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

August 19, 2020

This project was supported by Aquatic Group and by NSF Grant 1440415

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 runtime effect 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

List of Algorithms

List of Figures

List of Tables

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 its models to make profits - if they predict a stock's price will rise, they will buy it. 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 correctly guess 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. Many companies in this field, such as Aquatic, use high-end computer clusters to predict and execute trades to tackle this problem. We address the problem with limited computing power in our project 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 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 ways of solving the memory and disk-io bounds of 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 Mathematical 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 this report's optimization problem, 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

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

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

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

If we let $\beta = (\beta_1, \ldots, \beta_p)$, 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 most straightforward 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 [15]. 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 rest's weights 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 [15], uses the L_1 norm of β instead, so the problem becomes

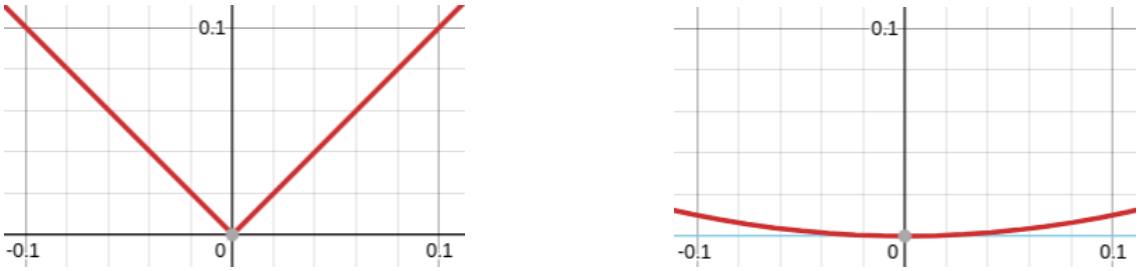
$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \frac{1}{2N} \|X\beta - y\|_2^2 + \lambda \|\beta\|_1$$

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. Thus, the ridge penalty does not provide a sufficient penalty for 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 OLS's above problems. 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 behavior is a decline in the model's interpretability: the zeroed features could have a strong correlation with the target, but it appears as though they have little correlation.



(a) $|x|$ (b) x^2

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 [19]. 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:

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

λ_1 and 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 [19]. 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 in nonsmooth so the elastic net inherits the lasso's desirable feature selection property.

Since all three terms of the argument in equation (2.1) are strictly convex, elastic net is a strictly convex optimization problem. Thus, 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 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. For the remainder of this report, i_k is the coordinate selected based on iteration k, β_{i_k} is the i_kth coordinate of the β vector, and β^k is the predicted β for the kth iteration. In a general framework each step of CD chooses an index $i_k \in \{1, 2, \ldots, n\}$, 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 $j \neq i_k$. The 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:

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

where $f : \mathbb{R}^n \to \mathbb{R}$ is continuous. Algorithm 1 from [17] describes coordinate descent:

Algorithm 1 Basic CD

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

Different variants of the Algorithm 1 exist, incorporating modifications for the update step such as block coordinate minimization, proximal point update, prox-linear update, and extrapolation, described in [12]. 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, making 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 gradient descent method. The purpose of estimating the gradient is the 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:

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

where $Q : \mathbb{R}^n \to \mathbb{R}$ 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 [18] describes the discussed procedure.

Algorithm 2 Basic SGD

set $k \longleftarrow 0$, choose $w^0 \in \mathbb{R}^n$, and choose learning rate η
repeat
  Randomly shuffle examples in the training set
  for $i = 1, 2, ..., n$ **do**
    $w^{k+1} \coloneqq w^k - \eta \nabla Q_i(w)$
  end for
  $k \longleftarrow k + 1$
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 our research's central problem 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 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:

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

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

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

The standard dual problem is defined as:

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

The duality gap is the difference between the primal problem's value and the dual problem's value. An important property of strongly convex problems is called strong duality. 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, our problem is the same as given in Section 2.4:

$$\min_{\beta} \frac{1}{2N} \|y - X\beta\|_2^2 + \lambda_1 \|\beta\|_1 + \lambda_2 \|\beta\|_2^2$$

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],

$$\begin{aligned} \min_{\beta} \quad & \frac{z^T z}{2N} + \lambda_1 \|\beta\|_1 + \lambda_2 \|\beta\|_2^2 \\ \text{s.t.} \quad & z = y - X\beta. \end{aligned}$$

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

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

To determine $\inf_{\beta,z} L(\beta, z, \nu)$, we can split it up as follows:

$$\inf_{\beta, z} L(\beta, z, \nu) = \inf_{\beta} \inf_{z} L(\beta, z, \nu).$$

Now, $\inf_z L(\beta, z, \nu)$ can be calculated by setting the partial gradient of $L(\beta, z, \nu)$ with respect to z to $\vec{0}$:

$$\frac{\partial L(\beta, z, \nu)}{\partial z} = \frac{z}{N} - \nu = \vec{0}$$

Thus, $z = n\nu$. So, plugging it back into the equation, our updated problem is

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

Now, we can solve this problem by setting the partial gradient of $L(\beta, z, \nu)$ with respect to β to $\vec{0}$:

$$\frac{\partial L(\beta, z, \nu)}{\partial \beta} = \lambda_1\frac{\partial\|\beta\|_1}{\partial\beta} + 2\lambda_2\beta - X^T\nu = \vec{0}.$$

$\frac{\partial\|\beta\|_1}{\partial\beta}$ is not defined everywhere and is given as follows:

$$\frac{\partial\|\beta\|_1}{\partial\beta_i} = \begin{cases}\text{sign}(\beta_i), & \beta_i \neq 0\\ [-1, 1], & \beta_i = 0\end{cases} \text{ for } i = 1, ..., p.$$

$L(\beta, z, \nu)$ must reach a minimum by the constraints of the duality problem. In order for this to happen, it has been proven in [6] that for $L(\beta, z, \nu)$ to obtain an optimum, $\left\|2\lambda_2\beta - X^T\nu\right\|_\infty \leq \lambda_1$. Therefore, this part of $L(\beta, z, \nu)$ has an optimum of 0. Simplifying, the optimum becomes

$$\inf_{\beta, z} L(\beta, z, \nu) = \nu^T y - \frac{N}{2}\nu^T\nu.$$

The dual function is then

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

The Lagrange dual problem is then given by

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

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

$$G(\nu) = \nu^T y - \frac{N}{2}\nu^T\nu.$$

The duality gap is then given by

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

It was shown in [6] that we can derive a feasible dual variable ν for an arbitrary β with the following formula:

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

$$s = \frac{N\lambda_1}{\|(2X^T(X\beta - y))\|_\infty}.$$

Strong duality holds for least squares with the elastic net regularizer and, thus, we can use the duality gap as a certificate of optimality as shown in [14].

2.8 Iteration Complexity of Coordinate Descent

Iteration complexity is how the minimum and the maximum number of iterations for a given problem changes with different inputs. For gradient descent algorithms, this is well understood. For CD, however, iteration complexity is not as well researched and still has room for improvement. The iteration bounds for cyclic and random with replacement CD are given in this section. First, we will discuss some fundamental ideas about strongly convex functions.

2.8.1 Strongly Convex Functions

General Strongly Convex Function

For any twice differentiable function $f(x)$, it is called μ-strongly convex if

$$\nabla^2 f(x) \succeq \mu I.$$

Let $f(x) = x^T Ax$. Then, $\nabla^2 f(x) = 2A^T$. So, $f(x)$ is μ-strongly convex if all the eigenvalues of A are greater than or equal to μ. Another important property of strongly convex functions is that if functions $f(x)$, $g(x)$, and $h(x)$ are defined as

$$h(x) = f(x) + g(x),$$

where $f(x)$ is strongly convex and $g(x)$ is convex, then $h(x)$ is strongly convex.

Least Squares with Elastic Net

In our problem, we can define $F(x)$ as

$$F(x) = f(x) + \Psi(x)$$

where

$$f(x) = \frac{1}{2N}\|X\beta - y\|_2^2 = \frac{1}{2N}\left(\beta^T X^T X\beta - 2\beta^T X^T y + y^T y\right)$$

and

$$\Psi(x) = \lambda_1\|\beta\|_1 + \lambda_1\|\beta\|_2^2.$$

So, the strong convexity parameter for $f(x)$ is the minimum eigenvalue of $\frac{X^T X}{N}$ and the strong convexity parameter for $\Psi(x)$ is $2\lambda_2$.

2.8.2 Cyclic Coordinate Descent

Our analysis of the iteration complexity is based on the block CD iteration complexity analysis in [7]. In [7], is has been proven that for coordinate descent on a strongly convex optimization problem, we need

$$\left\lceil \frac{\mu L_{\min}^{\mu} + 16L^2\log^2(3p)}{\mu L_{\min}^{\mu}} \log\left(\frac{\epsilon_{\text{init}}}{\epsilon}\right)\right\rceil$$

iterations. $L_{\min}^{\mu}$ is defined in [7] as

$$L_{\min}^{\mu} = \min_{i\in\{1,\ldots,p\}} L_i + \mu_i$$

where p is the number of features. The rest of the variables used are described as follows:

1. L_i is the Lipschitz constant of feature i. L_i is greater than or equal to the spectral norm of $X_i^T X_i$.

2. μ_i is the strong convexity constant of feature i.

3. L is the Lipschitz constant of the whole problem. L is greater than or equal to the spectral norm of $X^T X$.

4. μ is the strong convexity constant of the whole problem.

5. ϵ is the desired error.

6. ϵ_{init} is the initial error in objective function value.

2.8.3 Random Coordinate Descent

It was proven in [11] that to reach an error of ϵ with probability $1 - \rho$, random coordinate descent methods takes k iterations where

$$k \geq p \frac{1 + \mu_\Psi}{\mu_f + \mu_\Psi} \log \left(\frac{\epsilon_{\text{init}}}{\epsilon \rho} \right)$$

where p is the number of features, μ_Ψ is the strong convexity parameter for the regularizer (in our case, elastic net), μ_f is the strong convexity parameter for loss function (in our case squared loss), and ϵ_{init} is the initial error in objective function value. We do not know what ϵ_{init} is without knowing the optimal solution. However, it can be overestimated as the duality gap.

2.9 Data Statistical Properties and Synthesis

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 statistical 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 $\mathcal{N}(0, 1)$). 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, and corresponding columns X_a and X_b of X, the correlation between them is

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

   Around 15% of features fall into correlated blocks. Each pair of features a, b within a correlated block have correlation coefficient $r_{ab} \approx 0.8$. The remaining 85% of features c have $r_{cd} \approx 0$ for every other feature d.

3. The distribution of Pearson correlations r_{jy} between each feature j and the target y have mean and standard deviation 0 and 0.0075, respectively. We fit

$$r_{jy} = \frac{\text{Cov}[X_j, y]}{\sigma(x_j)\sigma(y)}.$$

Since X_j is standardized, this simplifies to

$$\frac{\sum_{i=1}^{N} X_{ij}(y_i - \mu_y)}{\sigma(y)}.$$

4. The entries of the target y should have mean 0 and standard deviation 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 a random (selected without replacement) 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

$$y = X\hat{\beta} + \epsilon$$

where ϵ is a noise vector whose magnitude is chosen with regard to p and sparsity of $\hat{\beta}$ such that property 4 is satisfied. X and y are inputs to the optimizer. Since we know $\hat{\beta}$, we can test the accuracy of the optimizer's predicted β.

Chapter 3

Coordinate Descent Optimization Methods without Accuracy Loss

3.1 Prior Related Work

Before creating our coordinate descent (CD) implementations, 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 data. Suppose the columns within the data are not correlated. In that case, 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 correlated, then simultaneously updating multiple coordinates will be suboptimal, requiring more iterations. Squared loss is a partially separable function when the feature matrix X is sparse, as shown in [12].

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 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 update its weight without waiting 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, it was shown in [12] that asynchronous methods can achieve speedups over serial methods of 25x compared to the 4x speedup of synchronization. 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 [17], 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. Accelerated methods of CD result in faster convergence rates 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), 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 is in [13]. This application solves a different problem (index tracking) then the problem Aquatic aims to solve. Still, the problem's mathematical formulation is identical to the formulation discussed above, with a few added constraints. The researchers in [13] use *adaptive elastic net* (Aenet), a modification to elastic net proposed in [20] in 2009. Aenet weights coordinates according to an initial estimate of $\boldsymbol{\beta}$, giving it desirable theoretical consistency properties that ordinary elastic net does not have. The study in [13] solves 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 future exploration area.

3.1.4 Hybrid Coordinate Descent (Hydra)

This method, introduced in [10], 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. Hydra 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. Each of these methods are compared against each other in Section 5.6.

3.2.1 Basic methods

Cyclic

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

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

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 $\{1, ..., n\}$.

Random without Replacement

A variant to the random with replacement feature selection method is to use random selection without replacement. Thus, 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 $\beta_j^0 = \beta_j^{\text{fin}}$ if $j \neq n$ where β^0 is the initial guess of β^{fin} is the vector for the global minimum. The simpler methods 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 can select coordinates farthest away from that global minimum.

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

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

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 [12] and is easily adjustable for elastic net since elastic net can be derived from Lasso.

$$i_k = \underset{1 \leq j \leq n}{\arg\max}\ g_j(x^{k-1}) \tag{3.3}$$

where

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

The shrink function from the equation above is defined as:

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

The GS-s method results in a computational complexity of $O(Np^2)$ for each feature selection, where N is the number of samples and p is the number of features. While this complexity is much larger than the previous methods, it is proven in [8] 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 greedy methods may not be worthwhile for later iterations of CD, where the maximum change of the weights shrinks 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. The probability of selecting each coordinate is weighted by the expected loss function decrease resulting from updating that coordinate.

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

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

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

$$\boldsymbol{\beta}^{t-1} = (\beta_1, \ldots, \beta_i^{t-1}, \ldots, \beta_p)$$
$$\boldsymbol{\beta}^{t} = (\beta_1, \ldots, \beta_i^{t}, \ldots, \beta_p)$$

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

$$\Delta = f(\boldsymbol{\beta}^t) - f(\boldsymbol{\beta}^{t-1}). \tag{3.7}$$

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

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

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

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

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

Algorithm 3 ACF probability update

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

decrease P_i if $\Delta < \overline{\Delta}$. Step 2 changes $P_{\texttt{sum}}$ 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 $\frac{P_k}{P_{\texttt{sum}}}$. 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 $P_1, \ldots, P_p$ 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 gradient's magnitude 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 $k = 1, \ldots, p$, we set

$$g_k = |\nabla_k f(\boldsymbol{\beta}_0)|$$

where $\boldsymbol{\beta}_0 = \mathbf{0}$ is the initial all-zero weight vector. Then for each k initialize

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

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 writing equation (2.1) is

$$\min_{\beta \in \mathbb{R}^p} \frac{1}{2N} \sum_{i=1}^{N} (y_i - x_i^T \beta)^2 + \lambda_1 ||\beta||_1 + \lambda_2 ||\beta||_2^2 \tag{3.8}$$

Consider a coordinate decent minimization step for equation (3.8) 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 (3.8). 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 shows that the coordinate-wise update has the form

$$\tilde{\beta}_j \leftarrow \frac{S(\frac{1}{N}\sum_{i=1}^{N} x_{ij}(y_i - \tilde{y}_i^{(j)}), \lambda_1)}{1 + 2\lambda_2} \tag{3.9}$$

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 (3.9).

