DOUBLY SEPARABLE MODELS AND DISTRIBUTED PARAMETER ESTIMATION

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Hyokun Yun

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2014

Purdue University

West Lafayette, Indiana

To my family.

ACKNOWLEDGMENTS

For an incompetent person such as myself to complete the Ph.D program of Statistics at Purdue University, exceptional amount of effort and patience from other people were required. Therefore, the most natural way to start this thesis is by acknowledging contributions of these people.

My advisor, Prof. S.V.N. (Vishy) Vishwanathan, was clearly the person who had to suffer the most. When I first started the Ph.D program, I was totally incapable of thinking about anything carefully since I had been too lazy to use my brain for my entire life. Through countless discussions we have had almost every day for past five years, he patiently taught me the habit of thinking. I am only making baby steps yet - five years were not sufficient even for Vishy to make me decent - but I sincerely thank him for changing my life, besides so many other wonderful things he has done for me.

I would also like to express my utmost gratitude to my collaborators. It was a great pleasure to work with Prof. Shin Matsushima at Tokyo University; it was his idea to explore double separability beyond the matrix completion problem. On the other hand, I was very lucky to work with extremely intelligent and hard-working people at University of Texas at Austin, namely Hsiang-Fu Yu, Cho-Jui Hsieh and Prof. Inderjit Dhillon. I also give many thanks to Parameswaran Raman for his hard work on RoBiRank.

On the other hand, I deeply appreciate the guidance I have received from professors at Purdue University. Especially, I am greatly indebted to Prof. Jennifer Neville, who has strongly supported every step I took in the graduate school from the start to the very end. Prof. Chuanhai Liu motivated me to always think critically about statistical procedures; I will constantly endeavor to meet his high standard on Statistics. I also thank Prof. David Gleich for giving me invaluable comments to improve the thesis.

In addition, I feel grateful to Prof. Karsten Borgwardt at Max Planck Institute, Dr. Chaitanya Chemudugunta at Blizzard Entertainment, Dr. A Kumaran at Microsoft Research, and Dr. Guy Lebanon at Amazon for giving me amazing opportunities to experience these institutions and work with them.

Furthermore, I thank Prof. Anirban DasGupta, Sergey Kirshner, Olga Vitek, Fabrice Baudoin, Thomas Sellke, Burgess Davis, Chong Gu, Hao Zhang, Guang Cheng, William Cleveland, Jun Xie and Herman Rubin for inspirational lectures that shaped my knowledge on Statistics. I also deeply appreciate generous help from the following people and those who I have unfortunately omitted: Nesreen Ahmed, Kuk-Hyun Ahn, Kyungmin Ahn, Chloé-Agathe Azencott, Nguyen Cao, Soo Young Chang, Lin-Yang Cheng, Hyunbo Cho, Mihee Cho, InKyung Choi, Joon Hee Choi, Meena Choi, Seungjin Choi, Sung Sub Choi, Yun Sung Choi, Hyonho Chun, Andrew Cross, Douglas Crabill, Jyotishka Datta, Alexander Davies, Glen DePalma, Vasil Denchev, Nan Ding, Rebecca Doerge, Marian Duncan, Guy Feldman, Ghihoon Ghim, Dominik Grimm, Ralf Herbrich, Jean-Baptiste Jeannin, Youngjoon Jo, Chi-Hyuck Jun, Kyuhwan Jung, Yushin Hong, Qiming Huang, Whitney Huang, Seung-sik Hwang, Suvidha Kancharla, Byung Gyun Kang, Eunjoo Kang, Jinhak Kim, Kwang-Jae Kim, Kangmin Kim, Moogung Kim, Young Ha Kim, Timothy La Fond, Alex Lamb, Baron Chi Wai Law, Duncan Ermini Leaf, Daewon Lee, Dongyoon Lee, Jaewook Lee, Sumin Lee, Jeff Li, Limin Li, Eunjung Lim, Diane Martin, Sai Sumanth Miryala, Sebastian Moreno, Houssam Nassif, Jeongsoo Park, Joonsuk Park, Mijung Park, Joel Pfeiffer, Becca Pillion, Shaun Ponder, Pablo Robles, Alan Qi, Yixuan Qiu, Barbara Rakitsch, Mary Roe, Jeremiah Rounds, Ted Sandler, Ankan Saha, Bin Shen, Nino Shervashidze, Alex Smola, Bernhard Schölkopf, Gaurav Srivastava, Sanvesh Srivastava, Wei Sun, Behzad Tabibian, Abhishek Tayal, Jeremy Troisi, Feng Yan, Pinar Yanardag, Jiasen Yang, Ainur Yessenalina, Lin Yuan, Jian Zhang.

Using this opportunity, I would also like to express my deepest love to my family. Everything was possible thanks to your strong support!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

SGD	Stochastic Gradient Descent
SSO	Stochastic Saddle-point Optimization
ODE	Ordinary Differential Equation
DSGD	Distributed Stochastic Gradient Descent
DSSO	Distributed Stochastic Saddle-point Optimization
NOMAD	Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized optimization
RERM	REgularized Risk Minimization
IRT	Item Response Theory

ABSTRACT

Yun, Hyokun Ph.D., Purdue University, May 2014. Doubly Separable Models and Distributed Parameter Estimation. Major Professor: S.V.N. Vishwanathan.

It is well known that stochastic optimization algorithms are both theoretically and practically well-motivated for parameter estimation of large-scale statistical models. Unfortunately, in general they have been considered difficult to parallelize, especially in distributed memory environment. To address the problem, we first identify that stochastic optimization algorithms can be efficiently parallelized when the objective function is doubly separable; lock-free, decentralized, and serializable algorithms are proposed for stochastically finding minimizer or saddle-point of doubly separable functions. Then, we argue the usefulness of these algorithms in statistical context by showing that a large class of statistical models can be formulated as doubly separable functions; the class includes important models such as matrix completion and regularized risk minimization. Motivated by optimization techniques we have developed for doubly separable functions, we also propose a novel model for latent collaborative retrieval, an important problem that arises in recommender systems.

1. INTRODUCTION

Numerical optimization lies at the heart of almost every statistical procedure. Majority of frequentist statistical estimators can be viewed as M-estimators [73] and thus are computed by solving an optimization problem; the use of (penalized) maximum likelihood estimator, a special case of M-estimator, is the dominant method of statistical inference. On the other hand, Bayesians also use optimization methods to approximate the posterior distribution [12]. Therefore, in order to apply statistical methodologies on massive datasets we confront in today's world, we need optimization algorithms that can scale to such data; development of such an algorithm is the aim of this thesis.

It is well known that stochastic optimization algorithms are both theoretically [13, 63, 64] and practically [75] well-motivated for parameter estimation of large-scale statistical models. To briefly illustrate why they are computationally attractive, suppose that a statistical procedure requires us to minimize a function $f(\theta)$, which can be written in the following form:

$$f(\theta) = \sum_{i=1}^{m} f_i(\theta), \tag{1.1}$$

where m is the number of data points. The most basic approach to solve this minimization problem is the method of gradient descent, which starts with a possibly random initial parameter θ and iteratively moves it towards the direction of the negative gradient:

$$\theta \leftarrow \theta - \eta \cdot \nabla_{\theta} f(\theta), \tag{1.2}$$

where η is a step-size parameter. To execute (1.2) on a computer, however, we need to compute $\nabla_\theta f(\theta)$; and this is where computational challenges arise when dealing with large-scale data. Since

$$\nabla_\theta f(\theta) = \sum_{i=1}^{m} \nabla_\theta f_i(\theta), \tag{1.3}$$

computation of the gradient $\nabla_\theta f(\theta)$ requires $O(m)$ computational effort. When m is a large number, that is, the data consists of large number of samples, repeating this computation may not be affordable.

In such a situation, stochastic gradient descent (SGD) algorithm [58] can be very effective. The basic idea is to replace $\nabla_\theta f(\theta)$ in (1.2) with an easy-to-calculate stochastic estimator. Specifically, in each iteration the algorithm draws a uniform random number i between 1 and m, and then instead of the exact update (1.2), it executes the following *stochastic* update:

$$\theta \leftarrow \theta - \eta \cdot \left\{ m \cdot \nabla_\theta f_i(\theta) \right\}. \tag{1.4}$$

Note that the SGD update (1.4) can be computed in $O(1)$ time, independently of m. The rational here is that $m \cdot \nabla_\theta f_i(\theta)$ is an unbiased estimator of the true gradient:

$$\mathbb{E}\left[m \cdot \nabla_\theta f_i(\theta) \right] = \nabla_\theta f(\theta), \tag{1.5}$$

where the expectation is taken over the random sampling of i. Since (1.4) is a very crude approximation of (1.2), the algorithm will of course require much more number of iterations than it would with the exact update (1.2). Still, Bottou and Bousquet [13] shows that SGD is asymptotically more efficient than algorithms which exactly calculate $\nabla_\theta f(\theta)$, including not only the simple gradient descent method we introduced in (1.2) but also much more complex methods such as quasi-Newton algorithms [53].

When it comes to parallelism, however, the computational efficiency of stochastic update (1.4) turns out to be a disadvantage; since the calculation of $\nabla_\theta f_i(\theta)$ typically requires very little amount of computation, one can rarely expect to speed it up by splitting it into smaller tasks. An alternative approach is to let multiple processors simultaneously execute (1.4) [43, 56]. Unfortunately, the computation of $\nabla_\theta f_i(\theta)$ can possibly require reading any coordinate of θ, and the update (1.4) can also change any coordinate of θ, and therefore every update made by one processor has to be propagated across all processors. Such a requirement can be very costly in distributed memory environment which the speed of communication between processors is considerably slower than that of the update (1.4); even within shared memory architecture, the cost of inter-process synchronization significantly deteriorates the efficiency of parallelization [79].

To propose a parallelization method that circumvents these problems of SGD, let us step back for now and consider what would be an ideal situation for us to parallelize an optimization algorithm, if we are given two processors. Suppose the parameter θ can be partitioned into $\theta^{(1)}$ and $\theta^{(2)}$, and the objective function can be written as

$$f(\theta) = f^{(1)}(\theta^{(1)}) + f^{(2)}(\theta^{(2)}). \tag{1.6}$$

Then, we can effectively minimize $f(\theta)$ in parallel; since the minimization of $f^{(1)}(\theta^{(1)})$ and $f^{(2)}(\theta^{(2)})$ are independent problems, processor 1 can work on minimizing $f^{(1)}(\theta^{(1)})$ while processor 2 is working on $f^{(2)}(\theta^{(2)})$, without having any need to communicate with each other.

Of course, such an ideal situation rarely occurs in reality. Now let us relax the assumption (1.6) to make it a bit more realistic. Suppose θ can be partitioned into four sets, $\mathbf{w}^{(1)}$, $\mathbf{w}^{(2)}$, $\mathbf{h}^{(1)}$ and $\mathbf{h}^{(2)}$, and the objective function can be written as

$$\begin{aligned} f(\theta) =& f^{(1,1)}(\mathbf{w}^{(1)}, \mathbf{h}^{(1)}) + f^{(1,2)}(\mathbf{w}^{(1)}, \mathbf{h}^{(2)}) \\ &+ f^{(2,1)}(\mathbf{w}^{(2)}, \mathbf{h}^{(1)}) + f^{(2,2)}(\mathbf{w}^{(2)}, \mathbf{h}^{(2)}). \end{aligned} \tag{1.7}$$

Note that the simple strategy we deployed for (1.6) cannot be used anymore, since (1.7) does not admit such a simple partitioning of the problem anymore.

Surprisingly, it turns out that the strategy for (1.6) can be adapted in a fairly simple fashion. Let us define

$$f_1(\theta) = f^{(1,1)}(\mathbf{w}^{(1)}, \mathbf{h}^{(1)}) + f^{(2,2)}(\mathbf{w}^{(2)}, \mathbf{h}^{(2)}), \tag{1.8}$$

$$f_2(\theta) = f^{(1,2)}(\mathbf{w}^{(1)}, \mathbf{h}^{(2)}) + f^{(2,1)}(\mathbf{w}^{(2)}, \mathbf{h}^{(1)}). \tag{1.9}$$

Note that $f(\theta) = f_1(\theta) + f_2(\theta)$, and that $f_1(\theta)$ and $f_2(\theta)$ are both in form (1.6). Therefore, if the objective function to minimize is $f_1(\theta)$ or $f_2(\theta)$ instead of $f(\theta)$, it can be efficiently minimized in parallel. This property can be exploited by the following simple two-phase algorithm:

- $f_1(\theta)$-phase: processor 1 runs SGD on $f^{(1,1)}(\mathbf{w}^{(1)}, \mathbf{h}^{(1)})$, while processor 2 runs SGD on $f^{(2,2)}(\mathbf{w}^{(2)}, \mathbf{h}^{(2)})$.

- $f_2(\theta)$-phase: processor 1 runs SGD on $f^{(1,2)}(\mathbf{w}^{(1)}, \mathbf{h}^{(2)})$, while processor 2 runs SGD on $f^{(2,1)}(\mathbf{w}^{(2)}, \mathbf{h}^{(1)})$.

Gemulla et al. [30] shows under fairly mild technical assumptions that if we switch between these two phases periodically, the algorithm converges to the local optimum of the *original* function $f(\theta)$.

This thesis is structured to answer the following natural questions one may ask at this point. First, how can the condition (1.7) be generalized for arbitrary number of processors? It turns out that the condition can be characterized as *double separability*; in Chapter 2 and Chapter 3, we will introduce double separability and propose efficient parallel algorithms for optimizing doubly separable functions.

The second question would be: How useful are doubly separable functions in building statistical models? It turns out that a wide range of important statistical models can be formulated using doubly separable functions. Chapter 4 to Chapter 7 will be devoted to discussing how such a formulation can be done for different statistical models. In Chapter 4, we will evaluate the effectiveness of algorithms introduced in Chapter 2 and Chapter 3 by comparing them against state-of-the-art algorithms for matrix completion. In Chapter 5, we will discuss how regularized risk minimization

(RERM), a large class of problems including generalized linear model and Support Vector Machines, can be formulated as doubly separable functions. A couple more examples of doubly separable formulations will be given in Chapter 6. In Chapter 7 we propose a novel model for the task of latent collaborative retrieval, and propose a distributed parameter estimation algorithm by extending ideas we have developed for doubly separable functions. Then, we will provide the summary of our contributions in Chapter 8 to conclude the thesis.

1.1 Collaborators

Chapter 3 and 4 were joint work with Hsiang-Fu Yu, Cho-Jui Hsieh, S.V.N. Vishwanathan and Inderjit Dhillon.

Chapter 5 was joint work with Shin Matsushima and S.V.N. Vishwanathan.

Chapter 6 and 7 were joint work with Parameswaran Raman and S.V.N. Vishwanathan.

2. BACKGROUND

2.1 Separability and Double Separability

The notion of separability [47] has been considered as an important concept in optimization [71], and was found to be useful in statistical context as well [28]. Formally, separability of a function can be defined as follows:

Definition 2.1.1 (Separability)**.** *Let* $\{\mathbb{S}_i\}_{i=1}^m$ *be a family of sets. A function* $f : \prod_{i=1}^m \mathbb{S}_i \to \mathbb{R}$ *is said to be* separable *if there exists* $f_i : \mathbb{S}_i \to \mathbb{R}$ *for each* $i = 1, 2, \ldots, m$ *such that*

$$f(\theta_1, \theta_2, \ldots, \theta_m) = \sum_{i=1}^m f_i(\theta_i), \tag{2.1}$$

where $\theta_i \in \mathbb{S}_i$ *for all* $1 \leqslant i \leqslant m$*.*

As a matter of fact, the codomain of $f(\cdot)$ does not necessarily have to be a real line $\mathbb{R}$ as long as the addition operator is defined on it. Also, to be precise we are defining *additive* separability here, and other notions of separability such as multiplicative separability do exist. Only additively separable functions with codomain $\mathbb{R}$ are of interest in this thesis, however, thus for the sake of brevity *separability* will always imply *additive separability.* On the other hand, although $\mathbb{S}_i$'s are defined as general arbitrary sets, we will always use them as subsets of finite-dimensional Euclidean spaces.

