \dots d_k \dots 0)) \geq f(z), \forall d_k \in \mathbb{R}^{nk} \tag{6}$$

Where all points in the vector $(0, \dots d_k \dots 0)$ are 0 except at the coordinate block d_k.

We also define regularity of a function, and say that f is regular at $z \in domf$ if:

$$\begin{aligned} &f'(z; d) \geq 0, \forall d = (d_1, \dots d_N), \\ &\text{s.t } f'(z + (0, \dots d_k \dots 0)) \geq 0, \quad k = 1, \dots N \end{aligned} \tag{7}$$

It follows that a coordinate-wise minimum point z of f is a stationary point of f whenever f is regular at z. Where a stationary point can be thought of as the global minimum in a convex optimization problem.

Referring back to (4), the regularity of f can be ensured by having 2 assumptions on g. The first is that g is Gatueux-differentiable on $domg$ and the second is that g is Gateaux-differentiable on $int(domg)$ and, for every $z \in domf \cap bdry(domf)$, $\exists k \in \{1, \dots N\}$ and $d_k \in \mathbb{R}^{nk}$ such that $f(z + (0, \dots d_k \dots 0)) < f(z)$.

Without proving or defining these assumptions, the objective f under these assumptions will allow BCD to converge.

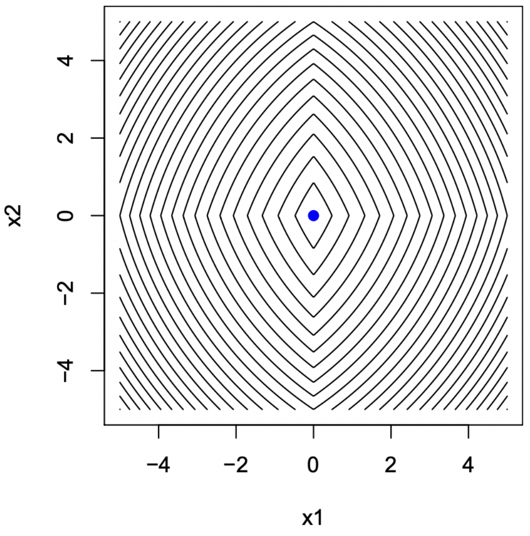

Figure 2: Coordinate Descent (Success)

Due to separability, which implies regularity, along coordinates x_1 and x_2, the function will always decrease to the minimum level curve and coordinate descent converges.

2.3 Conditions for Success on P1

For P1 specifically, when matching it to (4), as mentioned before, it is required that $g : \mathbb{R}^p \mapsto \mathbb{R}$ is a convex and differentiable function, and the univariate functions $h_j : \mathbb{R} \mapsto \mathbb{R}$ are convex.

Since $g(\beta) = \frac{1}{2}\|y - X\beta\|_2^2$, is a composite linear function of the l_2 norm it is convex and differentiable, and satisfies the first part of the sufficient property. For univariate functions h_j, which are unspecified, since they are already grouped in summation form, the second term as a whole is separable (hence regular), and hence the only condition for BCD to reach the stationary point is convexity of the univariate functions h_j.

3 Dykstra's Projection Algorithm

3.1 Description of Dykstra's Algorithm

Dykstra's algorithm is an algorithm for computing a point in the intersection of convex sets. It is given as follows:

Initialize $u^{(0)} = y$, $z^{(-d+1)} = \cdots = z^{(0)} = 0$, then repeat for $k = 1, 2, \ldots$

$$\begin{aligned} u^{(k)} &= P_{c_{[k]}}(u^{(k-1)} + z^{(k-d)}), \\ z^{(k)} &= u^{(k-1)} + z^{(k-d)} - u^{(k)}, \end{aligned} \tag{8}$$

Where $P_c(x)$ is the euclidean projection: $\arg\min_{c \in C} \|x - c\|_2^2$ onto a closed convex set C, $[\cdot]$ is the modulo operator, and z is the auxiliary or error variable.

3.2 Dykstra's algorithm and alternating projection algorithm

Dykstra's algorithm generally converges to the solution for closed convex sets of non-empty intersection and is a variant of the alternating projection algorithm.

Consider the standard alternating projection for 2 sets for $x \in \mathcal{C}_1 \cap \mathcal{C}_2$:

$$x_{k+1} = \mathcal{P}_C \left(\mathcal{P}_D(x_k) \right). \tag{9}$$

and the standard alternating projection for d sets for $x \in \mathcal{C}_1 \cap \mathcal{C}_2 \ldots \mathcal{C}_d$:

$$x_{k+1} = \mathcal{P}_{C_d} \left(\ldots \mathcal{P}_{C_2} \left(\mathcal{P}_{C_1}(x_k) \right) \right). \tag{10}$$

Compared to (8), Dykstra's projection algorithm uses additional auxiliary variables such as z in between projections. These auxiliary variables $z^{(k)}$ (not to be confused with z in P2) are also known as dual variables and they track (in a cyclic fashion) the residual of projecting onto $C_1, C_2, \ldots C_d$.

The key difference between the two is that: when there are multiple points in the intersection of closed sets, Dykstra's will return the projection of the initial point used, where as the standard alternating projection algorithm will return an arbitrary point.

It is due to this difference that for P2, the alternating projection will not even converge to the solution unless $C_1, C_2, \ldots C_d$ are subspaces. (in which case Dykstra's and alternating projections will coincide). As such, $C_1, C_2, \ldots C_d$ being subspaces is the **condition for equivalence** of the two methods (Dykstra's and alternating projections).

A re-written form of the algorithm is as follows: Initialize $u^{(0)} = y$, $z^{(-d+1)} = \cdots = z^{(0)} = 0$, then repeat for $k = 1, 2, \ldots$

$$\begin{aligned} & u_0^{(k)} = u_d^{(k-1)}, \\ & \left. \begin{aligned} u_i^{(k)} &= P_{C_i}(u_{i-1}^{(k)} + z_i^{(k-1)}), \\ z_i^{(k)} &= u_{i-1}^{(k)} + z_i^{(k-1)} - u_i^{(k)} \end{aligned} \right\} \quad \text{for } i = 1 \ldots d \end{aligned} \tag{11}$$

Where the $u_i^{(k)}$ vector represents z in (P2) and z is the dual/auxiliary variable.

4 Relationship between P1 and P2

4.1 Duality

For the general forms expressed in (P1) and (P2), they can be duals of each under conditions regarding the relationship between the sets $C_1, C_2, \dots C_d$ and penalty functions $h_1, h_2, \dots h_d$. Under such conditions, Coordinate Descent on (P1) and Dykstra's algorithm on (P2) are solving the same problem.

4.2 Equivalence Conditions

The conditions on which (P1) and (P2) are duals of each other are as follow:

1. That the functions h_i in (P1) each form a support function of a closed convex set $D_i \subseteq \mathbb{R}^{\text{'⊐}}$ i.e :

$$h_i(v) = \max_{d \in D_i} \langle d \; , \; v \rangle, \quad i = 1, \dots d \tag{12}$$

2. and that the sets $C_1, C_2, \dots C_d$ in (P2) are preimages of D_i under an affine map X_i^T for all $i = 1, \dots d$:

$$C_i = (X_i^T)^{-1}(D_i) = \{v \in \mathbb{R}^n : X_i^T v \in D_i\} \tag{13}$$

Under these 2 conditions and the earlier mentioned assumption that the associated coordinate block $X_i \in \mathbb{R}^{n \times p_i}$ has full rank, then for any $y \in \mathbb{R}^{\ltimes}$, we have the resulting Lenma:

$$\hat{\beta}_i = \operatorname*{arg\,min}_{\beta_i \in \mathbb{R}^{p_i}} \frac{1}{2} \|y - X_i \beta_i\|_2^2 + h_i(\beta_i) \iff X_i \hat{\beta}_i = (I_d - P_{C_i}) y \tag{14}$$

Where $\hat{\beta}$ is the minimizer of (P1), $I_d \in \mathbb{R}^{d \times d}$ is the identity map and all other variables have been defined.

