

Research in Industrial Projects for Students



Sponsor
Aquatic

Final Report

Memory-Bound Elastic Net Over Dense Matrices with Applications in Quantitative Trading

Student Members

Ben Young (Project Manager), *Case Western Reserve University*,
bmy11@case.edu

Atanas Dinev, *Princeton University*
Jacob Feitelberg, *Johns Hopkins University*

Gerson C. Kroiz, *University of Maryland, Baltimore County*

Academic Mentor

Arash Vahabpour, vahabpour@ucla.edu

Sponsoring Mentors

Dan Shiber, d.shiber@aquatic.com

July 24, 2020

Abstract

Quantitative analysis of U.S equity trading frequently involves processing large datasets whose predictive features often have very weak correlations with their target variables. One common method for regression analysis in quantitative trading is elastic net, a regularized form of least squares estimation. This project aims to solve elastic net using limited amounts of computational power. To overcome memory limitations when processing arbitrarily large data, we implement matrix batching methods whereby correlations between features are calculated per batch and combined into an overall result. Similarly, we introduce *MiniCD*, a general framework to estimate coefficients across subsets of samples and combine them into a prediction for the whole input. We compare several methods of ensembling estimated batch coefficients. We also explore several strategies to speed up Coordinate Descent. We compare cyclic, random, greedy, and adaptive methods for choosing update axes. We explore computationally cheap ways to determine convergence and several applications of a warm start. We study the effect on runtime of reducing iterations to convergence via data preconditioning. Our improvements achieve significant speedups relative to and can process larger datasets than out-of-the-box elastic net solvers such as Python's Sci-Kit Learn.

Key Words: Elastic Net, Coordinate Descent, Stochastic Gradient Descent, Ensembling methods, Memory-Bound Optimization

Acknowledgments

We would first like to acknowledge the support from the Aquatic team, lead by our sponsor mentor Dr. Dan Shiber. Throughout the project, Dr. Shiber provided insightful comments on concepts to study and the datasets used in our project from Aquatic. We would also like to thank the following members from Aquatic who occasionally joined our weekly meetings with Dr. Shiber and provided insightful comments on our work: Marco Santoli and Stefano Mosso. We also wanted to acknowledge the immense support from our academic mentor, Arash Vahabpour from UCLA. Special thanks to Dr. Susana Serna, the IPAM RIPS Program Director and the remaining staff of the RIPS program at UCLA for making this opportunity possible despite the COVID-19 pandemic.

Contents

Abstract	3
Acknowledgments	5
List of Figures	11
List of Tables	13
1 Introduction	15
1.1 Problem and Motivation	15
1.2 Approach	15
1.3 Report Overview	16
2 Mathematical Background	17
2.1 General Optimization	17
2.2 Least Squares Regression	17
2.3 Lasso and General Penalized Least Squares Regression	18
2.4 Elastic Net	19
2.5 Coordinate Descent	19
2.6 Stochastic Gradient Descent	20
2.7 Determining Convergence through the Duality Gap	21
2.8 Data Properties and Generation	22
3 CD Optimization Methods without Accuracy Loss	25
3.1 Prior Related Work	25
3.2 Feature Choice Rules	27
3.3 Update Rules	30
3.4 Front Heavy Covariance Method	32
3.5 Numerical Linear Algebra	33
3.6 Thresholding Rules	33
3.7 Disk I/O	33
3.8 Preconditioning X	34
3.9 Warm Start	41
4 CD Optimization Methods with Accuracy Loss	43
4.1 Computer Precision	43
4.2 Gram Matrix Estimation	43
4.3 MiniCD	44
4.4 Ensemble Methods	46
4.5 Early Stopping	50

5 Results	53
5.1 Hardware and Data Sepcifications	53
5.2 Parameter Study for λ_1 and λ_2 values	53
5.3 MiniCD results	54
5.4 Ensembling methods results	58
5.5 Early Stopping Study	61
5.6 Study of Feature Choice Methods	61
5.7 Comparison of Naive and Covariance Update Rules	64
5.8 Front Heavy Covariance CD vs Standard Covariance CD	65
5.9 Thresholding	65
5.10 Loading Data Studies	66
5.11 Lower Precision Studies	66
5.12 Gram Matrix Estimation Study	67
5.13 Preconditioning Study	68
5.14 Warm Start Study	70
5.15 Study of Ensemble Methods	71
5.16 Comparison with Scikit-Learn Elastic Net	71
6 Conclusions	73
7 Future Work	75
A Abbreviations	77
Selected Bibliography Including Cited Works	79

List of Algorithms

1	Basic CD	20
2	Basic SGD	20
3	ACF probability update	29
4	Front Heavy Covariance Method	32
5	MiniCD	44
6	Threshold voting	49
7	Tree Voting Algorithm	51

List of Figures

2.1	Comparison of lasso and ridge penalties in one dimension	19
3.1	CD on correlated and uncorrelated features	35
3.2	Subgradients of $ x $ at 0	38
3.3	Subgradients of $ x - 1 + -2x + 1 $ at 1	39
4.1	Mean and ground truth magnitude of active features	46
4.2	6 highest-magnitude features in Large dataset	47
4.3	Percent active vs coefficient magnitude	50
4.4	Tree voting schematic	52
5.1	Effect of λ_1 and λ_2 on solution sparsity	55
5.2	Active features across days for Large dataset	56
5.3	Estimate vs MSE for different batch sizes	57
5.4	Estimate vs support accuracy for different batch sizes	57
5.5	Support accuracy across batch sizes for various c	57
5.6	Convergence study of feature selection methods	63
5.7	Thresholding value vs cross-validation loss.	65

List of Tables

5.1	Proportion of nonzero coefficients for various λ_1 and λ_2 in Medium dataset	53
5.2	Proportion of nonzero coefficients for various λ_1 and λ_2 in Large dataset	54
5.3	Number of active features for varying thresholds and batch sizes	58
5.4	Support precision for varying thresholds and batch sizes	59
5.5	Support recall for varying thresholds and batch sizes	59
5.6	MSE between predictions and target for varying thresholds and batch sizes	59
5.7	Relative distance from true weights for varying thresholds and batch sizes	60
5.8	$(w - w^*)^T \Sigma (w - s^*)$ for varying thresholds and batch sizes	60
5.9	Study of optimal hybrid configuration	62
5.10	Runtime comparison of covariance and naive update methods	64
5.11	Runtime comparison of front heavy and standard covariance methods	65
5.12	Runtime comparison of methods for loading and calculating Gram matrix	66
5.13	Performance study of numerical linear algebra with various levels of precision	67
5.14	Performance study of precision impact on CD	67
5.15	Runtime and accuracy effects of Gram matrix estimation	68
5.16	Preconditioning performance study	69
5.17	Performance study of warm start for rolling windows	70
5.18	Performance study of warm start for distributed network	70

Chapter 1

Introduction

Founded in 2018 by Jonathan Graham, Aquatic is a startup quantitative trading and investment company based in Chicago. Their small and modernized team of researchers and engineers is building a fully automated investment engine based on a proprietary prediction machine. While Aquatic is currently testing their machine on small stock trades, their prediction machine will start actively trading stocks within the fourth quarter (Q4) of 2020.

1.1 Problem and Motivation

This project, sponsored by Aquatic, aims to develop a forecasting model of equity trading of U.S. stocks based on state-of-the-art techniques such as coordinate descent (CD) and stochastic gradient descent (SGD). In a broad sense, quantitative trading refers to using statistical models to predict stock price fluctuations. A quantitative trading firm uses their models to make profits - if they predict a stock's price will rise they will buy it and if they predict a stock's price will fall they will sell it. In general financial data has a very low signal-to-noise ratio, so even the best models find only weak correlations. A model may guess correctly on the direction of a stock's price only 51% of the time, but if it makes a lot of trades this small advantage can add up. Thus models should be able to make predictions over very short time frames. Our project therefore prioritizes improving algorithms' runtime over improving their accuracy.

Our research also focuses on efficiently handling large datasets. Quantitative trading models aim to determine a linear fit between hundreds or thousands of possible factors and return on investment. Within the U.S., there are around 3,000 various equities. Keeping track of frequent changes to stock values, volumes, and other characteristics over long periods often leads to large data sets. To tackle this problem, many companies in this field, such as Aquatic, use high-end computer clusters to predict and execute trades. For our project, we address the problem with limited computing power to show that the problem can be solved feasibly with less computing resources. With our work, Aquatic will have the resources to replicate the designs we develop on their computing clusters with larger datasets to possibly improve their current prediction machine.

1.2 Approach

Our project aims to develop an optimized prediction model using lower amounts of computation power. We start with basic implementations of optimizations algorithms such as

CD and SGD with Elastic Net as our regressor. Using these, we delve into various methods of improving the algorithm within the scopes of computational performance and accuracy. We also address means of addressing the memory and disk-io bounds associated with the quantitative trading problem on the limited hardware.

1.3 Report Overview

The remainder of this report is structured as follows. Chapter 2 discusses the general mathematical background of our problem and the optimization algorithms utilized. In Chapters 3, we discuss various techniques without an influence on accuracy that we looked into and implemented with aims to optimize the basic algorithms explained in the previous chapter. Chapter 4 discusses further optimization techniques that induce some accuracy loss in the result of CD. Chapter 5 shows preliminary results of each of the methods we explored. Based on the results, Chapter 6 provides general conclusions followed by future work in Chapter 7.

Chapter 2

Mathematical Background

2.1 General Optimization

Mathematical optimization is the selection of the best element from a set of available alternatives. Optimization problems arise in all quantitative disciplines, from computer science, engineering to operations research and economics. The development of solution methods has been of interest to mathematicians for centuries. In the simplest case, an optimization problem consists of maximizing or minimizing a real function by systematically choosing input values from within an allowed set and computing the function's value. More generally, optimization can be interpreted as finding the “best available” value of some objective function given a defined domain (or input), including a variety of functions and different types of domains.

2.2 Least Squares Regression

To introduce the optimization problem this report addresses, consider a simple example. Say we want to predict the price of Apple’s stock (the *target*) and that we have identified $p = 3$ possible *predictors* (or *features*): iPhone sales, MacBook sales, and Samsung’s stock price. We have measured each feature value and Apple’s stock price at each of N points in time. Thus we have an $N \times p$ matrix X storing the p feature values at each of N times and a N -length vector y storing Apple’s stock prices at each time.

We assume that Apple’s stock price (y) can be modeled as a linear combination of the feature values - that is

Apple’s stock price $\approx \beta_1 \cdot (\text{iPhone sales}) + \beta_2 \cdot (\text{MacBook sales}) + \beta_3 \cdot (\text{Samsung’s stock price})$

for coefficients $\beta_1, \beta_2, \beta_3$. In general, since the values of feature i are stored in x_i - the i th column of X -

$$y \approx \sum_{i=1}^p \beta_i x_i.$$

If we let $\beta = (\beta_1, \dots, \beta_p)$, this reduces to the matrix multiplication

$$y \approx X\beta$$

The goal of the optimization problem is to determine β such that $X\beta$ approximates y as closely as possible. The simplest measure of an approximation’s accuracy is the squared

Euclidian distance between $X\beta$ and y . Thus the solution to the problem is $\hat{\beta}$, where

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \|X\beta - y\|_2^2$$

Since we're minimizing only the sum of squared differences between $X\beta$ and y , this is called the *ordinary least squares* (OLS) problem.

2.3 Lasso and General Penalized Least Squares Regression

Ordinary least squares suffers from several problems [12]. First, due to its lack of restrictions on β values, it tends towards high variance and overfits to noise in the data. Second, when p is large, we can improve the model's interpretability and simplicity by selecting a subset of features with the strongest effects and setting the weights of the rest to 0. OLS, however, generally predicts every weight to be nonzero.

Ridge regression, a modification to OLS, addresses the first problem. Ridge regression adds a term penalizing the L_2 norm of β , giving the following formula:

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \frac{1}{2N} \|X\beta - y\|_2^2 + \lambda \|\beta\|_2^2$$

where

$$\|\beta\|_2^2 = \sum_{i=1}^p \beta_i^2.$$

The L_2 norm penalty term encourages the optimizer to shrink the values of β . By restricting the values of β in this way we reduce the model's ability to overfit to small variations in the training data. But the smooth L_2 norm does not set any values in β to 0. To address this, the “least absolute shrinkage and selection operator” (lasso), proposed in [12], uses the L_1 norm of β instead, so the problem becomes

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \frac{1}{2N} \|X\beta - y\|_2^2 + \lambda \|\beta\|_1$$

where

$$\|\beta\|_1 = \sum_{i=1}^p |\beta_i|.$$

Unlike ridge's L_2 penalty, lasso's L_1 penalty shrinks small weights to 0. To see the intuition behind this, consider Figure 2.1 comparing the L_1 norm $|x|$ and L_2 norm x^2 in one dimension. The lasso penalty always keeps its nonsmooth ‘V’ shape at any scale, but zooming in on the ridge penalty, at 0, it flattens out to a line. The ridge penalty thus does not provide a sufficient penalty to weights near 0 to reduce them to exactly 0, whereas the lasso penalty does.

The L_1 norm also shrinks the weights, so the lasso addresses both of the above problems with OLS. However, the L_1 norm doesn't 'group' highly correlated features by assigning them similar weights. Instead, it assigns one feature from the group a strong coefficient and leaves the others out. The resulting model is then highly sensitive to noise in the feature lasso selects to represent the whole group. Another drawback of this anti-grouping behaviour is a decline in the model's interpretability - the zeroed features could in fact have a strong correlation with the target, but it appears as though they have little correlation.

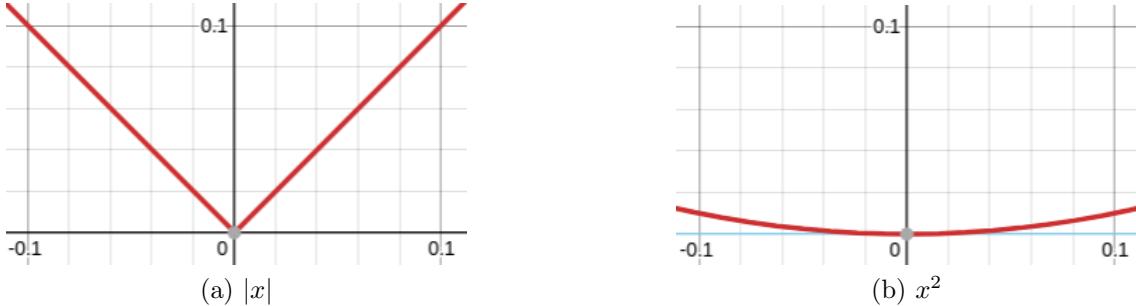


Figure 2.1: Comparison of lasso and ridge penalties in one dimension

2.4 Elastic Net

To address these issues with ridge regression and the lasso, *elastic net* was proposed in [16]. The elastic net's penalty term is a weighted sum of the L_1 and L_2 norms, so it can be viewed as a combination of ridge and lasso:

$$\hat{\beta} = \operatorname{argmin}_{\beta} \frac{1}{2N} \|X\beta - y\|_2^2 + \lambda_1 \|\beta\|_1 + \frac{1}{2} \lambda_2 \|\beta\|_2^2 \quad (2.1)$$

λ_1, λ_2 are scalar parameters whose values determine the relative influence of the ridge and lasso penalties. Elastic net avoids the drawbacks of both methods while keeping their benefits [16]. Assuming $\lambda_2 > 0$, the presence of the L_2 norm eliminates the lasso's anti-grouping problem. Furthermore, assuming $\lambda_1 > 0$, the overall penalty is nonsmooth so the elastic net inherits the lasso's desirable feature selection property.

It is important to note that, since all three terms of the argument in equation (2.1) are strictly convex, elastic net is a strictly convex optimization problem: the global minimum is the only local minimum, so algorithms don't have to worry about getting stuck in non-optimal local minima.

2.5 Coordinate Descent

CD is an optimization algorithm that successively minimizes along coordinate directions to find the minimum of a function. At each iteration, the algorithm determines a coordinate or coordinate block via a coordinate selection rule, then exactly or inexactly minimizes over the corresponding coordinate hyperplane while fixing all other coordinates or coordinate blocks. CD is applicable in both differentiable and subdifferentiable contexts. The algorithm is based on the idea that the minimization of a multivariable function $f(\beta)$ for $\beta \in \mathbb{R}^n$ can be achieved by minimizing it along one direction at a time i.e., solving univariate (or at least much simpler) optimization problems in a loop. For the remainder of this report, i_k is the coordinate selected based on iteration k , β_{i_k} is the i_k th coordinate of the β vector, and β^k is the predicted β for the k th iteration. In a general framework each step of CD chooses an index $i_k \in \{1, 2, \dots, n\}$, and updates β_{i_k} to $\beta_{i_k}^k$ by a certain scheme depending on β^{k-1} and f , while keeping β_j unchanged for all $j \neq i_k$. This update step consists of adjusting the i_k -th component of β in the opposite direction of the gradient ∇f with a step size of α_k . The process is repeated until a termination condition is satisfied.

The basic underlying problem in all settings is the following unconstrained optimization problem:

$$\min_{x, \beta \in \mathbb{R}^n} f(\beta)$$

where $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is continuous. Algorithm 1 from [14] describes coordinate descent:

Algorithm 1 Basic CD

- 1: set $k \leftarrow 0$ and choose $x^0 \in \mathbb{R}^n$
 - 2: **repeat**
 - 3: Choose index $i_k \in \{1, 2, \dots, n\}$
 - 4: $\beta^{k+1} \leftarrow \beta^k - \alpha_k [\nabla f(\beta^k)]_{i_k} e_{i_k}$ for some $\alpha_k > 0$
 - 5: $k \leftarrow k + 1$
 - 6: **until** termination test satisfied
-

Different variants of the Algorithm 1 exist, which incorporate modifications for the update step such as block coordinate minimization, proximal point update, prox-linear update, and extrapolation, described in [9]. Variants for choosing the index (or block) to update at each step include cyclic, randomized, and greedy variants such as Gauss-Southwell rules. In practice, these various methods can have different performance based on the data's characteristics, which makes the exploration of such methods a central part of our research.

2.6 Stochastic Gradient Descent

SGD is an iterative method to optimize an objective function by estimating the gradient from a subset of the data and using the estimated gradient to apply a coordinate or gradient descent method. The purpose of estimating the gradient is to reduce the computational burden of calculating the gradient if the dataset is large. SGD converges to a global or local minimum, almost surely when the objective function is convex. The problem that SGD solves is the following:

$$\min_w Q(w) := \frac{1}{n} \sum_{i=1}^n Q_i(w)$$

where $Q : \mathbb{R}^n \rightarrow \mathbb{R}$ and $w \in \mathbb{R}^n$. To compute the full gradient for the above problem, one would have to compute

$$\nabla Q(w) = \frac{1}{n} \sum_{i=1}^n \nabla Q_i(w)$$

by processing every training example in the dataset. This is often infeasible, and we can instead sample either one, or a small batch of loss functions Q_i , called *mini-batch*, to compute a sub-sampled gradient in place of the full gradient. For example, this can be achieved by randomly shuffling the training set. Algorithm 2 from [15] describes the discussed procedure.

Algorithm 2 Basic SGD

- 1: set $k \leftarrow 0$, choose $w^0 \in \mathbb{R}^n$, and choose learning rate η
 - 2: **repeat**
 - 3: Randomly shuffle examples in the training set
 - 4: **for** $i = 1, 2, \dots, n$ **do**
 - 5: $w^{k+1} := w^k - \eta \nabla Q_i(w)$
 - 6: **end for**
 - 7: $k \leftarrow k + 1$
 - 8: **until** termination test satisfied
-

SGD is a useful tool for estimating the gradient when using the full gradient is computationally inefficient. Since the dataset involved in the central problem for our research is large, SGD is a good candidate for our exploration.

2.7 Determining Convergence through the Duality Gap

The main problem of a convex optimization problem is referred to as the primal problem. There also exists a companion problem to the primal called the dual problem, typically the Lagrangian dual problem. For convex problems, the standard primal form is defined as:

$$\begin{aligned} \min_{\beta} \quad & f_0(\beta) \\ \text{s.t.} \quad & f_i(\beta) \leq 0 \text{ for } i = 1, 2, \dots, j, \\ & h_i(\beta) \leq 0 \text{ for } i = 1, 2, \dots, k \end{aligned}$$

where j and k are integers ≥ 1 that depend on the specific problem. The standard Lagrangian dual function is defined as:

$$L(\beta, \lambda, \nu) = f_0 + \sum \lambda_i f_i + \sum \nu h_i$$



The standard dual problem is defined as:

$$\begin{aligned} \max_{\beta} \quad & \inf_{\lambda} L(\beta, \lambda, \nu) \\ \text{s.t.} \quad & \lambda \geq 0. \end{aligned}$$

The duality gap is then defined as the difference between the primal problem's value and the value of the dual problem. By Weak Duality, the duality gap should always be greater than or equal to 0. If Strong Duality holds, then the solution is optimal if and only if the duality gap is equal to 0.