Note that the separability of a function is a very strong condition, and objective functions of statistical models are in most cases not separable. Usually, separability can only be assumed for a particular term of the objective function [28]. Double separability, on the other hand, is a considerably weaker condition:

Definition 2.1.2 (Double Separability)**.** *Let $\{\mathbb{S}_i\}_{i=1}^m$ and $\left\{\mathbb{S}'_j\right\}_{j=1}^n$ be families of sets. A function $f : \prod_{i=1}^m \mathbb{S}_i \times \prod_{j=1}^n \mathbb{S}'_j \to \mathbb{R}$ is said to be* doubly separable *if there exists $f_{ij} : \mathbb{S}_i \times \mathbb{S}'_j \to \mathbb{R}$ for each $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$ such that*

$$f(w_1, w_2, \ldots, w_m, h_1, h_2, \ldots, h_n) = \sum_{i=1}^m \sum_{j=1}^n f_{ij}(w_i, h_j). \tag{2.2}$$

It is clear that separability implies double separability.

Property 1. *If f is separable, then it is doubly separable. The converse, however, is not necessarily true.*

Proof. Let $f : \mathbb{S}_i \to \mathbb{R}$ be a separable function as defined in (2.1). Then, for $1 \leqslant i \leqslant m - 1$ and $j = 1$, define

$$g_{ij}(w_i, h_j) := \begin{cases} f_i(w_i) & \text{if} \quad 1 \leqslant i \leqslant m-2 \\ f_i(w_i) + f_n(h_j) & \text{if} \quad i = m-1 \end{cases}. \tag{2.3}$$

It can be easily seen that $f(w_1, \ldots, w_{m-1}, h_j) = \sum_{i=1}^{m-1} \sum_{j=1}^{1} g_{ij}(w_i, h_j)$.

The counter-example of the converse can be easily found: $f(w_1, h_1) = w_1 \cdot h_1$ is doubly separable, but not separable. If we assume that $f(w_1, h_1)$ is doubly separable, then there exist two functions $p(w_1)$ and $q(h_1)$ such that $f(w_1, h_1) = p(w_1) + q(h_1)$. However, $\nabla_{w_1,h_1}(w_1 \cdot h_1) = 1$ but $\nabla_{w_1,h_1}(p(w_1) + q(h_1)) = 0$, which is contradiction. □

Interestingly, this relaxation turns out to be good enough for us to represent a large class of important statistical models; Chapter 4 to 7 are devoted to illustrate how different models can be formulated as doubly separable functions. The rest of this chapter and Chapter 3, on the other hand, aims to develop efficient optimization algorithms for general doubly separable functions.

The following properties are obvious, but are sometimes found useful:

Property 2. *If f is separable, so is $-f$. If f is doubly separable, so is $-f$.*

Proof. It follows directly from the definition. □

Property 3. *Suppose f is a doubly separable function as defined in* (2.2). *For a fixed* $(h_1^*, h_2^*, \ldots, h_n^*) \in \prod_{j=1}^n \mathbb{S}'_j$, *define*

$$g(w_1, w_2, \ldots, w_n) := f(w_1, w_2, \ldots, w_n, h_1^*, h_2^*, \ldots, h_n^*). \tag{2.4}$$

Then, g is separable.

Proof. Let

$$g_i(w_i) := \sum_{j=1}^{n} f_{ij}(w_i, h_j^*). \tag{2.5}$$

Since $g(w_1, w_2, \ldots, w_n) = \sum_{i=1}^m g_i(w_i)$, g is separable. □

By symmetry, the following property is immediate.

Property 4. *Suppose f is a doubly separable function as defined in* (2.2). *For a fixed* $(w_1^*, w_2^*, \ldots, w_n^*) \in \prod_{i=1}^m \mathbb{S}_i$, *define*

$$q(h_1, h_2, \ldots, h_m) := f(w_1^*, w_2^*, \ldots, w_n^*, h_1, h_2, \ldots, h_n). \tag{2.6}$$

Then, q is separable.

2.2 Problem Formulation and Notations

Now, let us describe the nature of optimization problems that will be discussed in this thesis. Let f be a doubly separable function defined as in (2.2). For brevity, let $W = (w_1, w_2, \ldots, w_m) \in \prod_{i=1}^m \mathbb{S}_i$, $H = (h_1, h_2, \ldots, h_n) \in \prod_{j=1}^n \mathbb{S}'_j$, $\theta = (W, H)$, and denote

$$f(\theta) := f(W, H) := f(w_1, w_2, \ldots, w_m, h_1, h_2, \ldots, h_n). \tag{2.7}$$

In most objective functions we will discuss in this thesis, $f_{ij}(\cdot, \cdot) = 0$ for large fraction of (i, j) pairs. Therefore, we introduce a set $\Omega \subset \{1, 2, \ldots, m\} \times \{1, 2, \ldots, n\}$ and rewrite f as:

$$f(\theta) = \sum_{(i,j) \in \Omega} f_{ij}(w_i, h_j). \tag{2.8}$$

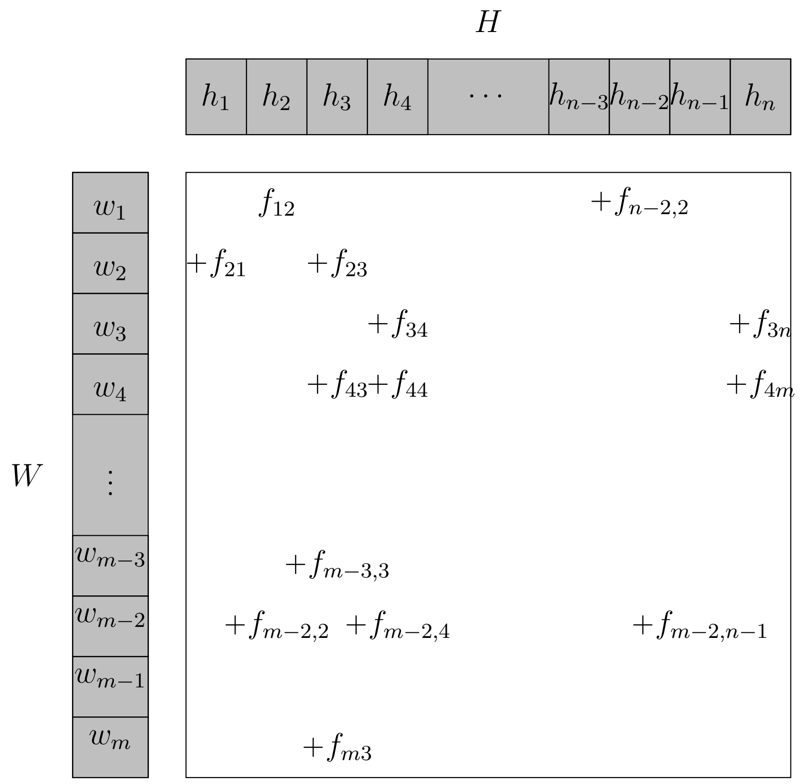


Figure 2.1.: Visualization of a doubly separable function. Each term of the function f interacts with only one coordinate of W and one coordinate of H. The locations of non-zero functions are sparse and described by Ω.

This will be useful in describing algorithms that take advantage of the fact that $|\Omega|$ is much smaller than $m \cdot n$. For convenience, we also define $\Omega_i = \{j : (i,j) \in \Omega\}$, $\bar{\Omega}_j = \{i : (i,j) \in \Omega\}$. Also, we will assume $f_{ij}(\cdot,\cdot)$ is continuous for every i,j, although may not be differentiable.

Doubly separable functions can be visualized in two dimensions as in Figure 2.1. As can be seen, each term f_{ij} interacts with only one parameter of W and one parameter of H. Although the distinction between W and H is arbitrary because they are symmetric to each other, for the convenience of reference we will call $w_1, w_2, \ldots, w_m$ as *row* parameters, and $h_1, h_2, \ldots, h_n$ as *column* parameters.

In this thesis, we are interested in two kinds of optimization problem on f, the minimization problem and the saddle-point problem.

2.2.1 Minimization Problem

The minimization problem is formulated as follows:

$$\min_{\theta} f(\theta) = \sum_{(i,j)\in\Omega} f_{ij}(w_i, h_j). \tag{2.9}$$

Of course, maximization of f is equivalent to minimization of $-f$; since $-f$ is doubly separable as well (Property 2), (2.9) covers both minimization and maximization problems. For this reason, we will only discuss the minimization problem (2.9) in this thesis.

The minimization problem (2.9) frequently arises in parameter estimation of matrix factorization models and a large number of optimization algorithms are developed in that context. However, most of them are specialized for the specific matrix factorization model they aim to solve, and thus we defer the discussion of these methods to Chapter 4. Nonetheless, the following useful property frequently exploitted in matrix factorization algorithms is worth mentioning here: when $h_1, h_2, \ldots, h_n$ are fixed, thanks to Property 3 the minimization problem (2.9) decomposes into n independent minimization problems:

$$\min_{w_i} \sum_{j\in\Omega_i} f_{ij}(w_i, h_j) \tag{2.10}$$

for $i = 1, 2, \ldots, m$. On the other hand, when W is fixed, the problem is decomposed into n independent minimization problems by symmetry. This can be useful for two reasons; first, the dimensionality of each optimization problem in (2.10) is only $1/m$ fraction of the original problem, so if the time complexity of an optimization algorithm is superlinear to the dimensionality of the problem, an improvement can be made by solving one sub-problem at a time. Also, this property can be used to parallelize an optimization algorithm, as each sub-problem can be solved independently of each other.

Note that the problem of finding local minimum of $f(\theta)$ is equivalent to finding locally stable points of the following ordinary differential equation (ODE) (Yin and Kushner [77], Chapter 4.2.2):

$$\frac{d\theta}{dt} = -\nabla_\theta f(\theta). \tag{2.11}$$

This fact is useful in proving asymptotic convergence of stochastic optimization algorithms by approximating them as stochastic processes that converge to stable points of an ODE described by (2.11). The proof can be generalized for non-differentiable functions as well (Yin and Kushner [77], Chapter 6.8).

2.2.2 Saddle-point Problem

Another optimization problem we will discuss in this thesis is the problem of finding a saddle-point (W^*, H^*) of f, which is defined as follows:

$$f(W^*, H) \leqslant f(W^*, H^*) \leqslant f(W, H^*), \tag{2.12}$$

for any $(W, H) \in \prod_{i=1}^m \mathbb{S}_i \times \prod_{j=1}^n \mathbb{S}'_j$. The saddle-point problem often occurs when a solution of constrained minimization problem is sought; this will be discussed in Chapter 5. Note that a saddle-point is also the solution of the minimax problem

$$\min_W \max_H f(W, H), \tag{2.13}$$

and the maximin problem

$$\max_H \min_W f(W, H), \tag{2.14}$$

at the same time [8]. Contrary to the case of minimization problem, however, neither (2.13) nor (2.14) can be decomposed into independent sub-problems as in (2.10).

The existence of saddle-point is usually harder to verify than that of minimizer or maximizer. In this thesis, however, we will only be interested in settings which the following assumptions hold:

Assumption 2.2.1.
- *$\prod_{i=1}^m \mathbb{S}_i$ and $\prod_{j=1}^n \mathbb{S}'_j$ are nonempty closed convex sets.*

- *For each W, the function $f(W, \cdot)$ is concave.*
- *For each H, the function $f(\cdot, H)$ is convex.*
- *W is bounded, or there exists H_0 such that $f(W, H_0) \to \infty$ when $\|W\| \to \infty$.*
- *H is bounded, or there exists W_0 such that $f(W_0, H) \to -\infty$ when $\|H\| \to \infty$.*

In such a case, it is guaranteed that a saddle-point of f exists (Hiriart-Urruty and Lemaréchal [35], Chapter 4.3).

Similarly to the minimization problem, we prove that there exists a corresponding ODE which the set of stable points are equal to the set of saddle-points.

Theorem 2.2.2. *Suppose that f is a twice-differentiable doubly separable function as defined in* (2.2)*, which satisfies Assumption 2.2.1. Let G be a set of stable points of the ODE defined as below:*

$$\frac{dW}{dt} = -\nabla_W f(W, H), \tag{2.15}$$

$$\frac{dH}{dt} = \nabla_H f(W, H), \tag{2.16}$$

and let G' be the set of saddle-points of f. Then, $G = G'$.

Proof. Let (W^*, H^*) be a saddle-point of f. Since a saddle-point is also a critical point of a function, $\nabla f(W^*, H^*) = 0$. Therefore, (W^*, H^*) is a fixed point of the ODE (2.16) as well. Now we show that it is a stable point as well. For this, it suffices to show the following stability matrix of the ODE is nonpositive definite when evaluated at (W^*, H^*): (Tabor [68], Section 1.4)

$$S(W, H) = \begin{bmatrix} -\nabla_W \nabla_W f(W, H) & -\nabla_H \nabla_W f(W, H) \\ \nabla_W \nabla_H f(W, H) & \nabla_H \nabla_H f(W, H) \end{bmatrix}. \tag{2.17}$$

Let $v_1 \in \mathbb{R}^m$ and $v_2 \in \mathbb{R}^n$ be arbitrary vectors, and let $v^T = (v_1^T, v_2^T)$. Then,

$$v^T S(W, H) v = -v_1^T \nabla_W \nabla_W f(W, H) v_1 + v_2^T \nabla_H \nabla_H f(W, H) v_2 \leqslant 0, \tag{2.18}$$

due to assumed convexity of $f(\cdot, H)$ and concavity of $f(W, \cdot)$. Therefore, the stability matrix is nonpositive definite everywhere including (W^*, H^*), and therefore $G' \subset G$.

On the other hand, suppose that (W^*, H^*) is a stable point; then, by definition of stable point, $\nabla f(W^*, H^*) = 0$. Now to show that (W^*, H^*) is a saddle-point, we need to prove the Hessian of f at (W^*, H^*) is indefinite; this immediately follows from convexity of $f(\cdot, H)$ and concavity of $f(W, \cdot)$. □

2.3 Stochastic Optimization

2.3.1 Basic Algorithm

A large number of optimization algorithms have been proposed for the minimization of a general continuous function [53], and popular batch optimization algorithms such as L-BFGS [52] or bundle methods [70] can be applied to the minimization problem (2.9). However, each iteration of a batch algorithm requires exact calculation of the objective function (2.9) and its gradient; as this takes $O(|\Omega|)$ computational effort, when Ω is a large set the algorithm may take a long time to converge.

In such a situation, an improvement in the speed of convergence can be found by appealing to stochastic optimization algorithms such as stochastic gradient descent (SGD) [13]. While different versions of SGD algorithm may exist for a single optimization problem according to how the stochastic estimator is defined, the most straightforward version of SGD on the minimization problem (2.9) can be described as follows: starting with a possibly random initial parameter θ, the algorithm repeatedly samples $(i, j) \in \Omega$ uniformly at random and applies the update

$$\theta \leftarrow \theta - \eta \cdot |\Omega| \cdot \nabla_\theta f_{ij}(w_i, h_j), \tag{2.19}$$

where η is a step-size parameter. The rational here is that since $|\Omega| \cdot \nabla_\theta f_{ij}(w_i, h_j)$ is an unbiased estimator of the true gradient $\nabla_\theta f(\theta)$, in the long run the algorithm will reach the solution similar to what one would get with the basic gradient descent algorithm, which uses the following update:

$$\theta \leftarrow \theta - \eta \cdot \nabla_{\theta} f(\theta). \tag{2.20}$$

Convergence guarantees and properties of this SGD algorithm are well known [13].

Note that since $\nabla_{w_{i'}} f_{ij}(w_i, h_j) = 0$ for $i' \neq i$ and $\nabla_{h_{j'}} f_{ij}(w_i, h_j) = 0$ for $j' \neq j$, (2.19) can be more compactly written as

$$w_i \leftarrow w_i - \eta \cdot |\Omega| \cdot \nabla_{w_i} f_{ij}(w_i, h_j), \tag{2.21}$$

$$h_j \leftarrow h_j - \eta \cdot |\Omega| \cdot \nabla_{h_j} f_{ij}(w_i, h_j). \tag{2.22}$$

In other words, each SGD update (2.19) reads and modifies only two coordinates of θ at a time, which is a small fraction when m or n is large. This will be found useful in designing parallel optimization algorithms later.

On the other hand, in order to solve the saddle-point problem (2.12), it suffices to make a simple modification on SGD update equations in (2.21) and (2.22):

$$w_i \leftarrow w_i - \eta \cdot |\Omega| \cdot \nabla_{w_i} f_{ij}(w_i, h_j), \tag{2.23}$$

$$h_j \leftarrow h_j + \eta \cdot |\Omega| \cdot \nabla_{h_j} f_{ij}(w_i, h_j). \tag{2.24}$$

Intuitively, (2.23) takes stochastic *descent* direction in order to solve minimization problem in W, and (2.24) takes stochastic *ascent* direction in order to solve maximization problem in H. Under mild conditions, this algorithm is also guaranteed to converge to the saddle-point of the function f [51]. From now on, we will refer to this algorithm as SSO (Stochastic Saddle-point Optimization) algorithm.