Under this lenma, (P1) and (P2) are duals of each other and their solutions $\hat{\beta}$ and $\hat{z}$ respectively satisfy:

$$\hat{z} = y - X_i \hat{\beta} \tag{15}$$

4.3 Consequence of Equivalence

As a result of the previous equation, given matrix X and a vector y, we can run Coordinate Descent on an unconstrained optimization problem of the form (P1) to get $\hat{\beta}$ and Dykstra's algorithm on a constrained optimization of the form (P2) to get $\hat{z}$, and compare their results via (15).

In addition to comparing the final results to be equal, one can even validate the two algorithms to be equal across each iteration k and column index i. The below equations will be satisfied: for all k and i:

$$\hat{z}^{(k)} = X_i w_i^{(k)} \text{ and } u_i^{(k)} = y - \sum_{j \leq i} X_j w_j^{(k-1)} \tag{16}$$

This result will also be used to verify whether the algorithms were implemented correctly. As one can see, the consequences of duality leads to the two algorithms being deeply related.

5 Lasso

5.1 Lasso Intro and Description

A popular class of optimization problems in machine learning and statistics is the Lasso problem. Instead of simply minimizing the squared-error loss like in Ordinary Least Squares (OLS), the Lasso combines the least squares loss with a l_1 norm constraint. This effectively shrinks the effect of the weights.

One may consider Lasso for better prediction accuracy (by trading bias for better variance), as well as interpretation, so one can identify the subset of predictors exhibiting the strongest effects.

When the h_i term in (P1) takes the form $h_i = \lambda \|\beta_i\|$ and $d = p$, this problem is known as Lasso:

$$\min_{\beta \in \mathbb{R}^d} \frac{1}{2} \|y - X\beta\|_2^2 + \sum_{i=1}^{d} |\beta_i| \tag{17}$$

In other words, the second term is simply the l_1 norm where we sum all the absolute values of the elements in the vector β.

Lasso is also often used to recover sparse β^*, $|\beta^*|_0 = s$ from the measurements: $y = X\beta^* + w$.

5.2 Lasso Duality

Given Lasso (17) as the primal and rewritten as:

$$p^* = \min_{\beta} \frac{1}{2} \|y - X\beta\|_2^2 + \lambda \|\beta\|_1 \tag{18}$$

It's dual is given as follows:

$$\begin{aligned} &d^* = \min_{\beta} \frac{1}{2} \left(\|y - X\beta\|_2^2 - \|y\|_2^2 \right) \\ &\text{subject to} \\ &\|X^T \beta\|_\infty \leq \lambda \end{aligned} \tag{19}$$

An equivalent form of the dual which will be used in implementation is (Note equivalent but not identical):

$$d_e^* = \min_{u \in \mathbb{R}^n} \|y - u\|_2^2 \text{ subj to } \|X^T u\|_\infty \leq \lambda \tag{20}$$

5.3 Lasso Duality Derivation

In this subsection, the dual of the Lasso will be derived.

Although the primal (18) has no constraints, the residual vector $r = y - X\beta$ is introduced so that the primal can be rewritten as:

$$\min_{\beta} \frac{1}{2}\|r\|_2^2 + \lambda\|\beta\|_1 \text{ subj to } r = y - X\beta \tag{21}$$

This lets us write the Lagrangian as:

$$L(\beta, r, \theta)\frac{1}{2}\|r\|_2^2 + \lambda\|\beta\|_1 - \theta^T(r - y + X\beta) \tag{22}$$

Where θ is the Lagrange multiplier vector. The dual objective is hence to minimize the above expression with respect to β and r. Since they aren't coupled, we can minimize each separately. Minimizing β first:

$$\min_{\beta} -\theta^T X\beta + \lambda\|\beta\|_1 = \begin{cases} 0 & \text{if } \|X^T\theta\|_\infty \leq \lambda \\ -\infty & \text{otherwise} \end{cases} \tag{23}$$

Next, isolating terms with r, we get the following when $r = \theta$:

$$\min_{r} \frac{1}{2}\|r\|_2^2 + -\lambda^T r = \frac{1}{2}\theta^T\theta \tag{24}$$

Substituting (24) into the Lagrangian (22) and minimizing, we obtain:

$$\min_{r,\theta} L(\beta, r, \theta) = \min_{r,\theta} -\frac{1}{2}\theta^T\theta - \theta^T X\beta + \lambda\|\beta\|_1 + \theta^T y \tag{25}$$

Substituting (23) into the Lagrangian (25) and considering the cases, the dual is obtained as maximizing the infimum of the Lagrangian:

$$\text{Dual} := \max_{\theta} \frac{1}{2}\left(\|y\|_2^2 - \|y - \theta\|_2^2\right) \text{ subj to } \|X^T\theta\|_\infty \leq \lambda \tag{26}$$

Which is same as (19) if $\theta = \beta$ and minimizing the negative.

5.4 Interpretations and Relations of the Dual of Lasso

From equation (26) and (19), which was derived as the dual of (18), one can see that it is different to the form in (P2) and (20), but it is an equivalent problem in that the minimizer obtained will be the same.

In implementation, Dykstra's algorithm will be performed on (20) to recover the minimizer and the minimum value (cost) of (19) will be obtained via plugging in the solved minimizer.

More formally, it will be obtained as:

$$d^* = \frac{1}{2}(\|y\|_2^2 + d_e^*) \tag{27}$$

5.5 Lasso: Subgradients and Soft-Thresholding

Due to non-differentiability of the l_1 norm, we use the stationary point property and optimize coordinate-wise. However, the absolute value function is still non-differentiable.

Differentiability for a function at a point is defined as follows:

$$\lim_{x\to 0+} \frac{f(x+h)-f(x)}{h} = \lim_{x\to 0-} \frac{f(x+h)-f(x)}{h} \tag{28}$$

Meaning that the derivative of a function at point x should be equal approaching from the left and right.

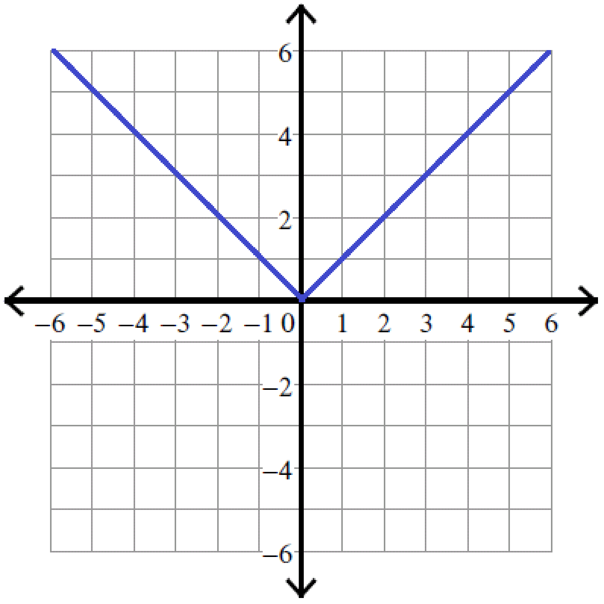

Figure 3: $|x|$

The function $|x|$ is not differentiable at $x = 0$, but we can define the subdifferetial as a set δ. For the function $|x|$, its subdifferetial at the origin would be the interval $[-1, 1]$.