In the case of least squares with the elastic net regularizer, we can define a new variable $z = X\beta - y$, to create an equivalent problem as the primal given in the first section, as described in [6],

$$\min_{\beta} \quad z^T z + n\lambda_1 \|\beta\|_1 + n\lambda_2 \|\beta\|_2^2$$



We can associate the dual variable $\nu_i \in \mathbb{R}, i = 1, \dots, m$. Thus, the Lagrangian becomes

$$L(\beta, z, \nu) = \frac{z^T z}{2n} + n\lambda_1 \|\beta\|_1 + n\lambda_2 \|\beta\|_2^2 + \nu^T (X\beta - y - z).$$



The dual function is

$$\inf_{\beta, z} L(\beta, z, \nu) = \begin{cases} -(1/2) \|\nu\|_2^2 - \nu^T y, & \|(X^T \nu - 2n\lambda_2 \beta)\|_\infty \leq n\lambda_1 \\ -\infty, & \text{otherwise.} \end{cases}$$

The Lagrange dual problem is then given by

$$\begin{aligned} \max_{\nu} \quad & G(\nu) \\ \text{s.t.} \quad & \|(X^T \nu - 2\lambda_2 \beta)\|_\infty \leq n\lambda_1 \end{aligned}$$

where the dual objective $G(\nu)$ is

$$G(\nu) = -(1/4)\nu^T \nu - \nu^T y.$$

The duality gap is then given by

$$\eta = \|X\beta - y\|_2^2 + n\lambda_1\|\beta\|_1 + n\lambda_2\|\beta\|_2^2 - G(\nu)$$

We can derive a feasible dual variable ν for an arbitrary β with the following formula:

$$\nu = 2s(X\beta - y)$$

$$s = \min\{n\lambda_1/|(2X^T(X\beta - y))_i|, i = 1, \dots, p\}$$

where p is the number of features. Strong duality holds for least squares with the elastic net regularizer and, thus, we can use the duality gap as a certificate of optimality [11].

2.8 Data Properties and Generation

Aquatic provided us with three datasets: ‘‘Small’’, ‘‘Medium’’, and ‘‘Large’’. They have $p = 10$, $p = 162$, and $p = 1768$, respectively, and all three have $N \approx 17 \cdot 10^6$. To test our algorithms’ robustness, we also implemented routines to generate random data mimicking Aquatic’s datasets. Our generated data inherited the following properties from Aquatic’s real data:

1. Each column j of X is sampled from a normal distribution with mean $\mu_j = 0$ and standard deviation $\sigma_j = 1$ (denoted $N(0, 1)$). Thus the features are ‘standardized’ in the sense that, for each feature j ,

$$\sum_{i=1}^N x_{ij} = 0 \text{ and } \sum_{i=1}^N x_{ij}^2 = 1$$

2. For features a, b , the correlation between them is calculated as

$$r_{ab} = \frac{1}{N} \sum_{i=1}^N x_{ia} x_{ib}$$

~~or equivalently~~, with x_a and x_b being the a and b -th columns of X , respectively,

$$r_{ab} = x_a \cdot x_b$$

Some features fall into correlated blocks. Each pair of features a, b within a correlated block have correlation coefficient $r_{ab} \approx 0.8$, but for ~~most~~ pairs of features $r_{ab} \approx 0$.

3. The distribution of correlations r_{jy} between each feature j and the target y should roughly $N(0, 0.0075)$. ~~Since y is not necessarily standardized,~~

$$r_{jy} = \frac{\sum_{i=1}^N X_{ij}(y_i - \mu_y)}{\sigma_y}.$$

4. The entries of the target y ~~should approximate $N(0, 10^{-3})$~~ .

The basic data generation procedure is as follows: first, we sample the ‘true’ weights $\hat{\beta}$ from a normal distribution centered at 0 with a standard deviation chosen to satisfy property 3, as shrinking the norm of $\hat{\beta}$ relative to the noise in y decreases the values of r_{jy} .

Next, we initially sample X according to property 1. To achieve property 2, one specifies ahead of time how many blocks of correlated features the dataset should have and the number of features, on average, in each correlated block. After generating the columns of X independently, for each block, we randomly select the specified number of features, plus or minus 33%. Each of the selected features i (other than the first chosen) is correlated with an arbitrary feature j in the block using the formula

$$X_i = \sqrt{1 - 0.8^2} X_i + 0.8^2 X_j.$$



Next we compute

$$y = Xw + \epsilon$$

where ϵ is a noise vector sampled from $N(0, 10^{-3})$ to satisfy property 4. X and y are inputs to the optimizer, and since we know $\hat{\beta}$ we can test the accuracy of the optimizer’s predicted β .

Chapter 3

CD Optimization Methods without Accuracy Loss

3.1 Prior Related Work

Before creating our own implementations of CD, we looked into several recently developed CD algorithms as a general guideline of possible optimizations.

3.1.1 Parallel Coordinate Descent

Across Features

One way to parallelize CD is across features, updating multiple features at a time. The effectiveness of this method depends on ~~the separability~~ of the data. If ~~the data is entirely separable~~ (i.e., the columns within the data are not correlated), then the computational cost of optimizing across all the features one after the other is equal to the computational cost of optimizing across all features independently. However, if the features are ~~not separable~~, then updating multiple coordinates simultaneously will be suboptimal, thus requiring more iterations. Squared loss is a partially separable function when the feature matrix X is sparse, as shown in [9].

Distributed Parallel Coordinate Descent

For problems with large datasets, loading in all the data may be impossible, and thus we cannot parallelize as much as we would like. A solution to this problem is the parallelize not just across cores of an individual processor, but across nodes of a network of computers. However, this method is not without its own bottlenecks. One major problem with distributed CD is that communication speeds across nodes are much slower than within a node. Another problem with many distributed methods is that after each iteration, the algorithm has to synchronize the updated weights across nodes. Thus, the whole system is only as fast as the slowest node.

Asynchronous Parallel Coordinate Descent

A solution to the problem in distributed CD and other parallel CD methods is asynchrony. Rather than synchronizing all the updated weights after each iteration, asynchrony allows each node to constantly update its weight, without having to wait for the slowest node.

This method is much newer and relatively unstudied compared to synchronous methods. A reason for this is the difficulty in analyzing convergence for asynchronous CD methods compared to synchronous ones. However, in some cases, asynchronous methods can achieve speedups over serial methods of 25x as compared to the 4x speedup of synchronization [9]. Proving that asynchronous CD methods converge depends on how “stale” we are willing to let β get. In other words, how far ahead are we willing to let one process get compared to the slower nodes. If we let one process go too far ahead, then convergence cannot be proved. However, if we set an upper limit on the “staleness” of β , a type of CD called “partially asynchronous” in [14], then convergence holds.

3.1.2 Accelerated Coordinate Descent

Another method of optimizing CD algorithms is via acceleration, where the step size for each iteration is adaptive to how far away the values of β are from the global minimum. As described in [1], for smooth convex minimization, randomized coordinate descent methods have convergent rates of $1/\epsilon$, where $\epsilon > 0$ is an additive error. Using an accelerated method of CD results in a faster convergence rate of $1/\sqrt{\epsilon}$ as each iteration updates the coordinate value based on the distance from the minimum.

One accelerated CD algorithm we looked into is accelerated, parallel, and proximal CD (APPROX), originally proposed in [2]. In addition to having the advantages of parallelism described in the previous subsections, APPROX is accelerated and is proximal. The CD method is also proximal, allowing for implementations of the algorithm with various regressors such as Lasso or Elastic Net. However, its convergence rate is dependent on the amount of parallelism utilizable, and may not be appropriate for our project where the level of parallelism achievable is extremely limited.

3.1.3 Adaptive Elastic Net

One of the most recent applications of solving elastic net via coordinate descent in quantitative finance ~~comes from Shu, Shi, and Tian in [10] in April 2020. Shu, Shi, and Tian~~ solve a different problem (index tracking) than the problem Aquatic aims to solve, but the problem’s mathematical formulation is identical to the formulation discussed above, with a few added constraints. They use *adaptive elastic net* (Aenet), a modification to elastic net proposed in [17] in 2009. Aenet weights coordinates according to an initial estimate of β , giving it desirable theoretical consistency properties that ordinary elastic net does not have. ~~Shu, Shi, and Tian~~ solve the Aenet using coordinate descent and finds that the predicted weights are sparser than those predicted by ordinary elastic net, but not significantly more accurate. Due to the extra computational cost associated with Aenet we have not yet decided to explore it for our problem, but it is a potential area of future exploration.

3.1.4 Hybrid Coordinate Descent (Hydra)

This method, introduced in [8], utilizes parallelism at two levels: across a cluster of computer nodes and parallel processing within each node. In every CD iteration, Hydra first splits up the coordinates into random subsets, assigns each subset to a node, and then each node updates its assigned coordinates. This is a synchronous parallel CD algorithm because, after each iteration, data is synchronized across the nodes. Hydra is similar to other parallel CD methods but extends them to work on a distributed network. We do not have access to a

network of nodes, but can still explore distributed methods in a simulated manner on our single machine.

3.2 Feature Choice Rules

Within Algorithm 1, the core of the algorithm consists of the feature selection (Line 3) and update (Line 4) steps. Previous studies have shown optimized variations of each step. This section looks into various methods for optimal feature selection. There are two desired attributes to have an optimal feature selection method: low computational complexity and high importance in the selected features. Low computational complexity leads to faster run time, and choosing features based on update priority leads to faster convergence. The following methods for determining i_k will show various computational complexities and feature choice importance.

3.2.1 Basic methods

Cyclic

Cyclic based feature selection is the simplest form of determining which feature to use.

$$i_{k+1} = (k \bmod n) + 1, k \in n. \quad (3.1)$$

Random with Replacement

Another method of choosing a feature is via random probability. The random with replacement selection method chooses feature i_k with uniform probability from $\{1, \dots, n\}$.

Random without Replacement

A variant to the random with replacement feature selection method is to use random selection without replacement. By this, once i_k is chosen based on a uniform probability of the features, its probability of being chosen is 0 until all other each feature has been selected.

3.2.2 Greedy

The basic methods described in the previous section are extremely computationally efficient, each with $O(1)$ complexity, but are not ideal feature selection methods for convergence in few iterations. For example:

Let $\beta_j^0 = \beta_j^{fin}$ if $j \neq n$ where β^0 is the initial guess of β^{fin} is the vector for the global minimum. For the simpler methods, it may take many different feature selections before selecting the only β_n the only feature that needs an update. Hence, there is motivation to explore other feature selection methods, such as greedy methods, that choose feature β_n in fewer iterations. Many greedy variants have large computational complexities for each iteration by calculating the gradient-based on each feature but are able to select coordinates farthest away from that global minimum.

Selecting features via Greedy rules are defined by the following equation:

$$i_k = \arg \max_{1 \leq j \leq n} \left\| \nabla_j f(x^{k-1}) \right\| \quad (3.2)$$

For our studies, we implemented the Gaussian-Southwell-s (GS-s) rule, one of many greedy variants for feature selection. We chose this rule for several reasons. 1) The method is well known and has lots of background literature to support its definitions and convergence rate. 2) The GS method for Lasso regression was previously defined in [9] and is easily adjustable for elastic net since elastic net can be derived from Lasso.

$$i_k = \arg \max_{1 \leq j \leq n} g_j(x^{k-1}) \quad (3.3)$$

where

$$g_j(x^{k-1}) = \begin{cases} \|X_{:,j}^T(X\beta^{k-1} - y)/N + \lambda_1 + 2\lambda_2\beta_j^{k-1}\| & \text{if } \beta_j^{k-1} > 0 \\ \|X_{:,j}^T(X\beta^{k-1} - y)/N - \lambda_1 + 2\lambda_2\beta_j^{k-1}\| & \text{if } \beta_j^{k-1} < 0 \\ \|\text{shrink}(X_{:,j}^T(X\beta^{k-1} - y)/N, \lambda_1 + 2\lambda_2\beta_j^{k-1})\| & \text{if } \beta_j^{k-1} = 0 \end{cases} \quad (3.4)$$

The shrink function from the equation above is defined as:

$$\text{shrink}(x, \mu) = \begin{cases} x - \mu & \text{if } x > \mu \\ 0 & \text{if } -\mu \leq x \leq \mu \\ x + \mu & \text{if } x < -\mu \end{cases} \quad (3.5)$$

Calculating the GS-s method results in a complexity of $O(Np^2)$ for each feature selection, where N = number of samples and p = number of features. While this complexity is much larger than the previous methods, it is proven in [7] that GS-s converges in fewer iterations compared to randomized in all except a few extreme cases that don't apply to our constraints.

3.2.3 Hybrid

In general, through a CD algorithm, the distance between the loss function value approximated at an iteration i_k and the global minimum decreases as the iteration i_k increases. As CD goes through many iterations, the update step size decreases to allow for finer adjustments, and therefore, the feature selection has less of an impact on the change of the loss function value. It is important to consider this concept when choosing an optimal feature selection method. For example, the high cost of performance greedy methods may not be worthwhile for later iterations of CD where the change of the weights shrink in magnitude.

One way to solve this problem is by utilizing hybrid methods, where the first few iterations select feature selection methods with large computational-complexities and the last few iterations select simple feature selection methods. For example, a CD algorithm could start with a greedy feature selection method, but alternate to a cyclic feature selection method once the change of the weights reaches a certain tolerance. Section 5.6 looks into studies of create an optimal hybrid method.

3.2.4 Adaptive Coordinate Frequencies

Adaptive Coordinate Frequencies (ACF) is an online feature selection method proposed in [4] in 2013. It is a combination of the Greedy and Random with Replacement selection methods in that it selects features probabilistically, but the probability of selecting each coordinate is weighted by the expected loss function decrease resulting from updating that coordinate.

Let (P_1, \dots, P_p) be the probabilities of selecting features $1, \dots, p$ on a given update step. For example the random without replacement method samples from the uniform distribution

$$(P_1, \dots, P_p) = \left(\frac{1}{N}, \dots, \frac{1}{N} \right) \quad (3.6)$$

on every update.

ACF initializes the feature selection probabilities according to equation (3.6). Suppose coordinate i is chosen on step t . The algorithm minimizes the loss function along coordinate i by updating β_i^{t-1} , the value of β_i before step t , to β_i^t . If we let

$$\begin{aligned} \boldsymbol{\beta}^{t-1} &= (\beta_1, \dots, \beta_i^{t-1}, \dots, \beta_p) \\ \boldsymbol{\beta}^t &= (\beta_1, \dots, \beta_i^t, \dots, \beta_p) \end{aligned}$$

then step t 's update changed the value of the loss function f by

$$\Delta = f(\boldsymbol{\beta}^t) - f(\boldsymbol{\beta}^{t-1}).$$

The larger Δ is, the more progress we have made in moving the loss function towards its optimal value. Hence if Δ is large relative to other recent updates, we want to increase P_i relative to the other probabilities to increase our odds of selecting coordinate i in the future. On the other hand, if Δ is relatively small, we want to decrease P_i , as we have not made much progress updating along coordinate i .

Let $\bar{\Delta}$ be the (running) average of Δ values for each step up to t . Let

$$P_{\text{sum}} = \sum_{j=1}^p P_j.$$

Since P_1, \dots, P_p are initialized according to equation (3.6), $P_{\text{sum}} = 1$ initially. Let P_{max} and P_{min} limit from above and below, respectively, the values P_1, \dots, P_p . Define the notation

$$[x]_{P_{\text{min}}}^{P_{\text{max}}} = \begin{cases} P_{\text{max}} & x \geq P_{\text{max}} \\ x & P_{\text{min}} < x < P_{\text{max}} \\ P_{\text{min}} & x \leq P_{\text{min}} \end{cases}$$

Finally, let c and η be learning rate parameters. The following algorithm introduced in [4] updates P_i by comparing Δ to $\bar{\Delta}$. Steps 1 and 3 of algorithm 3 increase P_i if $\Delta/\bar{\Delta} > 1$ and

Algorithm 3 ACF probability update

- 1: $P_{\text{new}} \leftarrow \left[\exp \left\{ \left(c(\Delta/\bar{\Delta} - 1) \right) P_i \right\} \right]_{P_{\text{min}}}^{P_{\text{max}}}$
 - 2: $P_{\text{sum}} \leftarrow P_{\text{sum}} + (P_{\text{new}} - P_i)$
 - 3: $P_i \leftarrow P_{\text{new}}$
 - 4: $\bar{\Delta} \leftarrow (1 - \eta)\bar{\Delta} + \eta\Delta$
-

decrease P_i if $\Delta < \bar{\Delta}$. Step 2 changes P_{sum} to reflect the change in P_i . Step 4 adds Δ for this step to the running average.

To select a feature at each step, we then sample from a discrete distribution where the probability of selecting feature k is $\frac{P_k}{P_{\text{sum}}}$. As currently implemented, sampling from the discrete distribution runs in constant time. Calculating Δ and hence algorithm 3 is

$O(p)$. Adjusting the discrete distribution to reflect the change in P_i each step requires $O(p)$ time. Thus ACF adds $O(2p)$ time to each coordinate update. We are exploring alternative methods to adjust the discrete distribution to run in $O(\log p)$ or $O(1)$, which would reduce ACF's overall runtime per update to just $O(p)$.

ACF-GPI

The choice to initialize P_1, \dots, P_p as a uniform distribution is somewhat arbitrary. We propose an alternate scheme called ACF with greedy probability initialization (ACF-GPI). The primary drawback with the greedy scheme proposed in Section 3.2.2 is the cost of calculating the gradient along each coordinate *at every update step*. Performing one such calculation, however, is not prohibitively expensive. Thus, ACF-GPI initializes each P_k relative to the magnitude of the gradient along coordinate k , but performs the ACF feature choice instead of the greedy feature choice at every subsequent update. This alternate initialization gives ACF a ‘head start’ on selecting coordinates providing significant progress towards the optimum. Specifically, for each $k = 1, \dots, p$, we set

$$g_k = |\nabla_k f(\beta_0)|$$

where $\beta_0 = \mathbf{0}$ is the initial all-zero weight vector. Then for each k initialize

$$P_k = \frac{g_k}{\sum_{j=1}^p g_j}.$$

Each of these methods are compared against each other in Section 5.6.

3.3 Update Rules

The update rules shown here are based on those presented in [4]. Here we describe the mathematical inner workings behind these update rules as presented in [3]. Let $X = (x_{ij})_{ij}$. Notice that another way of rewriting equation (2.1) is

$$\min_{\beta \in \mathbb{R}^p} \frac{1}{2N} \sum_{i=1}^N (y_i - x_i^T \beta)^2 + \lambda_1 \|\beta\|_1 + \lambda_2 \|\beta\|_2^2 \quad (3.7)$$

Consider a coordinate decent minimization step for equation (3.7) in the coordinate j . Suppose we have estimates for $\tilde{\beta}_l$ for $l \neq j$, and we wish to partially optimize with respect to β_j . Denote by $R(\beta)$ the objective function in equation (3.7). We would like to compute the gradient at $\beta_j = \tilde{\beta}_j$, which only exists if $\tilde{\beta}_j \neq 0$. If $\tilde{\beta}_j > 0$, then

$$\frac{\partial R}{\partial \beta_j} \Big|_{\beta=\tilde{\beta}} = -\frac{1}{N} \sum_{i=1}^N x_{ij} (y_i - x_i^T \tilde{\beta}) + 2\lambda_2 \beta_j + \lambda_1$$

If $\tilde{\beta}_j < 0$, then the expression is

$$\frac{\partial R}{\partial \beta_j} \Big|_{\beta=\tilde{\beta}} = -\frac{1}{N} \sum_{i=1}^N x_{ij} (y_i - x_i^T \tilde{\beta}) + 2\lambda_2 \beta_j - \lambda_1$$

Finally if $\tilde{\beta}_j = 0$, then

$$\frac{\partial R}{\partial \beta_j} \Big|_{\beta=\tilde{\beta}} = -\frac{1}{N} \sum_{i=1}^N x_{ij} (y_i - x_i^T \tilde{\beta}) + 2\lambda_2 \beta_j + \lambda_1 \alpha$$

where $\alpha \in \partial_0 |\beta_j|$ is the subdifferential of $|\beta_j|$ at 0. Simple calculus show that the coordinate-wise update has the form

$$\tilde{\beta}_j \leftarrow \frac{S\left(\frac{1}{N} \sum_{i=1}^N x_{ij}(y_i - \tilde{y}_i^{(j)}), \lambda_1\right)}{1 + 2\lambda_2} \quad (3.8)$$

where

- $\tilde{y}_i^{(j)} = \tilde{\beta}_0 + \sum_{l \neq j} x_{il} \tilde{\beta}_l$ is the fitted value excluding the contribution from x_{ij} , and hence $y_i - \tilde{y}_i^{(j)}$ is partial residual for fitting β_j . Because of the standardization, $\frac{1}{N} \sum_{i=1}^N x_{ij}(y_i - \tilde{y}_i^{(j)})$ is the simple least-squares coefficient when fitting this partial residual to x_{ij} .
- $S(z, \gamma)$ is the soft-thresholding operator with value

$$S(z, \gamma) = \begin{cases} z - \gamma & z > \gamma \\ 0 & z \in [-\gamma, \gamma] \\ z + \gamma & z < -\gamma \end{cases}$$

Thus, we compute the least-squares coefficient on the partial residual, apply soft-thresholding to take care of the lasso contribution to the penalty, and then apply a proportional shrinkage for the ridge penalty. Now that we have an explicit formula for the update rule in coordinate minimization we consider two different update schemes, which implement the update rule in equation (3.8).