3.3.1 Naive Updates

Looking more closely at equation (3.8), 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

$$\frac{1}{N}\sum_{i=1}^{N} x_{ij}(y_i - \tilde{y}_i^{(j)}) = \frac{1}{N}\sum_{i=1}^{N} x_{ij}r_i + \tilde{\beta}_j \tag{3.10}$$

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 (3.10) 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 described next.

3.3.2 Covariance Updates

Further efficiencies can be achieved in computing the update in equation (3.9). We can write the first term on the right (up to a factor of $\frac{1}{N}$) as

$$\sum_{i=1}^{N} x_{ij} r_i = \langle x_j, y \rangle - \sum_{k:|\tilde{\beta_k}|>0} \langle x_j, x_k \rangle \tilde{\beta}_k \tag{3.11}$$

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, 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 randomly generated data. When N >> 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.

A formal comparison between the two methods is located in Section 5.7. However, the iteration complexities of both methods suggest that the covariance update rule is faster, and therefore is the foundation for further optimizing the coordinate descent algorithm.

3.4 Front Heavy Covariance Method

As we can see from equation (3.11), 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 running CD, we can calculate the update in equation (3.9) by simply computing the sum in equation (3.11) with our precomputed 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 (3.9) by using the sum in equation (3.11).

Algorithm 4 Front Heavy Covariance Method

1: Precompute and store $X^T y$ and $X^T X$
2: Run normal covariance using equation (3.11)

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, then standard covariance will never compute its inner products with the other columns. However, the front heavy method will always compute that feature's inner products with all of the other features, which could be costly. Section 5.8 shows results comparing front heavy covariance against standard covariance.

3.5 Numerical Linear Algebra

Intel's Math Kernel Library (MKL) allows us to achieve peak performance for linear algebra operations on an Intel processor. One way the library does this is by parallelizing its operations through vectorization and an open-source parallel programming library OpenMP. Vectorization, also known as array programming, is 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 simultaneously on multiple processors.

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 [16], 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 [16] 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". Our studies are shown in Section 5.9.

3.7 Disk I/O

3.7.1 Main Problem

One dataset that we are working with is 115 GB in binary format with single precision. If we increase to double-precision, then that dataset, when loaded in, is 230 GB. Since we are only working with 16 GB of RAM, we required a method to load in data partially.

3.7.2 Gram-based Coordinate Descent

The method described in Section 3.4 is also called gram-based CD. 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:

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

To calculate $X^T X$, we can 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. We only have to load in X_i, which can be as small as a row. We also only store the much smaller gram matrix, which has dimensions p by p. For this reason, gram-based approaches are mostly used when $N >> p$. $X^T y$ is calculated in similarly to the gram matrix. With this method, our limit for loading in data is no longer defined by the size of X. Instead, our memory limit is mainly determined by the sizes of 2 rows of X, y, the gram matrix, and $X^T y$. We studied how the performance of this method in Section 5.10

3.8 Preconditioning X

Another family of methods for speeding up coordinate descent aims 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. Section 5.13 shows the performance of this preconditioning concept.

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 formulation in equation (2.1), giving

$$\hat{\beta_1} = \underset{\beta}{\operatorname{argmin}} \frac{1}{2N} \|X((X^T X)^{-1/2} \beta_1) - y\|_2^2 + \lambda_1 \|(X^T X)^{-1/2} \beta_1\|_1 + \frac{1}{2} \lambda_2 \|(X^T X)^{-1/2} \beta_1\|_2^2 \tag{3.12}$$

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

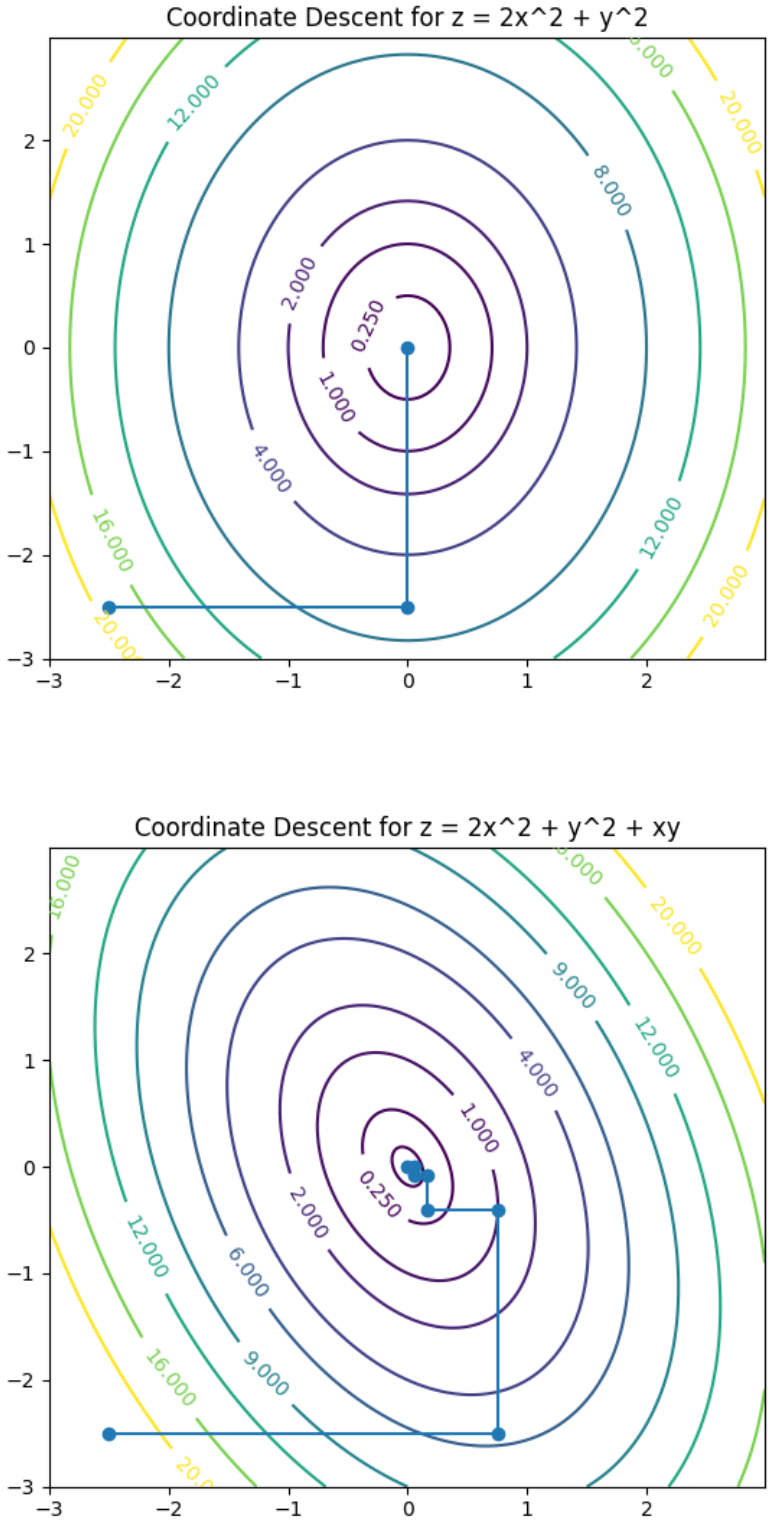


Figure 3.1: CD converges in one iteration on $z = 2x^2 + y^2$ where features x and y are "uncorrelated" but takes many iterations to converge for $z = 2x^2 + y^2 + xy$ due to the xy term.

semidefinite matrix X^TX and hence is also positive semidefinite. One can calculate the square root of such a matrix by finding its eigen/spectral decomposition

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

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

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

so that

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

If we view the first term of equation (3.12) as $\frac{1}{2N}\|(X(X^TX)^{-1/2})\beta_1\|_2^2$, we see that we have a new elastic net-like problem with

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

in place of X, so we can rewrite equation (3.12) as

$$\hat{\beta}_1 = \underset{\beta}{\operatorname{argmin}} \frac{1}{2N}\|Z\beta_1 - y\|_2^2 + \lambda_1\|G\beta_1\|_1 + \frac{1}{2}\lambda_2\|G\beta_1\|_2^2. \tag{3.13}$$

This point of this substitution is that now

$$\begin{aligned} Z^TZ &= (X(X^TX)^{-1/2})^TX(X^TX)^{-1/2} \\ &= (X^TX)^{-1/2}\big((X^TX)^{-1}\big)^{-1}(X^TX)^{-1/2} \\ &= Q\sqrt{\Lambda}Q^{-1}\big(Q\Lambda Q^{-1}\big)^{-1}Q\sqrt{\Lambda}Q^{-1} \\ &= I \end{aligned}$$

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 (3.13) 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 (3.13) 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 (3.13) is a modification of the original elastic net formulation in equation (2.1), so we need to derive a corresponding new version of the coordinate update step in equation (3.9). 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

$$\frac{\partial}{\partial\beta_k}\frac{1}{2}\lambda_2\|G\beta\|_2^2 = \lambda_2\sum_{l=1}^{p}\beta_l\langle G_k, G_l\rangle. \tag{3.14}$$

and, using an idea similar to the covariance update in equation (3.11),

$$\frac{\partial}{\partial\beta_k}\frac{1}{2N}\|Z\beta - y\|_2^2 = \frac{1}{N}\left(-\langle Z_k, y\rangle + \sum_{j=1}^{p}\langle Z_k, Z_j\rangle\beta_j\right). \tag{3.15}$$

$Z^TZ = I \implies \langle Z_k, Z_k\rangle = 1$ and $\langle Z_k, Z_j\rangle = 0$ for $j \neq k$ so we can rewrite equation (3.15) as

$$\frac{\partial}{\partial\beta_k}\frac{1}{2N}\|Z\beta - y\|_2^2 = \frac{1}{N}\left(\beta_k - \langle Z_k, y\rangle\right). \tag{3.16}$$

$\frac{\partial}{\partial\beta_k}\lambda_1\|G\beta\|_1$, 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 $\lambda_1\|\beta\| = \sum_{j=i}^{p}|\beta_j|$ 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

$$\beta_k \in \texttt{ND} = \left\{-\frac{r_{jk}}{g_{jk}}|j \in \{1,\ldots,p\}\right\} \tag{3.17}$$

With the exception of the separate case where $\beta = \vec{0}$ (the update for this case is given in equation (3.24) 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

$$\forall x, y \in \mathbb{R} : f(y) \geq f(x) + f'(x)(y - x) \text{ [12].}$$

Geometrically, we interpret this as the graph of f, never dipping below those of its tangent lines. 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:

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

For example, Figure 3.2 shows some subgradients of $f(x) = |x|$ at the non-differentiable point $x = 0$.

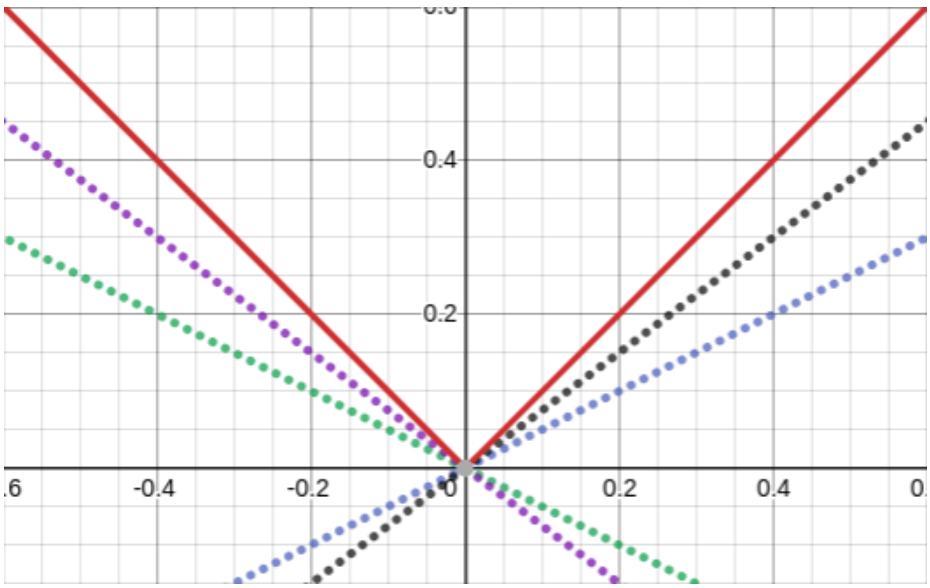


Figure 3.2: $\partial f(0) = [-1, 1]$ for $f(x) = |x|$. Any line $y = mx$ with $|m| \leq 1$ will lie below the graph of $|x|$ (in red), including the dotted $y = \pm\frac{1}{2}x$ and $y = \pm\frac{3}{4}x$

If f is differentiable at x, then the only line tangent to f at x has slope $f'(x)$, 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 $\beta_k \notin \texttt{ND}$. 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

$$c_1 = -\frac{r_k^{(1)}}{g_k^{(1)}}, \ldots, c_p = -\frac{r_k^{(p)}}{g_k^{(p)}} \tag{3.18}$$

be the points of ND sorted from least to greatest. This lets us define, for $\beta_k \notin \texttt{ND}$,

$$h'(\beta_k) = \begin{cases} s_1 = -\sum_{l=1}^{p} |g_k^{(l)}| & \beta_k < c_1 \\ s_2 = -\sum_{l=2}^{p} |g_k^{(l)}| + |g_k^{(1)}| & c_1 < \beta_k < c_2 \\ \quad \vdots & \\ s_m = -\sum_{l=m}^{p} |g_k^{(l)}| + \sum_{t=1}^{m-1} |g_k^{(t)}| & c_{m-1} < \beta_k < c_m \\ \quad \vdots & \\ s_{p+1} = \sum_{t=1}^{p} |g_k^{(t)}| & \beta_k > c_p. \end{cases} \tag{3.19}$$

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 $j = 1, \ldots, p$:

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

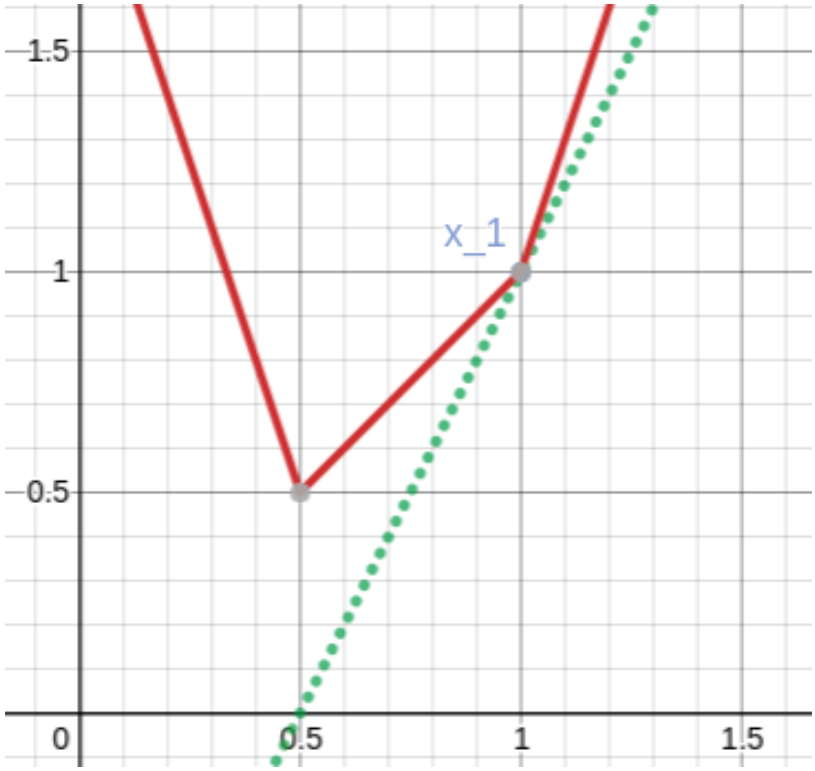


Figure 3.3: $\partial f(1) = [1, 3]$ for the red line $f(x) = |x-1|+|-2x+1|$. Any slope between the slopes of the portions of the graph immediately to the left and right of $x = 1$ is a subgradient. For example, the dotted green line has slope 2, a subgradient.