For the second term in the Lasso problem, we can define a soft-threshold for its subgradient:

$$\delta_{\beta_i} \lambda \sum_{i=1}^{n} |\beta_i| = \delta_{\beta_i} \lambda \|\beta_i\|_1 = \begin{cases} -\lambda & \text{if } \beta_i < 0 \\ [-\lambda, \lambda] & \text{if } \beta_i = 0 \\ \lambda & \text{if } \beta_i > 0 \end{cases} \tag{29}$$

In the implementation of BCD for Lasso, by following algorithm (2) or taking the gradient of (18), one can see that the coordinate-wise minimum will satisfy:

$$X_i^T X_i \beta_i + X_i^T (X_{-i}\beta_{-i} - y) + \lambda s_i \tag{30}$$

Where $s_i \in \delta|\beta_i|$ and i represents the ith column or index and $-i$ represents everything but the ith column or index.

By shifting and grouping the terms, if we define $\rho_i = X_i^T(y - X_{-i}\beta_{-i})$ and $\beta_i z_j = \beta_i X_i^T X_i$, then we get the closed form solution for the optimal coordinate point as a soft-threshold:

$$\beta_i^* = \begin{cases} \dfrac{\rho_i + \lambda}{z_i} & \text{if } \rho_i < -\lambda \\ 0 & \text{if } -\lambda \geq \rho_i \leq \lambda \\ \dfrac{\rho_i - \lambda}{z_i} & \text{if } \rho_i > \lambda \end{cases} \tag{31}$$

5.6 Lasso: Projection onto Intersection of Halfspaces

In the implementation of Dykstra's projection algorithm for the dual of Lasso (20), the algorithm (11) projects coordinate-wise onto $2p$ half-spaces in p operations in one iteration k. These intersection of these halfspaces are defined by the constraint in the dual problem but individually can be defined as (13), and more precisely as:

$$C_i = (X_i^T)^{-1} D_i = \{v \in \mathbb{R}^n : |X_i^T v| \leq \lambda\} \tag{32}$$

Which is also an intersection of 2 half-spaces, resulting in $2p$ half-spaces for $C_1 \cap C_2 \dots C_p$.

For one half-space, the least squares projection problem:

$$\arg\min_x \|x - y\|_2^2 \text{ subject to } a^T x \leq b \tag{33}$$

by solving KKT is given as:

$$x = \begin{cases} y & \text{if } a^T y \leq b \\ y - \dfrac{a^T y - b}{\|a\|_2^2} a & \text{if } a^T y > b \end{cases} \tag{34}$$

Although one can simply double the amount of operations per iterations in Dykstra's algorithm to project onto each half-space via (34), the projection of 2 half-spaces can be reduced via symmetry to:

$$x = \begin{cases} y - \dfrac{-a^T y - b}{\|a\|_2^2} a & \text{if } -a^T y > b \\ y & \text{if } -b \leq a^T y \leq b \\ y - \dfrac{a^T y - b}{\|a\|_2^2} a & \text{if } a^T y > b \end{cases} \tag{35}$$

And hence the projection $P_{C_i}(x)$ is defined for the Lasso and Dykstra's algorithm can be implemented.

6 Experiments

6.1 Intro to Experiments

Different regimes: m, n, s, namely rows, columns and sparsity respectively, will be analyzed for performance. Specifically, the l_2 error: $\|\beta - \beta^*\|_2$, and the support recovery performance will be analyzed. The speed and convergence performance of Coordinate Descent and will also be qualitatively measured and compared to the CVX solver's performance for large problem sizes as well.

6.2 Results of General Experiments

Experiment 1

m	n	s	λ	obj val	l_2 Error	SRP	Runtime	Iters
20	20	4	1	3.972278	0.919909	0.950000	0.021088	94
20	40	4	1	3.934002	0.677962	0.950000	0.023293	104
20	60	4	1	40.792009	1.494900	0.950000	0.021091	95
40	20	4	1	6.149437	0.548588	1.000000	0.023863	104
40	40	4	1	14.027883	0.772388	0.950000	0.032725	143
40	60	4	1	29.482783	0.879961	0.950000	0.022945	99
60	20	4	1	6.398909	0.494304	0.950000	0.036178	154
60	40	4	1	8.996638	0.489277	0.925000	0.029135	123
60	60	4	1	9.793597	0.575788	0.966667	0.035213	150
20	20	18	10	9.398607	2.051634	1.000000	0.028038	126
20	40	18	10	84.250268	1.984428	0.825000	0.041910	190
20	60	18	10	185.524195	3.312044	0.800000	0.040102	182
40	20	18	10	11.814567	1.067842	0.850000	0.039004	173
40	40	18	10	134.112941	1.542076	0.775000	0.053944	239
40	60	18	10	433.660666	2.304971	0.766667	0.054202	240
60	20	18	10	14.030280	0.767754	0.850000	0.055461	239
60	40	18	10	133.291568	1.187213	0.825000	0.045203	194
60	60	18	10	293.796884	1.629398	0.800000	0.049856	214
20	20	4	10	5.053791	0.338553	1.000000	0.016531	74
20	40	4	10	4.639295	0.999818	0.950000	0.009809	44
20	60	4	10	14.791151	0.847609	0.966667	0.014482	64
40	20	4	10	5.224256	0.496669	1.000000	0.016168	71
40	40	4	10	41.515673	0.951166	0.925000	0.028009	123
40	60	4	10	16.789454	0.858016	0.966667	0.025999	114
60	20	4	10	6.748126	0.581032	1.000000	0.025794	111
60	40	4	10	37.414766	0.922727	0.950000	0.025834	111
60	60	4	10	47.102510	0.863366	0.933333	0.034203	147
20	20	18	1	9.041539	1.160448	0.950000	0.050447	227
20	40	18	1	78.739221	2.878436	0.775000	0.048224	220
20	60	18	1	204.394010	3.035103	0.833333	0.042366	191
40	20	18	1	13.965787	0.772522	0.750000	0.034946	154
40	40	18	1	256.961380	2.283832	0.725000	0.057403	253
40	60	18	1	464.782856	2.521892	0.783333	0.047905	211
60	20	18	1	12.664225	0.568755	0.700000	0.076631	332
60	40	18	1	590.283736	2.105875	0.675000	0.066927	290
60	60	18	1	370.609827	1.654103	0.783333	0.073764	319