2.3.2 Distributed Stochastic Gradient Algorithms

Now, we will discuss how SGD and SSO algorithms introduced in the previous section can be efficiently parallelized using traditional techniques of batch synchronization. For now, we will denote each parallel computing unit as a *processor*; in a shared memory setting a processor is a thread and in a distributed memory architecture a processor is a machine. This abstraction allows us to present parallel algorithms in a unified manner. The exception is Chapter 3.5 which we discuss how to take advantage of hybrid architecture where there are multiple threads spread across multiple machines.

As discussed in Chapter 1, in general stochastic gradient algorithms have been considered to be difficult to parallelize; the computational cost of each stochastic gradient update is often very cheap and thus it is not desirable to divide this computation across multiple processors. On the other hand, this also means that if multiple processors are executing the stochastic gradient update in parallel, parameter values of these algorithms are very frequently updated; therefore the cost of communication for synchronizing these parameter values across multiple processors can be prohibitive [79], especially in the distributed memory setting.

In the literature of matrix completion, however, there exists stochastic optimization algorithms that can be efficiently parallelized by avoiding the need for frequent synchronization. It turns out that the only major requirement of these algorithms is double separability of the objective function; therefore, these algorithms have great utility beyond the task of matrix completion, as will be illustrated throughout the thesis.

In this subsection, we will introduce Distributed Stochastic Gradient Descent (DSGD) of Gemulla et al. [30] for the minimization problem (2.9) and Distributed Stochastic Saddle-point Optimization (DSSO) algorithm, our proposal for the saddle-point problem (2.12). The key observation of DSGD is that SGD updates of (2.21) and (2.21) involve on only one row parameter w_i and one column parameter h_j; given $(i, j) \in \Omega$ and $(i', j') \in \Omega$, if $i \neq i'$ and $j \neq j'$ then one can simultaneously perform SGD updates (2.21) on w_i and $w_{i'}$ and (2.21) on h_j and $h_{j'}$. In other words, updates to w_i and h_j are independent of updates to $w_{i'}$ and $h_{j'}$ as long as $i \neq i'$ and $j \neq j'$. The same property holds for DSSO; this opens up the possibility that $\min(m, n)$ pairs of parameters (w_i, h_j) can be updated in parallel.

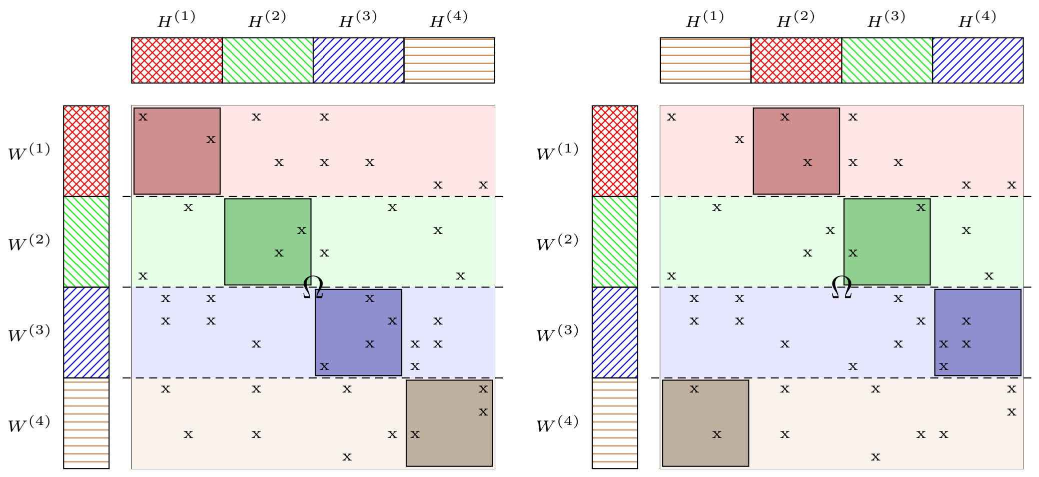


Figure 2.2.: Illustration of DSGD/DSSO algorithm with 4 processors. The rows of Ω and corresponding f_{ij}s as well as the parameters W and H are partitioned as shown. Colors denote ownership. The active area of each processor is shaded dark. Left: initial state. Right: state after one bulk synchronization step. See text for details.

We will use the above observation in order to derive a parallel algorithm for finding the minimizer or saddle-point of $f(W, H)$. However, before we formally describe DSGD and DSSO we would like to present some intuition using Figure 2.2. Here we assume that we have access to 4 processors. As in Figure 2.1, we visualize f with a $m \times n$ matrix; non-zero interaction between W and H are marked by x. Initially, both parameters as well as rows of Ω and corresponding f_{ij}'s are partitioned across processors as depicted in Figure 2.2 (left); colors in the figure denote ownership *e.g.*, the first processor owns a fraction of Ω and a fraction of the parameters W and H (denoted as $W^{(1)}$ and $H^{(1)}$) shaded with red. Each processor samples a non-zero entry (i, j) of Ω within the dark shaded rectangular region (active area) depicted in the figure, and updates the corresponding W_i and H_j. After performing a fixed number of updates, the processors perform a bulk synchronization step and exchange coordinates of H. This defines an epoch. After an epoch, ownership of the H variables and hence the active area changes as shown in Figure 2.2 (left). The algorithm iterates over the epochs until convergence.

Now let us formally introduce DSGD and DSSO. Suppose p processors are available, and let $I_1, \ldots, I_p$ denote p partitions of the set $\{1, \ldots, m\}$ and $J_1, \ldots, J_p$ denote p partitions of the set $\{1, \ldots, n\}$ such that $|I_q| \approx |I_{q'}|$ and $|J_r| \approx |J_{r'}|$. Ω and corresponding f_{ij}'s are partitioned according to $I_1, \ldots, I_p$ and distributed across p processors. On the other hand, the parameters $\{w_1, \ldots, w_m\}$ are partitioned into p disjoint subsets $W^{(1)}, \ldots, W^{(p)}$ according to $I_1, \ldots, I_p$ while $\{h_1, \ldots, h_d\}$ are partitioned into p disjoint subsets $H^{(1)}, \ldots, H^{(p)}$ according to $J_1, \ldots, J_p$ and distributed to p processors. The partitioning of $\{1, \ldots, m\}$ and $\{1, \ldots, d\}$ induces a $p \times p$ partition on Ω:

$$\Omega^{(q,r)} := \{(i,j) \in \Omega \ : \ i \in I_q, j \in J_r\}, \quad q, r \in \{1, \ldots, p\}.$$

The execution of DSGD and DSSO algorithm consists of epochs; at the beginning of the r-th epoch ($r \geqslant 1$), processor q owns $H^{(\sigma_r(q))}$ where

$$\sigma_r(q) = \{(q + r - 2) \mod p\} + 1, \tag{2.25}$$

and executes stochastic updates (2.21) and (2.22) for the minimization problem (DSGD) and (2.23) and (2.24) for the saddle-point problem (DSSO), only on coordinates in $\Omega^{(q,\sigma_r(q))}$. Since these updates only involve variables in $W^{(q)}$ and $H^{(\sigma(q))}$, no communication between processors is required to perform these updates. After every processor has finished a pre-defined number of updates, $H^{(q)}$ is sent to $H^{\sigma^{-1}_{(r+1)}(q)}$ and the algorithm moves on to the $(r+1)$-th epoch. The pseudo-code of DSGD and DSSO can be found in Algorithm 1.

It is important to note that DSGD and DSSO are serializable; that is, there is an equivalent update ordering in a serial implementation that would mimic the sequence of DSGD/DSSO updates. In general, serializable algorithms are expected to exhibit faster convergence in number of iterations, as there is little waste of computation due to parallelization [49]. Also, they are easier to debug than non-serializable algorithms which processors may interact with each other in unpredictable complex fashion.

Nonetheless, it is not immediately clear whether DSGD/DSSO would converge to the same solution the original serial algorithm would converge to; while the original

Algorithm 1 Pseudo-code of DSGD and DSSO

```
1: $\{\eta_r\}$: step size sequence
2: Each processor $q$ initializes $W^{(q)}, H^{(q)}$
3: while Convergence do
4:     // start of epoch $r$
5:     Parallel Foreach $q \in \{1, 2, \ldots, p\}$
6:         for $(i, j) \in \Omega^{(q, \sigma_r(q))}$ do
7:             // Stochastic Gradient Update
8:             $w_i \leftarrow w_i - \eta_r \cdot |\Omega| \cdot \nabla_{w_i} f_{ij}(w_i, h_j)$
9:             if DSGD then
10:                $h_j \leftarrow h_j - \eta_r \cdot |\Omega| \cdot \nabla_{h_j} f_{ij}(w_i, h_j)$
11:            else
12:                $h_j \leftarrow h_j + \eta_r \cdot |\Omega| \cdot \nabla_{h_j} f_{ij}(w_i, h_j)$
13:            end if
14:        end for
15:        non-blocking send $H^{\sigma_r(q)}$ to machine $\sigma_{r+1}^{-1}(\sigma_r(q))$
16:        receive $H^{\sigma_{r+1}(q)}$
17:    Parallel End
18:    $r \leftarrow r + 1$
19: end while
```

algorithm samples (i, j)-pairs from entire Ω, the sampling of DSGD/DSSO is confined to $\Omega^{(1,\sigma_r(1))}, \Omega^{(2,\sigma_r(2))}, \ldots, \Omega^{(p,\sigma_r(p))}$, which is only $1/p$ fraction of the whole. Surprisingly, Gemulla et al. [30] proves that DSGD will converge to a local minimum of the objective function f; the proof can be readily adapted for DSSO.

The essential idea of the proof is that the difference between the execution of serial SGD and that of DSGD will be *washed out* if two algorithms are run for a sufficiently long time. This intuition can be quantified as follows. One can see that at r-th epoch, DSGD and DSSO are optimizing the following function:

$$f_r(W, H) = p \cdot \sum_{q=1}^{p} \sum_{(i,j)\in\Omega^{(q,\sigma_r(q))}} f_{ij}(w_i, h_j), \tag{2.26}$$

instead of the original function f. Note that the function is multiplied by p to match the scale of f. Then, the difference between updates made at r-th epoch of serial SGD and DSGD can then be quantified as $\eta_r\left(\nabla_\theta f_r(\theta) - \nabla_\theta f(\theta)\right)$. Theorem 6.1.1 of Yin and Kushner [77] shows that if this "difference" converges to zero, that is,

$$\lim_{s\to\infty} \eta_s \sum_{r=1}^{s} \left(\nabla_\theta f_r(\theta) - \nabla_\theta f(\theta)\right) \to 0, \tag{2.27}$$

then under reasonably mild conditions such as continuity of f and boundedness of the solution, DSGD and DSSO will converge to the set of stable points of ODEs (2.11) and (2.16) respectively, which is the desired result (recall the equivalence between solutions of an optimization problem and stable points of an ODE, as illustrated in Chapter 2.2).

By the construction of σ_r in (2.25),

$$\sum_{r=T}^{T+p-1} \left(\nabla_\theta f_r(\theta) - \nabla_\theta f(\theta)\right) = 0 \tag{2.28}$$

for any positive integer T, because each f_{ij} appears exactly once in p epochs. Therefore, condition (2.27) is trivially satisfied. Of course, there are other choices of σ_r that can also satisfy (2.27); Gemulla et al. [30] shows that if σ_r is a regenerative process, that is, each f_{ij} appears in the temporary objective function f_r in the same frequency, then (2.27) is satisfied.

3. NOMAD: NON-LOCKING, STOCHASTIC MULTI-MACHINE ALGORITHM FOR ASYNCHRONOUS AND DECENTRALIZED OPTIMIZATION

3.1 Motivation

Note that at the end of each epoch, DSGD/DSSO requires every processor to stop sampling stochastic gradients, and communicate column parameters between processors to prepare for the next epoch. In the distributed-memory setting, algorithms that bulk synchronize their state after every iteration are popular [19, 70]. This is partly because of the widespread availability of the MapReduce framework [20] and its open source implementation Hadoop [1].

Unfortunately, bulk synchronization based algorithms have two major drawbacks: First, the communication and computation steps are done in sequence. What this means is that when the CPU is busy, the network is idle and vice versa. The second issue is that they suffer from what is widely known as the *the curse of last reducer* [4, 67]. In other words, all machines have to wait for the slowest machine to finish before proceeding to the next iteration. Zhuang et al. [79] report that DSGD suffers from this problem even in the shared memory setting.

In this section, we present NOMAD (Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized optimization), a parallel algorithm for optimization of doubly separable functions, which processors exchange messages in an asynchronous fashion [11] to avoid bulk synchronization.

3.2 Description

Similarly to DSGD, NOMAD splits row indices $\{1, 2, \ldots, m\}$ into p disjoint sets $I_1, I_2, \ldots, I_p$ which are of approximately equal size. This induces a partition on the rows of the nonzero locations Ω. The q-th processor stores n sets of indices $\bar{\Omega}_j^{(q)}$, for $j \in \{1, \ldots, n\}$, which are defined as

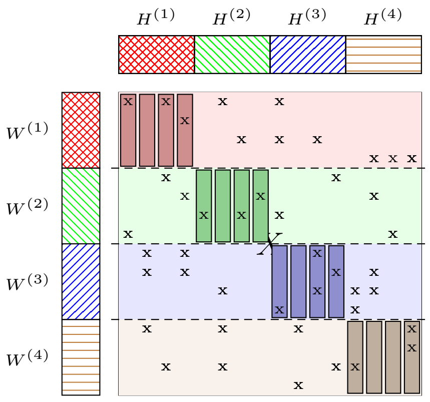
$$\bar{\Omega}_j^{(q)} := \left\{(i, j) \in \bar{\Omega}_j ; i \in I_q\right\},$$

as well as corresponding f_{ij}'s. Note that once Ω and corresponding f_{ij}'s are partitioned and distributed to the processors, they are never moved during the execution of the algorithm.

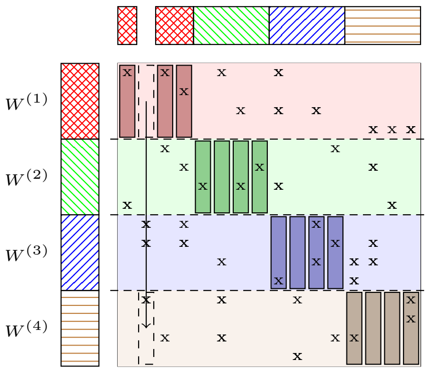
Recall that there are two types of parameters in doubly separable models: row parameters w_i's, and column parameters h_j's. In NOMAD, w_i's are partitioned according to $I_1, I_2, \ldots, I_p$, that is, the q-th processor stores and updates w_i for $i \in I_q$. The variables in W are partitioned at the beginning, and never move across processors during the execution of the algorithm. On the other hand, the h_j's are split randomly into p partitions at the beginning, and their ownership changes as the algorithm progresses. At each point of time an h_j variable resides in one and only processor, and it moves to another processor after it is processed, independent of other item variables. Hence these are called *nomadic* variables[1].

Processing a column parameter h_j at the q-th procesor entails executing SGD updates (2.21) and (2.22) or (2.24) on the (i, j)-pairs in the set $\bar{\Omega}_j^{(q)}$. Note that these updates only require access to h_j and w_i for $i \in I_q$; since I_q's are disjoint, each w_i variable in the set is accessed by only one processor. This is why the communication of w_i variables is not necessary. On the other hand, h_j is updated only by the processor that currently owns it, so there is no need for a lock; this is the popular *owner-computes* rule in parallel computing. See Figure 3.1.

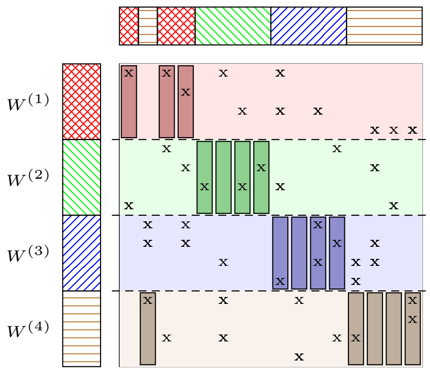
[1]Due to symmetry in the formulation, one can also make the w_i's nomadic and partition the h_j's. To minimize the amount of communication between processors, it is desirable to make h_j's nomadic when $n < m$, and vice versa.