Figure 3.3 below gives a simple illustration of this fact.

Now, using equation (3.19) and the fact that if h is differentiable at β_k then $\partial h(\beta_k) = h'(\beta_k)$,

$$\partial h(\beta_k) = \begin{cases} s_1 & \beta_k < c_1 \\ [s_1, s_2] & \beta_k = c_1 \\ s_2 & c_1 < \beta_k < c_2 \\ \quad \vdots & \\ [s_m, s_{m+1}] & \beta_k = c_m \\ s_{m+1} & c_m < \beta_k < c_{m+1} \\ \quad \vdots & \\ [s_p, s_{p+1}] & \beta_k = c_p \\ s_{p+1} & \beta_k > c_p. \end{cases} \tag{3.20}$$

Combining equation (3.20) with the results of equations (3.14) and (3.16) shows that $L(\beta_k)$ is minimized when

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

where

$$a_k = \frac{1}{N} + \lambda_2 \langle G_k, G_k \rangle \tag{3.21}$$

and

$$b_k = \frac{1}{N} \langle Z_k, y \rangle - \lambda_2 \sum_{l \neq k} \langle G_k, G_l \rangle \beta_l. \tag{3.22}$$

To speed up the calculation of a_k and b_k, note that

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

so if we store $(X^T X)^{-1}$ from the process of calculating G, we can easily obtain $\langle G_k, G_l \rangle = ((X^T X)^{-1})_{k,l}$ for any k and l.

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 $m = 1, \ldots, p$ $(\beta_k = c_m)$:

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

   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 $m = 2, \ldots, p-1$ $(c_{m-1} < \beta_k < c_m)$:

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

   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 $\beta \neq \vec{0}$.

$$\tilde{\beta_k} \leftarrow S_m(b_k) = \begin{cases} \frac{b_k - \lambda_1 s_1}{a_k} & b_k < a_k c_1 + \lambda_1 s_1 \\ c_1 & a_k c_1 + \lambda_1 s_1 < b_k < a_k c_1 + \lambda_1 s_2 \\ \frac{b_k - \lambda_1 s_2}{a_k} & a_k c_1 + \lambda_1 s_2 < b_k < a_k c_2 + \lambda_1 s_2 \\ \vdots & \\ c_m & a_k c_m + \lambda_1 s_m < b_k < a_k c_m + \lambda_1 s_{m+1} \\ \frac{b_k - \lambda_1 s_m}{a_k} & a_k c_m + \lambda_1 s_{m+1} < b_k < a_k c_{m+1} + \lambda_1 s_{m+1} \\ \vdots & \\ c_p & a_k c_p + \lambda_1 s_p < b_k < a_k c_p + \lambda_1 s_{p+1} \\ \frac{b_k - \lambda_1 s_{p+1}}{a_k} & b_k > a_k c_p + \lambda_1 s_{p+1} \end{cases} \tag{3.23}$$

If $\beta = \vec{0}$, then $\forall j : r_{jk} = 0$ so $\texttt{ND} = \{0\}$ 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

$$\tilde{\beta_k} \leftarrow \frac{S(b_k, \lambda_1 \|G_k\|_1)}{a_k} \tag{3.24}$$

where S is the soft threshold function from the original update rule in equation (3.9).

3.9 Warm Start

A warm start is setting the initial weights to close to the final solution to reduce computation. This is mainly employed when solving a similar problem beforehand. The potential speedup from a Warm Start can range from none speedup to a total speedup where CD runs for no iterations. Section 5.14 shows the performance of the warm start concept.

3.9.1 Rolling Window

CD 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:

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

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

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 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:

$$\hat{\beta} = \underset{\beta}{\text{argmin}} \frac{1}{2N} \|X_{\text{sub}}\beta - y_{\text{sub}}\|_2^2 + \lambda_1 \|\beta\|_1 + \lambda_2 \|\beta\|_2^2 \tag{3.25}$$

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

Chapter 4

Coordinate Descent 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 32-bit floats (`float32`), which stores values with up to 7-8 decimal digits. However, eight 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 suggests that we do not need to store 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 $\approx$ 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 and ensembling methods

4.3.1 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 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 out training data $\mathbf{Z} = (x_1, y_1), \ldots, (x_N, y_N)$, 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 $\mathbf{Z}^{*b}$, $b = 1, 2, \ldots, B$, we fit our model, giving prediction $\hat{f}^{*b}(x)$. The bagging estimate is defined by $\hat{f_{bag}}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x)$. 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 $\mathbf{Z}^* = (x_1^*, y_1^*), \ldots, (x_N^*, y_N^*)$ 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 $B \to \infty$. 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

unchanges. Assume our training observations (x_i, y_i), $i = 1, \ldots, N$ 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^*, $i = 1, 2, \ldots, N$ 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 concept 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 $\hat{f}_1(x), \ldots, \hat{f_M}(x)$, under squared-error loss, we can seek the weights $w = (w_1, \ldots, w_M)$ such that

$$\hat{w} = \text{argmin}_w E_{\mathcal{P}}[Y - \sum_{m=1}^{M} w_m \hat{f_m}(x)]^2 \tag{4.1}$$

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 $\hat{F}(x)^T = [\hat{f}_1(x), \ldots, \hat{f_M}(x)]$:

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

Now the full regression has smaller error than any single model

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

3. Stacking: One issue with model averaging is that overfitting can occur 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 $\hat{f_m}^{-i}(x_i), m = 1, 2, \ldots, M$. In detail, the stacking weights are given by

$$\hat{w}^{st} = \text{argmin}_w \sum_{i=1}^{N} [y_i - \sum_{m=1}^{M} w_m \hat{f_m}^{-i}(x_i)]^2 \tag{4.4}$$

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 are obtainable 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 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 the fitting method finds many local minima, bumping can help 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 $\mathbf{Z}^{*1}, \ldots, \mathbf{Z}^{*B}$ and fit our model to each, giving predictions $\hat{f}^{*b}(x)$, $b = 1, 2, \ldots, B$ 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

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

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

4.3.2 MiniCD

Loading the data is the main bottleneck of our algorithm based on the front heavy method. This bottleneck gives motivation to solutions that 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 in Section 4.3.1. The general outline of MiniCD is given in Algorithm 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 subproblem. Step 4 is the most open-ended one, and there are many methods one can apply. We mainly focused on two of them: averaging and voting. General exploratory data analysis of the partial solutions of MiniCD can be found in Section 5.3. This analysis helps us get a better understanding of how the partial solutions relate to the ground truth.

Algorithm 5 MiniCD

1: Split the dataset (X, y) by rows into batches $(X_1, y_1), \ldots, (X_N, y_N)$.
2: Select m of the chunks $(X_{i_1}, y_{i_1}), \ldots, (X_{i_m}, y_{i_m})$
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 $w_1, \ldots, w_m$ to approximate the ground truth w

4.3.3 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 randomly choose the batches. We implemented both methods and recorded the results. Our methods are described below:

1. We partition the dataset into m equally sized batches of rows $S_1, \ldots, S_m$, and obtain solution estimates $w_{S_1}, \ldots, w_{S_m}$. Then, we averaged the estimates to get an approximation solution $\hat{w} = \frac{1}{m}(\sum_{i=1}^{m} x_{S_i})$. Then, we compared $\hat{w}$ to the ground truth w by using the following metrics: Euclidean distance, relative distance (dist/norm),

and non-zero support accuracy. We also experimented with the number of batches $m = 2, 3, 4, 5, 6, 7, 8, 10, 20, 50$. For each of these values, we recorded our results in the table (refer to table). We used $\lambda_1 = 5 \cdot 10^{-5}$ 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 result suggests that as we combine more estimates, the estimated solution tends to get further away from the ground truth in the above metrics. Also note 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, it will be non-zero in the predicted average. Another issue with averaging is that it introduces a one-sided bias, discussed in the next point.

2. We sampled X equally sized batches of contiguous rows $S_1, \ldots, S_N$ of the data X and obtained solution estimates $w_{S_1}, \ldots, w_{S_N}$. 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. 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 does not 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 introduce bias. It would be helpful to empirically understand what the bias exactly is and correct for it by adding a constant. The possibility to correct for bias leads us to exploratory data analysis of many randomly generated solutions w_i and how the feature entries are distributed with respect to the true values.

Figure 4.1 shows the magnitudes of all active (non-zero) coefficients in the ground truth solution against the magnitudes of the coefficients of the estimated obtained by

just taking the mean of $w_1, \ldots, w_N$. Using the Large dataset, we used all 493 days for our batches, where each batch corresponds to 33,000-36,000 rows of the data. As the plot shows, the ground truth solutions' values are consistently larger than the ones for the mean of the partial estimates. This remark empirically demonstrates the presence of a one-sided bias. Notice that as the feature's importance 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 with their estimated coefficients from the partial solutions. Each feature's distribution from the estimates is always closer to zero than the ground truth value for the feature. This observation supports the one-sided bias previously explained.

We can see a one-sided bias in those weights, whose side depends on the sign of the true weights. This one-sided bias could be because we did not sample random days, but we merely used all of the days or because we did not have enough samples. Nevertheless, the plot is convincing enough to show that some form of bias exists, confirming our theoretical considerations above. Unfortunately, we could not estimate this bias successfully and therefor did not correct for it. Thus, we decided not to pursue averaging partial solutions any further.

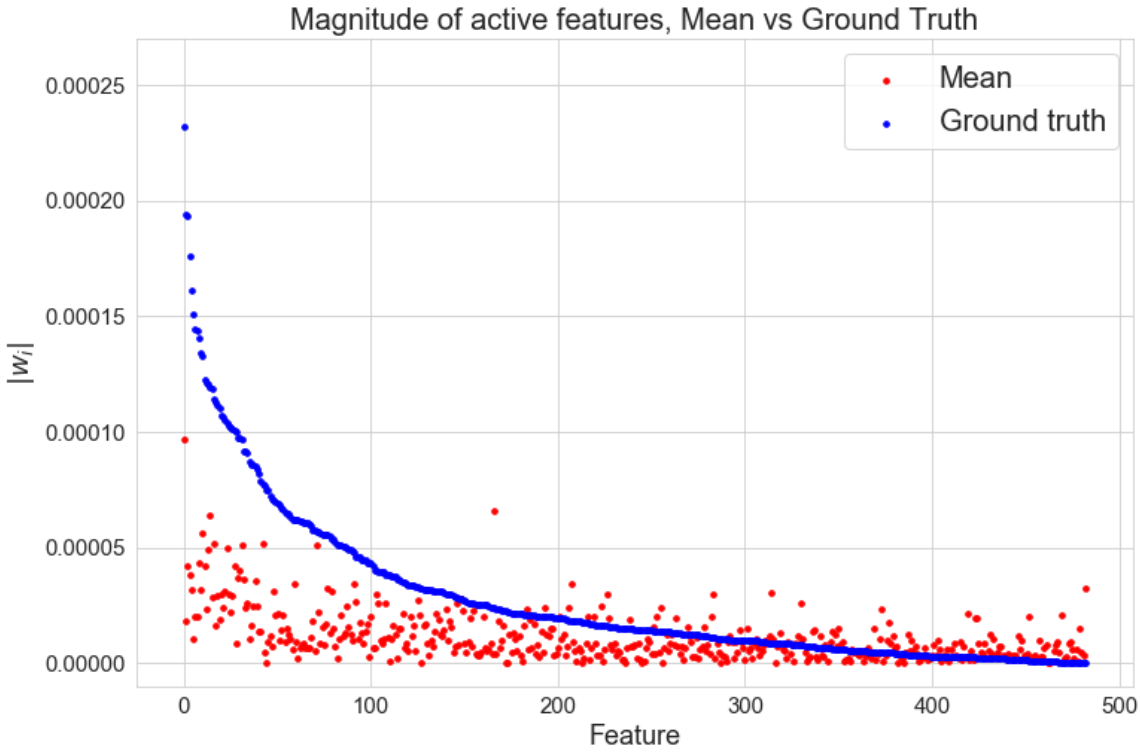


Figure 4.1: Magnitude of the active features in the ground truth sorted in a decreasing fashion and the estimate obtained by taking the mean of the partial solutions. We can see a form of one-sided bias for the more prominent features.