Table 1: Experiment 1

Experiment 2

m	n	s	λ	obj val	l_2 Error	SRP	Runtime	Iters
30	20	20	0.2	13.905019	1.546178	0.850000	0.033313	145
30	50	20	0.2	288.652021	2.730460	0.780000	0.039084	171
30	90	20	0.2	333.532472	2.770830	0.844444	0.031293	138
100	20	20	0.2	16.954688	0.540366	0.800000	0.055706	222
100	50	20	0.2	565.027750	1.612935	0.780000	0.087562	348
100	90	20	0.2	1628.055558	2.155520	0.800000	0.096959	386
60	20	20	0.2	15.068917	0.755052	0.800000	0.052380	225
200	50	20	0.2	1944.136299	1.903586	0.740000	0.138499	366
200	90	20	0.2	2691.578617	2.087058	0.844444	0.153308	407
30	20	10	50.0	7.197901	0.790855	0.800000	0.075162	196
30	50	10	50.0	121.269780	1.686624	0.860000	0.033936	122
30	90	10	50.0	126.111065	1.845078	0.900000	0.034585	136
100	20	10	50.0	14.922546	0.498327	0.900000	0.044683	162
100	50	10	50.0	349.295037	1.374499	0.840000	0.076556	283
100	90	10	50.0	375.277702	1.515757	0.911111	0.075555	291
60	20	10	50.0	8.379622	0.524019	0.900000	0.057078	242
200	50	10	50.0	773.207190	1.206699	0.880000	0.305333	331
200	90	10	50.0	894.764806	1.434773	0.911111	0.138730	341
30	20	20	50.0	12.916701	0.979004	0.850000	0.099179	352
30	50	20	50.0	431.474525	3.222204	0.740000	0.051255	197
30	90	20	50.0	433.378229	3.040022	0.811111	0.058133	203
100	20	20	50.0	18.766702	0.472843	0.600000	0.065412	251
100	50	20	50.0	2044.938733	2.508892	0.640000	0.099084	395
100	90	20	50.0	2160.082474	2.694521	0.800000	0.104245	417
60	20	20	50.0	15.811072	0.674070	0.750000	0.080446	345
200	50	20	50.0	2892.658291	2.091619	0.720000	0.146393	373
200	90	20	50.0	4601.602654	2.628287	0.822222	0.106958	332
30	20	10	0.2	8.352275	1.360410	0.900000	0.065813	178
30	50	10	0.2	62.422373	1.868473	0.900000	0.063014	231
30	90	10	0.2	145.204879	1.743754	0.900000	0.046106	190
100	20	10	0.2	13.584021	0.451194	0.850000	0.051828	159
100	50	10	0.2	105.156583	0.791878	0.920000	0.078763	268
100	90	10	0.2	225.108271	1.423551	0.911111	0.062545	250
60	20	10	0.2	10.007174	0.587249	0.900000	0.033968	145
200	50	10	0.2	272.739899	0.897175	0.860000	0.157884	337
200	90	10	0.2	1204.146314	1.444142	0.900000	0.194643	332

Table 2: Experiment 2

Experiment 3

m	n	s	λ	obj val	l_2 Error	SRP	Runtime	Iters
20	30	30	0.01	209.030195	2.687627	0.633333	0.147639	677
50	30	30	0.01	439.995869	2.568554	0.633333	0.085019	373
90	30	30	0.01	444.718196	1.762106	0.666667	0.089934	367
20	100	30	0.01	559.698551	6.088753	0.800000	0.048251	217
50	100	30	0.01	2116.557799	4.101392	0.740000	0.089458	390
90	100	30	0.01	2964.953885	3.491047	0.770000	0.069201	283
20	60	30	0.01	405.886315	3.946323	0.700000	0.075741	348
50	200	30	0.01	2323.901595	4.562216	0.865000	0.086330	308
90	200	30	0.01	3778.843964	3.378005	0.865000	0.095217	335
20	30	1	300.00	3.610885	0.402683	0.966667	0.010409	47
50	30	1	300.00	9.192000	0.684733	0.966667	0.038862	134
90	30	1	300.00	7.760227	0.360991	0.966667	0.033052	107
20	100	1	300.00	2.413092	0.646000	1.000000	0.018608	69
50	100	1	300.00	8.013228	0.544763	0.990000	0.034628	109
90	100	1	300.00	9.261357	0.387510	1.000000	0.026274	97
20	60	1	300.00	2.408152	0.425235	0.983333	0.027523	71
50	200	1	300.00	11.582139	0.653694	0.995000	0.028482	123
90	200	1	300.00	16.282556	0.531917	0.995000	0.048705	165
20	30	30	300.00	158.693076	2.811466	0.666667	0.059741	239
50	30	30	300.00	272.030960	1.811307	0.500000	0.066520	247
90	30	30	300.00	587.719867	1.660267	0.633333	0.113957	384
20	100	30	300.00	590.657382	4.610293	0.780000	0.068435	289
50	100	30	300.00	1988.744896	3.826379	0.740000	0.065398	287
90	100	30	300.00	3506.545126	3.429012	0.770000	0.109042	447
20	60	30	300.00	392.151414	4.029625	0.700000	0.080364	369
50	200	30	300.00	2337.296014	4.327491	0.860000	0.094136	270
90	200	30	300.00	6409.041395	4.658729	0.860000	0.129975	345
20	30	1	0.01	2.028124	0.551302	0.966667	0.016030	70
50	30	1	0.01	5.138893	0.336443	1.000000	0.023734	104
90	30	1	0.01	8.424413	0.343646	1.000000	0.030557	125
20	100	1	0.01	2.163777	0.548247	1.000000	0.011833	54
50	100	1	0.01	5.042878	0.504571	1.000000	0.012938	48
90	100	1	0.01	8.390910	0.369081	0.990000	0.042498	123
20	60	1	0.01	2.654515	0.394600	0.983333	0.019086	46
50	200	1	0.01	16.243974	0.602848	0.995000	0.039646	116
90	200	1	0.01	11.991961	0.403353	0.995000	0.060227	177

Table 3: Experiment 3

Experiment 4 (Initializing with zero)

m	n	s	λ	obj val	l_2 Error	SRP	Runtime	Iters
20	30	8	0.01	4.221218	0.832382	0.966667	0.029382	97
50	30	8	0.01	8.659950	0.674951	0.933333	0.070776	229
90	30	8	0.01	10.917428	0.490238	0.933333	0.092215	275
20	100	8	0.01	4.587632	1.100475	0.990000	0.603406	346
50	100	8	0.01	6.368652	0.730698	0.990000	0.923944	498
90	100	8	0.01	8.698268	0.530543	0.980000	1.673041	764
20	200	8	0.01	4.021029	0.723463	1.000000	1.783389	292
50	200	8	0.01	5.340347	0.800780	1.000000	14.497770	697
90	200	8	0.01	8.017258	0.640923	0.995000	19.529745	842
20	30	3	300.00	4.147118	0.885381	1.000000	0.062292	157
50	30	3	300.00	5.256502	0.487099	1.000000	0.072237	146
90	30	3	300.00	8.197883	0.365573	1.000000	0.064059	190
20	100	3	300.00	3.055928	0.701220	0.990000	0.702476	399
50	100	3	300.00	3.548507	0.487438	1.000000	0.370189	190
90	100	3	300.00	7.523687	0.487348	0.980000	1.169090	499
20	200	3	300.00	2.469018	1.242246	1.000000	1.055736	172
50	200	3	300.00	5.103518	0.325230	0.995000	4.967294	227
90	200	3	300.00	7.084490	0.508057	1.000000	14.854310	602
20	30	8	300.00	5.898084	1.315373	0.966667	0.058507	179
50	30	8	300.00	6.633682	0.718992	0.933333	0.061081	195
90	30	8	300.00	10.075018	0.364350	0.933333	0.090604	272
20	100	8	300.00	3.817545	1.297089	1.000000	0.794421	452
50	100	8	300.00	7.403264	0.845495	0.990000	1.021607	532
90	100	8	300.00	9.962595	0.646842	0.990000	2.050809	919
20	200	8	300.00	5.269186	2.317282	1.000000	6.285018	1024
50	200	8	300.00	5.456317	0.911367	0.990000	12.254736	686
90	200	8	300.00	7.724564	0.640457	0.995000	20.420292	843
20	30	3	0.01	3.942870	0.660821	1.000000	0.035820	86
50	30	3	0.01	5.430901	0.610462	0.966667	0.081374	191
90	30	3	0.01	7.792180	0.531047	0.966667	0.043437	128
20	100	3	0.01	3.086188	1.481536	1.000000	0.270687	154
50	100	3	0.01	4.476950	0.596213	1.000000	0.454810	236
90	100	3	0.01	7.983560	0.494891	0.990000	1.267187	554
20	200	3	0.01	3.116259	1.541826	1.000000	2.228319	368
50	200	3	0.01	4.590401	0.504387	1.000000	7.063599	302
90	200	3	0.01	6.006824	0.500134	1.000000	15.700361	604