3.3.1 Naive Updates

Looking more closely at equation (3.7), we see that

$$\begin{aligned} y_i - \tilde{y}_i^{(j)} &= y_i - \hat{y}_i + x_{ij}\tilde{\beta}_j \\ &= r_i + x_{ij}\tilde{\beta}_j \end{aligned}$$

where \hat{y}_i is the current fit of the model for observations i , and hence r_i the current residual. Thus

$$\frac{1}{N} \sum_{i=1}^N x_{ij}(y_i - \tilde{y}_i^{(j)}) = \frac{1}{N} \sum_{i=1}^N x_{ij}r_i + \tilde{\beta}_j \quad (3.9)$$

because the columns x_j are standardized. The first term of the right-hand side is the gradient of the loss with respect to β_j . It is clear from equation (3.9) why coordinate descent is computationally efficient. Many coefficients are zero, remain zero after thresholding, and so nothing gets changed. Such a step costs $O(N)$ operations - the sum to compute the gradient. On the other hand, if a coefficient does change after thresholding, r_i is changed in $O(N)$, and the step costs $O(2N)$. Thus a complete cycle through all p variables costs $O(pN)$ operations. We refer to this as the naive algorithm since it is generally less efficient than the covariance updating algorithm to follows.

3.3.2 Covariance Updates

Further efficiencies can be achieved in computing the update in equation (3.8). We can write the first term on the right (up to a factor of $\frac{1}{N}$) as

$$\sum_{i=1}^N x_{ij} r_i = \langle x_j, y \rangle - \sum_{k:|\tilde{\beta}_k|>0} \langle x_j, x_k \rangle \tilde{\beta}_k \quad (3.10)$$

where $\langle x_j, y \rangle = \sum_{i=1}^N x_{ij} y_i$. Hence we need to compute inner products of each feature with y initially, and then each time a new features x_k enters the model (for the first time), we need to compute and store its inner product with all the rest of the features ($O(Np)$ computations). We also keep an unordered set of those indices j for which $\tilde{\beta}_j$ are non-zero and update that set as the algorithm progresses. Hence, with m non-zero terms in the model, a complete cycle costs $O(pm)$ operations if no new variables become non-zero, and costs $O(Np)$ for each new variable entered. **Most importantly, $O(N)$ calculations do not have to be made at every step.** Because of this main advantage of the covariance update, it has the potential to perform very well if most of the features in the final solutions are zero. If that is indeed the case, then they may never be introduced as a non-zero coefficient during the algorithm, and we will never need to compute the inner products of their respective features with the other features.

The Naive update can be easily implemented by using linear algebra, and it is simple to implement. The covariance update is a bit more complex to implement because of the need to maintain an unordered set during the algorithm. We implemented both the covariance update and the naive update and compared their speed performance on the datasets from Aquatic and on randomly generated data. When $N \gg p$ (as in our case) the covariance method is expected to significantly outperform naive method because covariance does at most $O(p)$ operations per iteration and occasionally $O(N)$ (at most p times), while naive uses $O(N)$ operations at every step.

Based on this reduced computational complexity, we chose the covariance update as our main foundation for further optimizing the coordinate descent algorithm.

3.4 Front Heavy Covariance Method

As we can see from equation (3.10), the update rule in the covariance update method depends only on the inner products of between the target and the features and the inner products between the features. Thus, if we precompute these quantities before the algorithm has already started, we can calculate the update in equation (3.8) by simply computing the sum in equation (3.10) with our precomputed and stored inner products. In the front heavy covariance method, we do all the computations before the algorithm begins. In other words, we precompute and store the inner products $X^T y = (\langle x_i, y \rangle)_{i=1}^N$ and the Gram matrix $X^T X = (\langle x_i, x_j \rangle)_{ij}$. After that, we execute the update rule given by equation (3.8) by using the sum in equation (3.10).

Algorithm 4 Front Heavy Covariance Method

- 1: Precompute and store $X^T y$ and $X^T X$
 - 2: Run normal covariance using equation (3.10)
-

One disadvantage of the front heavy covariance method is that it may lead to unnecessary computations. For example, if there is a feature whose weight is always zero during the execution of standard covariance, then standard covariance will never compute its inner products with the other columns. However, the front heavy method will always compute the inner products of that feature with all of the other features, which could be costly.

3.5 Numerical Linear Algebra

3.5.1 Parallelized Numerical Linear Algebra

Intel's Math Kernel Library (MKL) allows us to achieve peak performance for linear algebra operations on an Intel processor. One of the ways the library does this is through parallelizing its operations through vectorization and an open-source parallel programming library OpenMP. Vectorization, also known as array programming, is the process of performing operations on an entire array at once. This significantly improves computing performance over traditional for loops. OpenMP is a library for parallelizing code by splitting up loops into tasks that can be performed on multiple processors at once.

3.6 Thresholding Rules

The thresholding method consists of discarding a feature, is the magnitude of its inner product with the target is below a certain threshold. In our literature review, we came across rules, such as those in [13], for discarding predictors in lasso-type problems like the elastic net. These screening rules are essentially thresholding bounds. If satisfied, it is guaranteed that the thresholded feature will be zero in the final solution, and this can be ignored in the covariance update. This leads to saving computations as we consider fewer features. The screening rules proposed in [13] lead us to try to control the thresholding parameter and evaluating its performance with cross-validation. At the beginning of the covariance update, we need to precompute all of the inner products $\langle y, x_j \rangle$ for all j . If some of these inner products are sufficiently “small,” or below some given threshold, we can discard their respective features and set them to zero in the final solution. This discarding of features is reasonable because a small inner product with the target likely indicates a lower correlation with the target, which means that β_j is likely to be 0. We tried out experimenting with the thresholding parameter to see which threshold value is the “best”. We used cross-validation to determine the “best” value of the thresholding parameter. For our train-validation split, we used the most recent 10 percent of the data as a validation set and the rest 90 percent as a train set. We performed our experiments on the Small dataset. As we can see in the table shown in the results, a good value for the thresholding parameter would be 0.02, as that is where the validation loss is the least.

3.7 Disk I/O

3.7.1 Main Problem

The Medium dataset that we are working with is 11 GB in binary format with single precision. If we increase to double-precision, then that dataset, when loaded in, is 22 GB.

Since we are only working with 16 GB of RAM, we needed a method of loading in data partially.

3.7.2 Gram-based Coordinate Descent

The method described previously where inner products are calculated between the matrix X^T , X , and y is also known as a gram-based CD method. The matrix $X^T X$ is called the gram matrix. After $X^T X$ and $X^T y$ are calculated, we no longer need to have X loaded into memory. The gram can be calculated in blocks by rows of X . Let $m \leq n$ be the number of blocks in X determined such that we load in as many rows as possible. X is thus structured as follows:

$$X = \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ \vdots \\ X_m \end{pmatrix}$$

To calculate $X^T X$, we can instead calculate

$$X^T X = \sum_{i=1}^m X_i^T X_i$$

Thus, we can calculate $X^T X$ without loading in all X all at once and instead store the much smaller gram matrix, which has dimensions p by p . For this reason, gram-based approaches are mostly used when $n \gg p$. With this method, our limit for loading in data is no longer defined by the size of X . Rather our memory limit is mainly determined by the sizes of 2 rows of X , y , the gram matrix, and $X^T y$.

3.8 Preconditioning X

Another family of methods for speeding up coordinate descent aim to reduce the number of iterations for which it runs. One such method is early stopping (discussed in Section 4.5), where the algorithm stops before it has fully converged, sacrificing a small amount of accuracy to save time. In this section we discuss a different approach: preconditioning the data such that the algorithm converges in only a few iterations.

Recall from Section 3.4 the concept of the Gram matrix $X^T X = \{\langle X_i, X_j \rangle\}_{i,j}$ where $\langle X_i, X_j \rangle$ is a measure of the correlation between features i and j . If $X^T X = I$, then $\langle X_i, X_j \rangle = 0$ for every $i \neq j$, implying that every column has no correlation with any other column. In this case coordinate descent converges in just one iteration, as it moves each feature independently to its value at the global minimum and thus never has to update a feature more than once. For a simple visualization of this concept, consider figure 3.1 below. For $z = 2x^2 + y^2$, the ellipses are oriented along the x and y axis, so x and y are immediately set to 0 on their first updates. But the xy term in $z = 2x^2 + y^2 + xy$ rotates the ellipses, so CD takes several iterations to converge. $X^T X = I$ has roughly the same effect (in a much higher-dimensional sense) as $c = 0$ in $z = ax^2 + by^2 + cxy$.

In this spirit, consider substituting the substitution $(X^T X)^{-1/2} \beta_1 = \beta$ in the elastic net

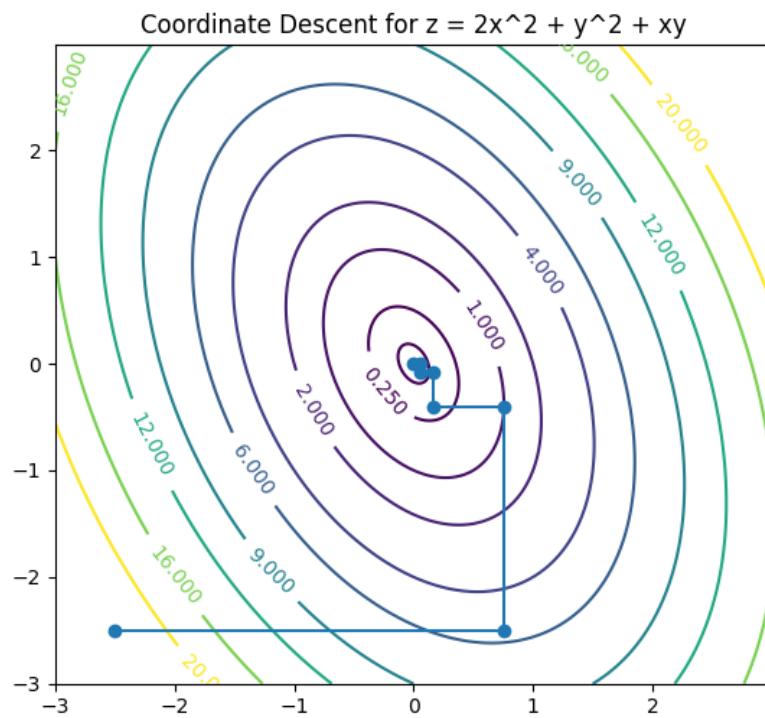
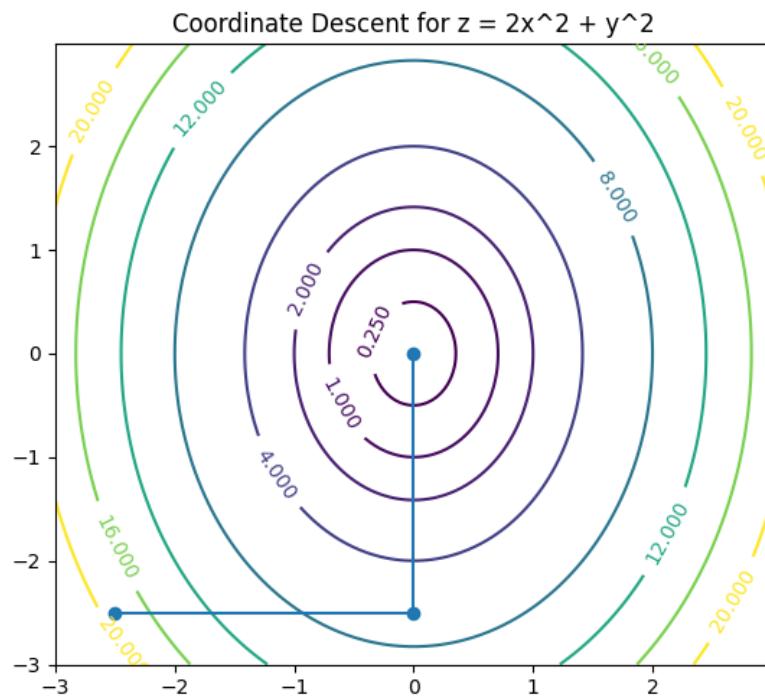


Figure 3.1: CD converges in one iteration on $z = 2x^2 + y^2$ where features x and y are “uncorrelated” but takes many iterations to converge for $z = 2x^2 + y^2 + xy$ due to the xy term.

formulation in equation (2.1), giving

$$\hat{\beta}_1 = \underset{\beta}{\operatorname{argmin}} \frac{1}{2N} \|X((X^T X)^{-1/2}\beta_1) - y\|_2^2 + \lambda_1\|(X^T X)^{-1/2}\beta_1\|_1 + \frac{1}{2}\lambda_2\|(X^T X)^{-1/2}\beta_1\|_2^2 \quad (3.11)$$

where $G = (X^T X)^{-1/2}$ is defined such that $G * G = (X^T X)^{-1}$. While not every matrix has a ‘square root’ in this sense, $(X^T X)^{-1}$ does because it is the inverse of the positive definite matrix $X^T X$ and hence is also positive definite. One can calculate the square root of such a matrix by finding its eigen/spectral decomposition

$$(X^T X)^{-1} = Q\Lambda Q^{-1},$$

calculating the square root of the diagonal matrix Λ by taking the square roots of each of its (positive) diagonal elements, and letting

$$(X^T X)^{-1/2} = Q\sqrt{\Lambda}Q^{-1}$$

so that

$$(X^T X)^{-1/2} \cdot (X^T X)^{-1/2} = Q\sqrt{\Lambda}Q^{-1}Q\sqrt{\Lambda}Q^{-1} = Q\sqrt{\Lambda}\sqrt{\Lambda}Q^{-1} = Q\Lambda Q^{-1} = (X^T X)^{-1}.$$

If we view the first term of equation (3.11) as $\frac{1}{2N}\|(X(X^T X)^{-1/2})\beta_1\|_2^2$, we see that we have a new elastic net-like problem with

$$Z = X(X^T X)^{-1/2}$$

in place of X , so we can rewrite equation (3.11) as

$$\hat{\beta}_1 = \underset{\beta}{\operatorname{argmin}} \frac{1}{2N} \|Z\beta_1 - y\|_2^2 + \lambda_1\|G\beta_1\|_1 + \frac{1}{2}\lambda_2\|G\beta_1\|_2^2. \quad (3.12)$$

This point of this substitution is that now

$$\begin{aligned} Z^T Z &= (X(X^T X)^{-1/2})^T X(X^T X)^{-1/2} \\ &= ((X^T X)^{-1/2})^T X^T X(X^T X)^{-1/2} \\ &= (X^T X)^{-1/2}(X^T X)(X^T X)^{-1/2} \quad ((X^T X)^{-1/2} \text{ is symmetric}) \\ &= (X^T X)^{-1/2}((X^T X)^{-1})^{-1}(X^T X)^{-1/2} \\ &= Q\sqrt{\Lambda}Q^{-1}(Q\Lambda Q^{-1})^{-1}Q\sqrt{\Lambda}Q^{-1} \\ &= Q\sqrt{\Lambda}Q^{-1}Q\Lambda^{-1}Q^{-1}Q\sqrt{\Lambda}Q^{-1} \\ &= Q\sqrt{\Lambda}\Lambda^{-1}\sqrt{\Lambda}Q^{-1} \\ &= QQ^{-1} \quad (\sqrt{\Lambda} \text{ and } \Lambda^{-1} \text{ are diagonal so they commute}) \\ &= I \end{aligned}$$

Hence multiplying X by G ‘rotates’ it in the same sense that rotating the ellipses in figure 3.1 to align with the x and y axes causes CD to converge in one iteration. Thus we can solve equation (3.12) using coordinate descent to obtain $\hat{\beta}_1$ and set $\hat{\beta} = G\hat{\beta}_1$ to recover a solution to the original problem in equation (2.1). If $\lambda_1 = \lambda_2 = 0$ then by the above discussion CD will solve the preconditioned problem in equation (3.12) in one iteration. The multiplication by G in $\lambda_1\|G\beta_1\|_1 + \frac{1}{2}\lambda_2\|G\beta_1\|_2^2$ adds some dependence between features so the preconditioned problem likely won’t fully converge within one iteration, but it should still converge in fewer iterations than the original problem.

3.8.1 Preconditioned Problem Update Rule

Equation (3.12) is a modification of the original elastic net formulation in 2.1, so we need to derive a corresponding new version of the coordinate update step in equation (3.8). An additional complication arises from the fact that, while we could assume that X was standardized, we cannot make the same assumption for Z . Suppose we are updating feature k , so we aim to minimize the modified loss function

$$L(\beta_k) = \frac{1}{2N} \|Z\beta - y\|_2^2 + \lambda_1 \|G\beta\|_1 + \frac{1}{2} \lambda_2 \|G\beta\|_2^2.$$

with respect to β_k . Straightforward calculus shows that

$$\begin{aligned} \frac{\partial}{\partial \beta_k} \frac{1}{2} \lambda_2 \|G\beta\|_2^2 &= \frac{\partial}{\partial \beta_k} \frac{1}{2} \lambda_2 \sum_{j=1}^p \left(\sum_{l=1}^p g_{jl} \beta_l \right)^2 \\ &= \lambda_2 \sum_{j=1}^p \left(g_{jk} \sum_{l=1}^p g_{jl} \beta_l \right). \end{aligned} \quad \boxed{=} \quad (3.13)$$

and

$$\begin{aligned} \frac{\partial}{\partial \beta_k} \frac{1}{2N} \|Z\beta - y\|_2^2 &= \frac{\partial}{\partial \beta_k} \frac{1}{2N} \sum_{i=1}^N \left(\sum_{j=1}^p (z_{ij} \beta_j) - y_i \right)^2 \\ &= \frac{1}{N} \left(\beta_k \sum_{i=1}^N (z_{ik}^2) - \sum_{i=1}^N z_{ik} \left(y_i - \sum_{j \neq k} z_{ij} \beta_j \right) \right). \end{aligned} \quad \boxed{=} \quad (3.14)$$

$Z^T Z = I \implies \langle Z_k, Z_k \rangle = \sum_{i=k}^N (z_{ik}^2) = 1$ and, using an idea similar to the covariance update in equation (3.10), we can write

$$\sum_{i=1}^N z_{ik} \left(y_i - \sum_{j \neq k} z_{ij} \beta_j \right) = \langle Z_k, y \rangle - \sum_{j \neq k} \langle Z_k, Z_j \rangle \beta_j = \langle Z_k, y \rangle$$

since $\langle Z_k, Z_j \rangle = 0$ for $j \neq k$. Thus we can rewrite equation (3.14) as

$$\frac{\partial}{\partial \beta_k} \frac{1}{2N} \|Z\beta - y\|_2^2 = \frac{1}{N} (\beta_k - \langle Z_k, y \rangle). \quad (3.15)$$

$\frac{\partial}{\partial \beta_k} \lambda_1 \|G\beta\|_1$, however, poses a greater challenge. Define

$$h(\beta_k) = \|G\beta\|_1 = \sum_{j=1}^p \left| \sum_{l=1}^p g_{jl} \beta_l \right| = \sum_{j=1}^p |g_{jk} \beta_k + r_{jk}|.$$

where

$$r_{jk} = \sum_{l \neq k} g_{jl} \beta_l$$

is the residual sum at row j of G .

As a function of β_k , the original $\lambda_1 \|\beta\| = \sum_{j=1}^p |\beta_j|$ fails to be differentiable only at $\beta_k = 0$. However, since β_k is present in every term of $\sum_{j=1}^p |g_{jk} \beta_k + r_{jk}|$, h fails to be differentiable at the p points where $g_{jk} \beta_k + r_{jk} = 0$, namely

$$\beta_k \in \text{ND} = \left\{ -\frac{r_{jk}}{g_{jk}} \mid j \in \{1, \dots, p\} \right\} \quad (3.16)$$

With the exception of the separate case where $\beta = \mathbf{0}$ (the update for this case is given in equation (3.22) below) we assume that ND contains j distinct points.