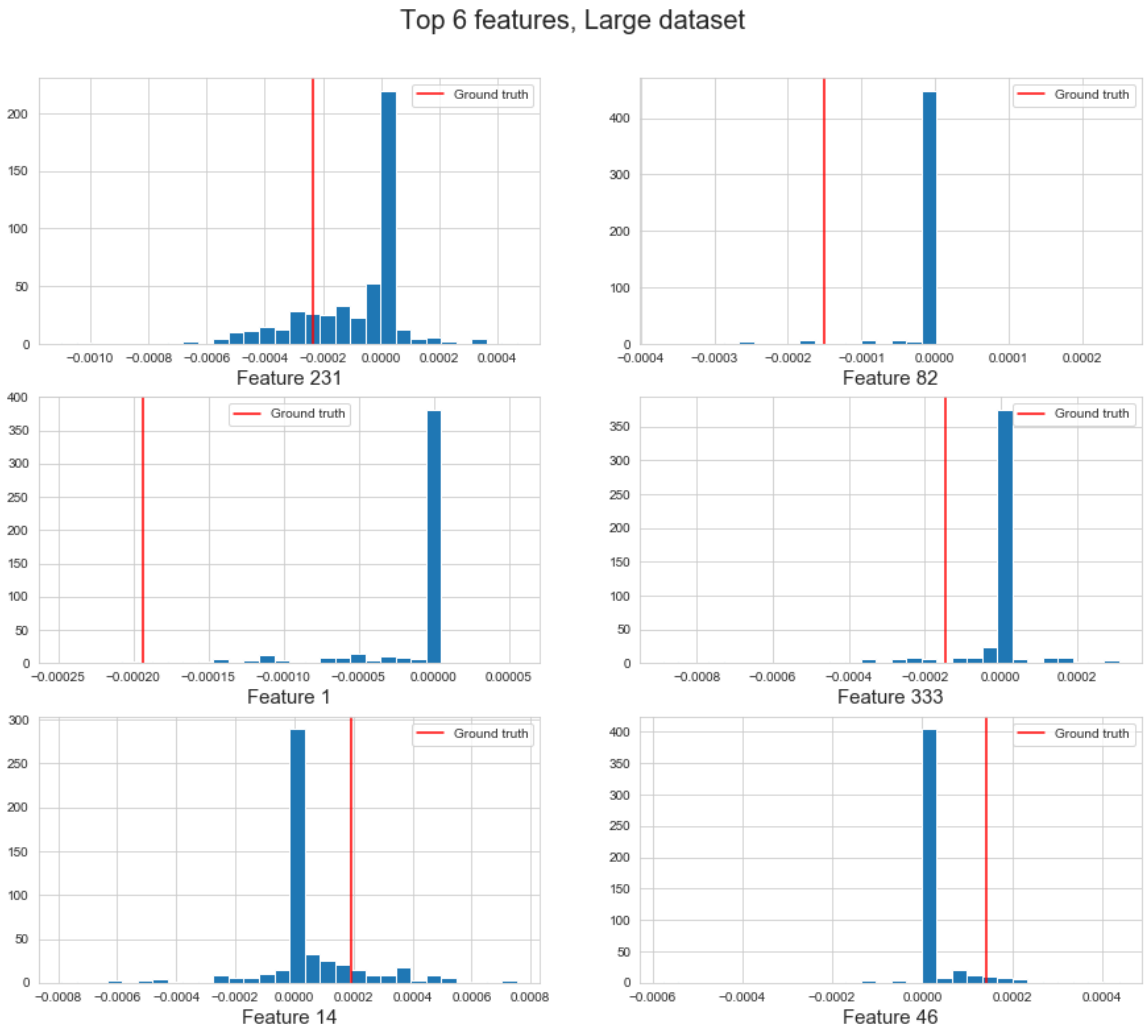


Figure 4.2: The top 6 features in magnitude for the Large dataset. The red vertical lines represent the true values of the coefficient, and the blue histogram represents the distribution of the coefficient over the partial solution for each of the 493 days.

4.4 Ensemble Methods

4.4.1 Ensembling with MiniCD

By using the MiniCD concept explained in Section 4.3.2, 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. When removing these weights from the computation, the problem's dimensions reduce and, consequently, the time of CD shortens. 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.

Ensemble methods commonly use a voting scheme to combine different solutions. This

report implemented two voting methods, threshold, and tree-based voting, to estimate which features are nonzero (active) and zero (inactive) in the ground truth.

4.4.2 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. 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** $k \in \{1, 2, .., p\}$ **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 larger than a given threshold, then this feature is considered active, and otherwise, it is not active. Once we have determined which features are active and which are not, we run CD only on the features claimed as 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 if the true active set of features is not entirely captured.

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 inactive features are highly inseparable 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. The trade-off between recall and precision shown in Section 5.4 further shows this inseparability. Results and analysis for our results from threshold voting are shown in Section 5.4.3.

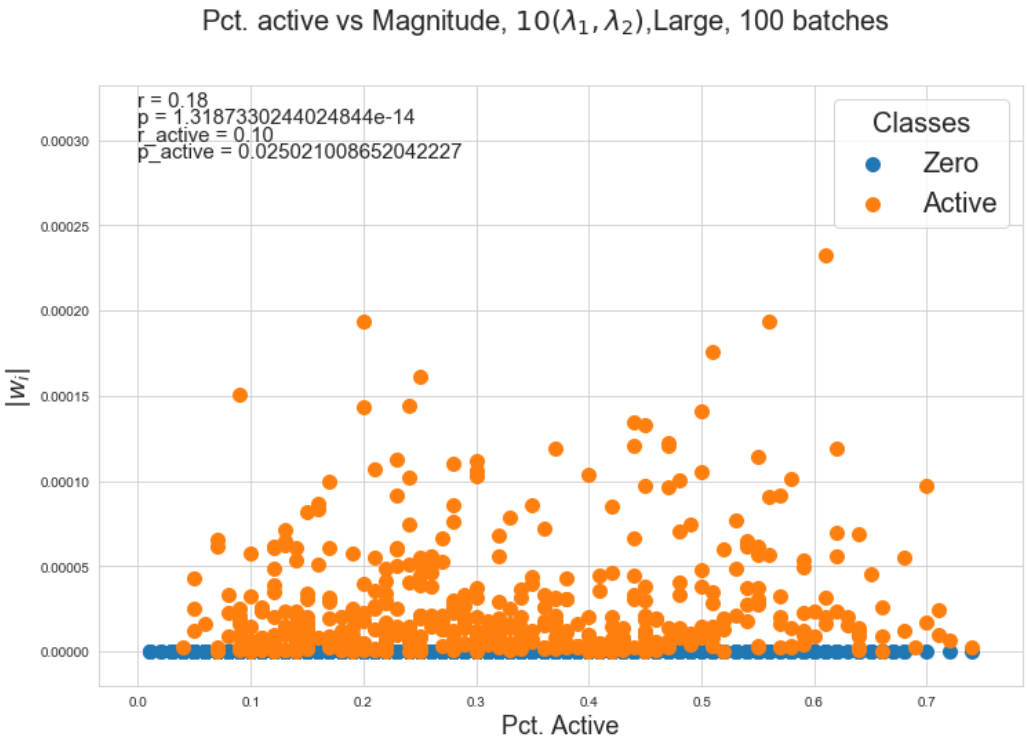


Figure 4.3: Percent active vs magnitude of a coefficient for the Large dataset. 100 batches (days) of size 35,000 were used to compute the percent active of each feature.

4.4.3 Tree Voting

Within voting via a threshold, each feature has an equal impact on the final decision. We also wanted to implement a voting scheme where some features may have a more significant impact on the final solution than others. One voting method where this inequality occurs is via 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.

This process initially starts with a set of weights, S, where each row is an individual feature resulting from all batches, and each column is the full set of weights from one individual batch. For a given feature k, this method takes S_k, a row, and assigns a node to each vector value. Initial ordering of node assignment is unnecessary as each value is calculated via a random sampling of data through the MiniCD method. Once each value has a node, Algorithm 7 determines the active set of features.