Table 4: Experiment 4 (Initializing with zero)

6.3 Results of Speed Experiments (CVX vs BCD vs Dykstra's)

Experiment 5 (Speed & Val)

m	n	s	λ	CVX val	CVX time	BCD val	BCD time	Dyk val	Dyk time
20	30	8	0.01	4.876049	0.004796	4.957497	0.041657	4.957497	0.046814
50	30	8	0.01	7.435948	0.006546	7.480736	0.065831	7.480736	0.100930
90	30	8	0.01	10.440851	0.007764	10.551148	0.120242	10.551148	0.191051
20	100	8	0.01	4.949804	0.006524	5.043532	1.209892	5.043532	0.377503
50	100	8	0.01	7.349174	0.011399	7.537898	0.719068	7.537898	2.254766
90	100	8	0.01	8.823168	0.021327	8.951439	1.576220	8.951439	4.225935
20	200	8	0.01	3.610647	0.009944	3.681416	3.190844	3.681416	0.569877
50	200	8	0.01	5.791027	0.017975	5.827004	9.580513	5.827004	3.151657
90	200	8	0.01	8.836155	0.040467	9.307302	11.714386	9.307302	6.293862
20	30	3	300.00	2.427903	0.004615	2.496149	0.053076	2.496149	0.060113
50	30	3	300.00	4.828962	0.013997	4.853728	0.058341	4.853728	0.098457
90	30	3	300.00	7.329149	0.008231	7.377902	0.063956	7.377902	0.101939
20	100	3	300.00	3.308528	0.007732	3.395241	0.404944	3.395241	0.144694
50	100	3	300.00	5.227856	0.010918	5.574757	0.461591	5.574757	1.011487
90	100	3	300.00	6.488225	0.021953	6.931340	0.817659	6.931340	1.475792
20	200	3	300.00	2.436815	0.010220	2.514544	2.081863	2.514544	0.288887
50	200	3	300.00	4.105282	0.021318	4.489477	4.664916	4.489477	0.736798
90	200	3	300.00	5.696998	0.040098	6.292656	9.094575	6.292656	4.876706
20	30	8	300.00	5.580971	0.008198	5.645138	0.042720	5.645138	0.053163
50	30	8	300.00	6.886182	0.006363	6.922718	0.071510	6.922718	0.097491
90	30	8	300.00	10.251838	0.007715	10.266735	0.137694	10.266735	0.226401
20	100	8	300.00	4.829324	0.008389	4.894104	0.879207	4.894104	0.238255
50	100	8	300.00	5.136297	0.013533	5.175212	1.378164	5.175212	2.507526
90	100	8	300.00	9.360214	0.023184	9.640360	1.735298	9.640360	4.682671
20	200	8	300.00	4.357964	0.011801	4.446421	2.369779	4.446421	0.478173
50	200	8	300.00	5.564783	0.017926	5.623969	8.256975	5.623969	3.095472
90	200	8	300.00	9.111001	0.082499	9.750816	20.087704	9.750816	10.451279
20	30	3	0.01	3.622890	0.008743	3.691469	0.061909	3.691469	0.094715
50	30	3	0.01	4.478632	0.006914	4.507356	0.062741	4.507356	0.085175
90	30	3	0.01	8.979457	0.009706	9.149849	0.060294	9.149849	0.093531
20	100	3	0.01	2.234934	0.008491	2.320701	0.241449	2.320701	0.084571
50	100	3	0.01	3.803970	0.012527	3.875944	0.356997	3.875944	1.313017
90	100	3	0.01	7.093156	0.021542	7.428734	0.801973	7.428734	1.392791
20	200	3	0.01	3.065387	0.010652	3.121146	1.316476	3.121146	0.210749
50	200	3	0.01	4.179682	0.022813	4.271213	4.488510	4.271213	1.013013
90	200	3	0.01	5.055395	0.043319	5.518773	14.208514	5.518773	7.047472

Table 5: Experiment 5 (Speed & Val)

Experiment 6 (Speed & Val)

m	n	s	λ	CVX val	CVX time	BCD val	BCD time	Dyk val	Dyk time
60	10	8	0.01	9.988694	0.004669	10.006907	0.007915	10.006907	0.015504
120	20	8	0.01	10.737367	0.007562	10.850918	0.051898	10.850918	0.096714
180	30	8	0.01	19.679133	0.011300	19.732903	0.152434	19.732903	0.247699
60	10	8	0.01	9.019796	0.004521	9.038458	0.006981	9.038458	0.012985
120	20	8	0.01	14.384284	0.007716	14.453050	0.039139	14.453050	0.071619
180	30	8	0.01	19.831501	0.010543	19.867077	0.167153	19.867077	0.279704
60	10	8	0.01	8.064587	0.004983	8.085213	0.007339	8.085213	0.014885
120	20	8	0.01	14.659065	0.006738	14.782062	0.058945	14.782062	0.109707
180	30	8	0.01	17.997177	0.010139	18.045484	0.181623	18.045484	0.300567
60	10	3	300.00	6.055692	0.005103	6.074416	0.007659	6.074416	0.015373
120	20	3	300.00	11.098472	0.007418	11.144826	0.036061	11.144826	0.062905
180	30	3	300.00	14.816367	0.012498	14.931732	0.117033	14.931732	0.190133
60	10	3	300.00	5.236196	0.004805	5.251821	0.005648	5.251821	0.010554
120	20	3	300.00	12.890935	0.007217	12.911277	0.032604	12.911277	0.056808
180	30	3	300.00	16.725454	0.012116	16.848380	0.096932	16.848380	0.158783
60	10	3	300.00	8.052576	0.004575	8.072846	0.006555	8.072846	0.012790
120	20	3	300.00	12.611987	0.007106	12.621002	0.033909	12.621002	0.057776
180	30	3	300.00	14.737180	0.012200	14.779286	0.110353	14.779286	0.183170
60	10	8	300.00	11.998313	0.004463	12.016796	0.008964	12.016796	0.017337
120	20	8	300.00	13.789236	0.007152	13.798289	0.075341	13.798289	0.133296
180	30	8	300.00	19.789829	0.010771	19.823529	0.130941	19.823529	0.215851
60	10	8	300.00	11.024570	0.004474	11.043864	0.006022	11.043864	0.011703
120	20	8	300.00	13.181581	0.007175	13.190430	0.057505	13.190430	0.099972
180	30	8	300.00	17.689367	0.011765	17.763889	0.179881	17.763889	0.301976
60	10	8	300.00	10.543644	0.004638	10.564304	0.007151	10.564304	0.013737
120	20	8	300.00	14.508077	0.007009	14.517388	0.054630	14.517388	0.100985
180	30	8	300.00	18.174153	0.011211	18.297554	0.169073	18.297554	0.294689
60	10	3	0.01	6.614231	0.004907	6.642891	0.004914	6.642891	0.009280
120	20	3	0.01	12.584535	0.007735	12.637978	0.039268	12.637978	0.067956
180	30	3	0.01	15.452043	0.011637	15.591903	0.127036	15.591903	0.208737
60	10	3	0.01	7.076365	0.004994	7.094739	0.007267	7.094739	0.013855
120	20	3	0.01	11.655638	0.007577	11.664785	0.037954	11.664785	0.068375
180	30	3	0.01	15.109919	0.011164	15.266814	0.148970	15.266814	0.242119
60	10	3	0.01	6.130565	0.004692	6.147824	0.005743	6.147824	0.010940
120	20	3	0.01	12.895651	0.007120	12.904984	0.027275	12.904984	0.046889
180	30	3	0.01	15.835604	0.011461	15.896271	0.088241	15.896271	0.144439