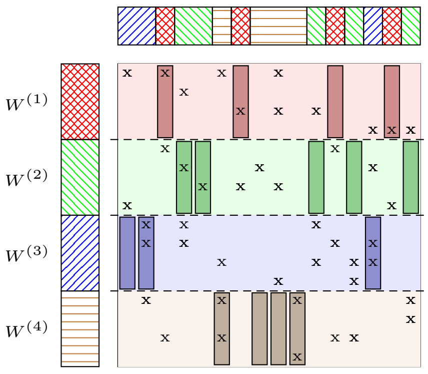
(a) Initial assignment of W and H. Each processor works only on the diagonal active area in the beginning.



(b) After a processor finishes processing column j, it sends the corresponding parameter w_j to another. Here, h_2 is sent from 1 to 4.



(c) Upon receipt, the component is processed by the new processor. Here, processor 4 can now process column 2.



(d) During the execution of the algorithm, the ownership of the component h_j changes.

Figure 3.1.: Graphical Illustration of the Algorithm 2

We now formally define the NOMAD algorithm (see Algorithm 2 for detailed pseudo-code). Each processor q maintains its own concurrent queue, $\texttt{queue}[q]$, which contains a list of columns it has to process. Each element of the list consists of the index of the column j ($1 \leqslant j \leqslant n$), and a corresponding column parameter h_j; this pair is denoted as (j, h_j). Each processor q pops a (j, h_j) pair from its own queue, $\texttt{queue}[q]$, and runs stochastic gradient update on $\bar{\Omega}_j^{(q)}$, which corresponds to functions in column j locally stored in processor q (line 14 to 22). This changes values of w_i for $i \in I_q$ and h_j. After all the updates on column j are done, a uniformly random processor q' is sampled (line 23) and the updated (j, h_j) pair is pushed into the queue of that processor, q' (line 24). Note that this is the only time where a processor communicates with another processor. Also note that the nature of this communication is asynchronous and non-blocking. Furthermore, as long as the queue is nonempty, the computations are completely asynchronous and decentralized. Moreover, all processors are symmetric, that is, there is no designated master or slave.

3.3 Complexity Analysis

First, we consider the case when the problem is distributed across p processors, and study how the space and time complexity behaves as a function of p. Each processor has to store $1/p$ fraction of the m row parameters, and approximately $1/p$ fraction of the n column parameters. Furthermore, each processor also stores approximately $1/p$ fraction of the $|\Omega|$ functions. Then, the space complexity per processor is $O((m + n + |\Omega|)/p)$. As for time complexity, we find it useful to use the following assumptions: performing the SGD updates in line 14 to 22 takes a time and communicating a (j, h_j) to another processor takes c time, where a and c are hardware dependent constants. On the average, each (j, h_j) pair contains $O\left(|\Omega|/np\right)$ non-zero entries. Therefore when a (j, h_j) pair is popped from $\texttt{queue}[q]$ in line 13 of Algorithm 2, on the average it takes $a \cdot \left(|\Omega|/np\right)$ time to process the pair. Since

Algorithm 2 the basic NOMAD algorithm

```
1: $\lambda$: regularization parameter
2: $\{\eta_t\}$: step size sequence
3: Initialize $W$ and $H$
4: // initialize queues
5: for $j \in \{1, 2, \ldots, n\}$ do
6:     $q \sim \text{UniformDiscrete}\,\{1, 2, \ldots, p\}$
7:     queue[$q$].push($(j, h_j)$)
8: end for
9: // start $p$ processors
10: Parallel Foreach $q \in \{1, 2, \ldots, p\}$
11:     while stop signal is not yet received do
12:         if queue[$q$] not empty then
13:             $(j, h_j) \leftarrow$ queue[$q$].pop()
14:             for $(i, j) \in \bar{\Omega}_j^{(q)}$ do
15:                 // Stochastic Gradient Update
16:                 $w_i \leftarrow w_i - \eta_r \cdot |\Omega| \cdot \nabla_{w_i} f_{ij}(w_i, h_j)$
17:                 if minimization problem then
18:                     $h_j \leftarrow h_j - \eta_r \cdot |\Omega| \cdot \nabla_{h_j} f_{ij}(w_i, h_j)$
19:                 else
20:                     $h_j \leftarrow h_j + \eta_r \cdot |\Omega| \cdot \nabla_{h_j} f_{ij}(w_i, h_j)$
21:                 end if
22:             end for
23:             $q' \sim \text{UniformDiscrete}\,\{1, 2, \ldots, p\}$
24:             queue[$q'$].push($(j, h_j)$)
25:         end if
26:     end while
27: Parallel End
```

computation and communication can be done in parallel, as long as $a \cdot (|\Omega| / np)$ is higher than c a processor is always busy and NOMAD scales linearly.

Suppose that $|\Omega|$ is fixed but the number of processors p increases; that is, we take a fixed size dataset and distribute it across p processors. As expected, for a large enough value of p (which is determined by hardware dependent constants a and b) the cost of communication will overwhelm the cost of processing an item, thus leading to slowdown.

On the other hand, suppose the work per processor is fixed, that is, $|\Omega|$ increases and the number of processors p increases proportionally. The average time $a \cdot (|\Omega| / np)$ to process an item remains constant, and NOMAD scales linearly.

Finally, we discuss the communication complexity of NOMAD. For this discussion we focus on a single column parameter h_j. In order to be processed by all the p processors once, it needs to be communicated p times. This requires $O(p)$ communication per item. There are n items, and if we make a simplifying assumption that during the execution of NOMAD each item is processed a constant c number of times by each processor, then the total communication complexity is $O(np)$.

3.4 Dynamic Load Balancing

As different processors have different number of nonzero functions per column, the speed at which a processor processes a set of functions $\bar{\Omega}_j^{(q)}$ for a column j also varies among processors. Furthermore, in the distributed memory setting different processors might execute updates at different rates dues to differences in hardware and system load. NOMAD can handle this by dynamically balancing the workload of processors: in line 23 of Algorithm 2, instead of sampling the recipient of a message uniformly at random we can preferentially select a processors which has fewer columns in its queue to process. To do this, a payload carrying information about the size of the $\texttt{queue}[q]$ is added to the messages that the processors send each other. The overhead of the payload is just a single integer per message. This scheme allows us to dynamically load balance, and ensures that a slower processor will receive smaller amount of work compared to others.

3.5 Hybrid Architecture

In a hybrid architecture we have multiple threads on a single machine as well as multiple machines distributed across the network. In this case, we make two improvements to the basic NOMAD algorithm. First, in order to amortize the communication costs we reserve two additional threads per machine for sending and receiving (j, h_j) pairs over the network. Intra-machine communication is much cheaper than machine-to-machine communication, since the former does not involve a network hop. Therefore, whenever a machine receives a (j, h_j) pair, it circulates the pair among all of its threads before sending the pair over the network. This is done by uniformly sampling a random permutation whose size equals to the number of processor threads, and sending the column variable to each thread according to this permutation. Circulating a variable more than once was found to not improve convergence in practice, and hence is not used in our implementations.

3.5.1 Implementation Details

Multi-threaded MPI was used for inter-machine communication. Instead of communicating single (j, h_j) pairs, we follow the strategy of [65], and accumulate a fixed number of pairs (e.g., 100) before transmitting them over the network.

NOMAD can be implemented with lock-free data structures since the only interaction between threads is via operations on the queue. We used the concurrent queue provided by Intel Thread Building Blocks (TBB) [3]. Although technically not lock-free, the TBB concurrent queue nevertheless scales almost linearly with the number of threads.

There is very minimal sharing of memory among threads in NOMAD. By making memory assignments in each thread carefully aligned with cache lines we can exploit memory locality and avoid cache ping-pong. This results in near linear scaling for the multi-threaded setting.

3.6 Related Work

3.6.1 Map-Reduce and Friends

There are other existing approaches to parallelize SGD for matrix completion, and they can also be generalized for doubly separable functions as well. In this section, we briefly discuss how NOMAD is conceptually different from those methods.

DSGD++ is an algorithm proposed by Teflioudi et al. [69] to address the aforementioned utilization issue of computation and communication resources. Instead of using p partitions, DSGD++ uses $2p$ partitions. While the p processors are processing p partitions, the other p partitions are sent over the network. This keeps both the network and CPU busy simultaneously. However, DSGD++ also suffers from the curse of the last reducer.

Another attempt to alleviate the problems of bulk synchronization in the shared memory setting is the FPSGD** algorithm of Zhuang et al. [79]; given p threads, FPSGD** partitions the parameters into more than p sets, and uses a task manager thread to distribute the partitions. When a thread finishes updating one partition, it requests for another partition from the task manager. It is unclear how to extend this idea to the distributed memory setting.

In NOMAD we sidestep all the drawbacks of bulk synchronization. Like DSGD++ we also simultaneously keep the network and CPU busy. On the other hand, like FPSGD** we effectively load balance between the threads. To understand why NOMAD enjoys both these benefits, it is instructive to contrast the data partitioning schemes underlying DSGD, DSGD++, FPSGD**, and NOMAD (see Figure 3.2). Given p number of processors, DSGD divides locations of non-zero functions Ω into $p \times p$ number of blocks; DSGD++ improves upon DSGD by further dividing each block to 1×2 sub-blocks (Figure 3.2 (a) and (b)). On the other hand, FPSGD**

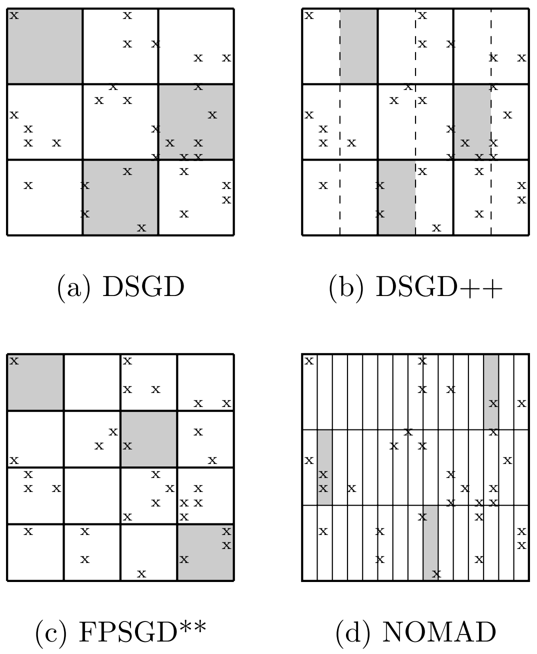


Figure 3.2.: Comparison of data partitioning schemes between algorithms. Example active area of stochastic gradient sampling is marked as gray.

splits A into $p' \times p'$ blocks with $p' > p$ (Figure 3.2 (c)), while NOMAD uses $p \times n$ blocks (Figure 3.2 (d)). In terms of communication there is no difference between various partitioning schemes; all of them require $O(np)$ communication for each column to be processed a constant c number of times. However, having smaller blocks means that NOMAD has much more flexibility in assigning blocks to processors, and hence better ability to exploit parallelism. Because NOMAD operates at the level of individual column parameters, h_j, it can dynamically load balance by assigning fewer columns to a slower processor. A pleasant side effect of such a fine grained partitioning coupled with the lock free nature of updates is that one does not require sophisticated scheduling algorithms to achieve good performance. Consequently, NOMAD has outperformed DSGD, DSGD++, and FPSGD** in our experiments (see Chapter 4.3).

3.6.2 Asynchronous Algorithms

There is growing interest in designing machine learning algorithms that do not perform bulk synchronization. See, for instance, the randomized (block) coordinate descent methods of Richtarik and Takac [57] and the Hogwild! algorithm of Recht et al. [56]. A relatively new approach to asynchronous parallelism is to use a so-called parameter server. A parameter server is either a single machine or a distributed set of machines which caches the current values of the parameters. Processors store local copies of the parameters and perform updates on them, and periodically synchronize their local copies with the parameter server. The parameter server receives updates from all processors, aggregates them, and communicates them back to the processors. The earliest work on a parameter server, that we are aware of, is due to Smola and Narayanamurthy [65], who propose using a parameter server for collapsed Gibbs sampling in Latent Dirichlet Allocation. PowerGraph [32], upon which the latest version of the GraphLab toolkit is based, is also essentially based on the idea of a parameter server. However, the difference in case of PowerGraph is that the responsibility of parameters is distributed across multiple machines, but at the added expense of synchronizing the copies.

Very roughly speaking, the asynchronously parallel optimization algorithm for matrix completion in GraphLab works as follows: W_i and H_j variables are distributed across multiple machines, and whenever W_i is being updated as a solution of the sub-problem (2.10), the values of H_j's for $j \in \Omega_i$ are retrieved across the network and read-locked until the update is finished. GraphLab provides functionality such as network communication and a distributed locking mechanism to implement this. However, frequently acquiring read-locks over the network can be expensive. GraphLab provides a complex job scheduler which attempts to minimize this cost, but then the efficiency of parallelization depends on the difficulty of the scheduling problem and the effectiveness of the scheduler.

Our empirical evaluation in Appendix A.3 shows that NOMAD performs significantly better than GraphLab. The reasons are not hard to see. First, because of the lock free nature of NOMAD, we completely avoid acquiring expensive network locks. Second, we use stochastic updates which allows us to exploit finer grained parallelism as compared to solving the minimization problem (2.10) which involves more number of coordinates than two. In fact, the GraphLab framework is not well suited for SGD (personal communication with the developers of GraphLab). Finally, because of the finer grained data partitioning scheme used in NOMAD, unlike GraphLab whose performance heavily depends on the underlying scheduling algorithms we do not require a complicated scheduling mechanism.

3.6.3 Numerical Linear Algebra

The concepts of asynchronous and non-blocking updates have also been studied in numerical linear algebra. To avoid the load balancing problem and to reduce processor idle time, asynchronous numerical methods were first proposed over four decades ago by Chazan and Miranker [18]. Given an operator $\mathcal{H} : \mathbb{R}^m \to \mathbb{R}^m$, to find the fixed point solution x^* such that $\mathcal{H}(x^*) = x^*$, a standard Gauss-Seidel-type procedure performs the update $x_i = (\mathcal{H}(x))_i$ sequentially (or randomly). Using the asynchronous procedure, each computational node asynchronously conducts updates on each variable (or a subset) $x_i^{\text{new}} = (\mathcal{H}(x))_i$ and then overwrites x_i in common memory by x_i^{new}. Theory and applications of this asynchronous method have been widely studied (see the literature review of Frommer and Szyld [29] and the seminal textbook by Bertsekas and Tsitsiklis [11]). The concept of this asynchronous fixed-point update is very closely related to the Hogwild algorithm of Recht et al. [56] or the so-called Asynchronous SGD (ASGD) method proposed by Teflioudi et al. [69]. Unfortunately, such algorithms are *non-serializable*, that is, there may not exist an equivalent update ordering in a serial implementation. In contrast, our NOMAD algorithm is not only asynchronous but also serializable, and therefore achieves faster convergence in practice.

On the other hand, non-blocking communication has also been proposed to accelerate iterative solvers in a distributed setting. For example, Hoefler et al. [36] presented a distributed conjugate gradient (CG) implementation with non-blocking collective MPI operations for solving linear systems. However, this algorithm still requires synchronization at each CG iteration, so it is very different from our NOMAD algorithm.

3.6.4 Discussion

We remark that among algorithms we have discussed so far, NOMAD is the only distributed-memory algorithm which is both asynchronous and lock-free. Other parallelizations of SGD such as DSGD and DSGD++ are lock-free, but not fully asynchronous; therefore, the cost of synchronization will increase as the number of machines grows [79]. On the other hand, GraphLab [49] is asynchronous but not lock-free, therefore depends on a complex job scheduler to reduce the side-effect of using locks.

4. MATRIX COMPLETION

As discussed in Chapter 2.3, many of the parallel SGD algorithms were specifically developed for the matrix completion model. However, there are many deterministic optimization algorithms for matrix completion as well; therefore, the matrix completion problem is an ideal benchmark test that can be used to evaluate the effectiveness of algorithms we have introduced so far.

Note that the matrix completion model itself has been considered as an important statistical model in machine learning and data mining community as well. It is partly thanks to the empirical success of the model in the Netflix prize challenge [41], which participants of the challenge were asked to predict unseen movie ratings of users based on given training data.

4.1 Formulation

Most of the notations defined here are consistent with those introduced in Chapter 2. We redefine some of them here, however, to illustrate their interpretation in this particular context of matrix completion.

Let $A \in \mathbb{R}^{m \times n}$ be a rating matrix, where m denotes the number of users and n the number of items. Typically $m \gg n$, although the algorithms we consider in this paper do not depend on such an assumption. Furthermore, let $\Omega \subseteq \{1 \ldots m\} \times \{1, \ldots, n\}$ denote the observed entries of A, that is, $(i, j) \in \Omega$ implies that user i gave item j a rating of A_{ij}. The goal here is to predict accurately the unobserved ratings. For convenience, we define Ω_i to be the set of items rated by the i-th user, i.e., $\Omega_i := \{j : (i, j) \in \Omega\}$. Analogously $\bar{\Omega}_j := \{i : (i, j) \in \Omega\}$ is the set of users who have rated item j. Also, let $\mathbf{a}_i^\top$ denote the i-th row of A.