The derivation of the original elastic net update uses the concept of subdifferentials to handle the case where $\beta_k = 0$. We consider a differentiable function f of one variable to be *convex* if

$$\forall x, y \in \mathbb{R} : f(y) \geq f(x) + f'(x)(y - x) [9].$$

Geometrically, we interpret this as the graph of f never dipping below those of its tangent lines at any point. For non-differentiable functions such as $f(x) = |x|$ we define a *subgradient* at x to be any slope $m \in \mathbb{R}$ such that the graph of f lies above the tangent line at x with slope m :

$$\forall y \in \mathbb{R} : f(y) \geq f(x) + m(y - x)$$

Then the *subdifferential* $\partial f(x)$ of f at x is the set of all subgradients at x :

$$\partial f(x) = \{m \in \mathbb{R} \mid \forall y \in \mathbb{R} : f(y) \geq f(x) + m(y - x)\}.$$

For example, figure 3.2 shows some subgradients of $f(x) = |x|$ at the non-differentiable point $x = 0$. If f is differentiable at x , then the only line tangent to f at x has slope $f'(x)$,

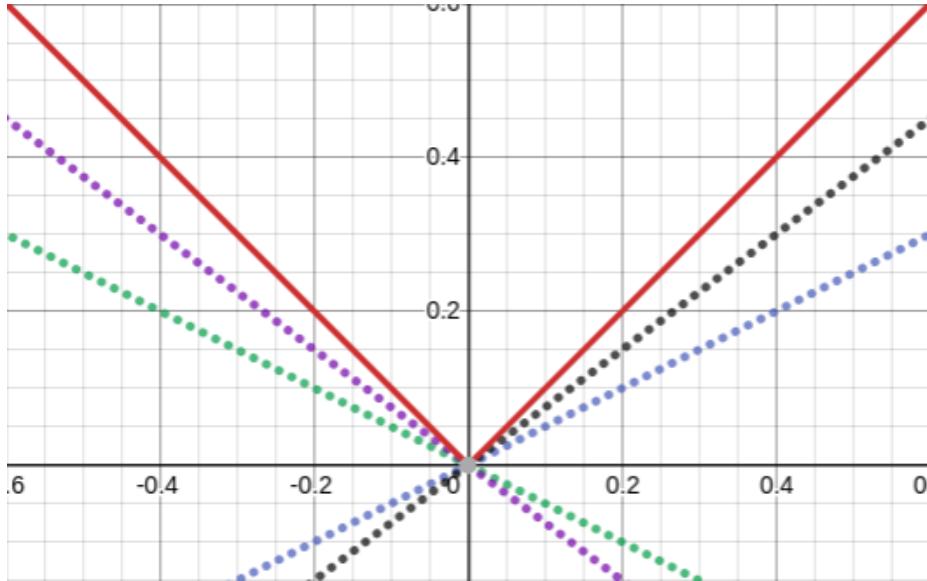


Figure 3.2: $\partial f(0) = [-1, 1]$ for $f(x) = |x|$. Any line $y = mx$ with $|m| \leq 1$ will lie below the graph of $|x|$ (in red), including the dotted $y = \pm\frac{1}{2}x$ and $y = \pm\frac{3}{4}x$

so

$$\partial f(x) = \{f'(x)\}.$$

β_k globally minimizes the convex function $h \iff$

$$\forall x \in \mathbb{R} : h(\beta_k) \geq h(x) \iff \forall x \in \mathbb{R} : h(\beta_k) \geq h(x) + 0(\beta_k - x) \iff 0 \in \partial h(\beta_k).$$

To calculate ∂h , we must first determine $h'(\beta_k)$ at points $\beta_k \notin \text{ND}$. For each j ,

$$\frac{\partial}{\partial \beta_k} |g_{jk}\beta_k + r_{jk}| = \begin{cases} -|g_{jk}| & \beta_k < -\frac{r_{jk}}{g_{jk}} \\ |g_{jk}| & \beta_k > -\frac{r_{jk}}{g_{jk}} \end{cases}$$

Let $c_1 = -\frac{r_k^{(1)}}{g_k^{(1)}}, \dots, c_p = -\frac{r_k^{(p)}}{g_k^{(p)}}$ be the points of ND sorted from least to greatest. This lets us define, for $\beta_k \notin \text{ND}$,

$$h'(\beta_k) = \begin{cases} s_1 = -\sum_{l=1}^p |g_k^{(l)}| & \beta_k < c_1 \\ s_2 = -\sum_{l=2}^p |g_k^{(l)}| + |g_k^{(1)}| & c_1 < \beta_k < c_2 \\ \vdots \\ s_m = -\sum_{l=m}^p |g_k^{(l)}| + \sum_{t=1}^{m-1} |g_k^{(t)}| & c_{m-1} < \beta_k < c_m \\ \vdots \\ s_{p+1} = \sum_{t=1}^p |g_k^{(t)}| & \beta_k > c_p. \end{cases} \quad (3.17)$$

Due to the convexity of h , for $\beta_k = c_j$, any tangent line with a slope between the slopes of the portions of the graph of h on either side of c_j will be a subgradient. That is, for $j = 1, \dots, p$:

$$\partial h(c_j) = [s_j, s_{j+1}]$$

Figure 3.3 below gives a simple illustration of this fact.

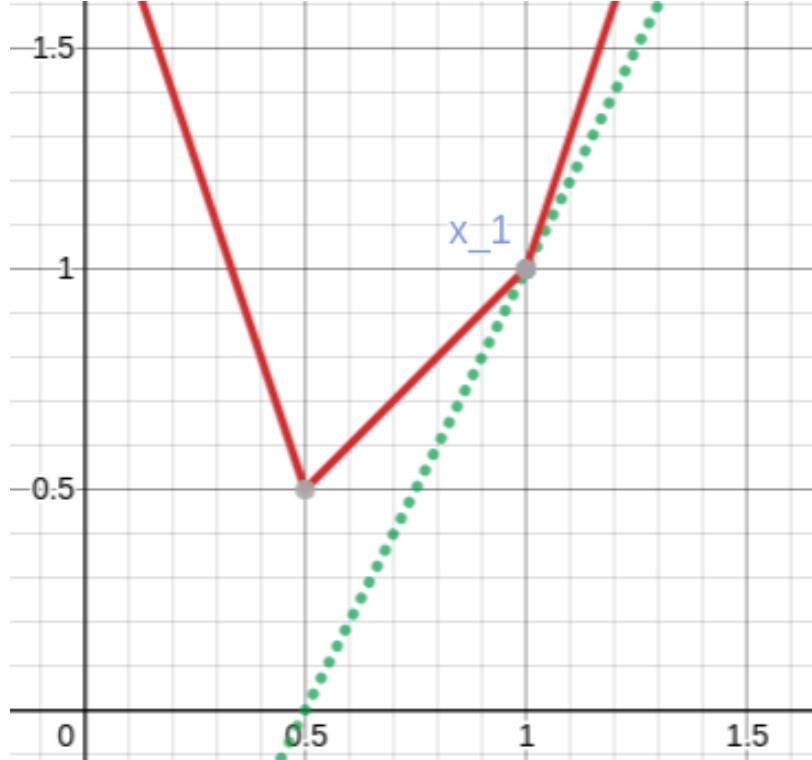


Figure 3.3: $\partial f(1) = [1, 3]$ for the red line $f(x) = |x - 1| + |-2x + 1|$. Any slope between the slopes of the portions of the graph immediately to the left and right of $x = 1$ is a subgradient. For example, the dotted green sline has slope 2, a subgradient.

Now, using equation (3.17) and the fact that if h is differentiable at β_k then $\partial h(\beta_k) =$

$h'(\beta_k)$,

$$\partial h(\beta_k) = \begin{cases} s_1 & \beta_k < c_1 \\ [s_1, s_2] & \beta_k = c_1 \\ s_2 & c_1 < \beta_k < c_2 \\ \vdots & \\ [s_m, s_{m+1}] & \beta_k = c_m \\ s_{m+1} & c_m < \beta_k < c_{m+1} \\ \vdots & \\ [s_p, s_{p+1}] & \beta_k = c_p \\ s_{p+1} & \beta_k > c_p. \end{cases} \quad (3.18)$$

Combining equation (3.18) with the results of equations 3.13 and 3.15 shows that $L(\beta_k)$ is minimized when

$$\begin{aligned} \partial L(\beta_k) &= \frac{1}{N} (\beta_k - \langle Z_k, y \rangle) + \lambda_2 \sum_{j=1}^p \left(g_{jk} \sum_{l=1}^p g_{jl} \beta_l \right) + \lambda_1 \partial h(\beta_k) \\ &= \left(\frac{1}{N} + \lambda_2 \sum_{j=1}^N (g_{jk}^2) \right) \beta_k - \frac{1}{N} \langle Z_k, y \rangle + \lambda_2 \sum_{j=1}^p \left(g_{jk} \sum_{l \neq k} g_{jl} \beta_l \right) + \lambda_1 \partial h(\beta_k) \\ &= a_k \beta_k - b_k + \lambda_1 \partial h(\beta_k) \ni 0 \end{aligned}$$

where

$$a_k = \frac{1}{N} + \lambda_2 \langle G_k, G_k \rangle \quad (3.19)$$

and

$$b_k = \frac{1}{N} \langle Z_k, y \rangle - \lambda_2 \sum_{j=1}^p \left(g_{jk} \sum_{l \neq k} g_{jl} \beta_l \right). \quad (3.20)$$

We must consider four types of cases for $\partial h(\beta_k)$.

1. $\partial h(\beta_k) = s_1$ ($\beta_k < c_1$):

$$a_k \beta_k - b_k + \lambda_1 s_1 = 0 \implies \beta_k = \frac{b_k - \lambda_1 s_1}{a_k} < c_1 \iff b_k < a_k c_1 + \lambda_1 s_1.$$

So if $b_k < a_k c_1 + \lambda_1 s_1$, L is minimized at $\beta_k = \frac{b_k - \lambda_1 s_1}{a_k}$.

2. $\partial h(\beta_k) = [s_m, s_{m+1}]$ for $m = 1, \dots, p$ ($\beta_k = c_m$):

$$\begin{aligned} a_k \beta_k - b_k + \lambda_1 [s_m, s_{m+1}] &= a_k c_m - b_k + [\lambda_1 s_m, \lambda_1 s_{m+1}] \\ &= [a_k c_m - b_k + \lambda_1 s_m, a_k c_m - b_k + \lambda_1 s_{m+1}] \ni 0 \\ &\iff a_k c_m - b_k + \lambda_1 s_m < 0 < a_k c_m - b_k + \lambda_1 s_{m+1} \\ &\iff a_k c_m + \lambda_1 s_m < b_k < a_k c_m + \lambda_1 s_{m+1} \end{aligned}$$

So if $a_k c_m + \lambda_1 s_m < b_k < a_k c_m + \lambda_1 s_{m+1}$, L is minimized at $\beta_k = c_m$.

3. $\partial h(\beta_k) = s_m$ for $m = 2, \dots, p-1$ ($c_{m-1} < \beta_k < c_m$):

$$\begin{aligned} a_k\beta_k - b_k + \lambda_1 s_m = 0 &\implies c_{m-1} < \beta_k = \frac{b_k - \lambda_1 s_m}{a_k} < c_m \\ &\iff a_k c_{m-1} + \lambda_1 s_m < b_k < a_k c_m + \lambda_1 s_m. \end{aligned}$$

So if $a_k c_{m-1} + \lambda_1 s_m < b_k < a_k c_m + \lambda_1 s_m$, L is minimized at $\beta_k = \frac{b_k - \lambda_1 s_m}{a_k}$.

4. $\partial h(\beta_k) = s_{p+1}$ ($\beta_k > c_p$):

$$a_k\beta_k - b_k + \lambda_1 s_{p+1} = 0 \implies \beta_k = \frac{b_k - \lambda_1 s_{p+1}}{a_k} > c_p \iff b_k > a_k c_p + \lambda_1 s_{p+1}.$$

So if $b_k > a_k c_p + \lambda_1 s_{p+1}$, L is minimized at $\beta_k = \frac{b_k - \lambda_1 s_{p+1}}{a_k}$.

Combining the four cases results in a sort of ‘multi soft threshold’ for the coordinate update rule for $\beta \neq \mathbf{0}$.

$$\tilde{\beta}_k \leftarrow S_m(b_k) = \begin{cases} \frac{b_k - \lambda_1 s_1}{a_k} & b_k < a_k c_1 + \lambda_1 s_1 \\ c_1 & a_k c_1 + \lambda_1 s_1 < b_k < a_k c_1 + \lambda_1 s_2 \\ \frac{b_k - \lambda_1 s_2}{a_k} & a_k c_1 + \lambda_1 s_2 < b_k < a_k c_2 + \lambda_1 s_2 \\ \vdots & \\ c_m & a_k c_m + \lambda_1 s_m < b_k < a_k c_m + \lambda_1 s_{m+1} \\ \frac{b_k - \lambda_1 s_m}{a_k} & a_k c_m + \lambda_1 s_{m+1} < b_k < a_k c_{m+1} + \lambda_1 s_{m+1} \\ \vdots & \\ c_p & a_k c_p + \lambda_1 s_p < b_k < a_k c_p + \lambda_1 s_{p+1} \\ \frac{b_k - \lambda_1 s_{p+1}}{a_k} & b_k > a_k c_p + \lambda_1 s_{p+1} \end{cases} \quad (3.21)$$

If $\beta = \mathbf{0}$, then $\forall j : r_{jk} = 0$ so $\text{ND} = \{0\}$ and

$$\partial h \beta_k = \begin{cases} -\|G_k\|_1 & \beta_k < 0 \\ [-\|G_k\|_1, \|G_k\|_1] & \beta_k = 0 \\ \|G_k\|_1 & \beta_k > 0 \end{cases}$$

so the the coordinate update is just

$$\tilde{\beta}_k \leftarrow \frac{S(b_k, \lambda_1 \|G_k\|_1)}{a_k} \quad (3.22)$$

where S is the soft threshold function from the original update rule in equation (3.8).

3.9 Warm Start

A warm start is setting the initial weights to close to the final solution so as to reduce computation. This is mainly employed when solving a similar problem beforehand. The potential speedup from a Warm Start can range from none at all to a complete speedup where Coordinate Descent runs for no iterations.

3.9.1 Rolling Window

Coordinate Descent for Aquatic will be employed on time series data in a rolling window fashion. This means that as new data comes into the system, old data is removed in the following way:

$$X_{old} = \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ \vdots \\ X_m \end{pmatrix}$$

$$X_{new} = \begin{pmatrix} X_2 \\ X_3 \\ \vdots \\ \vdots \\ X_{m+1} \end{pmatrix}$$

We can create a new gram matrix like this:

$$\text{Gram}_{new} = X_{old}^T X_{old} - X_1^T X_1 + X_{m+1}^T X_{m+1}$$

The new problem is very similar to the old problem. So, we can utilize a warm start for the new problem.

3.9.2 With a Distributed Network

We can also utilize a warm start when running CD on a distributed network. When using the front-heavy method, we can load in each X_i on a different computer node. After every node has completed calculating their respective $X_i^T X_i$, we can add up all of small grams to create the final gram matrix. Every computer node should theoretically run in the same amount of time. But, in practice, each node will take a different amount of time to compute $X_i^T X_i$ on node i . Once a certain percentage of nodes, we collect their grams and run CD on the subproblem:

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \frac{1}{2N} \|X_{\text{sub}}\beta - y_{\text{sub}}\|_2^2 + \lambda_1 \|\beta\|_1 + \lambda_2 \|\beta\|_2^2 \quad (3.23)$$

where X_{sub} and y_{sub} are made up from the nodes that have completed their calculations. Then, we can use the solution to the subproblem as a warm start for the whole problem after every node has completed its calculation.

Chapter 4

CD Optimization Methods with Accuracy Loss

This chapter describes various techniques that can improve CD computational time with the sacrifice of small amounts of accuracy.

4.1 Computer Precision

The data provided by Aquatic consists of single-precision floats (float32), which stores values with up to 7-8 decimal digits. However, 8 digits of precision are not realistic when using observed equity data, where values are only accurate up to an error of $1e^{-4}$. This limited accuracy means that there is little need to store and use all 7-8 decimal digits in computations. In theory, storing fewer digits is advantageous because 1) the data takes up less space in memory, and 2) the computations no longer need to use all of the digits, leading to possible speedup. The only downside to using lower precision is the loss in computational accuracy. With this promising concept, we looked at several possible methods of implementing lower precision described in Section 5.11.

4.2 Gram Matrix Estimation

Most of the calculations done by the covariance method are inner products between columns of \mathbf{X} . Hence the front heavy method in Section 3.4 derives its name from its initial computation of the Gram matrix doing most of the algorithm's work before the covariance updates even begin. Thus, for example, halving the time to calculate the Gram matrix will almost halve the algorithm's total runtime. Section 3.7.2 discusses a method of calculating the Gram matrix in chunks when available memory is limited. Here we propose a similar idea, but instead of calculating the Gram matrix exactly as in Section 3.7.2, we estimate it by limiting expensive computations to one chunk of \mathbf{X} and extrapolating the rest of \mathbf{X} .

Pearson Correlation Coefficients

Let a and b be two features corresponding to columns \mathbf{x}_a and \mathbf{x}_b of \mathbf{X} . Assuming the columns of \mathbf{X} are standardized such that their means are 0, the Pearson Correlation Coefficient between \mathbf{x}_a and \mathbf{x}_b is

$$r_{ab} = \frac{\langle \mathbf{x}_a, \mathbf{x}_b \rangle}{\sigma_a \sigma_b}$$

where

$$\sigma_z = \sqrt{\sum_{i=1}^N x_{zi}^2}$$

is the \sqrt{N} times the standard deviation of \mathbf{x}_z .

In studies of Aquatic's ≈ 17 million $\times 10$ dataset, we qualitatively observed that r_{ab} for each pair of features a, b is relatively invariant over time. If we divide \mathbf{X} into chunks by rows - say 17 chunks of 1 million rows - r_{ab} calculated using only the rows in the first chunk is approximately equal to r_{ab} calculated using only the rows in the second chunk and so on. The same invariance does not hold for σ_a for a given feature a , however, as some chunks have significantly larger σ_a values than others. This observation, along with the observation that

$$r_{ab} = \frac{\langle \mathbf{x}_a, \mathbf{x}_b \rangle}{\sigma_a \sigma_b} \implies \langle \mathbf{x}_a, \mathbf{x}_b \rangle = r_{ab} \sigma_a \sigma_b$$

motivates a method for estimating the Gram matrix values $\langle \mathbf{x}_a, \mathbf{x}_b \rangle$.

First calculate r_{ab} for each pair of features a, b using only the first chunk of \mathbf{X} . Then for the first chunk and each subsequent chunk calculate σ_a for each column a using only the rows of that chunk. Now for each pair a, b estimate the chunk's value of $\langle \mathbf{x}_a, \mathbf{x}_b \rangle$ as $r_{ab} \sigma_a \sigma_b$, using the r_{ab} calculated for the first chunk. Finally, for each a and b sum the estimated $\langle \mathbf{x}_a, \mathbf{x}_b \rangle$ values for each chunk into an estimated value for all of \mathbf{X} , as in Section 3.7.2. The collection of estimated $\langle \mathbf{x}_a, \mathbf{x}_b \rangle$ values over all pairs of features forms the estimated Gram matrix for \mathbf{X} .

Instead of performing the $O(Np^2)$ time calculation of the inner products of each pair of features over the entire matrix \mathbf{X} , the estimation performs this expensive calculation only for a small fraction of the rows of \mathbf{X} . For the remaining chunks, it only has to calculate σ of each column, requiring just $O(Np)$ time. The results of applying this method to Aquatic's 17 million \times 162 dataset can be viewed in Section 5.12.

4.3 MiniCD

Loading the data is the main bottleneck of our algorithm based on the Front Heavy variant of the Covariance update. That is why we decided to look into methods of solution which do not include the whole data, whose solutions hopefully approximate the ground truth. We developed a general scheme for such a procedure, which we refer to as MiniCD. Our methods in this section were motivated by the literature review that we did on ensembling methods (see 5.4). The general outline of MiniCD is given in 5. Notice that step 1 can be implemented in many ways. For example we can choose the subsets to be non-intersecting or we can allow intersections. There is also flexibility in step 3. as we can vary the lasso and ridge regularizers for each of the subproblems. Step 4 is the most open ended one and there are many methods one can apply. We mainly focused on 2 of them: averaging and voting.