Table 6: Experiment 6 (Speed & Val)

6.4 Analysis of general experiments (m,n,sparsity...)

For all the experiments conducted, m, n, s, λ represent the rows of X, columns of X, sparsity of β and scaling parameter λ. All time measurements are also expressed in seconds.

For the first 4 experiments, (obj val) represents the objective function's value, after plugging in the computed β from BCD, and the l_2 error represents the error:

$$\|\beta - \beta^*\|_2 \tag{36}$$

Where β is the true beta generated and β^* is the computed beta. SRP is the support recovery performance and is given as a ratio out of 1, where 1 indicates 100%. SRP is calculated as follows:

$$SRP = \frac{N - \#\text{ recovery success}}{N} \tag{37}$$

Where N is the number of columns in X and number of recovery success is determined via counting how many entries in β and β^* have difference of less than some tolerance ratio $\epsilon * min(\beta^*, \beta)$.

One can see from table 4 that in general, the l2 error is quite arbitrary due to the randomness of the experiments. The sparsity recovery performance in general is also very high, especially in experiment 4 where β is initialized as zero as it should be.

One small mistake in experiments 1 to 3 is that the β value was not initialized as zero and this affected the SRP rate by a lot. Nonetheless the recovery rates in general are very high even for max sparsity as seen in experiment 3. In general across all experiments, SRP is never below 50% and is only occurred once in experiment 3.

6.5 Analysis of speed and value experiments

For the speed and value experiments, namely experiment 5 and 6, columns CVX val, BCD val and Dyk val represent the CVX solver's objective function value, coordinate descent's objective function value and Dykstra's objective function value respectively. As one can expect, Dykstra's and BCD will obtain the same value.

One can also see that the values obtained by CVX are always less than from coordinate descent or Dykstra's, so CVX is a more precise/accurate minimizer.

In terms of speed, the CVX solver is also consistently much faster than BCD or Dykstra's and BCD is consistently faster than Dykstra's, though this could be due to unnecessary pseudoinverse calculation in Dykstra's for comparison.

One can also see from the speed experiments that in general, the runtimes take longer when m and n are higher with n seeming to have more of an effect than m. It is also noted from table 5, that sparsity seems to have an effect on the runtime as well as $m = 90, n = 200, s = 3$ took 6.29 seconds and $m = 90, n = 200, s = 8$ took 9.75 seconds for BCD.

7 Verifying Dykstra's and Coordinate Descent on Lasso

7.1 Equivalence in testing

Through testing and the experiments, the equivalence between Dykstra's algorithm and coordinate descent has been verified.

One can clone the code from github and run the Python script and see that through the Lasso class' $together(self, verbose = True)$ function, that at each iteration k and each index i, the relationship formulated in the earlier sections indicate that Dykstra's algorithm and coordinate descent are indeed equal for the primal and dual respectively.

8 Matrix Completion

8.1 Another problem in literature

A famous problem in literature is the Matrix Completion problem, in which one aims to learn a model from past incomplete data[3]. Common applications of this problem include recommender systems, or completing a partially filled survey.

The objective of matrix completion is to approximate training data via a low rank matrix. Given a matrix of incomplete data $A \in \mathbb{R}^{m\times n}$, and Ω be the set of indices of where data are available, the low rank matrix factorization problem is to find $W \in \mathbb{R}^{m\times k}$ and $H \in \mathbb{R}^{n\times k}$ such that $A \approx WH^H$ [2]. It can be formulated as an optimization problem:

$$\min_{W,H} \sum_{i,j\in\Omega} A_{ij} - w_i h_j^T + \frac{\lambda}{2}(\|W\|_F^2 + \|H\|_F^2) \tag{38}$$

Where w_i and is the i-th row vector of W, h_j is the j-th row vector of H, and $\|\cdot\|_F$ is the Frobenius matrix norm:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{i=1}^{n} |a_{ij}|^2} \tag{39}$$

The problem can be rewritten as:

$$\min_{W,H} \|A - WH\|_F^2 + \frac{\lambda}{2}(\|W\|_F^2 + \|H\|_F^2) \tag{40}$$

The coordinate descent algorithm applied at iteration k, where we fix one variable at the i-th index: w_{ik}, the optimal value w_{ik}^* is obtained as:

$$w_{ik}^* = \frac{\sum_{j\in\Omega_i}(A_{ij} - w_i h_j^T + w_{ik}h_{jk})h_{jk}}{\lambda + \sum_{j\in\Omega} h_{jk}^2} \tag{41}$$

If we notice the following identity[3]:

$$\|Z\|_* = \min_{Z=WH} \frac{1}{2}(\|W\|_F^2 + \|H\|_F^2) \tag{42}$$

, Where $\|\cdot\|$ is the nuclear norm, i.e sum of singular values $\sum \sigma_i$.

Then the rank constrained problem is given as follows:

$$\min_{rank(Z)\leq r} \|A - Z\|_F^2 + \lambda\|Z\|_* \tag{43}$$

Where $r \leq \min(m, n)$.

One can also acquire the relaxed convex program by dropping the rank constraint to acquire:

$$\min_{Z} \|A - Z\|_F^2 + \lambda\|Z\|_* \tag{44}$$

To determine the low rank approximation of A.

Though this form closely resembles P1, the dual of this was too difficult to derive, and thus only Coordinate Descent is shown, and Dykstra's algorithm is not.

9 Ridge Regression

9.1 Another problem in literature

Similar to Lasso, coordinate descent and Dysktra's projection algorithm can be used on Ridge Regression.