A standard model for matrix completion finds matrices $W \in \mathbb{R}^{m \times k}$ and $H \in \mathbb{R}^{n \times k}$, with $k \ll \min(m, n)$, such that $A \approx WH^\top$. One way to understand this model is to realize that each row $\mathbf{w}_i^\top \in \mathbb{R}^k$ of W can be thought of as a k-dimensional embedding of the user. Analogously, each row $\mathbf{h}_j^\top \in \mathbb{R}^k$ of H is an embedding of the item in the same k-dimensional space. In order to predict the (i, j)-th entry of A we simply use $\langle \mathbf{w}_i, \mathbf{h}_j \rangle$, where $\langle \cdot, \cdot \rangle$ denotes the Euclidean inner product of two vectors. The goodness of fit of the model is measured by a loss function. While DSGD or NOMAD can work with an arbitrary separable loss, for ease of exposition we will only discuss the square loss: $\frac{1}{2} \left(A_{ij} - \langle \mathbf{w}_i, \mathbf{h}_j \rangle \right)^2$. Furthermore, we need to enforce regularization to prevent over-fitting, and to predict well on the unknown entries of A. Again, a variety of regularizers can be handled by our algorithm, but we will only focus on the following weighted square norm-regularization in this paper: $\frac{\lambda}{2} \sum_{i=1}^m |\Omega_i| \cdot \|\mathbf{w}_i\|^2 + \frac{\lambda}{2} \sum_{j=1}^n |\bar{\Omega}_j| \cdot \|\mathbf{h}_i\|^2$, where $\lambda > 0$ is a regularization parameter. Here, $|\cdot|$ denotes the cardinality of a set, and $\|\cdot\|^2$ is the L_2 norm of a vector. Putting everything together yields the following objective function:

$$\min_{\substack{W \in \mathbb{R}^{m \times k} \\ H \in \mathbb{R}^{n \times k}}} J(W, H) := \frac{1}{2} \sum_{(i,j) \in \Omega} \left(A_{ij} - \langle \mathbf{w}_i, \mathbf{h}_j \rangle \right)^2 + \frac{\lambda}{2} \left(\sum_{i=1}^m |\Omega_i| \cdot \|\mathbf{w}_i\|^2 + \sum_{j=1}^n |\bar{\Omega}_j| \cdot \|\mathbf{h}_i\|^2 \right). \tag{4.1}$$

This can be further simplified and written as

$$J(W, H) = \frac{1}{2} \sum_{(i,j) \in \Omega} \left\{ \left(A_{ij} - \langle \mathbf{w}_i, \mathbf{h}_j \rangle \right)^2 + \lambda \left(\|\mathbf{w}_i\|^2 + \|\mathbf{h}_j\|^2 \right) \right\}.$$

Observe that this is in doubly separable form (2.2).

In the sequel we will let w_{il} and h_{il} for $1 \leqslant l \leqslant k$ denote the l-th coordinate of the column vectors $\mathbf{w}_i$ and $\mathbf{h}_j$, respectively. Furthermore, H_{Ω_i} (resp. $W_{\bar{\Omega}_j}$) will be used to denote the sub-matrix of H (resp. W) formed by collecting rows corresponding to Ω_i (resp. $\bar{\Omega}_j$).

Note that as illustrated in (2.10), if we fix H then the optimization problem (4.1) decomposes to m independent convex optimization problems, with each of them having the following form:

$$\min_{\mathbf{w}_i \in \mathbb{R}^k} J_i(\mathbf{w}_i) = \frac{1}{2} \sum_{j \in \Omega_i} \left(A_{ij} - \langle \mathbf{w}_i, \mathbf{h}_j \rangle\right)^2 + \lambda \left\| \mathbf{w}_i \right\|^2 . \tag{4.2}$$

Analogously, if we fix W then (4.1) decomposes into n independent convex optimization problems, each of which has the following form:

$$\min_{\mathbf{h}_j \in \mathbb{R}^k} \bar{J}_j(\mathbf{h}_j) = \frac{1}{2} \sum_{i \in \bar{\Omega}_j} \left(A_{ij} - \langle \mathbf{w}_i, \mathbf{h}_j \rangle\right)^2 + \lambda \left\| \mathbf{h}_j \right\|^2 .$$

The gradient and Hessian of $J_i(\mathbf{w})$ can be easily computed:

$$\nabla J_i(\mathbf{w}_i) = M\mathbf{w}_i - \mathbf{b}, \text{ and } \nabla^2 J_i(\mathbf{w}_i) = M,$$

where we have defined $M := H_{\Omega_i}^\top H_{\Omega_i} + \lambda I$ and $\mathbf{b} := H^\top \mathbf{a}_i$.

4.2 Batch Optimization Algorithms

While we have introduced stochastic optimization techniques in Chapter 2.3, here we will present two well known batch optimization strategies specifically developed for solving (4.1). These two approaches essentially differ in only two characteristics namely, the sequence in which updates to the variables in W and H are carried out, and the level of approximation in the update.

4.2.1 Alternating Least Squares

A simple version of the Alternating Least Squares (ALS) algorithm updates variables as follows: $\mathbf{w}_1$, $\mathbf{w}_2$, …, $\mathbf{w}_m$, $\mathbf{h}_1$, $\mathbf{h}_2$, …, $\mathbf{h}_n$, $\mathbf{w}_1$, … and so on. Updates to $\mathbf{w}_i$ are computed by solving (4.2) which is in fact a least squares problem, and thus the following Newton update gives us:

$$\mathbf{w}_i \leftarrow \mathbf{w}_i - \left[\nabla^2 J_i(\mathbf{w}_i)\right]^{-1} \nabla J_i(\mathbf{w}_i), \tag{4.3}$$

which can be rewritten using M and $\mathbf{b}$ as $\mathbf{w}_i \leftarrow M^{-1}\mathbf{b}$. Updates to $\mathbf{h}_j$'s are analogous.

4.2.2 Coordinate Descent

The ALS update involves formation of the Hessian and its inversion. In order to reduce the computational complexity, one can replace the Hessian by its diagonal approximation:

$$\mathbf{w}_i \leftarrow \mathbf{w}_i - \left[\operatorname{diag}\left(\nabla^2 J_i\left(\mathbf{w}_i\right)\right)\right]^{-1} \nabla J_i\left(\mathbf{w}_i\right), \tag{4.4}$$

which can be rewritten using M and $\mathbf{b}$ as

$$\mathbf{w}_i \leftarrow \mathbf{w}_i - \operatorname{diag}(M)^{-1}\left[M\mathbf{w}_i - \mathbf{b}\right]. \tag{4.5}$$

If we update one component of $\mathbf{w}_i$ at a time, the update (4.5) can be written as:

$$w_{il} \leftarrow w_{il} - \frac{\langle \mathbf{m}_l, \mathbf{w}_i \rangle - b_l}{m_{ll}}, \tag{4.6}$$

where $\mathbf{m}_l$ is l-th row of matrix M, b_l is l-th component of $\mathbf{b}$ and m_{ll} is the l-th coordinate of $\mathbf{m}_l$.

If we choose the update sequence $w_{11}, \ldots, w_{1k}, w_{21}, \ldots, w_{2k}, \ldots, w_{m1}, \ldots, w_{mk}, h_{11}, \ldots, h_{1k}, h_{21}, \ldots, h_{2k}, \ldots, h_{n1}, \ldots, h_{nk}, w_{11}, \ldots, w_{1k}$, and so on, then this recovers Cyclic Coordinate Descent (CCD) [37]. On the other hand, the update sequence $w_{11}, \ldots, w_{m1}, h_{11}, \ldots, h_{n1}, w_{12}, \ldots, w_{m2}, h_{12}, \ldots, h_{n2}$ and so on, recovers the CCD++ algorithm of Yu et al. [78]. The CCD++ updates can be performed more efficiently than the CCD updates by maintaining a residual matrix [78].

4.3 Experiments

In this section, we evaluate the empirical performance of NOMAD with extensive experiments. For the distributed memory experiments we compare NOMAD with DSGD [31], DSGD++ [69] and CCD++ [78]. We also compare against GraphLab, but the quality of results produced by GraphLab are significantly worse than the other methods, and therefore the plots for this experiment are delegated to Appendix A.3. For the shared memory experiments we pitch NOMAD against FPSGD** [79] (which is shown to outperform DSGD in single machine experiments) as well as CCD++. Our experiments are designed to answer the following:

- How does NOMAD scale with the number of cores on a single machine? (Section 4.3.2)
- How does NOMAD scale as a fixed size dataset is distributed across multiple machines? (Section 4.3.3)
- How does NOMAD perform on a commodity hardware cluster? (Chapter 4.3.4)
- How does NOMAD scale when both the size of the data as well as the number of machines grow? (Section 4.3.5)

Since the objective function (4.1) is non-convex, different optimizers will converge to different solutions. Factors which affect the quality of the final solution include 1) initialization strategy, 2) the sequence in which the ratings are accessed, and 3) the step size decay schedule. It is clearly not feasible to consider the combinatorial effect of all these factors on each algorithm. However, we believe that the overall trend of our results is not affected by these factors.

4.3.1 Experimental Setup

Publicly available code for FPSGD**[1] and CCD++[2] was used in our experiments. For DSGD and DSGD++, which we had to implement ourselves because the code is not publicly available, we closely followed the recommendations of Gemulla et al. [31] and Teflioudi et al. [69], and in some cases made improvements based on our experience. For a fair comparison all competing algorithms were tuned for optimal performance on our hardware. The code and scripts required for reproducing the experiments are readily available for download from `http://www.stat.purdue.edu/~yun3`. Parameters used in our experiments are summarized in Table 4.1.

[1] `http://www.csie.ntu.edu.tw/~cjlin/libmf/`

[2] `http://www.cs.utexas.edu/~rofuyu/libpmf/`

Table 4.1.: Dimensionality parameter k, regularization parameter λ (4.1) and step-size schedule parameters α, β (4.7)

Name	k	λ	α	β
Netflix	100	0.05	0.012	0.05
Yahoo! Music	100	1.00	0.00075	0.01
Hugewiki	100	0.01	0.001	0

Table 4.2.: Dataset Details

Name	Rows	Columns	Non-zeros
Netflix [7]	2,649,429	17,770	99,072,112
Yahoo! Music [23]	1,999,990	624,961	252,800,275
Hugewiki [2]	50,082,603	39,780	2,736,496,604

For all experiments, except the ones in Chapter 4.3.5, we will work with three benchmark datasets namely Netflix, Yahoo! Music, and Hugewiki (see Table 5.2 for more details). The same training and test dataset partition is used consistently for all algorithms in every experiment. Since our goal is to compare optimization algorithms, we do very minimal parameter tuning. For instance, we used the same regularization parameter λ for each dataset as reported by Yu et al. [78], and shown in Table 4.1; we study the effect of the regularization parameter on the convergence of NOMAD in Appendix A.1. By default we use $k = 100$ for the dimension of the latent space; we study how the dimension of the latent space affects convergence of NOMAD in Appendix A.2. All algorithms were initialized with the same initial parameters; we set each entry of W and H by independently sampling a uniformly random variable in the range $(0, \frac{1}{\sqrt{k}})$ [78, 79].

We compare solvers in terms of Root Mean Square Error (RMSE) on the test set, which is defined as:

$$\sqrt{\frac{\sum_{(i,j)\in\Omega^{\text{test}}} \left(A_{ij} - \langle \mathbf{w}_i, \mathbf{h}_j \rangle\right)^2}{|\Omega^{\text{test}}|}},$$

where Ω^{test} denotes the ratings in the test set.

All experiments, except the ones reported in Chapter 4.3.4, are run using the Stampede Cluster at University of Texas, a Linux cluster where each node is outfitted with 2 Intel Xeon E5 (Sandy Bridge) processors and an Intel Xeon Phi Coprocessor (MIC Architecture). For single-machine experiments (Chapter 4.3.2), we used nodes in the `largemem` queue which are equipped with 1TB of RAM and 32 cores. For all other experiments, we used the nodes in the `normal` queue which are equipped with 32 GB of RAM and 16 cores (only 4 out of the 16 cores were used for computation). Inter-machine communication on this system is handled by MVAPICH2.

For the commodity hardware experiments in Chapter 4.3.4 we used `m1.xlarge` instances of Amazon Web Services, which are equipped with 15GB of RAM and four cores. We utilized all four cores in each machine; NOMAD and DSGD++ uses two cores for computation and two cores for network communication, while DSGD and

CCD++ use all four cores for both computation and communication. Inter-machine communication on this system is handled by MPICH2.

Since FPSGD** uses single precision arithmetic, the experiments in Chapter 4.3.2 are performed using single precision arithmetic, while all other experiments use double precision arithmetic. All algorithms are compiled with Intel C++ compiler, with the exception of experiments in Chapter 4.3.4 where we used `gcc` which is the only compiler toolchain available on the commodity hardware cluster. For ready reference, exceptions to the experimental settings specific to each section are summarized in Table 4.3.

Table 4.3.: Exceptions to each experiment

Section	Exception
Chapter 4.3.2	• run on `largemem` queue (32 cores, 1TB RAM) • single precision floating point used
Chapter 4.3.4	• run on `m1.xlarge` (4 cores, 15GB RAM) • compiled with `gcc` • `MPICH2` for MPI implementation
Chapter 4.3.5	• Synthetic datasets

The convergence speed of stochastic gradient descent methods depends on the choice of the step size schedule. The schedule we used for NOMAD is

$$s_t = \frac{\alpha}{1 + \beta \cdot t^{1.5}}, \tag{4.7}$$

where t is the number of SGD updates that were performed on a particular user-item pair (i, j). DSGD and DSGD++, on the other hand, use an alternative strategy called bold-driver [31]; here, the step size is adapted by monitoring the change of the objective function.

4.3.2 Scaling in Number of Cores

For the first experiment we fixed the number of cores to 30, and compared the performance of NOMAD vs FPSGD**[3] and CCD++ (Figure 4.1). On Netflix (left) NOMAD not only converges to a slightly better quality solution (RMSE 0.914 vs 0.916 of others), but is also able to reduce the RMSE rapidly right from the beginning. On Yahoo! Music (middle), NOMAD converges to a slightly worse solution than FPSGD** (RMSE 21.894 vs 21.853) but as in the case of Netflix, the initial convergence is more rapid. On Hugewiki, the difference is smaller but NOMAD still outperforms. The initial speed of CCD++ on Hugewiki is comparable to NOMAD, but the quality of the solution starts to deteriorate in the middle. Note that the performance of CCD++ here is better than what was reported in Zhuang et al. [79] since they used double-precision floating point arithmetic for CCD++. In other experiments (not reported here) we varied the number of cores and found that the relative difference in performance between NOMAD, FPSGD** and CCD++ are very similar to that observed in Figure 4.1.

For the second experiment we varied the number of cores from 4 to 30, and plot the scaling behavior of NOMAD (Figures 4.2, 4.3 and 4.4). Figure 4.2 shows how test RMSE changes as a function of the number of updates. Interestingly, as we increased the number of cores, the test RMSE decreased faster. We believe this is because when we increase the number of cores, the rating matrix A is partitioned into smaller blocks; recall that we split A into $p \times n$ blocks, where p is the number of parallel processors. Therefore, the communication between processors becomes more frequent, and each SGD update is based on fresher information (see also Chapter 3.3 for mathematical analysis). This effect was more strongly observed on Yahoo! Music dataset than others, since Yahoo! Music has much larger number of items (624,961 vs. 17,770 of Netflix and 39,780 of Hugewiki) and therefore more amount of communication is needed to circulate the new information to all processors.

[3]Since the current implementation of FPSGD** in LibMF only reports CPU execution time, we divide this by the number of threads and use this as a proxy for wall clock time.

On the other hand, to assess the efficiency of computation we define *average throughput* as the average number of ratings processed per core per second, and plot it for each dataset in Figure 4.3, while varying the number of cores. If NOMAD exhibits linear scaling in terms of the speed it processes ratings, the average throughput should remain constant[4]. On Netflix, the average throughput indeed remains almost constant as the number of cores changes. On Yahoo! Music and Hugewiki, the throughput decreases to about 50% as the number of cores is increased to 30. We believe this is mainly due to cache locality effects.