Algorithm 5 MiniCD

- 1: Split the dataset (X, y) by rows into batches $(X_1, y_1), \dots, (X_N, y_N)$.
 - 2: Select m of the chunks $(X_{i_1}, y_{i_1}), \dots, (X_{i_m}, y_{i_m})$
 - 3: Solve elastic net with input data (X_{i_j}, y_{i_j}) , and obtain a solution w_j for each j
 - 4: Ensemble the solutions w_1, \dots, w_m to approximate the ground truth w
-

4.3.1 Averaging

We tried two ways of implementing step 1. First, we split the data evenly into N batches by rows and solved each of them. Secondly, we choose the batches in a random fashion. We implemented both methods and recorded our results. Our methods are described below:

1. We partition the dataset into N equally sized batches of rows S_1, \dots, S_N , and obtain solution estimates w_{S_1}, \dots, w_{S_N} . Then, we averaged the estimates to get an approximation solution $\hat{w} = \frac{1}{N}(\sum_{i=1}^N x_{S_i})$. Then, we compared \hat{w} to the ground truth w by using the metrics euclidean distance, relative distance (dist/norm), and non-zero support accuracy. We also experimented with the number of batches $N = 2, 3, 4, 5, 6, 7, 8, 10, 20, 50$. For each of these values we recorded our results in the table (refer to table). We used $\lambda_1 = 5 * 10^{-5}$ and $\lambda_2 = 10^{-1}$ because those values gave us the desired sparsity of 45 (out of 162) non-zeros in the medium dataset. From our results we can see that as the number of batches increases, both the euclidean and relative distances increase while the support accuracy decreases. This suggests that as we combine more estimates the estimated solution tends to get further away from the ground truth in the above metrics. It should be noted that when averaging these solutions, there is the danger of losing a lot of support accuracy. For example, if an insignificant feature is zero in all of the estimates but one, then it will be non-zero in the predicted average. Another issue with averaging is the fact that it introduces a one-sided bias which we will discuss in the next point.
2. We sampled X equally sized batches of contiguous rows S_1, \dots, S_N of the data X and obtained solution estimates w_{S_1}, \dots, w_{S_N} . Since we are using the Front heavy method, we rely of computing the Gram matrices $X_{S_i}^T X_{S_i}$ and the inner products $X_{S_i}^T y_{S_i}$. When X_{S_i} consists of randomly chosen rows of X , then one can show that $X_{S_i}^T X_{S_i}$ and $X_{S_i}^T y_{S_i}$ are unbiased estimators of $X^T X$ and $X^T y$. Because of that averaging the partial solutions may seem like a tempting idea at first. However, each of the solutions w_i based on the subset S_i incurs a one-sided bias as our industry mentor Dan Shiber from Aquatic pointed out. Therefore, as we average all of the solution the one-sided bias will not cancel and will remain one-sided. To make this more clear mathematically let's consider a simplified version where we only solve Ordinary Least Squares (the argument easily generalizes to Elastic Net). Then $X_{S_i}^T X_{S_i} = X^T X + \epsilon_1$ and $X_{S_i}^T y_{S_i} = X^T y + \epsilon_2$, where $E[\epsilon_1] = E[\epsilon_2] = 0$ and ϵ_1 and ϵ_2 are independent as the estimators are unbiased. For simplicity, ϵ_i for $i = 1, 2$ be univariate noise, since the argument can be easily generalized in the case where the noise is multivariate. Then by the solution of Ordinary Least Squares, we get $E[w_{S_i}] = E[(X^T X + \epsilon_1)^{-1}]E[X^T y + \epsilon_2]$. By above we have that $E[X^T y + \epsilon_2] = X^T y$ as $E[\epsilon_2] = 0$. Now, by doing a Taylor expansion of x^{-1} at $X^T X$ and using the fact that $E[\epsilon_1] = 0$ we see that

$$\begin{aligned} E[w_{S_i}] &= E[(X^T X + \epsilon_1)^{-1}]X^T y = \\ &= ((X^T X)^{-1} - (X^T X)^{-2}E[\epsilon_1] + 2(X^T X)^{-3}E[\epsilon_1^2] + o(\epsilon_1^2))X^T y = \\ &= ((X^T X)^{-1} + 2(X^T X)^{-3}E[\epsilon_1^2] + o(\epsilon_1^2))X^T y = \\ &= (X^T X)^{-1}X^T y + 2(X^T X)^{-3}E[\epsilon_1^2]X^T y + o(\epsilon_1^2)X^T y = \\ &= w + 2(X^T X)^{-3}E[\epsilon_1^2]X^T y + o(\epsilon_1^2)X^T y \end{aligned}$$

Notice that for small ϵ_1 the term $2(X^T X)^{-3}E[\epsilon_1^2]o(\epsilon_1^2)$ is always strictly positive-definite and dominates the term $o(\epsilon_1^2)X^T y$. Therefore, there will always be a one-sided finite-sample bias. It is important to note here that this doesn't contradict

the Central Limit Theorem. As the number of observations approaches infinity, we get that $E[\epsilon_1^2]$ goes to 0. However, the estimator w_{S_i} approaches the limit only from one side. Thus, averaging the solution estimates, will necessarily introduce a bias. It would be helpful if we can empirically understand what the bias exactly is and correct for it by adding a constant. This lead us to exploratory data analysis of many randomly generated solutions w_i and how the feature entries are exactly distributed with respect to the true values. Figure 4.1 shows the magnitudes of all active (non-zeros) coefficients in the ground truth solution against the magnitudes of the coefficients of the estimated obtained by just taking the mean of w_1, \dots, w_n . The plot was made for the Large dataset and we used all 493 days for our batches, where each batch corresponds to 33,000-35,000 rows of the data. As we can see in the plot there the values for the ground truth solutions seem to be consistently bigger than the ones for the mean of the partial estiamtes. This empirically demonstrates the presence of a one-sided bias. Notice that as the importance of the feature decreases the bias becomes less significant and it hard to see. Furthermore, in Figure 4.2 we plotted histograms of the estimated values for each of the top 6 features along of their estimated coefficients from the partial solutions. We can clearly see that there is a one-sided bias of in those weights, whose side depends on the sign of the true weights. This could be due to the fact that we did not sample random days, but simply used all of the says or due simply because we don't have enough samples. Nevertheless, the plot is convincing enough to show that some form of a bias exists, confirming our theoretical considerations above. Unfortunately, we couldn't estimate this bias successfully and couldn't correct for it. Thus, we decided to not pursue averaging partial solutions any further.

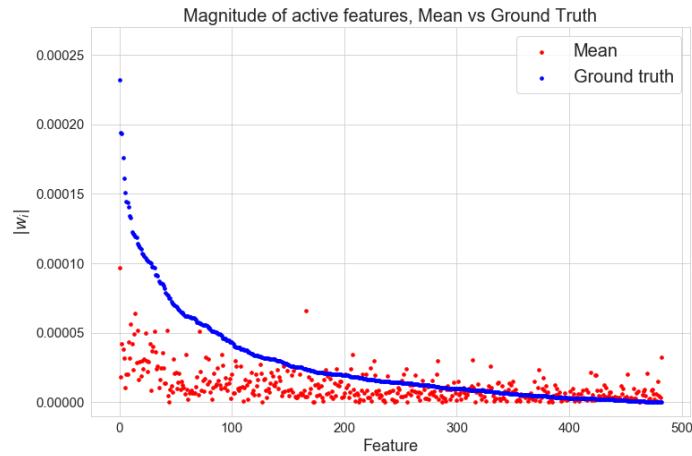


Figure 4.1: Magnitude of the active features in the ground truth sorted in a decreasing fashion and the estimate obtained by taking the mean of the partial solutions. We can see a form of one-sided bias for the bigger features.

4.4 Ensemble Methods

4.4.1 Literature Review

In many machine learning problems, ensemble methods combine solutions of different models to improve the accuracy over a single estimator. As a motivation for the ensembling methods

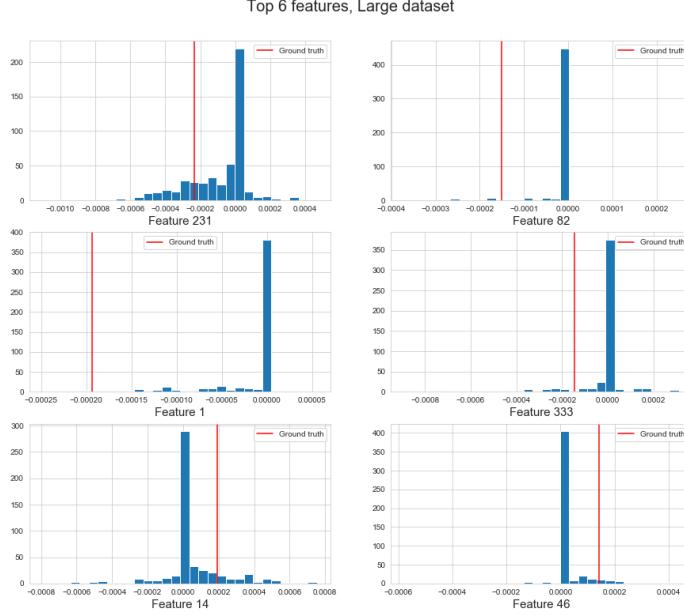


Figure 4.2: The top 6 features in magnitude for the Large dataset. The red vertical lines represent the true values of the coefficient and the blue histogram represents the distribution of the coefficient over the partial solution for each of the 493 days.

we used in our studies we did a general literature review of various ensembling methods. Chapter 8 of [5] provides a great overview of ensembling techniques and introduces the following methods:

1. Bagging: Consider a regression problem. Suppose we fit a model to our training data $\mathbf{Z} = (x_1, y_1), \dots, (x_N, y_N)$, obtaining the prediction $\hat{f}(x)$ at input x . Bootstrap aggregation or bagging averages this prediction over a collection of bootstrap samples, thereby reducing its variance. For each bootstrap sample \mathbf{Z}^{*b} , $b = 1, 2, \dots, B$, we fit our model, giving prediction $\hat{f}^{*b}(x)$. The bagging estimate is defined by $f_{bag}(x) = \frac{1}{B} \sum_{b=1}^B \hat{f}^{*b}(x)$. Denote by $\hat{\mathcal{P}}$ the empirical distribution putting equal probability $\frac{1}{N}$ on each of the data points (x_i, y_i) . In fact the "true" bagging estimate is defined by $E_{\hat{\mathcal{P}}} \hat{f}^*(x)$, where $\mathbf{Z}^* = (x_1^*, y_1^*), \dots, (x_N^*, y_N^*)$ and each $(x_i^*, y_i^*) \sim \hat{\mathcal{P}}$. Then the above expression is a Monte Carlo estimate of the true bagging estimate as $B \rightarrow \infty$. Bagging can dramatically reduce the variance of unstable procedures like trees, leading to improved prediction. A simple argument shows why bagging helps under squared-error loss, in short because averaging reduces variance and leaves bias unchanged. Assume our training observations (x_i, y_i) , $i = 1, \dots, N$ are independently drawn from a distribution \mathcal{P} , and consider the ideal aggregate estimator $f_{ag}(x) = E_{\mathcal{P}} \hat{f}^*(x)$. Here x is fixed and the bootstrap dataset \mathbf{Z}^* consists of observations x_i^*, y_i^* , $i = 1, 2, \dots, N$ samples from \mathcal{P} . Note that $f_{ag}(x)$ is a bagging estimate, drawing bootstrap samples from the actual population \mathcal{P} rather than the data. It is not an estimate that we can use in practice, but is convenient for analysis. We can write

$$\begin{aligned} E_{\mathcal{P}}[Y - \hat{f}^*(x)]^2 &= E_{\mathcal{P}}[Y - f_{ag}(x) + f_{ag}(x) - \hat{f}^*(x)]^2 = \\ &= E_{\mathcal{P}}[Y - f_{ag}(x)]^2 + E_{\mathcal{P}}[\hat{f}^*(x) - f_{ag}(x)]^2 \geq \\ &\geq E_{\mathcal{P}}[Y - f_{ag}(x)]^2 \end{aligned}$$

The extra error on the right-hand side comes from the variance of $\hat{f}^*(x)$ around its mean $f_{ag}(x)$. Therefore true population aggregation never increases mean square error. This

suggests that bagging- drawing samples from the training data - will often decrease mean-squared error.

2. Model Averaging: In this method let's have predictions $\hat{f}_1(x), \dots, \hat{f}_M(x)$, under squared-error loss, we can seek the weights $w = (w_1, \dots, w_M)$ such that

$$\hat{w} = \operatorname{argmin}_w E_{\mathcal{P}}[Y - \sum_{m=1}^M w_m \hat{f}_m(x)]^2 \quad (4.1)$$

Here the input value x is fixed and the N observations in the dataset \mathbf{Z} (and the target Y) are distributed according to \mathcal{P} . The solution is the population linear regression of Y on $\hat{F}(x)^T = [\hat{f}_1(x), \dots, \hat{f}_M(x)]$:

$$\hat{w} = E_{\mathcal{P}}[\hat{F}(x)\hat{F}(x)^{-1}]^T E_{\mathcal{P}}[\hat{F}(x)Y] \quad (4.2)$$

Now the full regression has smaller error than any single model

$$E_{\mathcal{P}}[Y - \sum_{m=1}^M w_m \hat{f}_m(x)]^2 \leq E_{\mathcal{P}}[Y - \hat{f}_m(x)]^2 \quad \forall m \quad (4.3)$$

3. Stacking: Since model averaging can suffer from overfitting due to some of the models having a much higher complexity than others. Stacking deals with this issue. We denote by $\hat{f}_m^{-i}(x)$ to be the prediction at x , using model m , applied to the dataset with the i -th observation removed. The stacking estimate of the weights is obtained from the least squares linear regression of y_i on $\hat{f}_m^{-i}(x_i)$, $m = 1, 2, \dots, M$. In detail the stacking weights are given by

$$\hat{w}^{st} = \operatorname{argmin}_w \sum_{i=1}^N [y_i - \sum_{m=1}^M w_m \hat{f}_m^{-i}(x_i)]^2 \quad (4.4)$$

The final prediction is $\sum_m \hat{w}^{st}$. By passing the cross-validated predictions $\hat{f}_m^{-i}(x)$, stacking avoids giving unfairly high weight to models with higher complexity. Better results can be obtained by restricting the weights to be nonnegative, and to sum to 1. There is a close connection between stacking and model selection via leave-one-out cross validation. The stacking idea is actually more general than described above. One can use any learning method, not just linear regression, to combine models and the weights could also depend on the input location of x .

4. Stochastic Search (Bumping): This method does not involve averaging or combining models, but rather is a technique for finding a better single model. Bumping uses bootstrap sampling to move randomly through model space. For problems where fitting method finds many local minima, bumping can help the method to avoid getting stuck in poor solutions. As in bagging, we draw bootstrap samples and fit a model to each. But rather than average the predictions, we choose the model estimated from a bootstrap sample that best fits the training data. In detail, we draw bootstrap samples $\mathbf{Z}^{*1}, \dots, \mathbf{Z}^{*B}$ and fit our model to each, giving predictions $\hat{f}^{*b}(x)$, $b = 1, 2, \dots, B$ at input x . We then choose the model that produces the smallest prediction error, averaged over the original training set. For squared error, for example, we choose the model obtained from bootstrap sample \hat{b} , where

$$\hat{b} = \operatorname{argmin}_b \sum_{i=1}^N [y_i - \hat{f}^{*b}(x_i)]^2$$

The corresponding model predictions are $\hat{f}^{*b}(x)$.

4.4.2 Ensembling with MiniCD

By using the MiniCD concept explained in Section 4.3, we can devise sets of estimates based on random samples of the data. To avoid the bias described in the previous subsection that arises when averaging these estimates, we can use ensemble methods to determine which features play an important role in obtaining ground truth to the minimization problem, which requires the entire dataset. Dependent on lasso and ridge penalty values, the weights considered as ground truth can be extremely sparse. The weights equal to zero provide no numerical impact to the solution, and therefore, are not essential to the overall computation of CD. By removing these weights from the computation, the dimensions of the problem is reduced and consequently the time of CD is reduced. If a large number of non-impactful weights are removed in a short amount of time, there is possible speedup to the overall minimization problem without losing significant amounts of accuracy.

In these ensemble methods, voting is a common scheme to combine different solutions. In this report, we implemented two voting methods, threshold and tree based voting, to estimate which features are nonzero (active) and zero (inactive) in the ground truth.

4.4.3 Threshold Voting

Let's suppose that our partial estimates obtained in step 3 in Algorithm 5 are stored in the rows of a matrix E with m rows and p columns, where p is the number of features. In order to use threshold voting we need to specify a threshold $t \in (0, 1)$ beforehand. The process of thresholding voting is shown in Algorithm 6.

Algorithm 6 Threshold voting

- 1: **Input:** threshold value $t \in (0, 1)$.
 - 2: Initialize the set of active features $A = \emptyset$
 - 3: **for all** $k \in \{1, 2, \dots, p\}$ **do**
 - 4: - Take the k -th column of E_k and compute the number of $j \in [1, m]$ such that $E_{jk} \neq 0$, let this number be N_k
 - 5: - Compute the percent active of feature k , which is $\frac{N_k}{m}$
 - 6: - If $\frac{N_k}{m} > t$, then add feature k to A
 - 7: **end for**
 - 8: Run Coordinate Descent only on the set of selected features A and produce a solution \hat{w}
-

For each feature we compute the percent of times it is active amongst the partial solutions. If this percentage is bigger than a given threshold, then we claim this feature is active and otherwise we say that it is not active. Once, we have determined which features are active and which are not, we run CD only on the features which have been claimed active. This method is promising since it is easily parallelizable and can potentially leave out many unimportant features, which will decrease computation time and make our algorithm more efficient, while sacrificing some accuracy in case we don't completely capture the true active set of features.

In Figure 4.3 we have plotted the percent active vs the magnitude for each feature in the Large dataset by using $m = 100$ batched (days). As we can see, there seems to be a mild correlation between the magnitude of the active features and their percent active which is a promising sign. However, we can also see from the plot that the active and zero

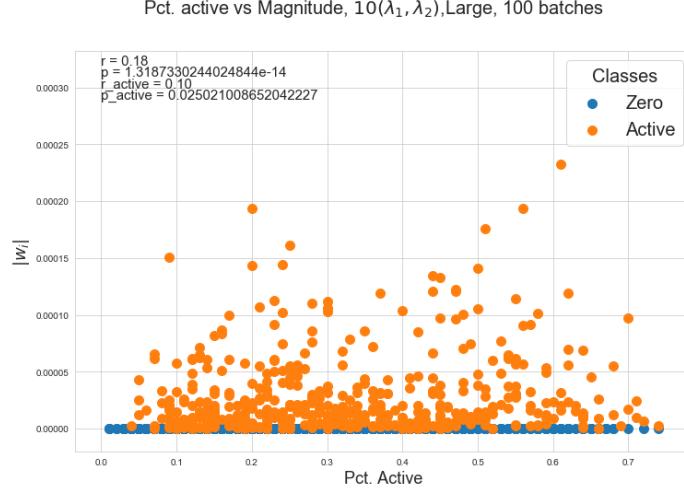


Figure 4.3: Percent active vs magnitude of a coefficient for the Large dataset. 100 batches (days) of size 35,000 were used to compute the percent active of each feature.

features are highly unseparable by a threshold (a vertical line), which indicates that we are not very likely to both capture the true support and leave out the zero features. This is also supported by the trade-off between recall and precision that we see in the results.

4.4.4 Tree Voting

The other voting scheme used in this report is ternary tree-based voting. For explanation purposes, this method considers one feature at a time. However, in practice, like the threshold voting scheme described in the previous subsection, this voting method is easily parallelizable.

For a given feature k , this method takes S_k and assigns a node to each value in the vector. Initial ordering of node assignment is not necessary as each value is calculated via a random sampling of data through the MiniCD method. Once each value has a node, the following algorithm can be used to determine whether the feature is active or inactive:

It is important to note that the tree voting mechanism will always use a majority based method for consolidating nodes only when the voting starts with m nodes, where m is divisible by a power of 3. Otherwise, there can be slight bias based on how many nodes there are. This bias can be shown in Figure 4.4. In this case, the last node with value 0 is compared to the condensed node with value 1 based on 9 original nodes. Using a majority vote here would result in large bias as 1 node would have equal weight to the remaining 9. As such, Line 8 of Algorithm 7 chooses the left node over the right node when comparing two nodes. This is a fair rule to define as the trees by design will either be symmetric or left-sided. Additionally, this method can be easily expanded to a random forest structure, where each node can be a tree itself.