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 ∈ {1, 2, .., 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
```

The algorithm takes many different values for a given feature and condenses them to a single value, active or inactive. Within the algorithm, three nodes aggregate at a time, where a majority vote occurs. If the majority of the nodes are active, the consolidated node will be active, and the opposite occurs if the majority of the nodes are inactive. This process repeats until one node remains. There are special cases defined in the algorithm for when less than three nodes aggregate. The tree structure uses a ternary structure since a majority vote for odd numbers cannot result in a tie. However, there is no reasoning for defining a tree around five nodes or larger odd numbers.

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, a large bias towards one node may arise on how many nodes there are. One example of this bias is in Figure 4.4. In this case, the last node with value 0 is compared to the condensed node with value one based on nine original nodes. Using a majority vote here would result in large bias as one node would have an equal weight to the remaining nine. As such, line 8 of Algorithm 7 chooses the left node over the right node when comparing two nodes. This rule will never result in larger inequalities for

each node's importance because the trees by design will either be symmetric or left-sided. Additionally, this method can expand to a random forest-like structure, where each node represents a voting tree itself. Results and analysis for our results from tree voting are shown in Section 5.4.3.

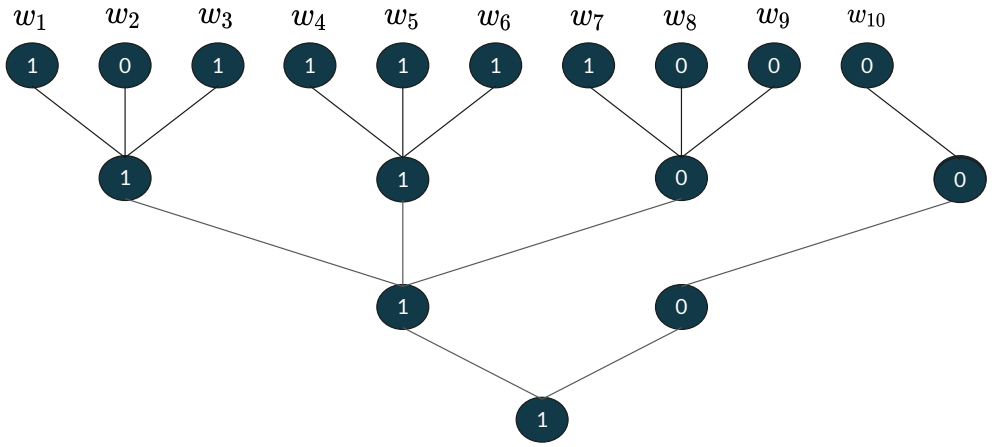


Figure 4.4: Arbitrary case with 10 inital nodes such that the node from w_{10} would have much stronger impact on the final decision without line 8 of Algorithm 7.

4.5 Early Stopping

Early stopping is a method of stopping an optimization algorithm early to reduce overfitting. However, it also can increase speed since the algorithm stops before unnecessary computation. Early stopping depends on the problem space and is determined by a tolerance parameter c. We can stop CD after the duality gap (η) is smaller than 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, we could significantly reduce the computation. 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 duality gap's computation would be to determine beforehand analytically how many iterations CD requires to converge. However, this runs off the risk of not running CD for long enough to converge and thus yielding a lower accuracy than we would like. Section 5.5 shows the impact of the early stopping concept on our CD algorithm.

Chapter 5

Results

5.1 Hardware and Data Specifications

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. The Medium and Large datasets are too large to fit into memory, so we use them to test our methods of overcoming memory bounds.

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 to achieve the 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.

Based on Tables A.1 and A.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 $(\lambda_1, \lambda_2) = (5\text{e}{-5}, \text{e}{-1})$ and $(\lambda_1, \lambda_2) = (\text{e}{-5}, \text{e}{-2})$ respectively.

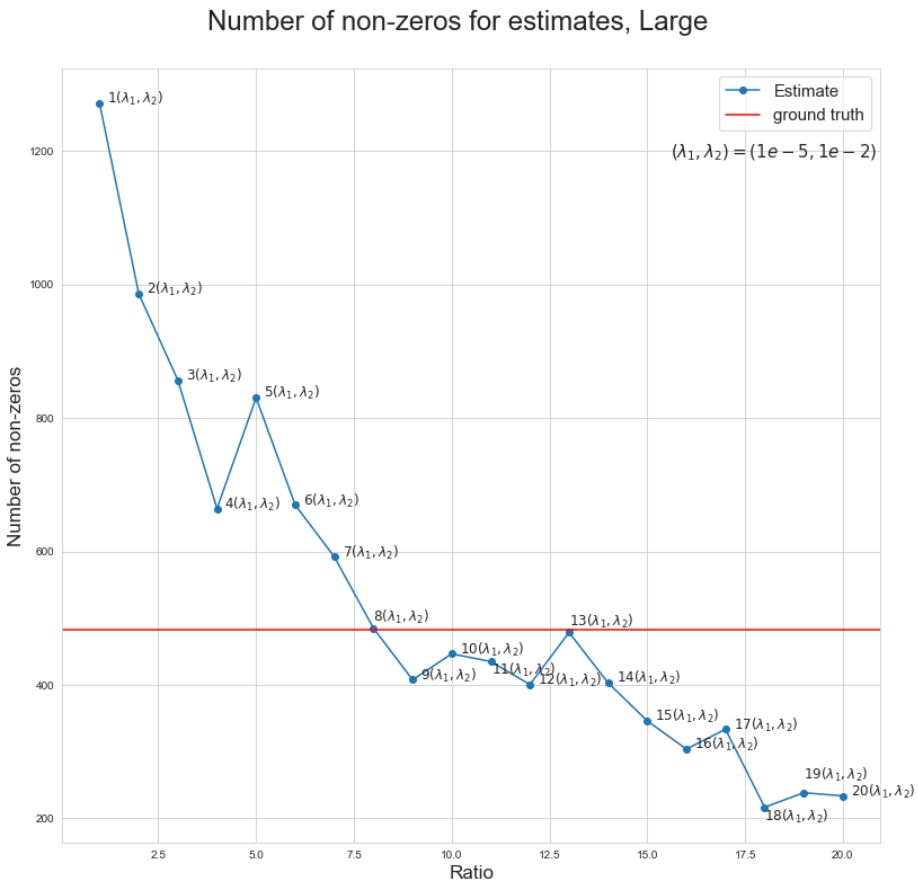
When implementing the voting thresholding method, we wanted to use values of (λ_1, λ_2) for step 3 of Algorithm 5 so that the sparsity of the partial solutions matches the sparsity of the ground true solutions. We wanted this property to hold so the voting thresholding method would not be biased to select fewer 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 $(\lambda_1, \lambda_2) = (1.5\text{e}{-4}, 3\text{e}{-1})$ for Medium and $(\lambda_1, \lambda_2) = (\text{e}{-4}, \text{e}{-1})$ for Large.

5.3 MiniCD exploratory data analysis

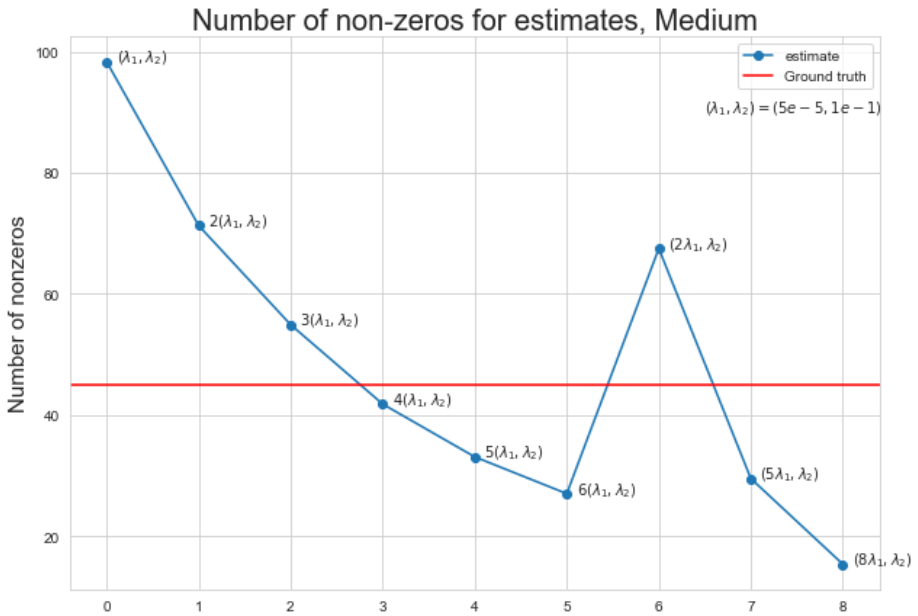
Here we present exploratory data analysis for the partial solutions from MiniCD introduced in Section 4.3.2. We solved the elastic net problem for each of the 493 days in the Large dataset and obtained 493 estimates. We wanted to visualize how does the support of theses partial solutions vary across 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 $17281517 \cdot c$ 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 trend follows our intuition 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 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 behavior. There is little variability for the extreme values of the batch size, and the variability amount 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 $17281517 \cdot c$ 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 decrease. We expect this trend 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.



(a)



(b)

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

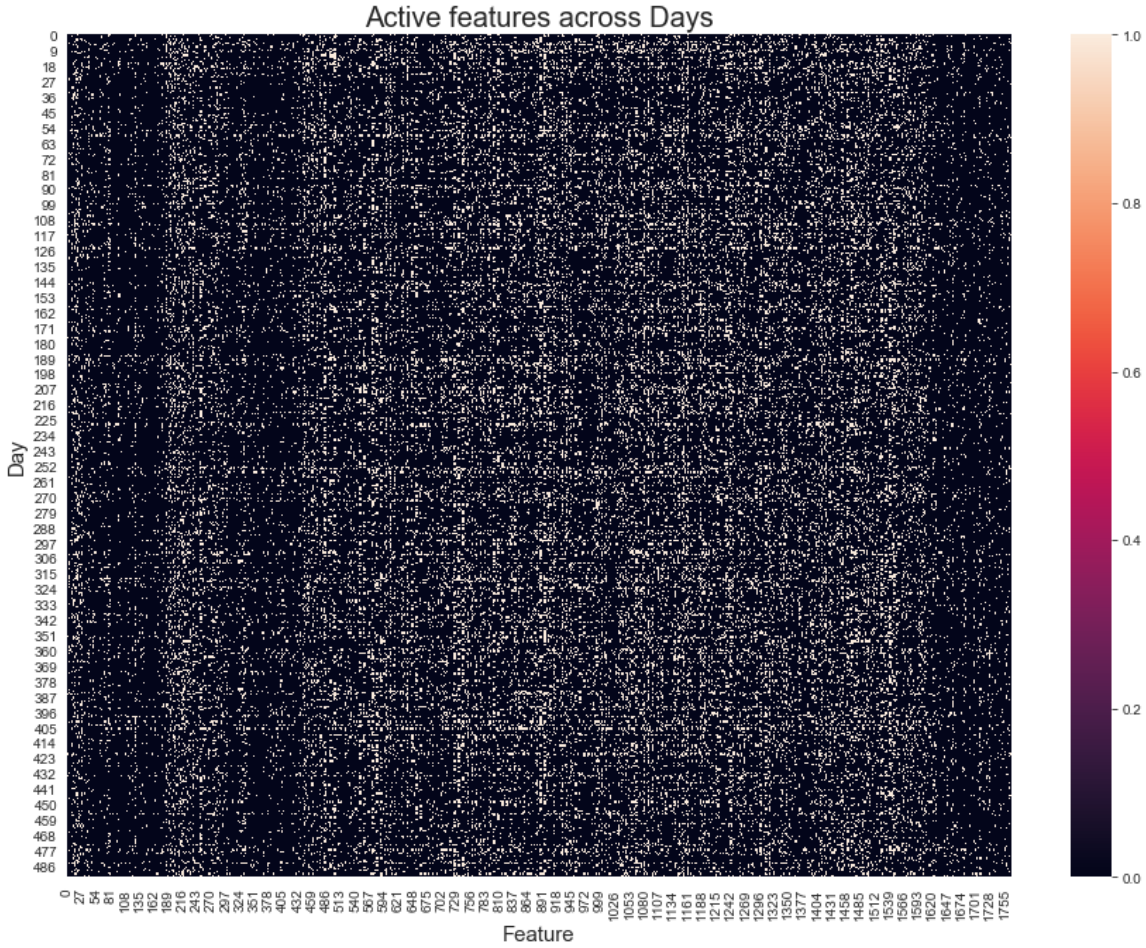


Figure 5.2: Active features across days in the Large dataset. A white square indicates the a feature is active and a dark square indicates that a feature is inactive(zero)

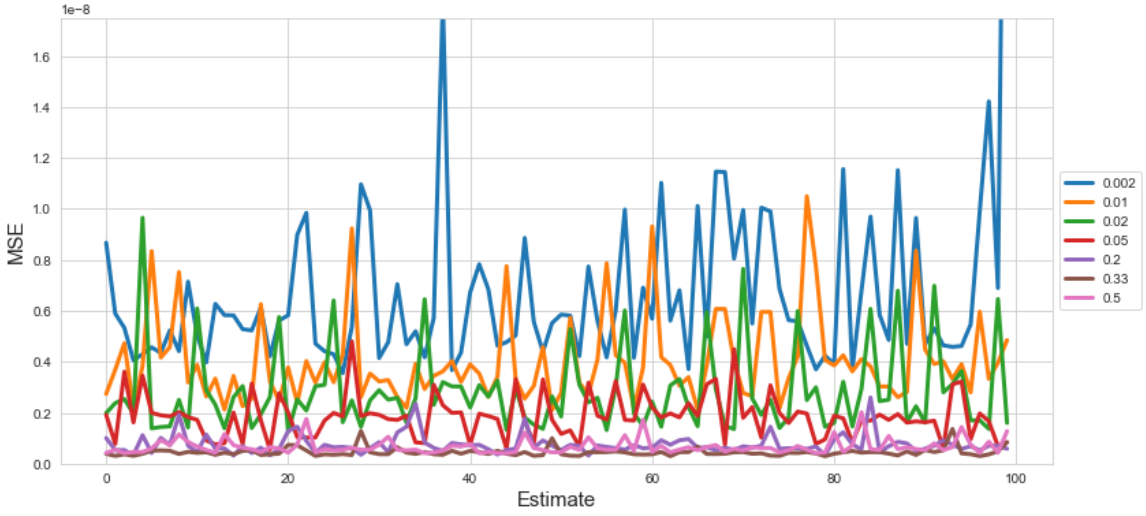


Figure 5.3: Lineplot of estimate vs MSE between the estimate and the ground truth soluton over different batch sizes on the Medium dataset.

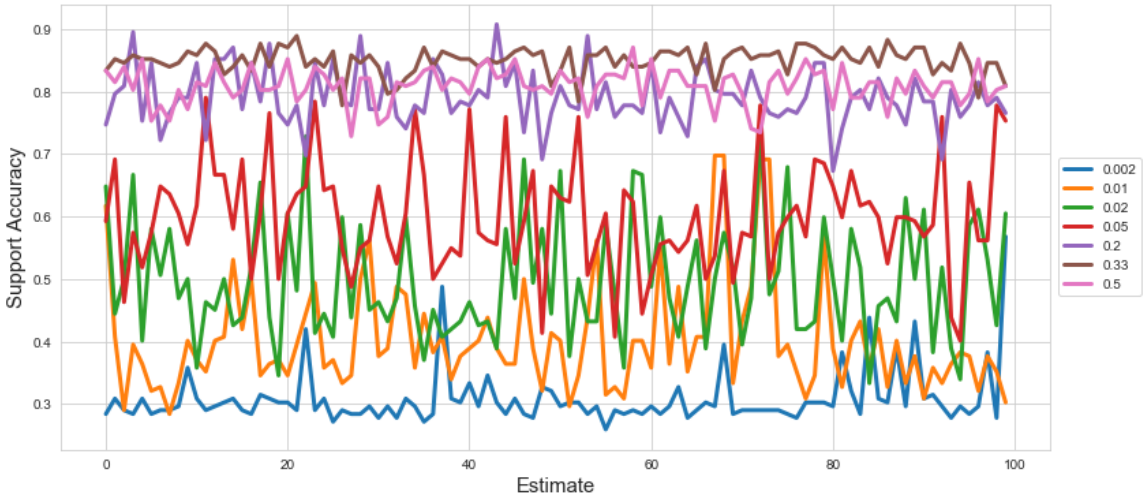


Figure 5.4: Lineplot of estimate vs non-zero accuracy (support accuracy) between the estimate and the ground truth soluton over different batch sizes on the Medium dataset.

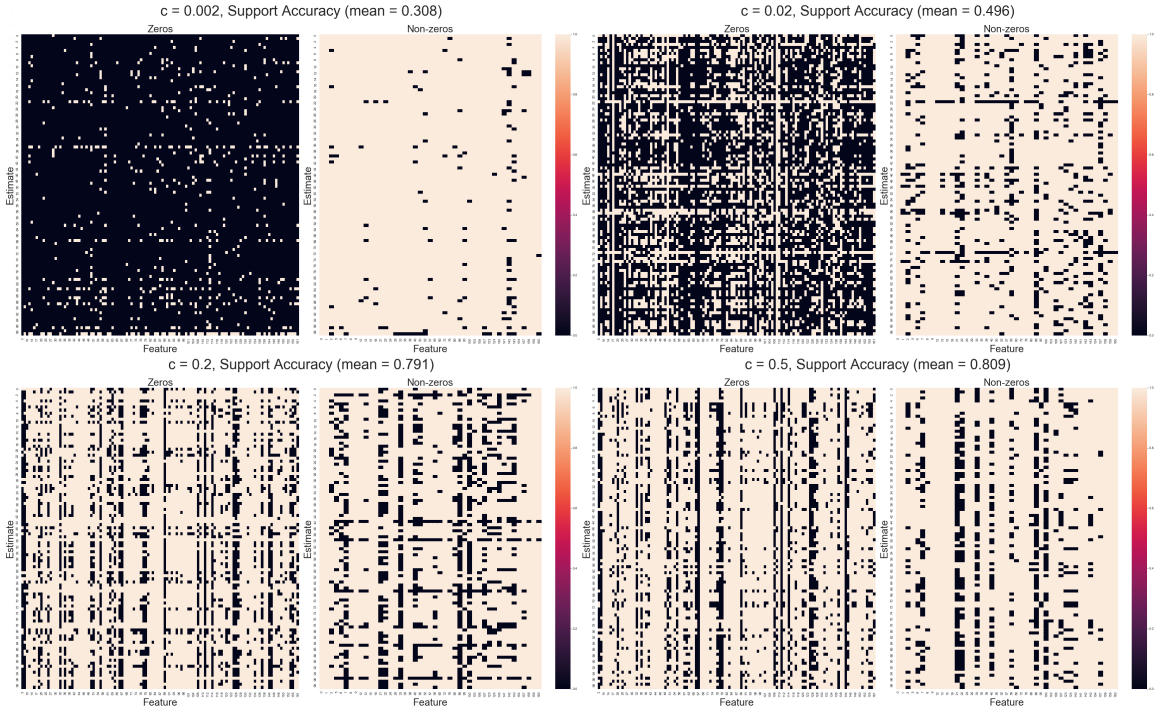


Figure 5.5: Support accuracy across batch sizes for $c = 0.002, 0.02, 0.2, 0.5$ on the Medium dataset. The x-axis is the feature and y-axis the partial weight estimate. A white square indicates a match and a black a mismatch.

5.4 Study of Ensembling Methods

5.4.1 Active Features based on λ for Tree Voting

Studies in Figure 5.1 of Section 5.2 shows multiples of the λ_1 and λ_2 and how those values determine the number of non-zeros produced by CD on a small batch. However, the number of active features after using an ensemble technique can vary based on technique and the number of batches. For threshold voting, this is easily adjustable through the threshold value. The adjustability of a threshold value does not translate over to the tree-based voting format. To compensate for this lack of customization, we can adjust the initial input weights to the tree ensemble method via the λ_1 and λ_2 values. This alternative method indirectly varies the number of active features resulting from the voting scheme.

m \ α	1	2	3	4	5	6	7	8	9	10	11	12
1	1200	1079	833	770	836	577	565	625	461	475	569	493
3	1396	1121	926	770	694	636	532	376	310	317	335	340
5	1423	1066	927	767	648	526	488	422	467	338	309	270
10	1505	1167	924	752	628	574	402	386	286	230	180	213
15	1488	1226	849	785	651	538	463	350	270	205	209	227
20	1267	911	616	451	350	257	204	153	122	135	97	74
50	1598	1232	928	726	555	432	386	268	228	218	194	149

Table 5.1: Number of active features for varying λ_1 and λ_2 for tree voting, and tree voting

Table 5.1 shows the average number of active features resulting from tree voting based on the number of batches, m, and the multiplier, α, used for λ_1 and λ_2, which were predetermined in Section 5.2 for the Large dataset. As α increases, the number of active features decreases. This is supported by the results from Tables A.1 and A.2, where using larger λ_1 values results in sparser weight vectors.

For the remaining studies with the tree voting scheme, we use $\alpha = 10$ for consistency with the threshold voting method. However, this table points out that using $\alpha = 10$ results in around 200 active features for larger batches, where the weights considered as ground truth have 400-500. It is possible that using a different α may result in more accurate results when using the tree voting method.

5.4.2 Metrics for Ensembling Methods

We performed a comprehensive study of both the threshold voting (Section 4.4.2) and the tree voting (Section 4.4.3) 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 tree voting study we used the same set for m with the same λ_1 and λ_2 values as the threshold voting. Both voting mechanisms use $(\lambda_1, \lambda_2) = (\mathrm{e}{-4},\ \mathrm{e}{-1})$ as described in Section 5.2. We recorded the following metrics:

1. Number of active features determined by the voting. The results are shown in Table A.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. (precision $= \frac{\mathrm{TP}}{\mathrm{TP+FP}}$, where TP = true positive and FP = false positive). The results are shown in Table A.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. (recall = $\frac{\text{TP}}{\text{TP+FN}}$, where FN = false negative). The results are shown in Table A.5.

4. We record the metric $\frac{\text{MSE}_w - \text{MSE}_{w^*}}{(\overline{|y|})^2}$, where $MSE_v = \frac{1}{N}||y - Xv||_2^2$ is the mean squared error for a weights vector v and $\overline{|y|} = \frac{1}{N}\sum_{i=1}^{N}|y_i|$ is the average absolute value of the target variable. Notice that since $(\overline{|y|})^2$ and $\text{MSE}_w - \text{MSE}_{w^*}$ are on the same scale, this metric is invariant under scaling and thus does not depend on units. The interpretation of this metric is the difference between the MSE of the estimate w produced by threshold voting to the MSE of the ground truth w^* taken with the sign to account for whether the MSE decreases or increases. For reference $\text{MSE}_{w^*} = 0.00032973$ and $\overline{|y|} = 0.012114$.

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 the following: how close the estimated weights are to the ground truth with respect to the magnitude of the ground truth. We call this metric the relative distance. The results are shown in Table A.7.

6. The quantity $\frac{(w-w^*)^T\Sigma(w-w^*)}{\text{MSE}_{w^*}}$, where $\Sigma = \frac{1}{N}X^TX$ is the sample covariance matrix. This quantity tells us the mean square error between the predictions given by w and w^*, which are Xw and Xw^* respectively, normalized by MSE_{w^*}, which makes the metric invariant under scaling. The interpretation of this quantity is how close the predictions are given by w and w^* are compared to the MSE of w^*. The normalization makes the results more interpretable as the real values are very small.

7. We also measured the time it took to run MiniCD for each pair (t, m) and recorded the times in Table A.9.

8. ROC curve for threshold voting for $m = 100$ batches shown in Figure 5.7. The x-axis shows the False Positive Rate ($\frac{\text{FN}}{\text{FN+TN}}$) and the y-axis show the True Positive Rate or recall ($\frac{\text{TP}}{\text{TP+FN}}$). In general, a ROC curve indicates the best performance at classification if the orange curve peaks in the upper left corner. The dashed blue line shows the curve of no predictive power (i.e., a random guess). Points above the diagonal represent good classifications, and points below represent bad classification results. The area below the curve is the probability that a classifier will rank a randomly chosen active feature higher than a randomly chosen zero feature. Thus, it is also an indicator of classification performance.

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

5.4.3 Analysis of the Ensembling Studies

Here we present an analysis of the results from threhsold voting (Section 4.4.2) and tree voting (Section 4.4.3). The full tables with our results are in the Appendix to Chapter 5 and here we will simply reference them and analyze them. Tables A.3-A.9 provide insight into the performances of both threshold and tree-based voting schemes. Each table displays a metric of accuracy based on a range of m, where m represents the number of batches. When selecting an individual batch, a measurement of accuracy can vary greatly due to the little representation a batch has over the entire dataset. However, when using many batches,

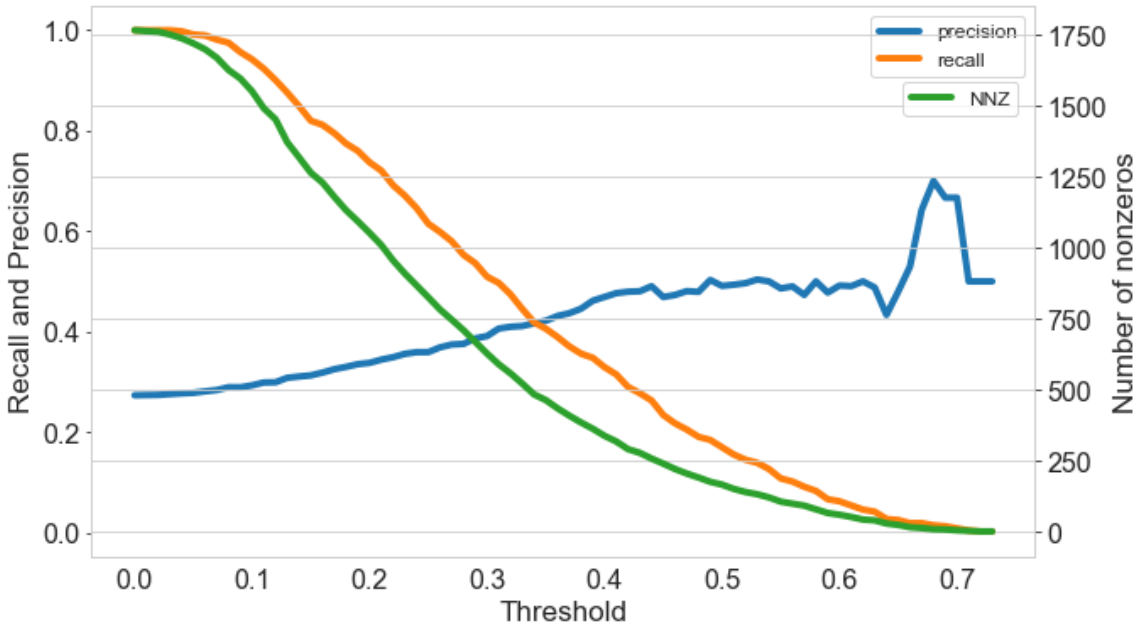


Figure 5.6: Trade off between precision and recall for $m = 100$

this variability in the accuracy measurements should decrease as the summation of data observed through many batches is more representative of the entire dataset. Furthermore, as an ensembling method takes additional batches, the significance that an outlier set of weights from one batch would have on the accuracy decreases.