Now we study how much speed-up NOMAD can achieve by increasing the number of cores. In Figure 4.4, we set y-axis to be test RMSE and x-axis to be the total CPU time expended which is given by the number of seconds elapsed multiplied by the number of cores. We plot the convergence curves by setting the # cores=4, 8, 16, and 30. If the curves overlap, then this shows that we achieve linear speed up as we increase the number of cores. This is indeed the case for Netflix and Hugewiki. In the case of Yahoo! Music we observe that the speed of convergence increases as the number of cores increases. This, we believe, is again due to the decrease in the block size which leads to faster convergence.

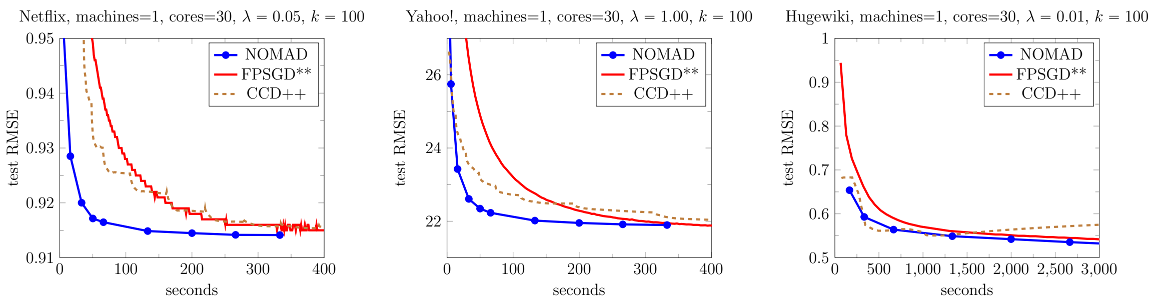


Figure 4.1.: Comparison of NOMAD, FPSGD**, and CCD++ on a single-machine with 30 computation cores.

[4]Note that since we use single-precision floating point arithmetic in this section to match the implementation of FPSGD**, the throughput of NOMAD is about 50% higher than that in other experiments.

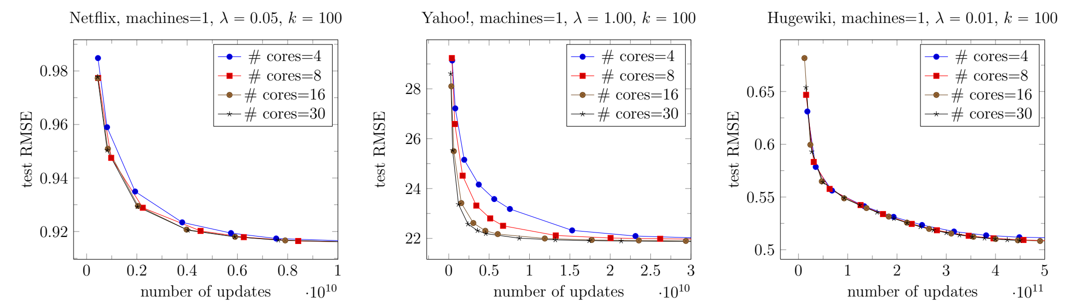


Figure 4.2.: Test RMSE of NOMAD as a function of the number of updates, when the number of cores is varied.

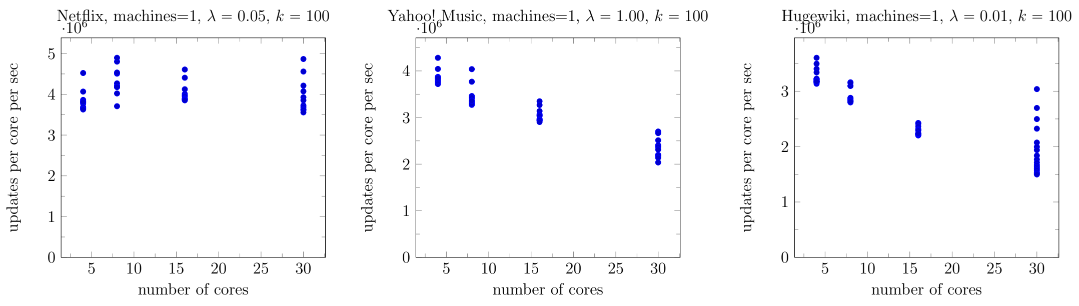


Figure 4.3.: Number of updates of NOMAD per core per second as a function of the number of cores.

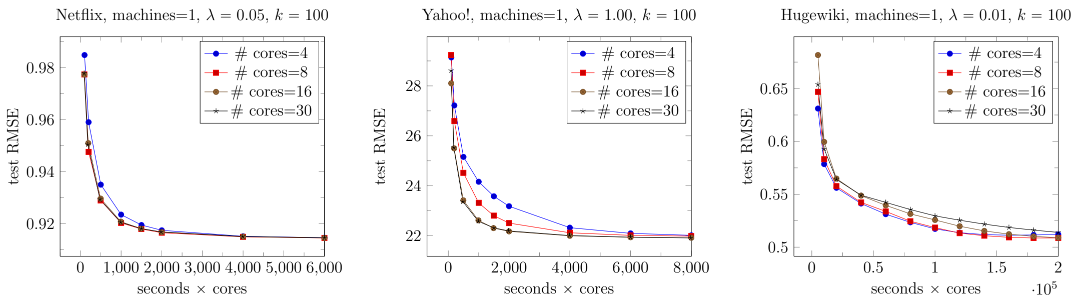


Figure 4.4.: Test RMSE of NOMAD as a function of computation time (time in seconds × the number of cores), when the number of cores is varied.

4.3.3 Scaling as a Fixed Dataset is Distributed Across Processors

In this subsection, we use 4 computation threads per machine. For the first experiment we fix the number of machines to 32 (64 for hugewiki), and compare the performance of NOMAD with DSGD, DSGD++ and CCD++ (Figure 4.5). On Netflix and Hugewiki, NOMAD converges much faster than its competitors; not only initial convergence is faster, it also discovers a better quality solution. On Yahoo! Music, four methods perform almost the same to each other. This is because the cost of network communication relative to the size of the data is much higher for Yahoo! Music; while Netflix and Hugewiki have 5,575 and 68,635 non-zero ratings per each item respectively, Yahoo! Music has only 404 ratings per item. Therefore, when Yahoo! Music is divided equally across 32 machines, each item has only 10 ratings on average per each machine. Hence the cost of sending and receiving item parameter vector $\mathbf{h}_j$ for one item j across the network is higher than that of executing SGD updates on the ratings of the item locally stored within the machine, $\bar{\Omega}_j^{(q)}$. As a consequence, the cost of network communication dominates the overall execution time of all algorithms, and little difference in convergence speed is found between them.

For the second experiment we varied the number of machines from 1 to 32, and plot the scaling behavior of NOMAD (Figures 4.6, 4.7 and 4.8). Figures 4.6 shows how test RMSE decreases as a function of the number of updates. Again, if NOMAD scales linearly the average throughput has to remain constant. On the Netflix dataset (left) convergence is mildly slower with two or four machines. However, as we increase the number of machines the speed of convergence improves. On Yahoo! Music (center), we uniformly observe improvement in convergence speed when 8 or more machines are used; this is again the effect of smaller block sizes which was discussed in Chapter 4.3.2. On the Hugewiki dataset, however, we do not see any notable difference between configurations.

In Figure 4.7 we plot the average throughput (the number of updates per machine per core per second) as a function of the number of machines. On Yahoo! Music the average throughput goes down as we increase the number of machines, because as mentioned above, each item has a small number of ratings. On Hugewiki we observe almost linear scaling, and on Netflix the average throughput even improves as we increase the number of machines; we believe this is because of cache locality effects. As we partition users into smaller and smaller blocks, the probability of cache miss on user parameters $\mathbf{w}_i$'s within the block decrease, and on Netflix this makes a meaningful difference: indeed, there are only 480,189 users in Netflix who have at least one rating. When this is equally divided into 32 machines, each machine contains only 11,722 active users on average. Therefore the $\mathbf{w}_i$ variables only take 11MB of memory, which is smaller than the size of L3 cache (20MB) of the machine we used and therefore leads to increase in the number of updates per machine per core per second.

Now we study how much speed-up NOMAD can achieve by increasing the number of machines. In Figure 4.8, we set y-axis to be test RMSE and x-axis to be the number of seconds elapsed multiplied by the total number of cores used in the configuration. Again, all lines will coincide with each other if NOMAD shows linear scaling. On Netflix, with 2 and 4 machines we observe mild slowdown, but with more than 4 machines NOMAD exhibits super-linear scaling. On Yahoo! Music we observe super-linear scaling with respect to the speed of a single machine on all configurations, but the highest speedup is seen with 16 machines. On Hugewiki, linear scaling is observed in every configuration.

4.3.4 Scaling on Commodity Hardware

In this subsection, we want to analyze the scaling behavior of NOMAD on commodity hardware. Using Amazon Web Services (AWS), we set up a computing cluster that consists of 32 machines; each machine is of type `m1.xlarge` and equipped with

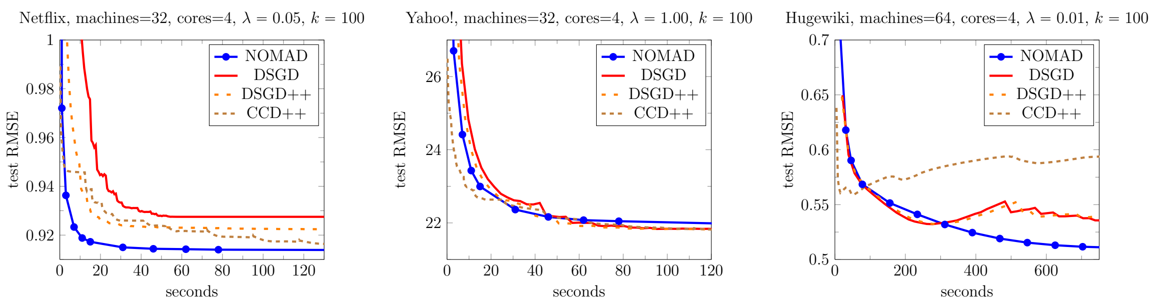


Figure 4.5.: Comparison of NOMAD, DSGD, DSGD++, and CCD++ on a HPC cluster.

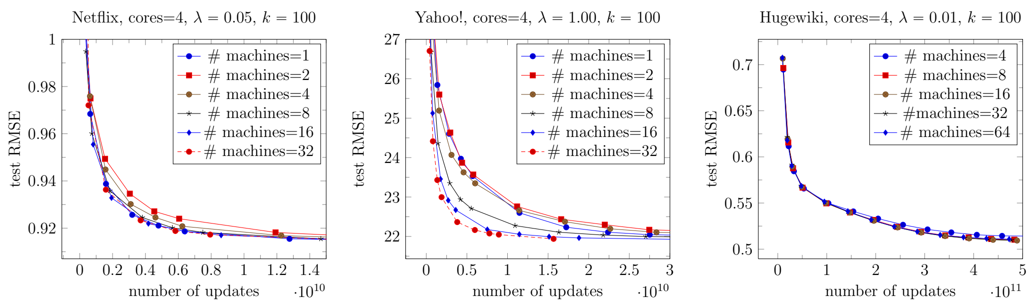


Figure 4.6.: Test RMSE of NOMAD as a function of the number of updates on a HPC cluster, when the number of machines is varied.

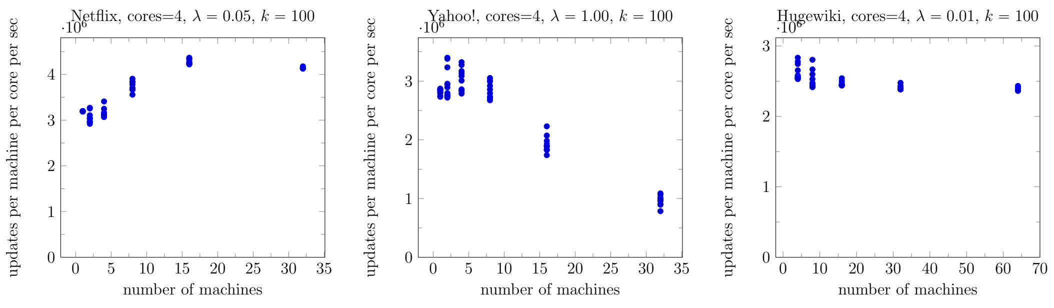


Figure 4.7.: Number of updates of NOMAD per machine per core per second as a function of the number of machines, on a HPC cluster.

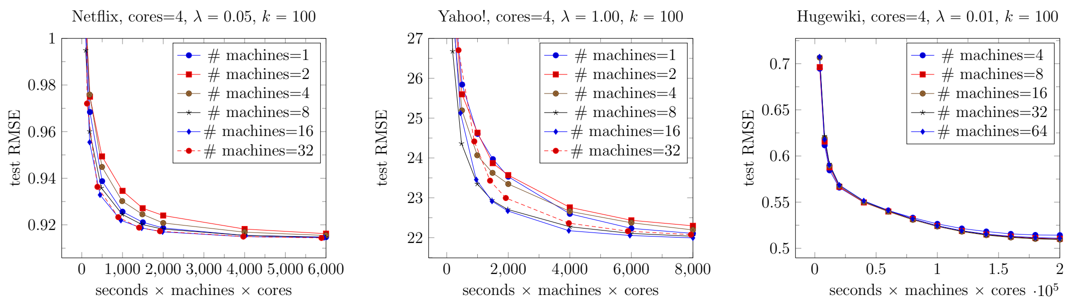


Figure 4.8.: Test RMSE of NOMAD as a function of computation time (time in seconds × the number of machines × the number of cores per each machine) on a HPC cluster, when the number of machines is varied.

quad-core Intel Xeon E5430 CPU and 15GB of RAM. Network bandwidth among these machines is reported to be approximately 1Gb/s[5].

Since NOMAD and DSGD++ dedicates two threads for network communication, on each machine only two cores are available for computation[6]. In contrast, bulk synchronization algorithms such as DSGD and CCD++ which separate computation and communication can utilize all four cores for computation. In spite of this disadvantage, Figure 4.9 shows that NOMAD outperforms all other algorithms in this setting as well. In this plot, we fixed the number of machines to 32; on Netflix and Hugewiki, NOMAD converges more rapidly to a better solution. Recall that on Yahoo! Music, all four algorithms performed very similarly on a HPC cluster in Chapter 4.3.3. However, on commodity hardware NOMAD outperforms the other algorithms. This shows that the efficiency of network communication plays a very important role in commodity hardware clusters where the communication is relatively slow. On Hugewiki, however, the number of columns is very small compared to the number of ratings and thus network communication plays smaller role in this dataset compared to others. Therefore, initial convergence of DSGD is a bit faster than NOMAD as it uses all four cores on computation while NOMAD uses only two. Still, the overall convergence speed is similar and NOMAD finds a better quality solution.

As in Chapter 4.3.3, we increased the number of machines from 1 to 32, and studied the scaling behavior of NOMAD. The overall pattern is identical to what was found in Figure 4.6, 4.7 and 4.8 of Chapter 4.3.3. Figure 4.10 shows how the test RMSE decreases as a function of the number of updates. As in Figure 4.6, the speed of convergence is faster with larger number of machines as the updated information is more frequently exchanged. Figure 4.11 shows the number of updates performed per second in each computation core of each machine; NOMAD exhibits linear scaling on Netflix and Hugewiki, but slows down on Yahoo! Music due to extreme sparsity of

[5]`http://epamcloud.blogspot.com/2013/03/testing-amazon-ec2-network-speed.html`

[6]Since network communication is not computation-intensive, for DSGD++ we used four computation threads instead of two and got better results; thus we report results with four computation threads for DSGD++.

the data. Figure 4.12 compares the convergence speed of different settings when the same amount of computational power is given to each; on every dataset we observe linear to super-linear scaling up to 32 machines.

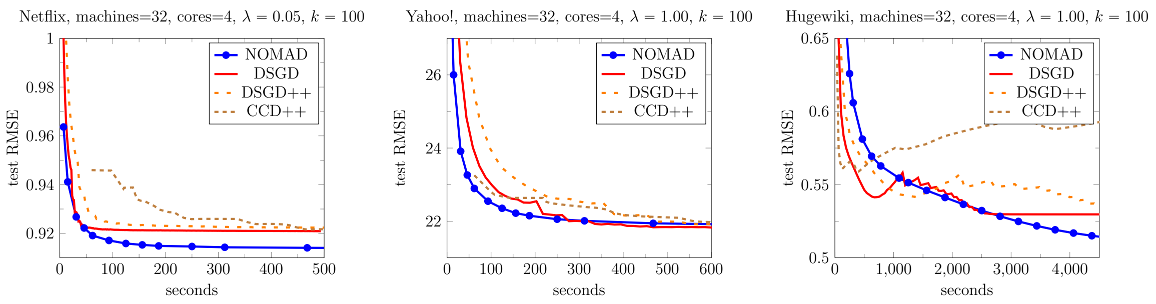


Figure 4.9.: Comparison of NOMAD, DSGD, DSGD++, and CCD++ on a commodity hardware cluster.