4.5 Early Stopping

Early stopping is a method of stopping an optimization algorithm early so as to reduce overfitting. However, it also can increase speed since the algorithm stops before unnecessary computation. Early stopping is completely determined by the problem space and is determined by a tolerance parameter c . We can stop CD after the duality gap (η) is smaller than

Algorithm 7 Tree Voting Algorithm

```
1: assign each node a binary value, 0 if existing value = 0 and 1 if existing value ≠ 0
2: for all  $k \in \{1, 2, \dots, p\}$  do
3:   Take  $S_k$ 
4:   repeat 
5:     if 3 consecutive nodes exist then
6:       consolidate to 1 node
       where value determined through majority value of the 3 nodes
7:     else if 2 consecutive nodes exist then
8:       consolidate to 1 node based on majority value of the 2 nodes
       if the two nodes have different values, then use the first node's value
9:     else
       no consolidation needed
10:    end if
11:    until 1 node exists, which is the final node
12: end for
```

c. Calculating the duality gap is computationally intensive compared to the computations needed for the rest of CD. If we could cut out all or even some of the duality gap calculation, then we could significantly reduce the computation necessary. We start checking the duality gap once the estimated weights stop changing that much from iteration to iteration. So, one way to reduce the duality gap computation would be to start calculating it as late as possible. Another way to reduce the computation of the duality gap would be to determine beforehand analytically how many iterations CD will need to run for. However, this runs of the risk of not running CD for long enough to converge and thus yielding a lower accuracy than we would like.

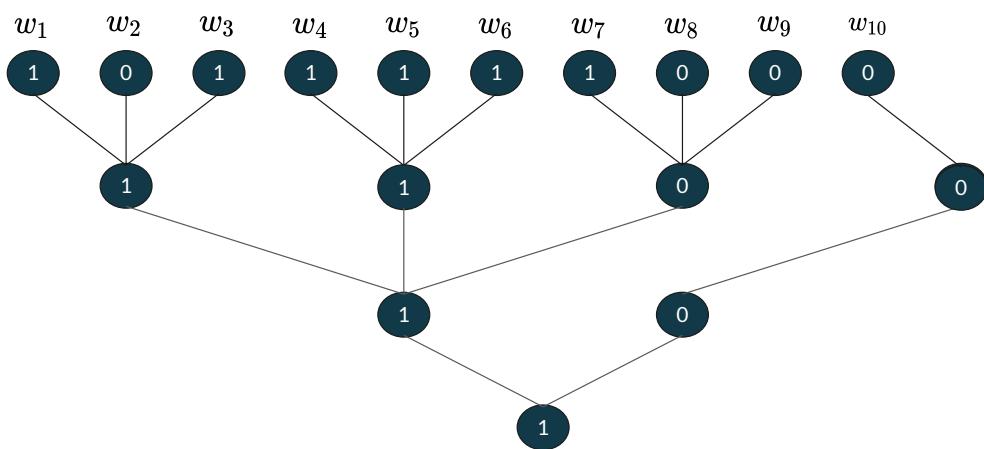


Figure 4.4: Arbitrary case with 10 initial nodes such that the node from w_{10} would have much stronger impact on the final decision without line 8 of Algorithm 7.

Chapter 5

Results

5.1 Hardware and Data Sepcifications

The studies for this project used 4 Intel(R) Xeon(R) Bronze 3104 CPUs, with 16 GB of RAM. The strict limitations on hardware for this problem lead to two different optimizations, based on whether the computational environment is memory-free or memory-bound. The problem becomes memory-bound when the data cannot be loaded into an algorithm all at once due to memory shortages and must be broken up into parts. For computing CD in a memory-free environment, we considered an input matrix of dimensions $17,281,517 \times 10$ (*small*). For a memory-bound environment, we used input data of sizes $17,281,517 \times 162$ (*medium*) and $17,192,783 \times 1768$ (*large*).

5.2 Parameter Study for λ_1 and λ_2 values

For each of our datasets we had to choose typical values of the λ_1 and λ_2 parameters in order to achieve a desired sparsity provided by Aquatic. For the Medium dataset this is around 30-50 (18%-31%) non-zero coefficients and for the Large dataset it is 300-500 (17%-28%) non-zero coefficients. To find the right λ_1 and λ_2 we performed a grid search for different values of λ_1 and λ_2 and recorded the number of non-zeros for each chosen pair. The results from our grid search are presented in 5.10. Based on the table and on Aquatic's recommendation the pair we chose the pair $(5 * 10^{-5}, 10^{-1})$ as our default parameters on the Medium dataset in our future studies.

λ_2	1e0	1e-1	1e-2	1e-3	1e-4
λ_1					
1e-4	0.19	0.13	0.1	0.1	0.1
5e-5	0.37	0.27	0.25	0.2	0.2
2*e-5	0.6	0.49	0.44	0.44	0.44
1.43*e-5	0.7	0.56	0.48	0.46	0.46
1e-5	0.75	0.66	0.56	0.55	0.54
1e-6	0.96	0.93	0.91	0.88	0.87
1e-7	0.99	0.98	0.98	0.99	0.99

Table 5.1: Proportion of non-zero coefficients in the ground truth for the Medium dataset across different pairs of λ_1 and λ_2 regularizers.

$\lambda_1 \backslash \lambda_2$	1e-1	5e-2	1e-2	5e-3	1e-3	5e-4	1e-4
1e-4	0.04	0.03	0.03	0.03	0.03	0.03	0.03
5e-5	0.09	0.07	0.06	0.05	0.05	0.05	0.05
1e-5	0.37	0.33	0.27	0.26	0.25	0.25	0.25
5e-6	0.55	0.5	0.43	0.42	0.39	0.39	0.39
1e-6	0.86	0.85	0.78	0.75	0.73	0.73	0.72
5e-7	0.93	0.91	0.88	0.86	0.84	0.83	0.82
1e-7	0.98	0.97	0.96	0.97	0.95	0.95	0.95

Table 5.2: Proportion of non-zero coefficients in the ground truth for the Large dataset across different pairs of λ_1 and λ_2 regularizers.

Based on Tables 5.1 and 5.2, we determined λ_1 and λ_2 values to be used for the remainder of the studies. For the Medium and Large datasets we used $[\lambda_1, \lambda_2] = [5e-5, 1e-1]$ and $[\lambda_1, \lambda_2] = [1e-5, 1e-2]$ respectively.

When implementing the voting thresholding method, we wanted to use values of (λ_1, λ_2) for step 3 of 5 so that the sparsity of the partial solutions matches the sparsity of the ground true solutions. We wanted this to hold in order for the voting thresholding method to not be biased to select less features than needed or more features than needed, as would be the case if the number of active coefficients in the partial solutions is lower or higher than in the ground truth. For both the medium and large datasets, we started from the default values of (λ_1, λ_2) , which we determined above and varied them until we achieved the desired sparsity. We used a batch size of around 35,000 rows for both Medium and Large. Based on our results shown in Figure 5.1 we decided to choose values $[\lambda_1, \lambda_2] = [3*5e-5, 3*1e-1]$ for Medium and $[\lambda_1, \lambda_2] = [10*1e-5, 10*1e-2]$ for Large.

5.3 MiniCD results

We solved the elastic net problem for each of the 493 days in the Large dataset and obtained 493 estimates. We wanted to visualize how does the support of these partial solutions vary across different days. This is illustrated in Figure 5.2. As we can see from the figure, there are some features (e.g. the ranges [1647,1755],[50,130],[320,400]), which are persistently inactive across the partial solutions. In terms of the overall support overlap, we can see that the support set tends to be almost invariant over time, which is an indicator that methods such as voting thresholding would work well, since even we sample a small subset of 10 days, the percent actives should remain close to the total percent actives.

We also did exploratory data analysis on how the individual estimates of the MiniCD framework compared to the ground truth solution for the medium dataset. We used two metrics: MSE and non-zero accuracy (support accuracy). We define non-zero accuracy as the number of matches between the partial coefficients and the ground truth based on whether a coefficient is active or not. We also varied the batch size as $c * 17281517$ for $c = 0.002, 0.01, 0.02, 0.05, 0.2, 0.33, 0.5$. The results from our studies are shown in Figure 5.3 and Figure 5.4 for MSE and non-zero accuracy respectively. As a general trend from the figures we can see that as we gradually increase the batch size the MSE drops and the non-zero accuracy increases. This is expected since as we have more rows in our batch our training set gets closer and closer to the full training set. For MSE the values range from

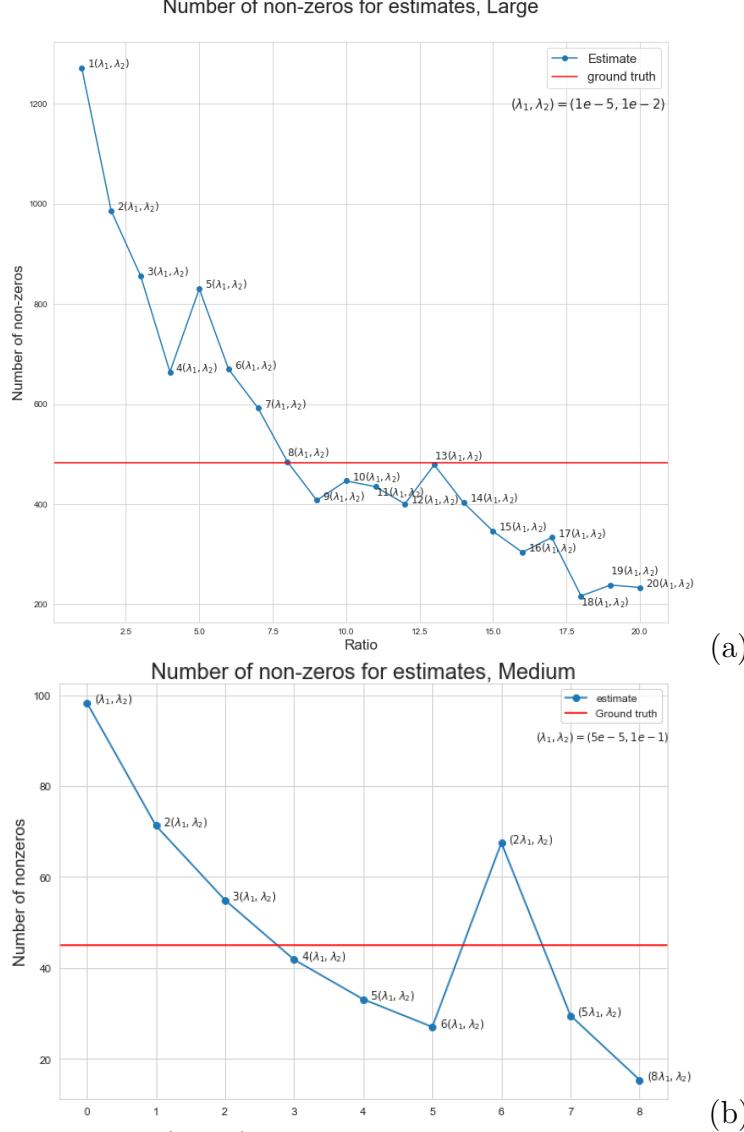


Figure 5.1: Search for suitable (λ_1, λ_2) for the desired sparsity in the partial solutions on the Large dataset.

0.02 for $c = 0.5$ to 1.75 for $c = 0.002$, and for non-zero accuracy the values range from 0.3 for $c = 0.002$ to 0.9 for $c = 0.5$. We can also see that as we decrease the batch size there appears to be more variability in the MSE. For the non-zero accuracy we observe a different behaviour. For the extreme values of the batch size there is little variability and the amount of variability peaks in the middle values ($c = 0.02, 0.05, 0.01$).

After looking at these plots we were also interested in exploring how the support sets overlap across all the partial weight estimates. This is illustrated in Figure 5.5. Each of the 4 subfigures correspond to different batch sizes of $c * 17281517$ resulting from $c = 0.002, 0.02, 0.2, 0.5$. The x-axis in each subplot represents the features and the y-axis the partial weight estimates derived from the batches. We have separated the heatmaps according to zero and non-zero features in the ground truth. We can notice that in all of the 4 cases the support matching support sets tend to be approximately invariant across the estimates. Additionally, we can see that as we increase the batch size the support accuracy of the estimates at the zeros increases, while the accuracy for the non-zeros tends to de-

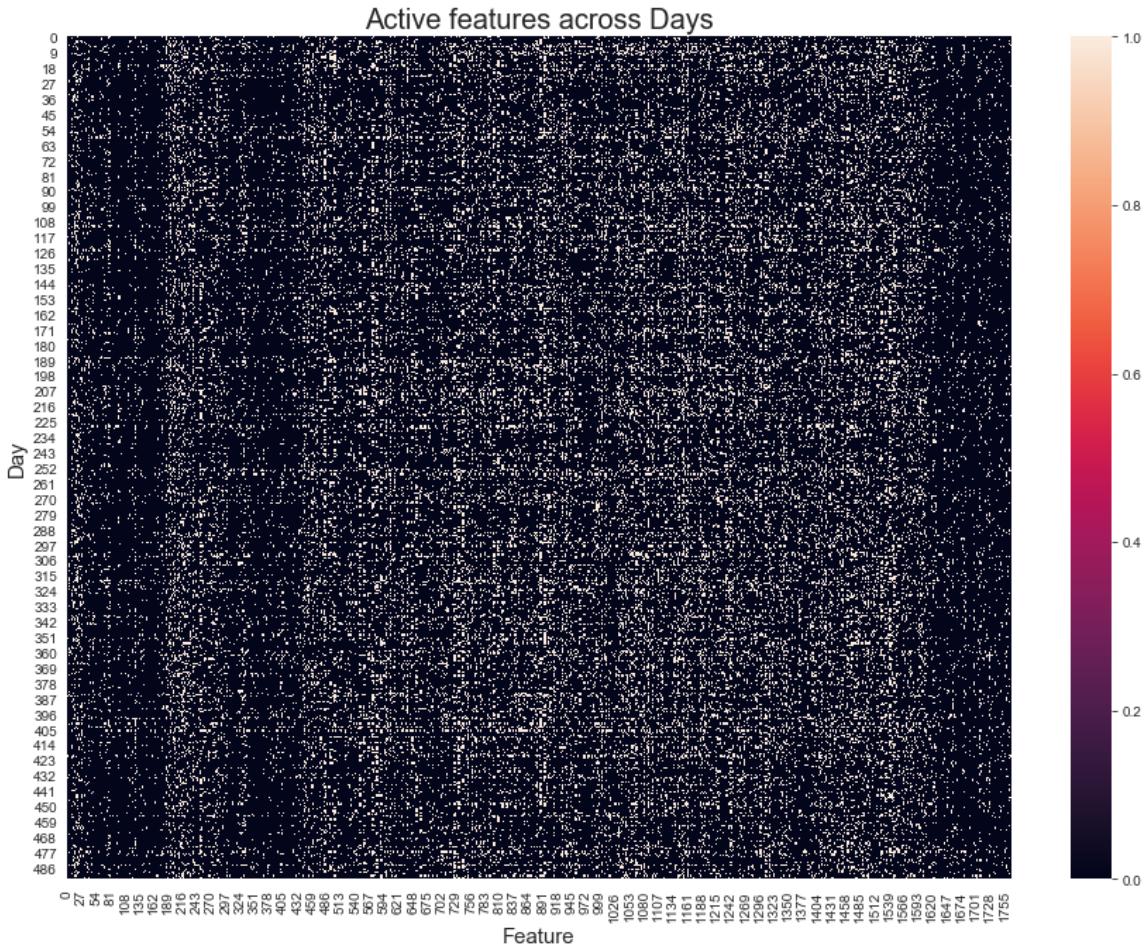


Figure 5.2: Active features across days in the Large dataset. A white square indicates the a feature is active and a dark square indicates that a feature is inactive(zero)

crease. This is expected as the higher the batch size, the closer our training set is to the full training set and thus the solutions will be more similar.

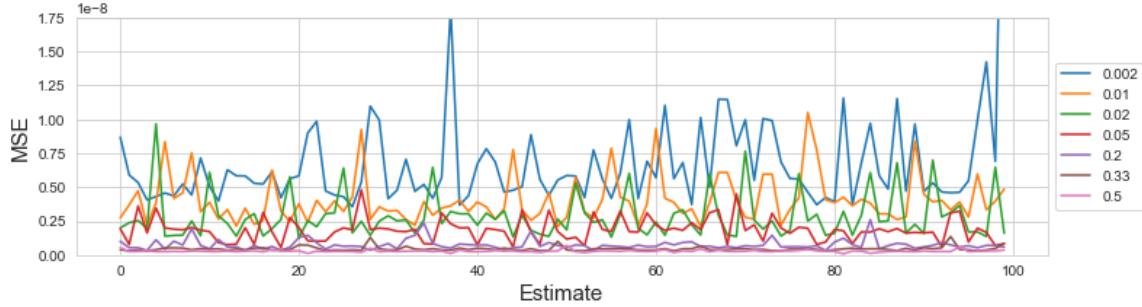


Figure 5.3: Lineplot of estimate vs MSE between the estimate and the ground truth solution over different batch sizes on the Medium dataset.

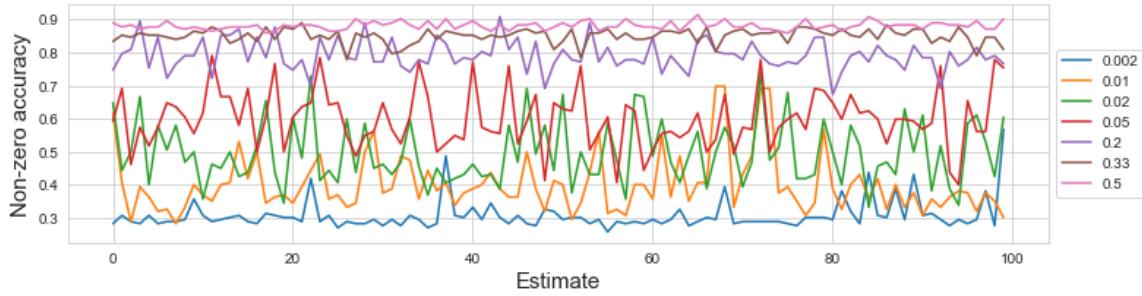


Figure 5.4: Lineplot of estimate vs non-zero accuracy (support accuracy) between the estimate and the ground truth solution over different batch sizes on the Medium dataset.

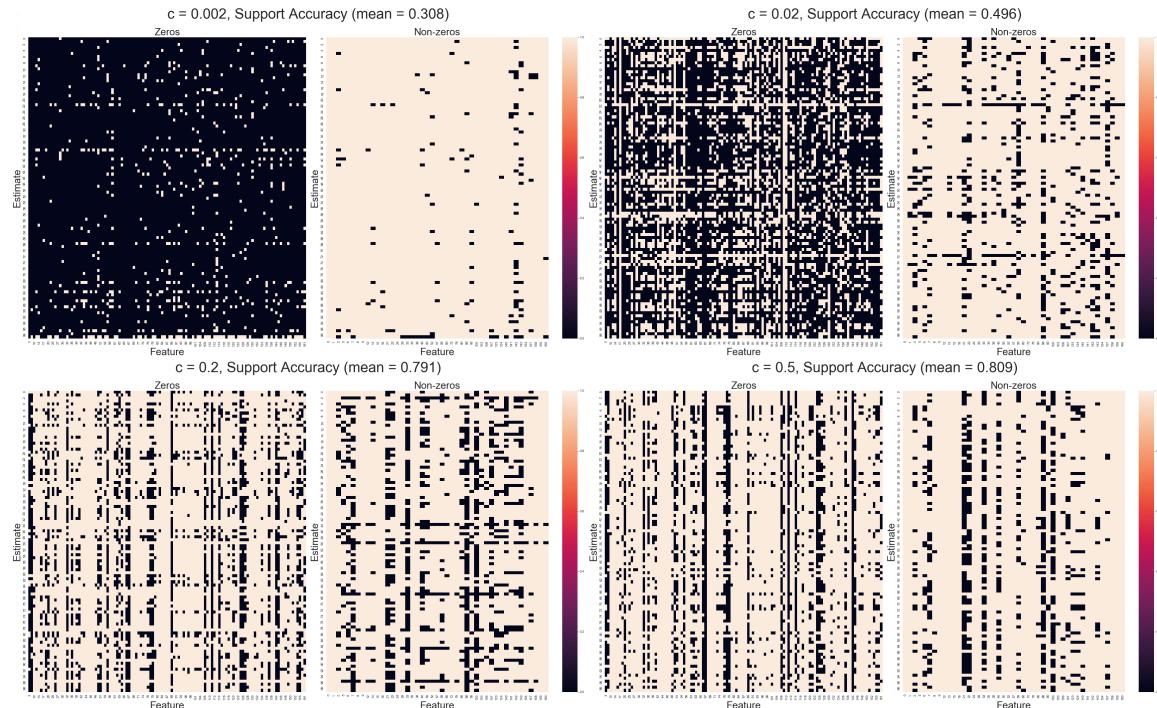


Figure 5.5: Support accuracy across batch sizes for $c = 0.002, 0.02, 0.2, 0.5$ on the Medium dataset. The x-axis is the feature and y-axis the partial weight estimate. A white square indicates a match, and a black a mismatch.