1. Selecting the support of the ground truth

One result shown in Table A.3 is that as the threshold increases, the number of active features decreases. This result supports our intuition as a higher threshold is more selective and therefore, should select less active features. It is important to note that for $m = 1$, each threshold value results in the same number of active features. There is no need for thresholding since only one batch determines active v. nonactive weights. The different number of active weights between the thresholding methods and the tree methods for $m = 1$ display that the studies for tree voting selected batches are independent of the batches used for the threshold voting results. In other words, these studies were computed separately. Regardless, the tree voting scheme shows results similar to $t \in [0.3, 0.6]$. Generally, the number of active features increases for small threshold values and decreases for large threshold values when the number of batches increases. This observation is supported by the description of batch number influence on results in the previous paragraph.

To see how well the thresholding and tree voting methods capture the true support of the ground truth, we compute the precision and recall metrics in Table A.4 and Table A.5 respectively. We notice that the precision tends to increase slightly for a fixed batch size as we increase the threshold while the recall decreases more drastically. This relationship shown clearer for $m = 100$ in Figure 5.6, where the lines for precision (blue) and recall (orange) are visually inversely proportional. The green line shows the number of non-zeros (active features). It is also important to note that the "bump" for $t \in (0.6, 0.9)$ in the precision is probably due to noise because of the very few active features (around 10) and the fact that batches are selected randomly. We notice that precision and recall are equal at a threshold of roughly $t = 0.33$. Besides the apparent trade-off between precision and recall, we can also notice in Table A.4 that almost all of the precision values are below 0.5, which indicates

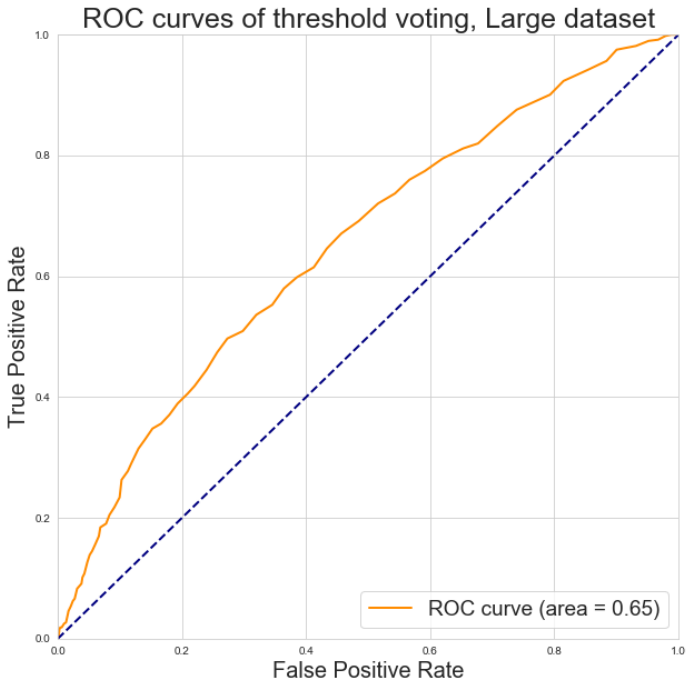


Figure 5.7: ROC curve for $m = 100$

that threshold voting and tree voting does not do a great job at exactly capturing the true support of the ground truth. To further investigate the ability of threshold voting to select the true support, we can look at the ROC curve for $m = 100$ batches in Figure 5.7. In this plot we fix $m = 100$ and vary t from 0 to 1. We see that the ROC curve of threshold voting is slightly above the diagonal, but closer to the diagonal than to the upper left corner. This trait of the plot indicates that thresholding voting has some predictive power in selecting the true support, but it does only moderately well. We also see that the AUC is 0.65 which is around 3 times closer to 0.5 than to 1. Based on that and our observations above from Figure 5.6 and tables A.4, A.5 we can conclude that threshold voting does a moderate job at predicting the true support, while its performance is far from optimal. As noted above, the tree voting results are similar for those for threshold voting for $t \in (0.3, 0.6)$, and thus we can make the same conclusion for tree voting.

2. Approximating the ground truth weights

Next, we analyze how close this solution is to the ground truth in terms of the metric $\frac{|w-w^*|}{|w^*|}$, shown in Table A.7. We interpret this metric as the relative distance. We notice that the relative distance generally increases for fixed batch sizes as we increase t and converges to 1. This trend is expected since the higher the threshold t, the closer the solution given by threshold voting is to the zero vector, which has a relative distance of 1. We also notice that for all values of t plus the tree voting, the relative distance tends to "mostly" decrease as we increase the batch size m. This pattern is more pronounced for the lower values of $t = 0.1, 0.2, 0.3, 0.4$ and starts to get less pronounced for $t \geq 0.5$. This tendency is most likely because when $t \geq 0.5$ as we see in Table A.3, the number of active features left in the model becomes very small and thus noise gets introduced as the batches are sampled randomly. In general, most of the relative distance values are > 0.7, with some exceptions for $t = 0.1, 0.2$ and $m > 10$, where the values range in $0.269 - 0.660$. Notice that in the first set of pairs (m, t) correspond to < 800 active features and the latter to $946 - 1523$ active features according to Table A.3. These results suggest that to get an estimate close to the

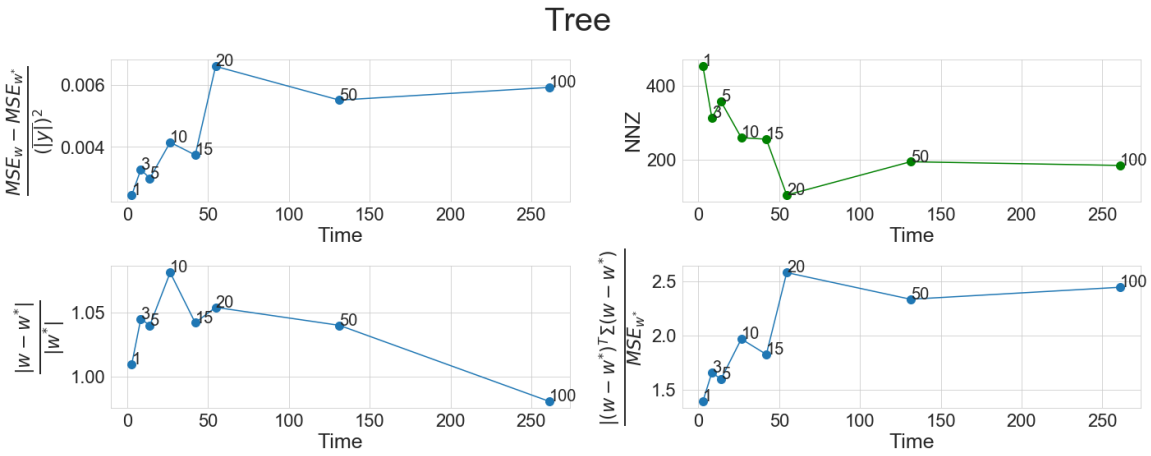


Figure 5.8: Plots of two accuracy metrics against time for Tree voting on different values for m

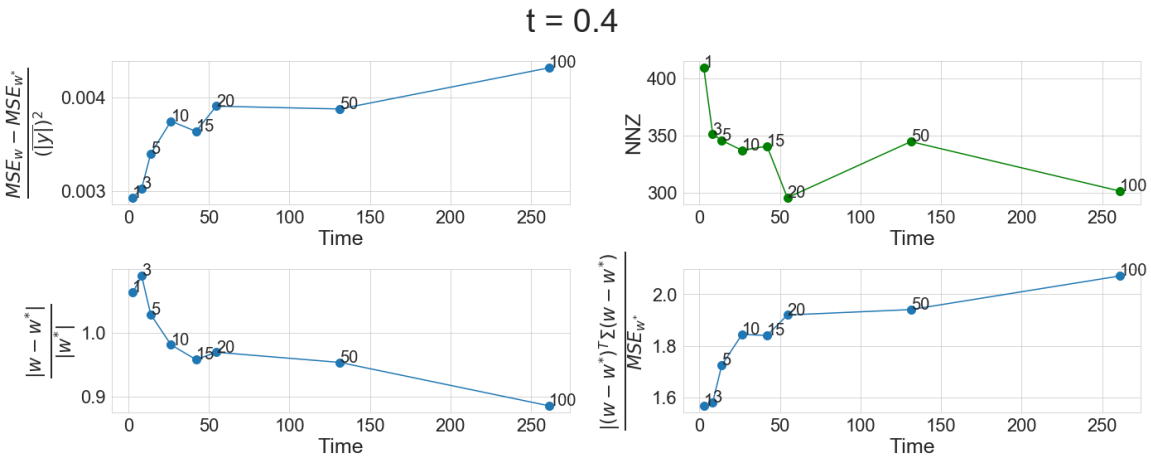


Figure 5.9: Plots of two accuracy metrics against time for threshold voting ($t = 0.4$) on different values for m

ground truth, we need to include most of the total features in the model and cannot do it by including a number of features that are close to the true number of active features (483). This observation is consistent with the previous paragraph's conclusion as threshold voting and tree voting do not do a great job a selecting the supports.

3. Approximating the target

Furthermore in Table A.6 we show the metric $\frac{\text{MSE}_w - \text{MSE}_{w^*}}{(|y|)^2}$. As we can see, all the values in the table are positive, and thus, none of our estimates from both voting schemes produces a lower MSE than the ground truth. Moreover, since our metric is normalized and all values in the table are < 0.02, we can conclude that in general, the difference between the MSE produced by voting is close to the MSE produced by the ground truth. We can also notice that the MSE produced by the threshold voting solution for a fixed batch size tends to increase as we increase t. We expect this trend as a higher threshold value should select fewer active features, and thus the solution gets farther from the ground truth. In general we can we a pretty significant difference between $t = 1$ and $t = 0.9$ as the batch size increases, where the values for $m = 100$ are 0.00009 and 0.01045 respectively. For $t = 0.3$, the threshold value that best balanced the tradeoff between precision and recall from Figure 5.7, we see that the values are in the range 0.0007-0.002. For the tree voting,

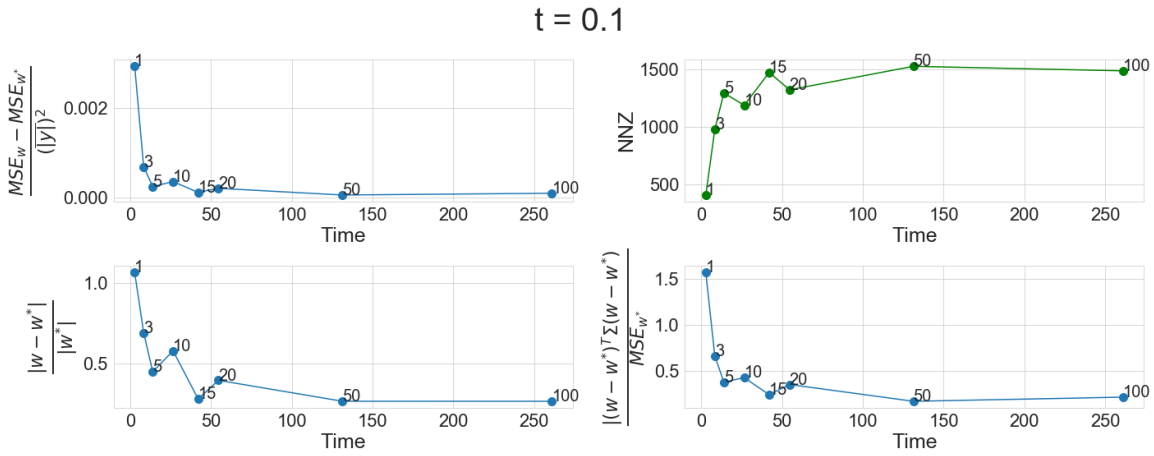


Figure 5.10: Plots of two accuracy metrics against time for threshold voting ($t = 0.7$) based on different values for m

the values are in the range 0.002-0.007. These results indicate that both threshold and tree voting do very well in approximating the ground truth in terms of MSE to the target, which means that if our ground truth gives good predictions of the target, so will the solution from MiniCD vice versa.

In Table A.8 we see the metric $\frac{(w-w^*)^T\Sigma(w-w^*)}{\text{MSE}_{w^*}}$ for different thresholds and the tree method. This metric computes the mean squared error between the predictions given by w and w^*. Based Table A.8, most values are > 1, except for the cases where $t = 0.1, 0.2$, in which cases more than half of the features are left in the model. This trend indicates that, for the most part, the predictions given by threshold voting and tree voting tend to be far from the predictions given by the ground truth. However, this conclusion is counter-intuitive with the previous paragraph's conclusion, but a reasonable explanation would be that the solutions from voting approximate the predictions from different sides. This solution would explain why their MSEs from the target are close to each other, but the actual predictions are not as close. We can also observe a general trend that the higher the number of active features left in the model, the closer the predictions from threshold and tree voting are to the ground truth predictions.

4. Runtime

Finally, in Table A.9, we measure the time it took to run threshold and tree voting. Notice that the times are roughly invariant for different thresholds and the tree voting for a fixed batch size. This invariance is because the time spent doing the actual voting and the coordinate descent is negligible compared to the time it takes to load the batches and compute the gram matrix and the inner products with the target. This result also explains why for a fixed t or tree, the times increase linearly with the number of batches. For $m = 1$ batch it takes around 2.784 seconds and the times increase almost linearly to around 260 for $m = 100$.

To combine our observations from the three metrics above and the NNZ (Number of non-zeros), we can look at Figures 5.8- 5.10. The three figures correspond to tree voting and and threshold voting with threshold $t = 0.1, 0.4$. The x-axis shows the time, the y-axis shows each one of the 4 metrics and each point corresponds to a different batch size $m = 1, 3, 5, 10, 15, 20, 50, 100$. Firstly, we can see a general trend in all 3 plots that as the number of NNZ increases, both metrics $\frac{(w-w^*)^T\Sigma(w-w^*)}{\text{MSE}_{w^*}}$ and $\frac{\text{MSE}_w - \text{MSE}_{w^*}}{(|y|)^2}$ decrease (i.e. we become more accurate) and when the number of NNZ decreases these metrics increase.

However, this trend happens in inverted fashion for $t = 0.4$ and tree versus $t = 0.1$. For $t = 0.4$ and tree voting we see that generally both $\frac{(w-w^*)^T\Sigma(w-w^*)}{\text{MSE}_{w^*}}$ and $\frac{\text{MSE}_w - \text{MSE}_{w^*}}{(|\overline{y}|)^2}$ increase with the batch size and NNZ decreases, while for $t = 0.1$ this metrics decreases and the NNZ increases. We can also observe that the metric $\frac{|w-w^*|}{|w^*|}$ has an "elbow" shape for $t = 0.1$ and $t = 0.4$ (Figures 5.10 and 5.9), which indicates a trade-off between runtime (number of batches) and how close we are to the ground truth in terms of relative distance. Value $m = 5, 10, 15$ around the "elbow" seem to be the best choices if we solely want to optimize the relative distance to the ground truth and also the runtime for $t = 0.1, 0.4$. We don't observe this "elbow" shape for the tree voting. Instead we see an increase until $m = 10$ and a decrease afterwards. In this case as seen in the Figure 5.8 the best value should be $m = 1$, because it achieves a slightly bigger relative distance only than $m = 100$ but it is 100 times faster.

5.5 Early Stopping Study

In Section 4.5, we introduced how we can reduce the 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 can converge in around 120 iterations.

Originally, we started checking the duality gap once the maximum absolute difference between weight values was below 1e-4. With this value for the Large dataset, we can 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 do not exceed 0.0001. However, when we change this tolerance to 1e-7, we achieve a time of 0.33 seconds and 120 iterations. With this new tolerance, we only check the duality gap on average two times. So, from the speedup, the duality gap took up around half of the computation. When we lower the threshold even further to 1e-8, 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 occurs 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 once every

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

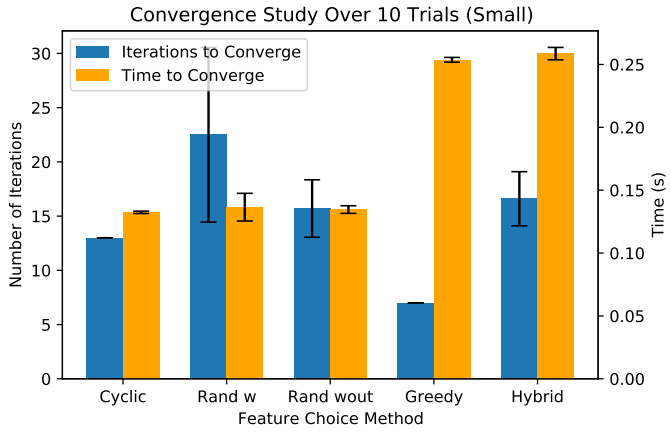
Table 5.2: Average times of CD on Large dataset over 10 trials for various threshold values and hybrid methods. Greedy → Cyclic = hybrid method starting with greedy and ending with cyclic feature selection. This also applies to Greedy → Rand w and Greedy → Rand wout, where *Randw* = random with replacement and Rand wout = random without replacement.

iteration, restricting any switch from one method to another to occur once an iteration has finished.

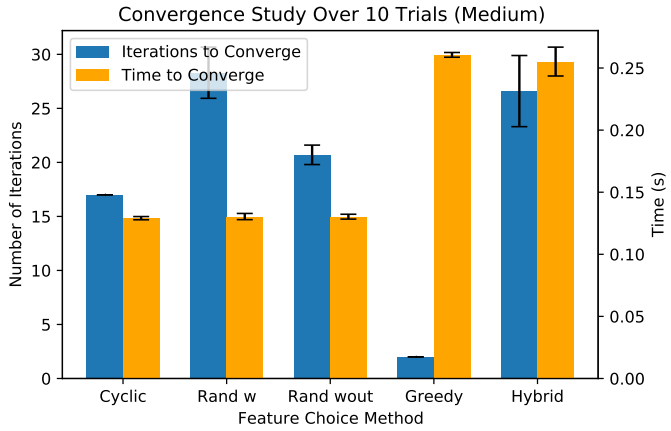
Table 5.2 shows performance times of variant hybrid methods, all starting with greedy feature selection switching the different basic feature selection methods at different threshold values. The result shows that the performance times plateau as the magnitude of the threshold value increases. As the threshold value increases, CD runs through fewer 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 5.2; a feature selection method alternates from greedy selection to random with replacement selection with a threshold value of 1e-4.