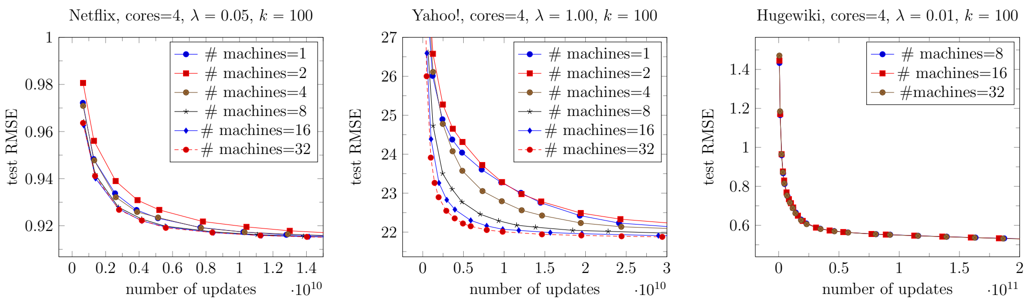


Figure 4.10.: Test RMSE of NOMAD as a function of the number of updates on a commodity hardware cluster, when the number of machines is varied.

4.3.5 Scaling as both Dataset Size and Number of Machines Grows

In previous sections (Chapter 4.3.3 and Chapter 4.3.4), we studied the scalability of algorithms by partitioning a fixed amount of data into increasing number of machines. In real-world applications of collaborative filtering, however, the size of the data should grow over time as new users are added to the system. Therefore, to match the increased amount of data with equivalent amount of physical memory and

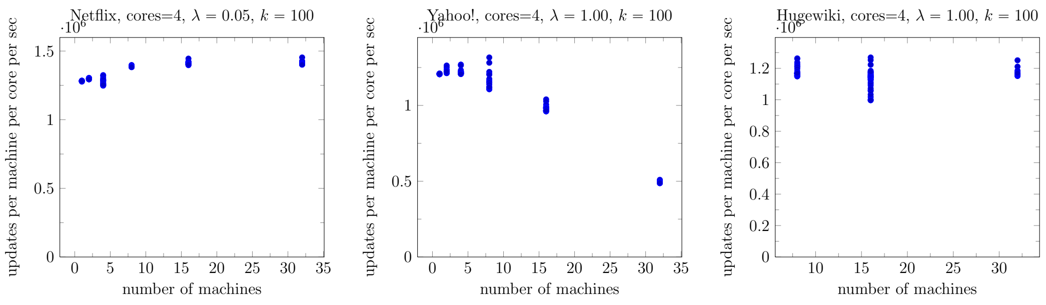


Figure 4.11.: Number of updates of NOMAD per machine per core per second as a function of the number of machines, on a commodity hardware cluster.

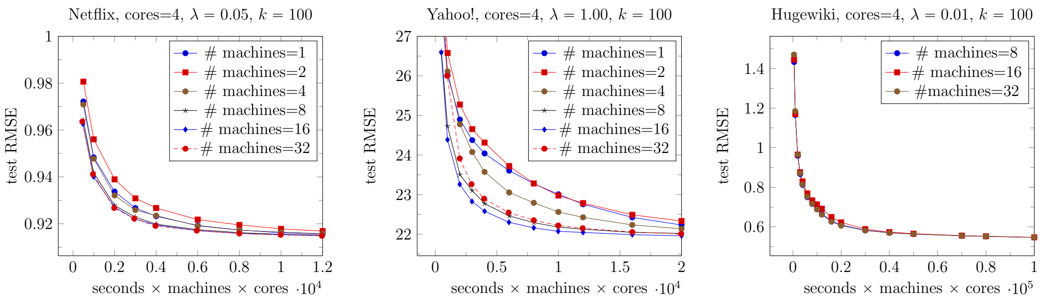


Figure 4.12.: Test RMSE of NOMAD as a function of computation time (time in seconds × the number of machines × the number of cores per each machine) on a commodity hardware cluster, when the number of machines is varied.

computational power, the number of machines should increase as well. The aim of this section is to compare the scaling behavior of NOMAD and that of other algorithms in this realistic scenario.

To simulate such a situation, we generated synthetic datasets which resemble characteristics of real data; the number of ratings for each user and each item is sampled from the corresponding empirical distribution of the Netflix data. As we increase the number of machines from 4 to 32, we fixed the number of items to be the same to that of Netflix (17,770), and increased the number of users to be proportional to the number of machines (480,189 $\times$ the number of machines[7]). Therefore, the expected number of ratings in each dataset is proportional to the number of machines (99,072,112 $\times$ the number of machines) as well.

Conditioned on the number of ratings for each user and item, the nonzero locations are sampled uniformly at random. Ground-truth user parameters $\mathbf{w}_i$'s and item parameters $\mathbf{h}_j$'s are generated from 100-dimensional standard isometric Gaussian distribution, and for each rating A_{ij}, Gaussian noise with mean zero and standard deviation 0.1 is added to the "true" rating $\langle \mathbf{w}_i, \mathbf{h}_j \rangle$.

Figure 4.13 shows that the comparative advantage of NOMAD against DSGD and CCD++ increases as we grow the scale of the problem. NOMAD clearly outperforms DSGD on all configurations; DSGD is very competitive on the small scale, but as the size of the problem grows NOMAD shows better scaling behavior.

4.3.6 Conclusion

From our experimental study we conclude that

- On a single machine, NOMAD shows near-linear scaling up to 30 threads.
- When a fixed size dataset is distributed across multiple machines, NOMAD shows near-linear scaling up to 32 machines.

[7] 480,189 is the number of users in Netflix who have at least one rating.

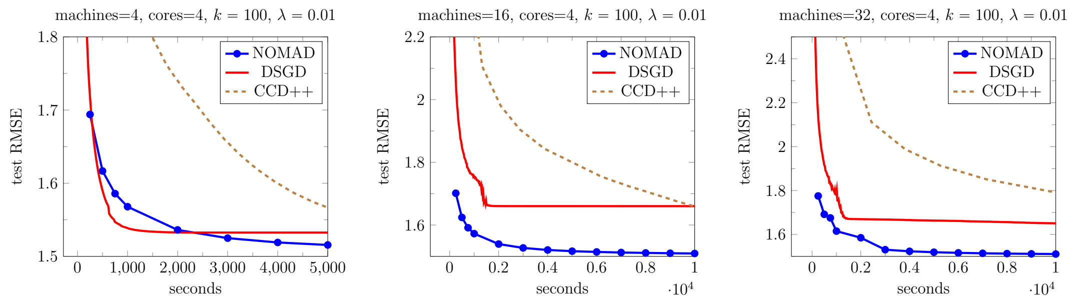


Figure 4.13.: Comparison of algorithms when both dataset size and the number of machines grows. Left: 4 machines, middle: 16 machines, right: 32 machines

- Both in shared-memory and distributed-memory setting, NOMAD exhibits superior performance against state-of-the-art competitors; in commodity hardware cluster, the comparative advantage is more conspicuous.

- When both the size of the data as well as the number of machines grow, the scaling behavior of NOMAD is much nicer than its competitors.

5. REGULARIZED RISK MINIMIZATION

5.1 Introduction

Numerous methods in statistics and machine learning minimize a regularized risk [70]:

$$P(\mathbf{w}) = \lambda \Omega(\mathbf{w}) + \frac{1}{m} \sum_{i=1}^{m} \ell(\mathbf{w}, \mathbf{x}_i, y_i). \tag{5.1}$$

Here, $\mathbf{w}$ is the parameter of the model, $\ell(\cdot, \cdot, \cdot)$ is a loss function, which is convex in $\mathbf{w}$, while $\Omega(\cdot)$ is a regularizer which penalizes complex models, and $\lambda > 0$ trade-offs between the average loss and the regularizer. The average loss is sometimes also called the empirical risk. Note that the loss is evaluated and averaged over m training data points $\mathbf{x}_i$ and their corresponding labels y_i. While more general models also fall under the regularized risk minimization umbrella [70], for the ease of exposition in this paper we will restrict ourselves to the following assumptions:

- the data $\mathbf{x}_i$ and the model parameter $\mathbf{w}$ lie in a d dimensional Euclidean space, that is, $\mathbf{x}_i, \mathbf{w} \in \mathbb{R}^d$
- the loss $\ell(\mathbf{w}, \mathbf{x}_i, y_i)$ can be written as $\ell_i(\langle \mathbf{w}, \mathbf{x}_i \rangle)$, where $\langle \mathbf{w}, \mathbf{x} \rangle$ denotes the Euclidean dot product
- the regularizer decomposes, that is, $\Omega(w)$ can be written as $\sum_j \phi_j(w_j)$ for some $\phi : \mathbb{R} \to \mathbb{R}$. Here w_j denotes the j-th coordinate of $\mathbf{w}$.

This yields the objective function

$$\min_{\mathbf{w}} P(\mathbf{w}) = \lambda \sum_{j=1}^{d} \phi_j(w_j) + \frac{1}{m} \sum_{i=1}^{m} \ell_i(\langle \mathbf{w}, \mathbf{x}_i \rangle). \tag{5.2}$$

A number of well known algorithms can be derived by specializing (5.2). For instance, if $y_i \in \{\pm 1\}$, then by setting $\phi_j(w_j) = w_j^2$ and letting $\ell_i(\langle \mathbf{w}, \mathbf{x}_i \rangle) =$

$\max(0, 1 - y_i \langle \mathbf{w}, \mathbf{x}_i \rangle)$ recovers binary linear support vector machines (SVMs) [61]. On the other hand, by using the same regularizer but changing the loss function to $\ell_i(\langle \mathbf{w}, \mathbf{x}_i \rangle) = \log(1 + \exp(-y_i \langle \mathbf{w}, \mathbf{x}_i \rangle))$ yields regularized logistic regression [12]. Similarly, setting $\ell_i(\langle \mathbf{w}, \mathbf{x}_i \rangle) = \frac{1}{2}(y_i - \langle \mathbf{w}, \mathbf{x}_i \rangle)^2$ and $\phi_j(w_j) = |w_j|$ leads to LASSO [34]. Also note that the entire class of generalized linear model [25] with separable penalty can be fit into this framework as well.

A number of specialized as well as general purpose algorithms have been proposed for minimizing the regularized risk. For instance, if both the loss and the regularizer are smooth, as is the case with logistic regression, then quasi-Newton algorithms such as L-BFGS [46] have been found to be very successful. On the other hand, for non-smooth regularized risk minimization Teo et al. [70] proposed a bundle method for regularized risk minimization (BMRM). Both L-BFGS and BMRM belong to the broad class of batch minimization algorithms. What this means is that at every iteration these algorithms compute the regularized risk $P(\mathbf{w})$ as well as its gradient

$$\nabla P(\mathbf{w}) = \lambda \sum_{j=1}^{d} \nabla \phi_j(w_j) \cdot \mathbf{e}_j + \frac{1}{m} \sum_{i=1}^{m} \nabla \ell_i(\langle \mathbf{w}, \mathbf{x}_i \rangle) \cdot \mathbf{x}_i, \tag{5.3}$$

where $\mathbf{e}_j$ denotes the j-th standard basis vector which contains a one at the j-th coordinate and zeros everywhere else. Both $P(\mathbf{w})$ as well as the gradient $\nabla P(\mathbf{w})$ take $O(md)$ time to compute, which is computationally expensive when m the number of data points is large. Batch algorithms overcome this hurdle by using the fact that the empirical risk $\frac{1}{m}\sum_{i=1}^{m} \ell_i(\langle \mathbf{w}, \mathbf{x}_i \rangle)$ as well as its gradient $\frac{1}{m}\sum_{i=1}^{m} \nabla \ell_i(\langle \mathbf{w}, \mathbf{x}_i \rangle) \cdot \mathbf{x}_i$ decompose over the data points, and therefore one can distribute the data across machines to compute $P(\mathbf{w})$ and $\nabla P(\mathbf{w})$ in a distributed fashion.

Batch algorithms, unfortunately, are known to be not favorable for machine learning both empirically [75] and theoretically [13, 63, 64], as we have discussed in Chapter 2.3. It is now widely accepted that stochastic algorithms which process one data point at a time are more effective for regularized risk minimization. Stochastic algorithms, however, are in general difficult to parallelize, as we have discussed so far.

Therefore we will reformulate the model as a doubly separable function to apply efficient parallel algorithms we introduced in Chapter 2.3.2 and Chapter 3.

5.2 Reformulating Regularized Risk Minimization

In this section we will reformulate the regularized risk minimization problem into an equivalent saddle-point problem. This is done by *linearizing* the objective function (5.2) in terms of $\mathbf{w}$ as follows: rewrite (5.2) by introducing an auxiliary variable u_i for each data point:

$$\min_{\mathbf{w},\mathbf{u}} \ \lambda \sum_{j=1}^{d} \phi_j(w_j) + \frac{1}{m}\sum_{i=1}^{m} \ell_i(u_i) \tag{5.4a}$$

$$\text{s.t. } \ u_i = \langle \mathbf{w}, \mathbf{x}_i \rangle \quad \forall\, i = 1, \ldots, m. \tag{5.4b}$$

Using Lagrange multipliers α_i to eliminate the constraints, the above objective function can be rewritten:

$$\min_{\mathbf{w},\mathbf{u}} \max_{\boldsymbol{\alpha}} \lambda \sum_{j=1}^{d} \phi_j(w_j) + \frac{1}{m}\sum_{i=1}^{m} \ell_i(u_i) + \frac{1}{m}\sum_{i=1}^{m} \alpha_i(u_i - \langle \mathbf{w}, \mathbf{x}_i \rangle).$$

Here $\mathbf{u}$ denotes a vector whose components are u_i. Likewise, $\boldsymbol{\alpha}$ is a vector whose components are α_i. Since the objective function (5.4) is convex and the constrains are linear, strong duality applies [15]. Thanks to strong duality, we can switch the maximization over $\boldsymbol{\alpha}$ and the minimization over $\mathbf{w}, \mathbf{u}$:

$$\max_{\boldsymbol{\alpha}} \min_{\mathbf{w},\mathbf{u}} \lambda \sum_{j=1}^{d} \phi_j(w_j) + \frac{1}{m}\sum_{i=1}^{m} \ell_i(u_i) + \frac{1}{m}\sum_{i=1}^{m} \alpha_i\left(u_i - \langle \mathbf{w}, \mathbf{x}_i \rangle\right).$$

Grouping terms which depend only on $\mathbf{u}$ yields

$$\max_{\boldsymbol{\alpha}} \min_{\mathbf{w},\mathbf{u}} \lambda \sum_{j=1}^{d} \phi_j(w_j) - \frac{1}{m}\sum_{i=1}^{m} \alpha_i \langle \mathbf{w}, \mathbf{x}_i \rangle + \frac{1}{m}\sum_{i=1}^{m} \alpha_i u_i + \ell_i(u_i).$$

Note that the first two terms in the above equation are independent of $\mathbf{u}$, and $\min_{u_i} \alpha_i u_i + \ell_i(u_i)$ is $-\ell_i^{\star}(-\alpha_i)$ where $\ell_i^{\star}(\cdot)$ is the Fenchel-Legendre conjugate of $\ell_i(\cdot)$

Name	$\ell_i(u)$	$-\ell_i^\star(-\alpha)$
Hinge	$\max(1-y_i u, 0)$	$y_i\alpha$ for $\alpha \in [0, y_i]$
Logistic	$\log(1+\exp(-y_i u))$	$-\{y_i\alpha\log(y_i\alpha)+(1-y_i\alpha)\log(1-y_i\alpha)\}$ for $\alpha \in (0, y_i)$
Square	$(u-y_i)^2/2$	$y_i\alpha - \alpha^2/2$

Table 5.1.: Different loss functions and their dual. $[0, y_i]$ denotes $[0, 1]$ if $y_i = 1$, and $[-1, 0]$ if $y_i = -1$; $(0, y_i)$ is defined similarly.