5.4 Ensembling methods results

We performed a comprehensive study of both the threshold voting (Section 4.4.3) and the tree voting (Section 4.4.4) mechanisms on the Large dataset. For our threshold voting study we used 8 values for number of batches (days) $m = 1, 3, 5, 10, 15, 20, 50, 100$ and the threshold value $t = 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9$. For our voting study we used 8 values for number of batches (days) $m = 1, 3, 5, 10, 15, 20, 50, 100$. We recorded 6 metrics:

1. Number of active features determined by the voting. The results are shown in Table 5.3.
2. The precision metric of the set of active features determined by the voting with respect to the true set of active features from the ground truth. ($\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$). The results are shown in Table 5.4.
3. The recall metric of the set of active features determined by the voting with respect to the true set of active features from the ground truth. ($\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$). The results are shown in Table 5.5.
4. The mean squared error (MSE) between the predictions given by the estimate given in step 8 of 6 and the target y . The quantities shown in Table 5.6 are multiplied by $1e5$ and rounded to the 3rd digit after the decimal. The corresponding MSE of the ground truth is 32.973 for comparison.
5. The quantity $\frac{|w - w^*|}{|w^*|}$, where w is the estimate given by step 8 of 6 and w^* is the ground truth. We can interpret this quantity as how close the estimated weights are to the ground truth with respect to the magnitude of the ground truth. The results are shown in Table 5.7.
6. The quantity $(w - w^*)^T \Sigma (w - s^*)$, where $\Sigma = \frac{1}{N} X^T X$ is the empirical covariance matrix. This quantity tells us the mean square error between the predictions given by w and w^* , which are Xw and Xw^* respectively. The quantities shown in Table 5.8 are multiplied by $1e5$ and rounded to the 3rd digit after the decimal.

We measured all of the metrics by repeating the random experiment for each pair (m, t) 5 times and averaging the results.

5.4.1 Ensembling methods: Threshold Voting

Threshold (t)	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	Tree
Batches (m)										
1	409	409	409	409	409	409	409	409	409	
3	976	976	976	351	351	351	73	73	73	
5	1292	750	750	345	345	105	105	20	20	
10	1186	845	554	337	186	90	33	8	1	
15	1470	946	710	340	227	83	40	3	0	
20	1320	889	545	295	149	60	17	3	0	
50	1523	1048	648	344	171	62	13	0	0	
100	1484	989	581	301	151	50	5	0	0	

Table 5.3: Number of active features for varying thresholds and batch sizes in Large dataset

Threshold (t)	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Batches (m)									
1	0.351	0.351	0.351	0.351	0.351	0.351	0.351	0.351	0.351
3	0.313	0.313	0.313	0.355	0.355	0.355	0.482	0.482	0.482
5	0.295	0.333	0.333	0.381	0.381	0.413	0.413	0.431	0.431
10	0.313	0.342	0.376	0.402	0.436	0.471	0.482	0.372	0.6
15	0.292	0.338	0.363	0.412	0.448	0.454	0.46	0.555	0.5
20	0.307	0.352	0.4	0.449	0.48	0.495	0.483	0.6	1.0
50	0.294	0.332	0.379	0.447	0.506	0.487	0.545	0.0	0.0
100	0.297	0.34	0.391	0.474	0.507	0.496	0.5	0.0	0.0

Table 5.4: Support precision for varying thresholds and batch sizes, Large dataset

Threshold (t)	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Batches (m)									
1	0.298	0.298	0.298	0.298	0.298	0.298	0.298	0.298	0.298
3	0.633	0.633	0.633	0.258	0.258	0.258	0.073	0.073	0.073
5	0.791	0.518	0.518	0.273	0.273	0.09	0.09	0.018	0.018
10	0.769	0.599	0.432	0.281	0.168	0.087	0.033	0.006	0.001
15	0.889	0.662	0.534	0.291	0.211	0.078	0.038	0.004	0.0
20	0.84	0.649	0.451	0.275	0.148	0.061	0.017	0.003	0.0
50	0.93	0.722	0.509	0.319	0.18	0.063	0.014	0.0	0.0
100	0.913	0.698	0.471	0.296	0.159	0.051	0.005	0.0	0.0

Table 5.5: Support recall for varying thresholds and batch sizes, Large dataset

Threshold (t)	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Batches (m)									
1	33.016	33.016	33.016	33.016	33.016	33.016	33.016	33.016	33.016
3	32.983	32.983	32.983	33.017	33.017	33.017	33.084	33.084	33.084
5	32.976	32.989	32.989	33.022	33.022	33.077	33.077	33.116	33.116
10	32.978	32.985	33.0	33.028	33.061	33.089	33.104	33.121	33.125
15	32.974	32.983	32.991	33.026	33.045	33.086	33.099	33.123	33.126
20	32.976	32.983	32.997	33.03	33.061	33.099	33.119	33.125	33.126
50	32.973	32.98	32.998	33.029	33.061	33.104	33.124	33.126	33.126
100	32.974	32.981	33.003	33.036	33.063	33.111	33.125	33.126	33.126

Table 5.6: MSE between predictions and target for varying thresholds and batch sizes, Large dataset

Threshold (t)	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Batches (m)									
1	1.062	1.062	1.062	1.062	1.062	1.062	1.062	1.062	1.062
3	0.691	0.691	0.691	1.088	1.088	1.088	1.089	1.089	1.089
5	0.451	0.766	0.766	1.028	1.028	1.069	1.069	1.025	1.025
10	0.579	0.753	0.902	0.98	1.023	1.05	1.022	1.01	1.002
15	0.283	0.618	0.764	0.958	1.015	1.054	1.035	1.006	1.0
20	0.398	0.66	0.844	0.969	1.011	1.005	1.016	1.0	1.0
50	0.269	0.578	0.825	0.953	1.01	1.013	1.006	1.0	1.0
100	0.269	0.519	0.84	0.886	0.978	0.999	1.0	0.999	1.0

Table 5.7: $\frac{|w - w^*|}{|w^*|}$ for varying thresholds and batch sizes, Large dataset

Threshold (t)	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Batches (m)									
1	51.761	51.761	51.761	51.761	51.761	51.761	51.761	51.761	51.761
3	21.739	21.739	21.739	52.222	52.222	52.222	91.892	91.892	91.892
5	12.303	28.656	28.656	56.912	56.912	88.68	88.68	106.552	106.552
10	13.929	25.057	39.588	60.864	80.661	94.245	101.5	108.799	110.719
15	7.773	22.122	32.084	60.705	72.193	93.357	99.169	109.905	110.958
20	11.444	22.994	38.176	63.34	81.046	99.345	108.292	110.557	110.962
50	5.424	19.179	39.105	64.032	82.566	101.923	110.167	110.953	110.966
100	6.9	21.18	43.712	68.295	83.295	104.781	110.54	110.966	110.966

Table 5.8: $(w - w^*)^T \Sigma (w - s^*)$ for varying thresholds and batch sizes, Large dataset

5.5 Early Stopping Study

In Section 4.5, we introduced how we can reduce computation of the duality gap by either precomputing how many iterations CD will run for or by checking the duality gap much later. After a thorough review of existing literature, all theoretical bounds for the iterations of CD in both cyclic and random variants are too large to be of use. For instance, the theoretical iteration bound of the random CD for the Large dataset is on the order of millions whereas we are able to converge in around 120 iterations.

Originally, we started checking the duality gap once the maximum absolute difference between weight values was below 0.0001. For the Large dataset, with this value, we are able to run CD (random with replacement) in 0.77 seconds and 120 iterations. We check the duality gap 120 times meaning that the changes in weights does not exceed 0.0001. However, when we change this tolerance to 0.0000001, we achieve a time of 0.33 seconds and 120 iterations. With this new tolerance, we only check the duality gap on average 2 times. So, from the speedup, the duality gap took up around half of the computation. When we lower the threshold even further to 0.00000001, we run for too many iterations (200), but we still run CD in 0.4 seconds which is still much faster than the 0.77 seconds we had before. With this method we run the risk of starting to check the duality gap too late. However, the computational cost of even 70 more iterations without checking the duality gap is not as much as checking the duality gap too soon. So, at best, we can achieve a $2 \times$ speedup over checking the duality gap too early.

5.6 Study of Feature Choice Methods

5.6.1 Configuration of Hybrid Feature Selection Method

The structure of a hybrid feature selection method described in Section 3.2.3 does not specify what methods to conglomerate and when to switch from one method to another. Because of this, we conducted several studies to determine an optimal hybrid configuration.

For the hybrid feature selection method implemented in this report, the switch between one method to another takes place once the largest gradient value in magnitude reaches a certain threshold. This gradient comparison method is superior to an alternative method comparing the largest magnitude of the weight changes per iteration as it can switch methods mid-iteration, where the largest magnitude of the weights can only be gathered every iteration, restricting any switch from one method to another to occur once an iteration has finished.

Table 5.9 shows performance times of variant hybrid methods, all starting with greedy feature selection switching the different basic feature selection methods at different threshold values. The result show that the performance times plateau as the magnitude of the threshold value increases. As the threshold value increases, CD runs through less iterations before the switch from a greedy to basic feature selection method occurs. This overall decrease in time when using less of a greedy feature selection method suggests that calculating the gradient at each iteration is a bottleneck of the computation. The results also show that using random with replacement results in the fastest hybrid computation times. The studies in the following section will use the fastest hybrid method from Table 5.9; a feature selection method alternates from greedy selection to random with replacement selection with a threshold value of 1e-4.

Threshold value	Greedy → Cyclic Time (s)	Greedy → Rand w Time (s)	Greedy → Rand wout Time (s)
1e-3	0.83	0.49	0.67
1e-4	0.83	0.49	0.67
1e-5	0.87	0.53	0.66
1e-6	0.85	0.71	0.79
1e-7	1.26	1.26	1.29
1e-8	1.83	1.85	1.83

Table 5.9: Average times of CD on Large dataset over 10 trials for various threshold values and hybrid methods. Greedy → Cyclic = hybrid method starting with greedy and ending with cyclic feature selection. This also applies to Greedy → Rand w and Greedy → Rand wout, where Randw = random with replacement and Rand wout = random without replacement.

5.6.2 Convergence study of feature selection methods

We conducted performance studies comparing overall time and iterations to converge for the feature choice methods described in Section 3.2. The convergence for the CD method was determined via the convergence of the dual gap value explained in Section 2.7 using the optimal results from the early stopping study in Section 5.5.

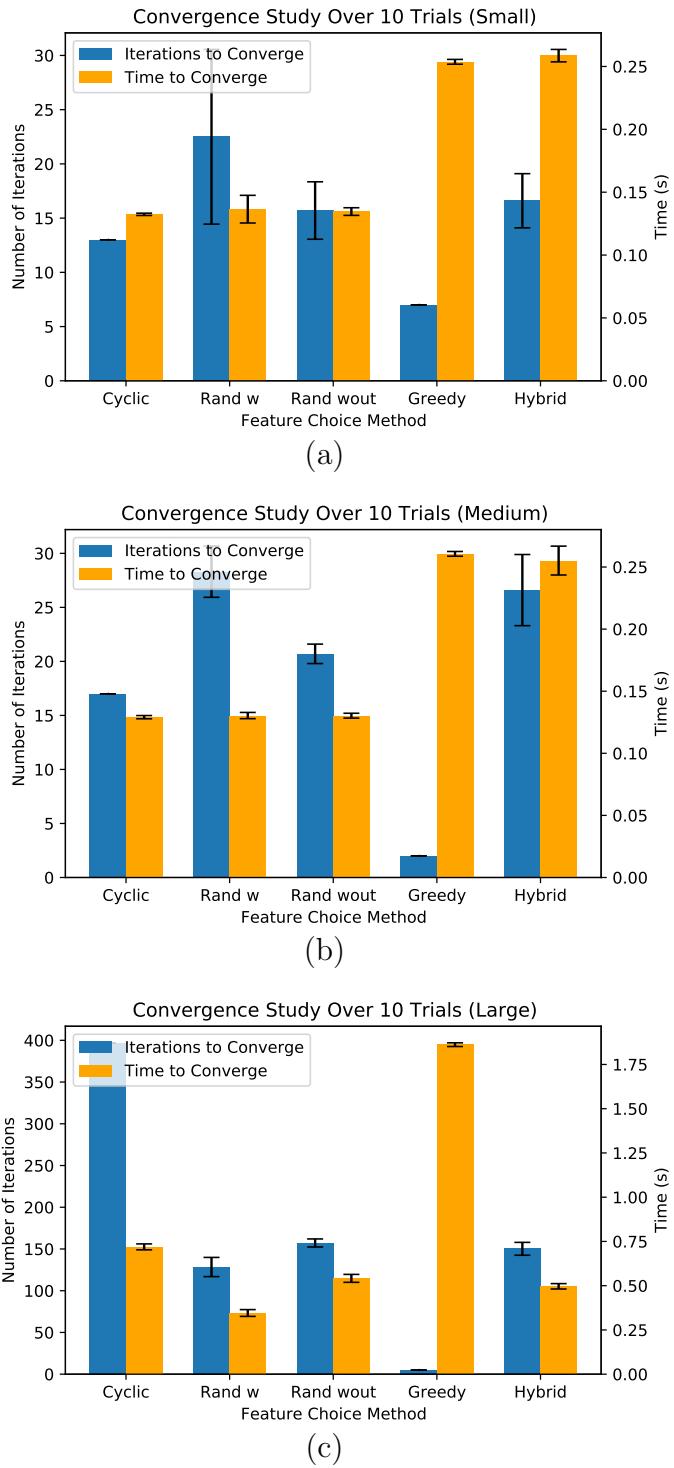


Figure 5.6: Convergence study showing number of iterations and overall time CD takes with various feature choice methods on the Small (a), Medium (b), and Large (c) dataset. Within the plot, abbreviations Rand w = random with replacement, Rand wout = random without replacement, Greedy = Gauss-Southwell-s, Hybrid = Hybrid method using Greedy until certain threshold in largest gradient value follow by cyclic

Figure 5.6 shows a comparision of the time and number of iterations required for various feature selection methods to converge on each of the datasets. Within the studies for the Small and Meidum datasets, it is difficult to distinguish the fastest method out of cyclic, random with replacement, and random without replacement. However, it is clear even on the smaller datasets that using greedy calculations slows down the overall time for CD to converge. In terms of the fastest computational time, the results support the idea that it is not worthwhile to increase the computational complexity of each iteration to lower iteration count. Using the Large dataset provides more separation between the times of the three basic selection methods. For the Large dataset, the hybrid method show results similar to the random without replacement, but is slower to the fastest method, random with replacement. While for these datasets and values for λ_1 and λ_2 , it is apparent that random with replacement is the fastest, this cannot be generalized for all λ_1 and λ_2 values nor for all sizes of data.

5.7 Comparison of Naive and Covariance Update Rules

We conducted a speed comparison study to compare the covariance update and the naive update rules. We ran both algorithms on different sizes of random generated data, in all of which the matrix X was “tall and thin” ($N >> p$). In this study, we did not compare both algorithms for convergence because, as shown in Section 3.2, they implement the same coordinate update rule. Therefore, we ran both update methods for 500 iterations and compared their runtimes. Table 5.10 shows the results of our study. As we can see from the table, when we keep the number of features low compared with the number of samples and increase the number of samples from 100 to 1,000,000 covariance tends to be consistently much faster than Naive. This result supports our theoretical considerations in Section 3.2. Based on these studies, we decided to use the covariance method as our main foundation for further optimization of coordinate descent.

Num. features	Num. samples	Naive Time (s)	Covariance Time (s)
10	100	0.004	0.002
10	1000	0.007	0.003
10	10000	0.01	0.003
10	100000	0.66	0.01
10	1000000	4.2	0.01
10	10000000	47.6	0.05
10	17281517	1100	1.24
100	100	0.8	0.1
100	1000	1.9	0.75
100	10000	6.5	0.8
100	100000	250	1.08
100	1000000	3300	5.03

Table 5.10: Speed comparison between the covariance and naive update methods. We ran both algorithms over 500 iterations and only compared their runtime on various sizes of “tall and thin” matrices ($N >> p$)

5.8 Front Heavy Covariance CD vs Standard Covariance CD

We conducted a runtime comparison between the front heavy covariance method and the standard covariance method on the Small dataset provided by Aquatic. The results from our study are shown in Table 5.11. As we can see in the table, the front heavy method takes more time to prepare the data due to the gram matrix calculation in the beginning. However, the overall time for the front heavy method (1.129) is significantly smaller than the overall time for standard covariance (5.626). From these results, we can conclude that Front Heavy covariance is significantly faster than standard covariance in practice despite the theoretical considerations in Section 3.3, which suggested that it may lead to unnecessary computations and slower performance.

Method	Front Heavy Covariance	Standard Covariance
Data Preparation Time (s)	0.839	0.553
CD Time (s)	0.167	5.073
Overall Time (s)	1.129	5.626
Number of Iterations	19	16

Table 5.11: Runtime comparison between front heavy covariance and standard covariance on the Small dataset

5.9 Thresholding

We used cross-validation on the Small dataset to tune the thresholding parameter discussed in Section 3.6. We used the most recent 10 percent of the data as a validation set and the rest 90 percent as a training set. We tried ten different values of the thresholding parameter and plotted the validation loss in Figure 5.7. From our studies, it seems like a value of 0.02 seems to have the lowest validation loss. In the future, we plan to explore the thresholding idea more thoroughly and have more conclusive results.

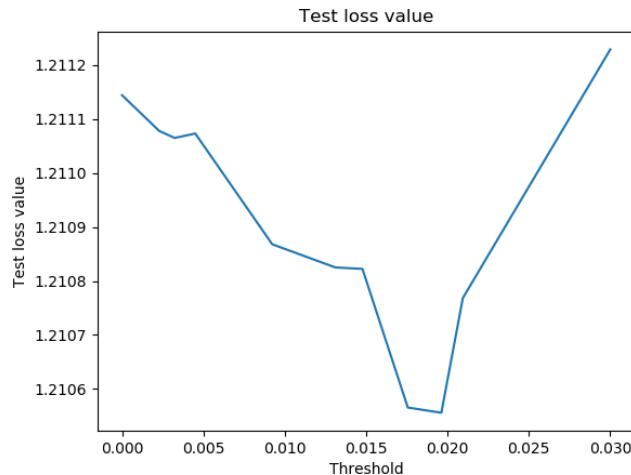


Figure 5.7: Thresholding value vs cross-validation loss.

5.10 Loading Data Studies

There are two main ways to overcome the limit on the size of our dataset. We could improve our hardware (i.e., increase our RAM capacity), or devise a way to load and process the data in parts. We chose to go with the latter since it is more scalable than the former. We devised two ways to calculate the gram matrix. The faster one is described in Section 3.7.2. Our first method was to load in each column as needed and discard it once we have computed the inner products with it. However, this method required loading in columns more than once and is not easily parallelized. The second method we developed was to load in parts of the matrix X by row rather than column and sum up the partial gram matrices as described in Section 3.7.2. The time to load and process X into a gram matrix on our hardware is stated in Table 5.12.

Our primary focus of the study is not how to load in data faster. We are researching how to make the CD algorithm faster. However, to make our methods feasible for Large datasets, we had to improve loading times to something reasonable (less than 1 minute).

Method	Dataset	Overall Time (s)
Load by Column	Small	2117.63
Load by Column	Medium	> 6 hours
Load by Row (50% loaded)	Small	0.993883
Load by Row (100% loaded)	Small	0.899703
Load by Row (35% loaded)	Medium	20.4896
Load by Row (50% loaded)	Medium	14.8203

Table 5.12: Performance study over 10 trials loading in and calculating $X^T X$ from our Small (dimensions $17,281,517 \times 10$) and Medium (dimensions $17,281,517 \times 162$) datasets.

5.11 Lower Precision Studies

As described in Section 4.1, using lower precision can be beneficial. For the computations in CD, pre-existing numerical linear algebra libraries immensely improve performance and should be used rather than developing our own methods. However, this project's hardware configuration does not have any AVX512 instruction with support for half precision. Furthermore, the libraries with *BLAS-like* functions that we tested, Intel's Math Kernel Library *MKL* and *Pytorch*, do not have support for matrix operations with float16 precision but have limited alternatives for int16. With this in mind, we looked at how lower precision calculations would influence the accuracy and computation time of the main computation used in coordinate descent: matrix vector multiplication.