5.6.2 Convergence study of feature selection methods

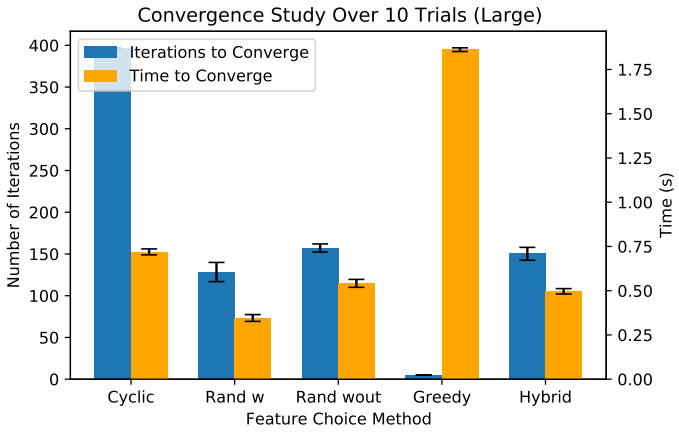
We conducted performance studies comparing overall time and iterations to converge for the feature choice methods described in Section 3.2. The convergence for the CD method was determined via the convergence of the dual gap value explained in Section 2.7 using the optimal results from the early stopping study in Section 5.5. We don't include ACF methods described in Section 3.2.4 due to issues with `float` precision in calculating the extremely small value $f(\boldsymbol{\beta}^t) - f(\boldsymbol{\beta}^{t+1})$ in equation 3.7. However, we predict that even if properly implemented, ACF would not be faster than the cyclic and random methods due to the dimensions of our data. Based on studies of ACF in [4], the method worked best for when $N \approx p$ or $N > p$, which is not the case for the data provided by Aquatic.



(a)



(b)



(c)

Figure 5.11: 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 replacmenet, Greedy = Gauss-Southwell-s, Hybrid = Hybrid method using Greedy until certain threshold in largest gradient value follow by cyclic

Figure 5.11 shows a comparison 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 Medium datasets, it is difficult to distinguish the fastest method out of cyclic, random with replacement, and random without replacement. However, even on the smaller datasets, it is clear 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 each iteration's computational complexity 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 randomly 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 3.3, they implement the same coordinate update rule. Therefore, we ran both update methods for 500 iterations and compared their runtimes. Table A.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.

5.8 Front Heavy Covariance CD vs Standard Covariance CD

Section 3.4 described an alternative method of the covariance CD, front heavy 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 A.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 (1.129s) is significantly smaller than the overall time for standard covariance (5.626s). 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.

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 thresholding parameter values and plotted the validation loss in Figure 5.12. From our studies, it seems like a value of 0.02 seems to have the lowest validation loss.

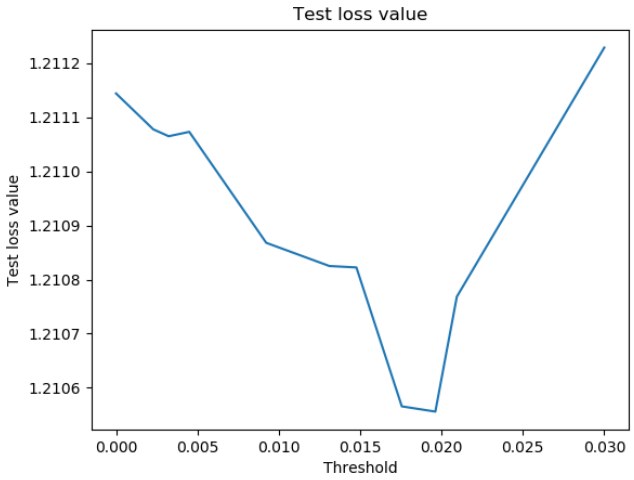


Figure 5.12: Thresholding value vs cross-validation loss.

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, where the faster one is described in Section 3.7.2. Our first method was to load 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 difficult to parallelize. 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 A.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 datasets, we had to improve loading times to something reasonable (less than 1 hour). As seen in Table A.12, we can achieve loading times below 15 minutes for every dataset, including the Large dataset. Storing data in binary format and loading in from disk in binary format is the fastest loading method because we do not have to convert from ASCII characters to numbers such as in parsing a CSV file.

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 16-bit float (`float16`) precision but have limited alternatives for 16-bit integers (`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.

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

Table 5.3: 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.

As shown in Table 5.3, we implemented several various matrix-vector multiplication methods to find the most optimal configuration. Both MKL implementations and the Pytorch method for single-precision 32-bit floats (`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.

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 5.4: 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 5.4 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 5.4, 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. The high memory usage 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 with 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 than 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) $= c \cdot N$. 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 5.5 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 $\hat{\boldsymbol{\beta}}^{(e)}$ is the weight vector predicted by the covariance optimizer using the estimated Gram matrix and $\hat{\boldsymbol{\beta}}^{(t)}$ is the 'ground truth' weight vector predicted by the optimizer using the true Gram matrix. "Loss function diff" refers to

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

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 5.5: 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 $\hat{\boldsymbol{\beta}}^{(t)}$ 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 estimated Gram's loss function difference is small compared to the ground truth loss function's value at the optimum, around 1200.

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 to explore the effect on the runtime of varying both N and p. Table 5.6 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 compute for the same problem for each N, p combination. Preconditioned CD ran for a fixed six iterations and thus never has to calculate the duality gap, while standard front heavy CD ran around 15 iterations before converging. "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 (3.23) and (3.24). "Cond total time" is simply the sum of the previous two columns - the total runtime of the precondition-and-CD algorithm. This metric is comparable 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 5.6 that the preconditioned algorithm's runs slower relative to the standard front heavy algorithm as p increases, even running 100 times slower for $N = 10^5, p = 300$. 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 perform several $O(p)$ operations, including those in equations (3.18), (3.19) and (3.23). The majority of preconditioned CD's runtime, however, comes from *sorting* the c values calculated in equation (3.18). For example, of the 0.22 seconds of preconditioned CD for $N = 10^5$ and $p = 300$, around 0.17 comes from calling c++'s `std::sort` to sort the cs. Standard front heavy CD also runs more iterations as p increases, but each iteration performs significantly less work than a preconditioned iteration.

Conversely, the preconditioned algorithm 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. Indeed, Cond total time stays roughly constant as N increases for all three p values. On the other hand, the front heavy total runtime for each p increases by around an

N	p	Cond prep time (s)	Cond CD time (s)	Cond total time (s)	Front heavy total time(s)
10^5	50	0.001	0.004	0.005	< 0.001
10^5	100	0.003	0.018	0.021	0.001
10^5	300	0.012	0.218	0.230	0.002
10^6	50	0.001	0.004	0.005	0.006
10^6	100	0.003	0.018	0.021	0.008
10^6	300	0.037	0.219	0.256	0.01
10^7	50	0.001	0.004	0.005	0.094
10^7	100	0.005	0.02	0.025	0.113
10^7	300	0.04	0.254	0.258	0.130

Table 5.6: 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 $(X^TX)^{-1/2}$ and "Cond CD time" refers to the time to perform two iterations of the preconditioned CD algorithm.

order of magnitude for every corresponding tenfold 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 a length of 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 of this experiment are stated in Table A.13. As shown from the table, using a warm start for new data can decrease the number of iterations we take in Coordinate Descent by half.

To determine 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 of this experiment are stated in Table A.14. 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.

5.15 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 storable 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

This project aims to reduce the runtime of coordinate descent when solving elastic net regression to predict stock prices using large financial datasets. In Chapter 5, we conducted studies of the CD optimization methods described in Chapters 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. Several methods, such as MiniCD, early stopping, feature selection, front heavy covariance, and warm start, produced successful results. Others, including lower precision calculations and preconditioning X, were not as successful. In this section, we provide concluding paragraphs addressing each of the methods explored above.

MiniCD and Ensembling Methods

We explored a general framework MiniCD. We tried averaging partial solutions and concluded the averaging introduces a one-sided bias (Section 4.3.3). Furthermore, we explored voting techniques. Section 5.4 showed preliminary studies of the threshold and voting based ensembling methods implemented for this report. These studies demonstrate that these methods do moderately well at selecting the true support of the ground truth. However, both voting methods can produce MSE, which is very close but slightly worse than the MSE produced by the ground truth. This result indicates that we can use voting methods to make predictions that perform almost as good as those produced by the ground truth. We also saw that those predictions tend to not be very close to the predictions made by the ground truth, but this is not contradictory to the previous result, as stated in the analysis. We also showed that for certain values of t and tree voting there is an inverse relationship between NNZ and two of our accuracy metrics (Metrics **4** and **6** from Section 5.4.2) and that for certain values of t there is an "elbow" shape between the runtime and relative distance.

Early Stopping

Section 5.5 showed how reducing the duality gap computation can give us a large speedup. At best, we could achieve a $2\times$ speedup by checking the duality gap as little as possible. Our results also showed that checking the duality gap too late still produced a speedup. Thus, late checking is stable and does not have to be fine-tuned to achieve a speedup.

Feature Choice Study

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

Naive vs Covariance 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 ($N >> p$). 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 **??**, 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 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 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 overhead that grows rapidly worse as p increases. The method shows signs of improvement over the standard method for very large N, but it is 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

Section 5.14 showed how a warm start could achieve a speedup for CD when applied in using a rolling window through the data or when running CD on a distributed network. With a rolling window, we were able to achieve 2×speedups. With a distributed network model, we were able to achieve a 1.57 speedup. This speedup shows that we can utilize a warm start for CD to fill in any computation gaps where CD is not running.

Scikit-learn Elastic Net Comparision

Section 5.15 showed how our CD implementation achieves significantly faster speeds than sklearn. We also demonstrated that our method supports arbitrarily large datasets, whereas the RAM limit bounds sklearn on a computer.

Chapter 7

Recommendations for Future Work

Most of this report's results in Chapter 5 were designed around the specific hardware configurations and the provided data from Aquatic. The conclusions from Chapter 6 should motivate to look into certain optimization fields of CD, as the results may differ when using different data or hardware. To strengthen this report's findings, one could explore 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 to see how CD with lower precision calculations compares to single-precision CD on hardware with 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 with large amounts of literature support. However, many other greedy variants 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 future researchers could explore alternative techniques.