The Ridge Regression problem is as follows:

$$\beta^{ridge} = \arg\min_{\beta} \sum_{i=1}^{N}(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \tag{45}$$

Where the coefficents β are shrunk by penalizing the residual sum of squares with a Lagrange multiplier λ. It can be written in matrix notation as the Lagrangian optimization:

$$L(X, t, \lambda^*, \beta) = \|y - X\beta\|_2^2 + \lambda^*(\|\beta\|_2^2 - t) \tag{46}$$

Where via solving the Karush-Kuhn-Tucker (KKT) conditions, the closed form solution is given as:

$$\beta^{ridge} = (X^T X - \lambda I)^{-1} X^T y \tag{47}$$

Where as Lasso uses the l_1 norm, Ridge uses the l_2 norm.

Similar to the Lasso, the second term (l_2 norm) is non-differentiable at 0. This can be proved via contradiction:

Suppose it were differentiable, then by the chain rule, if $v \in \mathbb{R}^d$ is a unit vector, then $f(t) = \|tv\|_2 = |t|$ would be a differentiable function, yielding a contradiction.

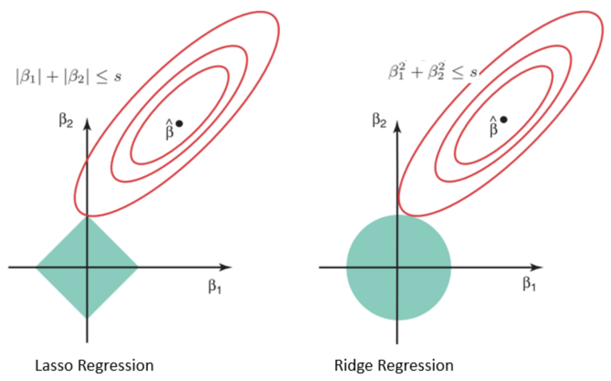

Figure 4: Ridge vs Lasso

Despite this, Ridge has a closed form solution unlike Lasso. Technically one could run coordinate descent and Dykstra's algorithm, but it wouldn't make sense for Ridge since it has a closed form solution.

10 Summary and Conclusions

In conducting research and experiments for this project, I've learnt a lot about optimization and math as a whole. Developing the Lasso Solver Program and exploring the derivations for the proofs was also very exciting. The fully working Lasso solver (via cylic coordinate descent and Dykstra's algorithm) code is uploaded to GitHub:

github.com/Johnkhk/Lasso-Solver-Coordinate-Descent-Dykstra-s-Projection-Algorithm-

Particularly, via exploring the equivalence dual relation between (P1) and (P2) as well as Coordinate Descent and Dykstra's projection algorithm, I've learnt a lot about rigorous proving of duality arguments via transforming constraints into Lagrangian variables.

Through implementation and experiments, I've learnt a lot about support recovery and sparsity in the context of design experiments.

Via writing this report, I've also learnt a lot about LaTeX

References

[1] Trevor Hastie, Robert Tibshirani, and Martin Wainwright. 2015. Statistical learning with sparsity. *Monographs on statistics and applied probability* 143 (2015), 143.

[2] Dongha Lee, Jinoh Oh, and Hwanjo Yu. 2020. OCAM: Out-of-core coor-

dinate descent algorithm for matrix completion. *Information Sciences* 514 (2020), 587–604.

[3] Spencer Sheen. [n.d.]. A Coordinate Descent Method for Robust Matrix Factorization and Applications. ([n. d.]).

[4] Ryan J Tibshirani. 2017. Dykstra's algorithm, ADMM, and coordinate descent: Connections, insights, and extensions. *Advances in Neural Information Processing Systems* 30 (2017).

[5] Paul Tseng. 2001. Convergence of a block coordinate descent method for nondifferentiable minimization. *Journal of optimization theory and applications* 109, 3 (2001), 475–494.