As shown in Table 5.13, we implemented several various methods of matrix vector multiplication to find the most optimal configuration. Both MKL implementations and the Pytorch method for single precision (Float32) appear to have the fastest overall times with a range of 0.03 seconds. While the Pytorch MatVect (Float32) method has the lowest average time, several factors could account for the time disparity, including the standard deviation of the trials and slight differences in the hardware's physical condition, such as temperature. The main difference between these three methods is that the MKL MatVect Int16 implementation only used half of the memory that the other two methods required. From these results, we decided to compare the performance of full CD implementations using MKL's Float32 and Int16 based matrix vector multiplication methods.

Method	Total Memory Used (GB)	Overall Time (s)
<i>MKL</i> MatVect (Float32)	7.38	0.359
<i>MKL</i> MatVect (Int16)	3.72	0.361
Basic MatVect (Float16)	3.69	135.2
<i>Pytorch</i> MatVect (Float32)	7.44	0.334
<i>Pytorch</i> MatVect (Int16)	3.75	2.371

Table 5.13: Performance study over 10 trials using matrix of size $12,500,000 \times 150$ and vector of size 150×1 . Other than the Basic MatVect (Float16) method which consists of two for loops, each method includes the library and data type.

Implementing the MKL Int16 method of CD required slight adjustments to the data. The provided data consisted of single precision floating point values. To convert these to short integers, we scaled each term by a factor of 10^N , where N is the number of digits saved from the conversion.

Method	Time (s)	Memory Used (kB)	Digits of Accuracy
Naive (Float32)	19.332	817,656	7-8
Naive (Int16)	120.613	707,744	3
Covariance (Float32)	0.577	817,868	7-8
Covariance (Int16)	N/A	N/A	0

Table 5.14: Performance study over 50 iterations for CD with single (Float32) and lower precision (Int16) with both naive and covariance update rules on the Small dataset.

With the conversion of the data, we implemented Int16 alternatives for CD with both naive and covariance update rules. Table 5.14 displays the performance of using lower precision versus single precision. The MKL Int16 MatVect function returns the resulting vector with data type Int32. Within the covariance update rule, calculating inner product values results in large terms that exceed the limits of Int32 when using the previously described scaling method. Exceeding this limit results in incorrect values for the inner products, invalidating the accuracy of the results produced by the covariance lower precision method.

When comparing CD using naive update rule for the two variants of precision in Table 5.14, it is apparent that the lower precision method is significantly slower and results in less accurate predicted weights. In theory, storing data as Int16 values should result in half of the memory used to store Float32 values. For these results, the memory usage for the Int16 method significantly exceeds half of the memory usage of the Float32 method. This is because not all MKL functions used in CD have lower precision alternatives, so various calculations are converted and stored as Float32 values. It is important to note that using larger datasets that have more columns will result in larger amounts of memory saved when using lower precision. This increase in memory saved is because the calculations converted and stored as Float32 values will take less space compared to the memory required to store the data.

5.12 Gram Matrix Estimation Study

In Section 4.2, we proposed a method of estimating the Gram matrix using Pearson Correlation Coefficients between features, which are invariant across chunks of rows. Here we give

the results of a study of this method on Aquatic’s 17 million rows by 162 column Medium dataset.

Let c refer to the proportion of rows of \mathbf{X} in each chunk - i.e. (chunk size) = $c * N$. Thus the smaller the value of c , the smaller the first chunk and the fewer rows we use to calculate r_{ab} , from which we estimate the feature inner products of the rest of the chunks. Table 5.15 below includes three additional metrics. First, “Time ratio” refers to the time to estimate the Gram matrix divided by the time to calculate it fully. “Avg absolute weight diff” refers to the quantity

$$\frac{1}{N} \sum_{i=1}^N |\hat{\beta}_i^{(e)} - \hat{\beta}_i^{(t)}|$$

where $\hat{\beta}^{(e)}$ is the weight vector predicted by the covariance optimizer using the estimated Gram matrix and $\hat{\beta}^{(t)}$ is the ‘ground truth’ weight vector predicted by the optimizer using the true Gram matrix. “Loss function diff” refers to

$$f(\hat{\beta}^{(e)}) - f(\hat{\beta}^{(t)})$$

where f is the elastic net loss function. The latter two metrics measure how much less optimal the solution is using the estimated Gram matrix.

c	Time ratio	Avg absolute weight diff	Loss function diff
0.5	1.001	$1.552 \cdot 10^{-6}$	0.04
0.2	0.586	$3.637 \cdot 10^{-6}$	0.16
0.1	0.538	$6.572 \cdot 10^{-6}$	0.51
0.05	0.596	$6.677 \cdot 10^{-6}$	0.59

Table 5.15: Runtime improvement and accuracy loss of estimating the Gram matrix using a decreasing proportion of the rows of X . As the amount of data used to make the estimate decreases, the time saved generally increases but the method’s accuracy decreases

We can see that as c decreases, the time saved estimating the Gram matrix increases, almost to the point where estimating saves half the time of calculating the Gram matrix fully at $c = 0.1$. We can also see that the discrepancy in predicted weights and loss function difference increase significantly. Note that the loss function difference is always positive, indicating that the optimizer estimating the Gram matrix finds a less optimal solution.

It is important to note that, while the values in the table’s third and fourth columns increase as c decreases, they remain relatively small. The average absolute value of an element of $\hat{\beta}^{(t)}$ is around $3 \cdot 10^{-5}$. Thus at $c = 0.05$ the average weight difference is only about 20% of the average weight, while this number is only 5% at $c = 0.5$. One caveat is that Small individual weights may have their values change by significantly more than 20%. However, these weights are Small and hence do not contribute significantly to the model.

The Gram matrix estimation method showed promising results on Aquatic’s Medium dataset, but should also be tested on a dataset with more features to ensure its robustness.

5.13 Preconditioning Study

In Section 3.8 we introduced the idea of preconditioning X with $(X^T X)^{-1/2}$ to encourage faster CD convergence. In this section we study the runtime performance of the preconditioned updates introduced in Section 3.8.1.

We ran this study on our generated data in order to explore the effect on runtime of varying both N and p . Table 5.16 below compares the overall runtimes of the precondition-and-CD procedure detailed in Section 3.8 to the front heavy covariance method described in Section 3.4 using the weight change criterion discussed in Section 4.5 to avoid excessive duality gap calculations. The two methods are run on the same problem for each N, p combination. Their predicted weights β are equal to within the tolerance of `float`. “Cond prep time” refers to the extra time required by the preconditioning method to calculate $(X^T X)^{-1/2}$ and $Z^T y = (X^T X)^{-1/2}(X^T y)$. “Cond CD time” is the runtime of the subsequent CD algorithm using the preconditioned coordinate updates in equations (3.21) and (3.22). The CD is run for exactly 2 iterations and thus never has to calculate the duality gap. “Cond total time” is simply the sum of the previous two columns - the total runtime of the precondition-and-CD algorithm. This is to be compared to “Front heavy total time” - the runtime of the standard front heavy method, which excludes the data loading and Gram matrix calculation (which the preconditioned method also must do) and hence is just the CD algorithm runtime. We can see from the last two columns of Table 5.16 that the

N	p	Cond prep time (s)	Cond CD time (s)	Cond total time (s)	Front heavy total time(s)
10^6	20	< 0.001	< 0.001	< 0.001	0.006
10^6	50	0.001	0.001	0.002	0.008
10^6	100	0.004	0.006	0.01	0.01
10^6	500	0.41	0.23	0.64	0.062
$5 \cdot 10^6$	20	< 0.001	< 0.001	< 0.001	0.039
$5 \cdot 10^6$	50	0.001	0.001	0.002	0.048
$5 \cdot 10^6$	100	0.006	0.006	0.012	0.05
$5 \cdot 10^6$	500	0.53	0.41	0.94	0.32

Table 5.16: Study of runtime of preconditioned front-heavy covariance vs standard front heavy covariance method for various numbers of features and samples. “Cond prep time” refers to the time to calculate $(X^T X)^{-1/2}$ and “Cond CD time” refers to the time to perform two iterations of the preconditioned CD algorithm.

preconditioned algorithm’s runs slower relative to the standard front heavy algorithm as p increases, even running 10 times slower for $N = 10^6, p = 500$. Both the preconditioning’s prep and CD runtime increase significantly with p . This matches our expectations, as $(X^T X)$ has dimensions $p \times p$ and each of the p CD iterations must calculate a sum of p terms. Optimized code can avoid calculating the sums in equations (3.19) and (3.20) for a_k and b_k at each iteration by determining them incrementally as it sets β values, but the $\langle G_k, G_k \rangle$ calculations in equation (3.19) are unavoidable. Standard front heavy CD also runs more iterations as p increases, but each iteration performs significantly less work than a preconditioned iteration.

On the other hand, the preconditioned algorithm’s runs faster relative to the standard front heavy algorithm as N increases. In fact, the preconditioned method’s prep and CD runtimes are theoretically invariant with respect to N , as N does not appear anywhere in either procedure. This is reflected for $p = 20, 50$, and 100 , although the algorithm runs slightly slower for larger N for $p = 500$. This change, however, is small compared to that of the front heavy runtime, which for all p increases about fivefold to match the fivefold increase in N . This discrepancy is likely due to the standard front heavy method’s need to

calculate the duality gap to determine convergence, as the duality gap calculation involves y , which has length N .

5.14 Warm Start Study

In Section 3.9, we introduced the idea of a warm start to speed up sequential CD. Here, we present simulated results to show how a warm start produces a speedup for CD. To determine how a warm start affected a rolling window CD, we studied its effects on the Small dataset. First, we ran CD on the first 10% of the rows. Then, we added 10% of the rows sequentially in the manner described in Section 3.9. The results from this experiment are stated in Table 5.17. As can be seen from the table, using a warm start for new data can decrease the amount of iterations we take in Coordinate Descent by half. To determine

Fraction of Rows	Iterations without Warm Start	Iterations with Warm Start
0.2	20	12
0.4	19	12
0.6	22	19
0.8	18	9
1.0	19	10

Table 5.17: Performance study of using a warm start for rolling windows on the Small dataset.

how a warm start affected a distributed front-heavy CD, we studied its effects on running CD for the Large dataset. The methodology is described in Section 3.9.2. This experiment used a random without replacement feature choice. To create consistent results, we ran CD ten times and averaged the results. The results from this experiment are stated in Table 5.18. As per the table, we can achieve a speedup of 1.5 once we have over 80% of the data loaded from a uniform distribution. Thus, we can utilize warm starts to fill in any time between the fastest and slowest nodes.

Fraction of Rows	Iterations with Warm Start	Iteration Speedup
0.20	106.2	1.14
0.25	102.6	1.84
0.33	107.9	1.13
0.50	97.1	1.25
0.67	93.9	1.30
0.84	77.5	1.57

Table 5.18: Performance study of using warm start for a distributed network on the Large dataset.

5.15 Study of Ensemble Methods

5.16 Comparison with Scikit-Learn Elastic Net

Scikit-Learn (sklearn), a popular Python package for machine learning and optimization, contains a solver for elastic net that uses CD, including covariance method support. However, sklearn only supports datasets that can be stored within a machine's RAM, and thus we can only test the sklearn on the Small dataset. On the Small dataset, sklearn can run CD with the same tolerance, lasso penalty, and ridge penalty in 2.66 seconds. Our implementation can run CD on the Small dataset in 0.66 seconds. Thus, we achieve both faster speeds and support for arbitrarily large datasets.

Chapter 6

Conclusions

In Chapter 5, we conducted studies of the CD optimizations methods described in Chapter 3 and 4. Our initial results provide insight on methods and techniques that lower computational time for CD using the Small, Medium, and Large datasets provided by Aquatic.

Feature Choice Study

Section 5.6 displays an analysis of each feature selection method on each of the datasets. The results suggest that the random selection with replacement is the fastest is either the fastest or amongst the fastest methods for each of the datasets.

Covariance vs Naive Update Rules

Based on our results in Section 5.7, we can conclude that the covariance update is significantly faster than the Naive method on “tall and thin” matrices ($N \gg p$). This result is consistent with our theoretical considerations in Section 3.3, where we highlighted that the covariance update does not do $O(N)$ operations per coordinate, unlike the Naive update. Based on that study, we decided to use the covariance method as our main foundation for further optimizing CD.

Front Heavy Covariance

The front heavy method allows us to move computations of $X^T X$ and $X^T y$ to the front of the algorithm before the main loop. In practice, as shown in Section 5.8, this method is faster than the standard covariance method because it simplifies the code and allows implicit parallelism from MKL when calculating the gram matrix and $X^T y$. This method also lets us to utilize the chunk-loading method in Section 3.7 to load in large matrices beyond our memory capacity.

Lower Precision

The results presented in Section 5.11 showed that lower precision implementations for CD using both naive and covariance update rules resulted in significant accuracy loss and slower computational performance. Furthermore, using lower precision on the smaller dataset did not result in significant saves in memory. These results suggest that the downsides of using lower precision on our hardware outweigh the only benefit, which is the memory save.

Estimating Gram Matrix

Estimating the Gram matrix with Pearson correlation coefficients yielded promising results, as shown in Section 5.12. Estimation using the optimal number of rows cut the algorithm’s total runtime almost in half while sacrificing only marginal accuracy. However, these results are based on only one dataset, and the method should be tested on other datasets to verify its robustness. The Pearson method’s success motivates the exploration of alternate estimation techniques that could reduce runtime even further.

Preconditioning

Section 5.13 shows mixed results for preconditioning X and solving the modified CD problem described in Section 3.8. While the preconditioned CD algorithm converges in fewer iterations, each coordinate update requires significantly more calculation than a corresponding standard coordinate update and calculating $(X^T X)^{-1/2}$ adds significant overhead that grows rapidly worse as p increases. The method shows signs of improvement over the standard method for very large N , but likely only because it avoids calculating the duality gap to check for convergence. Early stopping and other methods introduced in this paper to avoid the duality gap calculation are simpler and do not see the same slowdown for larger p , so they should be preferred.

Warm Start

Ensembling Methods

Scikit-learn Elastic Net Comparision

Chapter 7

Future Work

Most of the results for this report, in Chapter 5, were designed around the specific hardware configurations and the provided data from Aquatic. The conclusions from Chapter 6 should provide motivation to look into certain optimization fields of CD, as the results may differ when using different data or hardware. To further strengthen the CD findings within this report, we would like to look into several of the following concepts:

Lower Precision

Our lower precision studies were extremely limited due to the hardware configuration used in this project. The hardware did not support any computational speedup when using lower precision, and numerical linear algebra libraries only offered support for short integers, which lose too much accuracy for CD calculations. However, these issues disappear when using newer CPUs and GPUs. It would be interesting how CD with lower precision calculations would compare to single precision CD on hardware that has a greater range of lower precision support.

Feature selection methods

The feature selection methods covered in this report only scratch the surface of the research methods out there. For example, we implemented the Gaussian-Southwell-s greedy method, a commonly used greedy feature selection method that has large amounts of literature support. However, there are many other greedy variants that are designed to tackle the high per-iteration complexity when using a greedy method. Additionally, we only implemented the most commonly used cyclic and random methods, but would like to explore other techniques.

Update Rules

The covariance update rule used in to produce most of our results is best for when X has the property of $N \gg p$, which is the case for the data provided by Aquatic. However, the performance of this method deteriorates as this property of the data does not hold. There are several update methods we would like to explore that may provide faster computational performance when using data where N is not significantly greater than p .

CD performance on computer clusters

Many big data problems are computed on large computer clusters, providing both large amounts of parallelism and computational power to speed up calculations. For this project, the computations took place on a small hardware configuration with extreme limitations for parallelism. To not interfere with the MKL linear algebra functions that are most efficient when using all 4 available CPUs, we restricted the remaining portions of CD to run in sequential order, removing any possibilities of implementing block-based CD algorithms. However, using a computer cluster for CD provides new challenges such as moving data throughout processes and parallel concepts that we would like to further explore.

Ensemble Methods

For the ensembling methods used for the studies in this report, there was little to no motivation in selecting these over other commonly used methods. Additionally, it is difficult to determine when one ensembling method may produce more accurate results compared to another. **TBD**

Appendix A

Abbreviations

CD. Coordinate Descent.

SGD. Stochastic Gradient Descent.

GS-s. Gaussian-Southwell-s greedy feature choice method.

ACF. Adaptive coordinate frequencies.

ACF-GPI. ACF with gradient probability initialization.

MKL. Intel's Math Kernel Library.

IPAM. Institute for Pure and Applied Mathematics. An institute of the National Science Foundation, located at UCLA.

RIPS. Research in Industrial Projects for Students. A regular summer program at IPAM, in which teams of undergraduate (or fresh graduate) students participate in sponsored team research projects.

UCLA. The University of California at Los Angeles.

Selected Bibliography Including Cited Works

- [1] Z. ALLEN-ZHU, Z. QU, P. RICHTARIK, AND Y. YUAN, *Even faster accelerated coordinate descent using non-uniform sampling*, in Proceedings of The 33rd International Conference on Machine Learning, M. F. Balcan and K. Q. Weinberger, eds., vol. 48 of Proceedings of Machine Learning Research, New York, New York, USA, 20–22 Jun 2016, PMLR, pp. 1110–1119.
- [2] O. FERCOQ AND P. RICHTÁRIK, *Accelerated, Parallel and Proximal Coordinate Descent*, arXiv e-prints, (2013), p. arXiv:1312.5799.
- [3] J. FRIEDMAN, T. HASTIE, AND R. TIBSHIRANI, *Regularization paths for generalized linear models via coordinate descent*, Journal of Statistical Software, Articles, 33 (2010), pp. 1–22.
- [4] T. GLASMACHERS AND U. DOGAN, *Accelerated coordinate descent with adaptive coordinate frequencies*, in Proceedings of the 5th Asian Conference on Machine Learning, C. S. Ong and T. B. Ho, eds., vol. 29 of Proceedings of Machine Learning Research, Australian National University, Canberra, Australia, 13–15 Nov 2013, PMLR, pp. 72–86.
- [5] T. HASTIE, R. TIBSHIRANI, AND J. FRIEDMAN, *The Elements of Statistical Learning*, Springer Series in Statistics, Springer-Verlag New York, 2 ed., 2008.
- [6] S. KIM, K. KOH, M. LUSTIG, S. BOYD, AND D. GORINEVSKY, *An interior-point method for large-scale ℓ_1 -regularized least squares*, IEEE Journal of Selected Topics in Signal Processing, 1 (2007), pp. 606–617.
- [7] J. NUTINI, M. SCHMIDT, I. H. LARADJI, M. FRIEDLANDER, AND H. KOEPKE, *Coordinate descent converges faster with the gauss-southwell rule than random selection*, in Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37, C. S. Ong and T. B. Ho, eds., vol. 37 of Proceedings of Machine Learning Research, Australian National University, Canberra, Australia, 13–15 Nov 2015, PMLR, pp. 1632—1641.
- [8] P. RICHTARIK AND M. TAKAC, *Distributed coordinate descent method for learning with big data*, Journal of Machine Learning Research, 17 (2016), pp. 1–25.
- [9] H.-J. M. SHI, S. TU, Y. XU, AND W. YIN, *A Primer on Coordinate Descent Algorithms*, arXiv e-prints, (2016), p. arXiv:1610.00040.
- [10] L. SHU, F. SHI, AND G. TIAN, *High-dimensional index tracking based on the adaptive elastic net*, Quantitative Finance, 1 (2020), pp. 1–18.

- [11] V. SMITH, *System-Aware Optimization for Machine Learning at Scale*, PhD thesis, Univeristy of California at Berkeley, 2017.
- [12] R. TIBSHIRANI, *Regression shrinkage and selection via the lasso*, Journal of the Royal Statistical Society. Series B (Methodological), 58 (1996), pp. 267–288.
- [13] R. TIBSHIRANI, J. BIEN, J. FRIEDMAN, T. HASTIE, N. SIMON, J. TAYLOR, AND R. J. TIBSHIRANI, *Strong rules for discarding predictors in lasso-type problems*, Journal of the Royal Statistical Society: Series B (Statistical Methodology), 74 (2012), pp. 245–266.
- [14] S. J. WRIGHT, *Coordinate descent algorithms*, Mathematical Programming, 151 (2015), p. 3–34.
- [15] M. ZINKEVICH, M. WEIMER, L. LI, AND A. J. SMOLA, *Parallelized stochastic gradient descent*, in Advances in Neural Information Processing Systems 23, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, eds., Curran Associates, Inc., 2010, pp. 2595–2603.
- [16] H. ZOU AND T. HASTIE, *Regularization and variable selection via the elastic net*, Journal of the Royal Statistical Society: Series B (Statistical Methodology), 67 (2005), pp. 301–320.
- [17] H. ZOU AND H. H. ZHANG, *On the adaptive elastic-net with a diverging number of parameters*, Ann. Statist., 37 (2009), pp. 1733–1751.