Update Rules

The covariance update rule used to produce most of our results is best when X has the property of $N >> p$, which is the case for the data provided by Aquatic. However, the performance of this method deteriorates if this property of the data does not hold. Other update methods may provide faster computational performance when using data where N is not significantly greater than p.

Preconditioning

Further optimizations to the algorithm described in Section 3.8 could improve the preconditioned algorithm's performance relative to standard front heavy CD. Optimizing the $(X^TX)^{-1/2}$ calculation could significantly decrease the overall runtime, especially for very large p. Instead of determining $(X^TX)^{-1/2}$ exactly by first calculating $(X^TX)^{-1}$, then using the spectral decomposition of $(X^TX)^{-1}$ to take its square root, one can estimate $(X^TX)^{-1/2}$ using, for example, the method presented in [9]. Additionally, as discussed in Section 5.13, most of the preconditioned CD algorithm's runtime comes from sorting the c values calculated in equation (3.18), so an alternative method of calculating $h'(\beta_k)$ that avoids this sort will run several times faster, at least for the N and p values studied in Section 5.13.

Methods of Testing Convergence

The duality gap is only used to determine convergence. However, if we could determine beforehand how many iterations we would need to achieve convergence, then we would not need to compute the duality gap. Existing literature yields an iteration upper bound for both random and cyclic CD, but both of those upper bounds are too high to achieve a speedup. So determining a tighter upper bound on iterations eliminates the need to calculate the duality gap.

CD Performance on Computer Clusters

Many big data problems are computed on large computer clusters, providing 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 four available CPUs, we restricted the remaining portions of CD to run in sequential order, removing any possibilities of implementing block-based and other parallel algorithms. However, using a computer cluster for CD provides new challenges such as moving data throughout processes and parallel concepts that merit further exploration.

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 than another. The results from the ensembling methods should merely provide a glimpse of how their use can influence the overall accuracy of CD using portions of the data. To determine an optimal ensembling method, future researchers should explore various models in more detail. Additionally, our parameter studies were not extensive and only varied the number of batches and threshold values for the threshold method. Exploring the effects of other parameters could improve the performance of the ensembling methods.

Additionally, all tests using ensembling methods consolidated results from one data set. Since ensembling methods only use chunks of the data, it makes sense that the weights and predictions from ensembling are not as accurate as of the weights and prediction from the entire dataset. However, we did not explore how CD performs for out-of-sample testing, where training occurs on one dataset and testing is on another. One common issue with out-of-sample testing is overfitting the weights to the training data. It is possible that

determining weights with the entire dataset results in overfitting the data, where ensembling techniques may reduce or remove the overfitting entirely.

Appendix A

Tables from Chapter 5

λ_1 \ λ_2	1e0	1e-1	1e-2	1e-3	1e-4
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 A.1: Proportion of non-zero coefficients in the ground truth for the Medium dataset across different pairs of λ_1 and λ_2 regularizers.

λ_1 \ λ_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 A.2: Proportion of non-zero coefficients in the ground truth for the Large dataset across different pairs of λ_1 and λ_2 regularizers.

m \ t	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	Tree
1	409	409	409	409	409	409	409	409	409	453
3	976	976	976	351	351	351	73	73	73	313
5	1292	750	750	345	345	105	105	20	20	358
10	1186	845	554	337	186	90	33	8	1	260
15	1470	946	710	340	227	83	40	3	0	256
20	1320	889	545	295	149	60	17	3	0	105
50	1523	1048	648	344	171	62	13	0	0	195
100	1484	989	581	301	151	50	5	0	0	185

Table A.3: Number of active features for varying thesholds and batch sizes in Large dataset, and tree voting

m \ t	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	Tree
1	0.351	0.351	0.351	0.351	0.351	0.351	0.351	0.351	0.351	0.342
3	0.313	0.313	0.313	0.355	0.355	0.355	0.482	0.482	0.482	0.388
5	0.295	0.333	0.333	0.381	0.381	0.413	0.413	0.431	0.431	0.369
10	0.313	0.342	0.376	0.402	0.436	0.471	0.482	0.372	0.600	0.418
15	0.292	0.338	0.363	0.412	0.448	0.454	0.460	0.555	0.500	0.422
20	0.307	0.352	0.400	0.449	0.480	0.495	0.483	0.600	1.000	0.506
50	0.294	0.332	0.379	0.447	0.506	0.487	0.545	0.000	0.000	0.449
100	0.297	0.340	0.391	0.474	0.507	0.496	0.500	0.000	0.000	0.485

Table A.4: Support precision for varying thesholds and batch sizes, and tree voting, Large dataset

m \ t	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	Tree
1	0.298	0.298	0.298	0.298	0.298	0.298	0.298	0.298	0.298	0.320
3	0.633	0.633	0.633	0.258	0.258	0.258	0.073	0.073	0.073	0.252
5	0.791	0.518	0.518	0.273	0.273	0.090	0.090	0.018	0.018	0.273
10	0.769	0.599	0.432	0.281	0.168	0.087	0.033	0.006	0.001	0.225
15	0.889	0.662	0.534	0.291	0.211	0.078	0.038	0.004	0.000	0.223
20	0.840	0.649	0.451	0.275	0.148	0.061	0.017	0.003	0.000	0.110
50	0.930	0.722	0.509	0.319	0.18	0.063	0.014	0.000	0.000	0.181
100	0.913	0.698	0.471	0.296	0.159	0.051	0.005	0.000	0.000	0.186

Table A.5: Support recall for varying thesholds and batch sizes, and tree voting, Large dataset

m \ t	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	Tree
1	0.00293	0.00293	0.00293	0.00293	0.00293	0.00293	0.00293	0.00293	0.00293	0.00245
3	0.00068	0.00068	0.00068	0.00303	0.00303	0.00303	0.00759	0.00759	0.00759	0.00327
5	0.00024	0.00114	0.00114	0.0034	0.0034	0.00713	0.00713	0.00976	0.00976	0.00299
10	0.00035	0.00088	0.00185	0.00375	0.00603	0.00792	0.00895	0.0101	0.01041	0.00415
15	0.00011	0.00069	0.00129	0.00364	0.00492	0.00774	0.00863	0.01028	0.01045	0.00374
20	0.0002	0.00072	0.00169	0.00391	0.00604	0.0086	0.01	0.01038	0.01045	0.0066
50	0.00005	0.00051	0.00174	0.00388	0.00606	0.00895	0.0103	0.01045	0.01045	0.00551
100	0.00009	0.00059	0.00206	0.00432	0.00618	0.00942	0.01037	0.01045	0.01045	0.00592

Table A.6: $\frac{\mathrm{MSE}_w - \mathrm{MSE}_{w^*}}{(|\overline{y}|)^2}$ for varying thresholds and batch sizes, and tree voting, Large dataset

m \ t	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	Tree
1	1.062	1.062	1.062	1.062	1.062	1.062	1.062	1.062	1.062	1.010
3	0.691	0.691	0.691	1.088	1.088	1.088	1.089	1.089	1.089	1.045
5	0.451	0.766	0.766	1.028	1.028	1.069	1.069	1.025	1.025	1.040
10	0.579	0.753	0.902	0.980	1.023	1.050	1.022	1.010	1.002	1.081
15	0.283	0.618	0.764	0.958	1.015	1.054	1.035	1.006	1.000	1.042
20	0.398	0.660	0.844	0.969	1.011	1.005	1.016	1.000	1.000	1.054
50	0.269	0.578	0.825	0.953	1.010	1.013	1.006	1.000	1.000	1.040
100	0.269	0.519	0.840	0.886	0.978	0.999	1.000	0.999	1.000	0.981

Table A.7: $\frac{|w - w^*|}{|w^*|}$ for varying thresholds and batch sizes, and tree voting, Large dataset

m \ t	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	Tree
1	1.569	1.569	1.569	1.569	1.569	1.569	1.569	1.569	1.569	1.392
3	0.659	0.659	0.659	1.583	1.583	1.583	2.786	2.786	2.786	1.659
5	0.373	0.869	0.869	1.726	1.726	2.689	2.689	3.231	3.231	1.596
10	0.422	0.759	1.2	1.845	2.446	2.858	3.078	3.299	3.357	1.969
15	0.235	0.67	0.973	1.841	2.189	2.831	3.007	3.333	3.365	1.826
20	0.347	0.697	1.157	1.92	2.457	3.012	3.284	3.352	3.365	2.584
50	0.164	0.581	1.185	1.941	2.504	3.091	3.341	3.364	3.365	2.337
100	0.209	0.642	1.325	2.071	2.526	3.177	3.352	3.365	3.365	2.448

Table A.8: $\frac{(w - w^*)^T \Sigma (w - w^*)}{\mathrm{MSE}_{w^*}}$ for varying thresholds and batch sizes, and tree voting, Large dataset

m \ t	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	Tree
1	2.784	2.784	2.784	2.784	2.784	2.784	2.784	2.784	2.784	2.784
3	8.35	8.35	8.35	8.35	8.35	8.35	8.35	8.35	8.35	8.35
5	13.947	13.947	13.947	13.947	13.947	13.947	13.947	13.947	13.947	13.947
10	26.539	26.539	26.539	26.539	26.539	26.539	26.539	26.539	26.539	26.541
15	42.043	42.043	42.043	42.043	42.043	42.043	42.043	42.043	42.043	42.045
20	54.648	54.648	54.648	54.648	54.648	54.648	54.648	54.648	54.648	54.651
50	131.276	131.277	131.277	131.277	131.276	131.276	131.276	131.277	131.277	131.284
100	260.987	260.987	260.987	260.987	260.986	260.987	260.987	260.986	260.987	261.002

Table A.9: Time to run threshold voting for varying thresholds and batch sizes, and tree voting, Large dataset

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 A.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)

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 A.11: Runtime comparison between front heavy covariance and standard covariance on the Small dataset

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
Load by Row (0.2% loaded)	Large	902.19

Table A.12: Performance study over 10 trials loading in and calculating $X^T X$ from our Small (dimensions 17,281,517 × 10), Medium (dimensions 17,281,517 × 162), and Large (dimensions 17,192,783 × 1768) datasets.

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 A.13: Performance study of using a warm start for rolling windows on the Small dataset

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 A.14: Performance study of using warm start for a distributed network on the Large dataset

Appendix B

Abbreviations

ACF. Adaptive coordiante frequencies.

ACF-GPI. ACF with gradient probability initialization.

ASCII. American Standard Code for Information Interchange. ASCII is a character encoding standard for electronic communication.

CD. Coordinate Descent.

CSV. Comma-separated values. Commonly used file format for storing data.

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

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

MKL. Intel's Math Kernel Library.

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.

SGD. Stochastic Gradient Descent.

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] X. Li, T. Zhao, R. Arora, H. Liu, and M. Hong, *On faster convergence of cyclic block coordinate descent-type methods for strongly convex minimization*, Journal of Machine Learning Research, 18 (2018), pp. 1–24.

[8] 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.

[9] G. Pleiss, M. Jankowiak, D. Eriksson, A. Damle, and J. R. Gardner, *Fast Matrix Square Roots with Applications to Gaussian Processes and Bayesian Optimization*, arXiv e-prints, (2020), p. arXiv:2006.11267.

[10] P. Richtarik and M. Takac, *Distributed coordinate descent method for learning with big data*, Journal of Machine Learning Research, 17 (2016), pp. 1–25.

[11] P. Richtárik and M. Takác, *Iteration complexity of randomized block-coordinate descent methods for minimizing a composite function*, Springer-Verlag Berlin Heidelberg and Mathematical Optimization Society, (2012).

[12] 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.

[13] L. Shu, F. Shi, and G. Tian, *High-dimensional index tracking based on the adaptive elastic net*, Quantitative Finance, 1 (2020), pp. 1–18.

[14] V. Smith, *System-Aware Optimization for Machine Learning at Scale*, PhD thesis, Univeristy of California at Berkeley, 2017.

[15] R. Tibshirani, *Regression shrinkage and selection via the lasso*, Journal of the Royal Statistical Society. Series B (Methodological), 58 (1996), pp. 267–288.

[16] 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.

[17] S. J. Wright, *Coordinate descent algorithms*, Mathematical Programming, 151 (2015), p. 3–34.

[18] 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.

[19] 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.

[20] H. Zou and H. H. Zhang, *On the adaptive elastic-net with a diverging number of parameters*, Ann. Statist., 37 (2009), pp. 1733–1751.