11 Appendix Code

```python
from ast import arg
import numpy as np
import cvxpy as cp
import copy
import random
import time
import pandas as pd


class lasso_solver:
    def __init__(self, lambdu=1):
        self.lambdu = lambdu
        self.maxiters = 100
        self.termination_cond = 0.0001


    def generate_X_y_b(self, m=20,n=20):
        # X = np.random.rand(m,n).astype(np.longdouble)
        # y = np.random.rand(m).astype(np.longdouble)
        # b = np.random.rand(n).astype(np.longdouble)
        X = np.random.rand(m,n)
        y = np.random.rand(m)
        b = np.random.rand(n)
        self.m, self.n = X.shape
        # self.n = X.size

        self.select = list(range(n))
        return X,y,b

    def soft_thresh(self, rouj,zj):
        if rouj < -self.lambdu:
            return (rouj+self.lambdu)/zj

        elif rouj > self.lambdu:
            return (rouj-self.lambdu)/zj

        else:
            return 0

    def min_betaj(self,X,y,b,j):
        selector = [i for i in range(X.shape[1]) if i != j]
        rouj = X[:,j].T @ (y-X[:,selector] @ b[selector])
        zj = np.linalg.norm(X[:,j])**2
        thresh = self.soft_thresh(rouj,zj)
        return thresh

    def coord_desc(self,X,y,beta,b=False):
        """
        Cyclic Coordinate Descent on Lasso
        """
        if not b:
            beta = np.zeros(self.n)
        it=0
        while it < self.maxiters:
            for j in range(self.n):
                min_b = self.min_betaj(X,y,beta,j)
                beta[j]=min_b
            it+=1
        return beta

    def obj_func(self,y,X,b): # 2
        return np.linalg.norm(y-X@b)**2 + self.lambdu*np.linalg.norm(b,
ord=1)

    def cvx_min(self, X,y,b):
        # b = cp.Variable(self.n)
        b = cp.Variable(len(b))
        objective = cp.Minimize(cp.norm2(y - X @ b)**2 +
self.lambdu*cp.norm1(b)) #+ np.max(np.abs(b))
        prob = cp.Problem(objective)
        sol = prob.solve()
        # print("cvx beta: ",b.value)
        return sol

    def Pci(self,uz,Xcol):
        b = self.lambdu
        if Xcol.T @ uz > b:
            return uz - ((Xcol.T@uz - b)/np.linalg.norm(Xcol)**2)*Xcol
        elif - Xcol.T @ uz > b:
            Xcol=-Xcol
            return uz - ((Xcol.T@uz - b)/np.linalg.norm(Xcol)**2)*Xcol
        else:
            return uz

    def dykstras(self,X,y,b):
        """
        Dykstra's Projection Algorithm on Dual of Lasso
        """
        it = 0
        d = self.n
        u = [0]*(d+1)
        u[-1] = copy.deepcopy(y)
        z = [np.zeros(self.m) for i in range(self.n)]
        while it < self.maxiters:
            u[0] = u[-1].copy()
            for i in range(1,d+1):
                u[i] = copy.deepcopy(self.Pci(u[i-1].copy() + z[i-
1].copy(), X[:,i-1].copy()))
                z[i-1] = u[i-1].copy() + z[i-1] - u[i].copy()

            it+=1
        return u[-1]

    def together(self,X,y,b, verbose=True):
        """
        Coordinate Descent & Dykstra's Algorithm
        Together, same iterations for comparison
        """
        ### dykstra init ###
        it = 0
        d = self.n
        u = [0]*(d+1)
        u[-1] = copy.deepcopy(y)
        z = [np.zeros(self.m) for i in range(self.m)]

        ### coord desc init ###
        beta = np.zeros(self.n).astype(np.longdouble)
        it=0

        ### Main Loop ###
        # while it < self.maxiters:
        while True:
            betaprev = beta.copy()
            u[0] = u[-1].copy()
            for i in range(1,d+1):

                ### coord_desc ###
                min_b = self.min_betaj(X,y,beta,i-1)
                beta[i-1]=min_b

                ### dykstra ###
                u[i] = copy.deepcopy(self.Pci(u[i-1].copy() + z[i-1].copy(), X[:,i-1].copy()))
                z[i-1] = u[i-1].copy() + z[i-1] - u[i].copy()

                ### equal check ###
                if verbose:
                    print("###"*10," \n")
                    print(z[i-1])
                    print("\n")
                    print(X[:,i-1]*beta[i-1])
                    print("%%%"*10," \n")

            ### termination condition ###
            if np.linalg.norm(beta-betaprev)<self.termination_cond:
                print("iterations: ",it, "dist: ", np.linalg.norm(beta-betaprev))
                break

            it+=1
    def coord_desc_2(self,X,y,betaf,b=False):
        """
        Cyclic Coordinate Descent on Lasso
        Has termination condition rather than termination iteration
        """
        beta = np.zeros(betaf.shape)

        if not b:
            beta = np.zeros(self.n)
        it=0
        # while it < self.maxiters:
        while True:
            betaprev = beta.copy()
            for j in range(len(beta)):
                min_b = self.min_betaj(X,y,beta,j)
                beta[j]=min_b
            ## terminal condition ##
            if np.linalg.norm(beta-betaprev)<self.termination_cond:
                print("iterations: ",it, "dist: ", np.linalg.norm(beta-betaprev))
                break
            it+=1
        return beta, it
    def dykstras2(self,X,y,b):
        """
        Dykstra's Projection Algorithm on Dual of
        Has termination condition rather than termination iteration
        """
        it = 0
    # d = self.n
    d = X.shape[1]
    u = [0]*(d+1)
    u[-1] = copy.deepcopy(y)
    z = [np.zeros(X.shape[0]) for i in range(X.shape[1])]
    # compare = y-X@cbeta
    while True:
      u[0] = u[-1].copy()
      betaprev = np.linalg.pinv(X) @ (y-u[0].copy())

      for i in range(1,d+1):
        u[i] = copy.deepcopy(self.Pci(u[i-1].copy() + z[i-1].copy(), X[:,i-1].copy()))
        z[i-1] = u[i-1].copy() + z[i-1] - u[i].copy()

      beta = np.linalg.pinv(X) @ (y-u[-1].copy())
      if np.linalg.norm(beta-betaprev)<self.termination_cond:
        print("iterations: ",it, "dist: ", np.linalg.norm(beta-betaprev))
        break
      it+=1
    return u[-1]

  def gen_exp(self,m,n,s,l):
    X = np.random.rand(m,n) # random X
    idxs = random.sample(list(range(n)),n-s)
    b = np.random.rand(n)
    b[idxs] = 0 # sparse beta
    w = np.random.rand(m) # random noise
    y = X@b + w
    return X,y,b

  def performance(self,b,beta,y,X):
    """
    Compute Performance
      b: true beta
      beta: calculated beta
    """
    l2error = np.linalg.norm(beta-b)
    srp_tolerance = 0.1
    rec_suc, N = 0, len(beta)
    for j in range(N):
      if abs(b[j]-beta[j])<srp_tolerance*min(b[j],beta[j]):
        rec_suc+=1
    srp = (N-rec_suc)/N
    obj_val = self.obj_func(y,X,beta)

    return l2error,srp,obj_val

  def experiment(self,m_rows,n_cols,sparse, lambd):
    L = len(m_rows)

    ### exp ###
    results,speed_results=[],[]
    s,lam=sparse,lambd
    for i in range(L):
      m,n = m_rows[i],n_cols[i]
      X,y,b = self.gen_exp(m,n,s,lam)

      # cord desc #
```

```
            t1=time.time()
            beta,it_c = self.coord_desc_2(X,y,b,True)
            runtime_c = time.time() - t1
            l2error,srp,objval = self.performance(b,beta,y,X)
            results.append([m,n,s,lam,objval,l2error,srp,runtime_c,it_c])

            t1=time.time()
            cvx_sol = ls.cvx_min(X,y,b)
            runtime_s = time.time() - t1

            t1=time.time()
            d_sol = ls.dykstras2(X,y,b)
            runtime_d = time.time() - t1


 speed_results.append([m,n,s,lam,cvx_sol,runtime_s,objval,runtime_c,objval,
runtime_d])

        return results, speed_results




if __name__=="__main__":
    ls = lasso_solver()
    X,y,b = ls.generate_X_y_b()

    ### coordinate descent ###
    cbeta=ls.coord_desc(X,y,b)
    csol=ls.obj_func(y,X,cbeta)
    compare = y-X@cbeta

    ### dykstra projection ###
    du = ls.dykstras(X,y,b)

    ### cvx solver ###
    cvxsol = ls.cvx_min(X,y,b)

    ### compare results ###
    print(compare,"\n\n",du,"\n\n") # compare uhat of Dykstra and
Coord_desc
    print("cvx: ",cvxsol,"\ncoord_desc & dykstra: ",csol) # compare
(coord_desc, Dykstra) and CVX_solver

    ### together ###
    ls.together(X,y,b,verbose=False) # change verbose to True to check
equal conditions

    ### Experiments ###
    # m_rows = [20,20,20,40,40,40,60,60,60]
    # n_cols = [20,40,60,20,40,60,20,40,60]
    # m_rows = [30,30,30,100,100,100,60,200,200]
    # n_cols = [20,50,90,20,50,90,20,50,90]
    # m_rows = [60,120,180,60,120,180,60,120,180]
    # n_cols = [10,20,30,10,20,30,10,20,30]
    m_rows = [20,50,90,20,50,90,20,50,90]
    n_cols = [30,30,30,100,100,100,200,200,200]

    res1,sres1 = ls.experiment(m_rows,n_cols,sparse=8, lambd=0.01)
    res2,sres2 = ls.experiment(m_rows,n_cols,sparse=3, lambd=300)
    res3,sres3 = ls.experiment(m_rows,n_cols,sparse=8, lambd=300)
    res4,sres4 = ls.experiment(m_rows,n_cols,sparse=3, lambd=0.01)
    resf = res1+res2+res3+res4
    sresf = sres1+sres2+sres3+sres4

    ### get TEX file (Speed) ###
    headers=["m","n","s","lambda","CVX val","CVX time", "BCD val","BCD time", "Dyk val","Dyk time"]
    dfs = pd.DataFrame(sresf, columns=headers)
    dfs.to_latex(buf="table_s.tex",escape=False,index=False)

    ### Get TEX file ###
    headers=["m","n","s","lambda","obj val","l2 Error","SRP","Runtime","Iters"]
    df1 = pd.DataFrame(res1, columns=headers)
    df1.to_latex(buf="table1.tex",escape=False,index=False)
    df2 = pd.DataFrame(res2, columns=headers)
    df2.to_latex(buf="table2.tex",escape=False,index=False)
    df3 = pd.DataFrame(res3, columns=headers)
    df3.to_latex(buf="table3.tex",escape=False,index=False)
    df4 = pd.DataFrame(res4, columns=headers)
    df4.to_latex(buf="table4.tex",escape=False,index=False)
    dff = pd.DataFrame(resf, columns=headers)
    dff.to_latex(buf="tablef.tex",escape=False,index=False)
```