(see Table 5.1 for some examples) [59]. This yields our final objective function which we will henceforth denote by

$$\max_{\boldsymbol{\alpha}} \min_{\mathbf{w}} f(\mathbf{w}, \boldsymbol{\alpha}) := \lambda \sum_{j=1}^{d} \phi_j(w_j) - \frac{1}{m}\sum_{i=1}^{m} \alpha_i \langle \mathbf{w}, \mathbf{x}_i \rangle - \frac{1}{m}\sum_{i=1}^{m} \ell_i^\star(-\alpha_i).$$

At first glance, the above objective function seems unremarkable, except for the fact that it is a function of both the primal parameters $\mathbf{w}$ as well as the Lagrange multipliers $\boldsymbol{\alpha}$. However, it can be rewritten in the following form to reveal a very useful and interesting structure.

Let x_{ij} denote the j-th coordinate of $\mathbf{x}_i$, and $\Omega_i := \{j : x_{ij} \neq 0\}$ denote the non-zero coordinates of $\mathbf{x}_i$. Similarly, let $\bar{\Omega}_j := \{i : x_{ij} \neq 0\}$ denote the set of data points where the j-th coordinate is non-zero and $\Omega := \{(i,j) : x_{ij} \neq 0\}$ denote the set of all non-zero coordinates in the training dataset $\mathbf{x}_1, \ldots, \mathbf{x}_m$. Then, $f(\mathbf{w}, \boldsymbol{\alpha})$ can be rewritten as

$$f(\mathbf{w}, \boldsymbol{\alpha}) = \sum_{(i,j)\in\Omega} \frac{\lambda \phi_j(w_j)}{|\bar{\Omega}_j|} - \frac{\ell_i^\star(-\alpha_i)}{m\,|\Omega_i|} - \frac{\alpha_i w_j x_{ij}}{m}, \tag{5.5}$$

where $|\cdot|$ denotes the cardinality of a set. Remarkably, each component in the summation depends only one term of $\mathbf{w}$ and $\boldsymbol{\alpha}$: the function is in doubly separable form (2.2).

If we take the gradient of $f(\mathbf{w}, \boldsymbol{\alpha})$ in terms of $\mathbf{w}$ and set it to zero to eliminate $\mathbf{w}$, then we obtain so-called dual objective which is a function of $\boldsymbol{\alpha}$. Moreover, any $\mathbf{w}^*$ which is a solution of the primal problem (5.4), and any $\boldsymbol{\alpha}^*$ which is a solution of the dual problem is a saddle-point of $f(\mathbf{w}, \boldsymbol{\alpha})$ [15]. In other words, minimizing the primal, maximizing the dual, or finding a saddle-point of $f(\mathbf{w}, \boldsymbol{\alpha})$ are equivalent problems. The saddle-point of a doubly separable function can be numerically found by methods we introduced in Chapter 2.3 and Chapter 3.

5.3 Implementation Details

Our code is implemented in portable `C++` and uses Intel Thread Building Blocks [3] for multi-threading and the MPICH2 library which provides an implementation of the Message Passing Interface, `MPI`, for inter-machine communication. The parameters are initialized to 0. To prevent degeneracy in logistic regression, the value of α_j is projected to lie in the range $(\epsilon, 1-\epsilon)$ with $\epsilon = 10^{-6}$, while in the case of linear SVM it is naturally projected to its parameter space $[0, 1]$. Similarly, the w_i are restricted to lie in the interval $[-1/\lambda, 1/\lambda]$ for linear SVM and $[-\log(2)/\lambda, \log(2)/\lambda]$ for logistic regression.

As for step size tuning, we adapt the *bold driver heuristic* suggested by Gemulla et al. [31]. The main difference is that we are solving a saddle point problem and hence require reduction in the primal objective function and increase in the dual objective function for algorithm to make progress. Also, to speed up the convergence it will be beneficial to have different step sizes for $\mathbf{w}$ and α. Therefore, our criterion is as follows: if the primal objective function value has decreased from that of the last iteration, we increase the step size of $\mathbf{w}$ by multiplying it with 1.05. On the other hand, if the primal value has increased and the dual gap has widened, we drastically decrease the step size for $\mathbf{w}$ by multiplying it with 0.5. If the dual gap has decreased, however, we do not decrease the step size, since at least some improvement has been made in the previous iteration. The step size for $\boldsymbol{\alpha}$ is adjusted in the same way, but we monitor dual objective value instead of primal.

In the case of the parallel version of our algorithm, we partition the data and run dual coordinate descent [24] on each partition independently to initialize the parameters. To treat nomadic variables in a thread-safe way we used the concurrent queue provided by `TBB`. Each machine posseses two threads, sender and receiver, which are dedicated to inter-machine commutation of nomadic variables. Sender thread, as the name implies, keeps sending pairs of (j, w_j) to other machines. The receiver thread is tasked with receiving the pairs. Whenever a pair is received, it is circulated among all of threads before sending being sent to the sender thread for communication over the network.

5.4 Existing Parallel SGD Algorithms for RERM

Effective parallelization of SGD for RERM is an open problem, which has received significant research attention in recent years. As we mentioned above, the key difficulties in parallelizing SGD update are that 1) stochastic gradient calculation requires us to *read* each coordinate of $\mathbf{w}$, and 2) updates can *write* to every coordinate of $\mathbf{w}$. Due to 2), updates have to be executed in serial, leaving little room for parallelization. Existing work has focused on working around the limitation of stochastic optimization by either a) introducing strategies for computing the stochastic gradient in parallel, b) updating the parameter in parallel, or c) performing independent updates and combining the resulting parameter vectors. While the former two are popular in the shared memory setting, the latter is popular in the distributed memory setting. We will now briefly review these schemes.

An algorithm which uses strategy (a) was proposed by Langford et al. [43]. Their algorithm uses multiple slave threads which work with slightly outdated parameter vectors to compute gradients. These stale gradients are then used by a master thread to update the parameter vector. Langford et al. [43] show that in spite of using stale gradients, the algorithm converges. However, the master needs to write-lock and the slaves have to read-lock the parameter vector during access, which causes bottlenecks.

An algorithm which uses strategy (b) was proposed by [56]. Their algorithm, Hogwild!, allows multiple threads to update the vector simultaneously. This results in a lock-free parallel update of the parameter vector, that is, different threads can read and write the parameter vector without locks. However, the downside of this approach is that synchronizing the L1 caches of various threads causes considerable slowdown in throughput. It is unclear how to extend the above two algorithms to the distributed memory setting.

In the distributed memory setting, a number of proposals exist. Some of the earliest work that we are aware of includes algorithms by Bertsekas and Tsitsiklis [10] and their recent variants such as the algorithm of Ram et al. [55]. The basic idea here is that data is distributed across multiple processors, each of which works with their own version of the parameter vector. After a fixed number of updates, individual machines communicate their parameter vector to their neighbors. Each machine averages the parameter vectors received from its neighbors, and the iteration proceeds. These algorithms require frequent communication and synchronization. On the other extreme, Zinkevich et al. [80] propose to run stochastic optimization on a subset of data in each individual processor and finally average the results. There is no communication, but the empirical performance of such a method is usually inferior (see section 5.5).

Another class of algorithms use a so-called parameter server to synchronize local updates to the parameters Smola and Narayanamurthy [65]. In a nutshell, the idea here is that the updates to the parameters are continuously and asynchronously communicated to a processor which is designated as a parameter server, which in turn accumulates the updates and periodically transmits them to other machines. The main drawback of such a scheme is that it is not easy to "serialize" the updates, that is, to replay the updates on a single machine. This makes these algorithms slow to converge, and difficult to debug [49].

5.5 Empirical Evaluation

We conduct three preliminary experiments to test the efficacy of solving the saddle-point problem, and to compare it with state-of-the-art methods. The first experiment is to show that saddle point formulation is versatile, and can be applied to a variety of problems including linear SVM and logistic regression. In the second experiment we will study the behavior of the algorithms on medium sized datasets on a single machine. Finally, we study convergence in large datasets in a multi-machine setup. We will mainly report test error vs iterations in the main body of the paper.

5.5.1 Experimental Setup

We work with the following publicly available datasets: `real-sim`, `news20`, `worm`, `kdda`, `kddb`, `alpha`, `ocr`, and `dna` (see Table 5.2 for summary statistics). `news20` and `real-sim` are from Hsieh et al. [38], `worm` is from Franc and Sonnenburg [27], while `kdda` and `kddb` are from the KDD cup 2010 challenge[1]. The `alpha`, `dna`, `ocr` datasets are from the Pascal Large Scale Learning Workshop [66]. Wherever, training and test splits were available, we used them. Otherwise we randomly split the data and used 80% for training and 20% for testing. The same training test split is used for all algorithms in every experiment. We selected these datasets because they span a wide spectrum of values in terms of number of data points, number of features, sparsity, and class balance. They also represent data from a variety of application domains.

For simplicity we set the regularization parameter $\lambda = 10^{-3}$. Our goal is to compare optimizers. Therefore, tuning λ to obtain the best test performance is not our focus[2]. Moreover, note that since we are dealing with a convex regularized risk, all optimization algorithms will converge to the same solution. For a fair comparison, wherever possible, the algorithms are initialized with the same initial parameter values.

[1]`http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html`

[2]In fact, a large value of λ is favorable to the batch optimization algorithms.

Table 5.2.: Summary of the datasets used in our experiments. m is the total # of examples, d is the # of features, s is the feature density (% of features that are non-zero), $m_+ : m_-$ is the ratio of the number of positive vs negative examples, Datasize is the size of the data file on disk. M/G denotes a million/billion.

dataset	m	d	$\|\Omega\|$	$s(\%)$	$m_+ : m_-$	Datasize
`ocr`	2.8 M	1156	3.24G	100	0.96	43.18 GB
`dna`	40 M	800	8.00G	25.0	3e−3	63.04 GB
`kdda`	8.41M	20.22M	0.31G	1.82e-4	6.56	2.55 GB
`kddb`	19.26M	29.89M	0.59G	1.02e-4	7.91	4.90 GB
`worm`	0.82M	804	0.17G	25.12	0.06	0.93 GB
`alpha`	0.4M	500	0.20G	100	0.99	2.74 GB
`news20`	15960	1.36 M	7.26M	0.033	1.00	0.11 GB
`real-sim`	57763	20958	2.97M	0.245	0.44	0.07 GB

5.5.2 Parameter Tuning

For SSO and DSSO we used the bold-driver heuristic discussed in section 5.3. This requires tuning three parameters: $\eta_0^{\mathbf{w}}$ the initial step size for $\mathbf{w}$, $\eta_0^{\boldsymbol{\alpha}}$ the initial step size for $\boldsymbol{\alpha}$, and the period for updating the step size. We expect that as we gain more experience with saddle point optimization, an automatic mechanism for selecting these parameters will be developed. However, for the preliminary results reported in this paper, we do a semi-systematic search for the optimal values by letting $\eta_0^{\mathbf{w}} \in \{10^{-1}, 10^{-2}, \ldots, 10^{-6}\}$, $\eta_0^{\boldsymbol{\alpha}} \in \{10^{-1}, 10^{-2}, \ldots, 10^{-6}\}$, and period $\in \{1, 2, 5, 10\}$.

5.5.3 Competing Algorithms

For stochastic gradient descent, we used `sgd-2.1`, which is available for download from `http://leon.bottou.org/projects/sgd`. It is widely acknowledged as one of the fastest and most robust implementations for the single machine case. The stepsize schedule used is

$$\eta_t = \frac{\eta_0}{1 + \lambda \eta_0 t},$$

and initial stepsize η_0 is automatically tuned by using a small subset of the data.

For batch optimization we use BMRM, which is a first-order bundle method based solver which is specialized for smooth as well as non-smooth regularized risk minimization [70]. We also use L-BFGS, which is one of the fastest general purpose limited memory Quasi-Newton algorithms. For both algorithms, we implemented our own loss functions in C++ and used them with PETSc[3] and TAO[4] libraries, which provide a framework for efficient large scale linear algebra and optimization. In particular, we used the Limited Memory Variable Metric (lmvm) Quasi Newton optimization algorithm, as well as the BMRM solver from TAO. The code was compiled without debugging symbols for optimal performance, and the default stopping criterion was used for both solvers.

We also compare against the dual coordinate descent (DCD) algorithm, which performs very competitively in single machine experiments, and is the basis for the popular LibLinear library [24]. One way to view DCD is that it is SGD with automatic step size tuning (see e.g., Vishwanathan and Cheng [74]). We implemented our own version of DCD.

Finally for the distributed memory experiments we will compare DSSO and NOMAD against the parallel stochastic gradient descent solver of Zinkevich et al. [80]. This will be called PSGD in the sequel. We implemented a version of PSGD in our framework using C++ and MPI. Our implementation partitions the data across processors, and independently runs SGD on each partition independently. After every iteration of SGD, the parameters of the different processors are averaged during a bulk synchronization step, and the iteration proceeds.

[3]Version 3.4.3 from `http://www.mcs.anl.gov/petsc/`

[4]Version 2.2 from `https://bitbucket.org/sarich/tao-2.2`

5.5.4 Versatility

Our first experiment is designed to show the versatility of using the saddle-point formulation. By plugging in different loss functions, one recovers different well known models. As long as the loss is convex, and the regularizer is separable, our saddle-point formulation is valid. Here, we will restrict ourselves to a subset of the datasets (`real-sim` and `news20`) and work with the square norm-regularizer and two different loss functions: the hinge loss which leads to linear SVM, and the logistic loss which leads to regularized logistic regression. All experiments are conducted using a single machine. Note that L-BFGS can only handle smooth objective functions, it is excluded from the linear SVM experiments because the hinge loss is non-smooth. Since the computational complexity of all the algorithms is $O(md)$ for performing one iteration through the data, our main goal here is to show how the test error behaves as function of the number of iterations. A secondary goal is to show how the primal objective function (5.2) changes as a function of the number of iterations (see supplementary material). Results for `real-sim` can be found in Figure 5.1, while the results for `news20` can be found in Figures 5.2. In the case of `real-sim` for the hinge loss, SSO converges to a better solution must faster than SGD. The same story is repeated for logistic regression. However, in the case of `news20` SGD is able to converge to a marginally better solution. As expected, for small datasets and large values of λ, both the batch solvers exhibit competitive performance. DCD, which can be viewed as SGD with automatic step size tuning performs the best in all cases. We are investigating if similar step size tuning mechanisms can be adapted for SSO. In particular, note that the jumpy behavior of the test error for SSO, which happens because we use the bold driver heuristic to tune step sizes.

5.5.5 Single Machine Experiments

From now on we will only concentrate on logistic regression. In this section, we are interested in checking the convergence speed of SSO vs other algorithms on medium

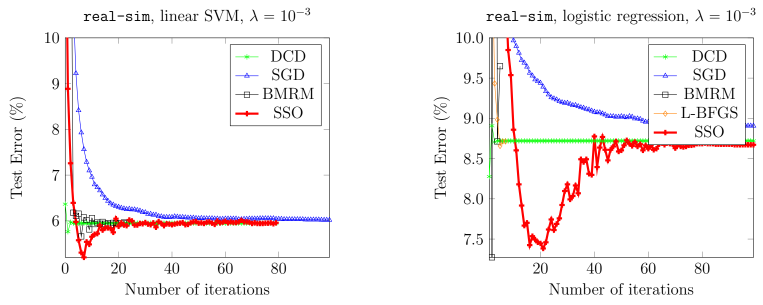


Figure 5.1.: Test error vs iterations for `real-sim` on linear SVM and logistic regression.

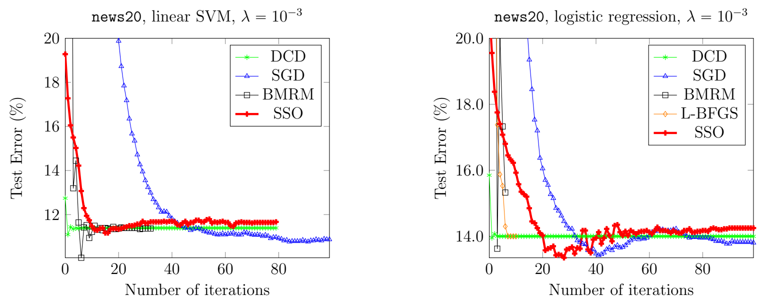
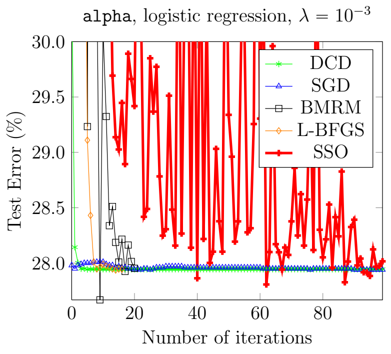


Figure 5.2.: Test error vs iterations for `news20` on linear SVM and logistic regression.



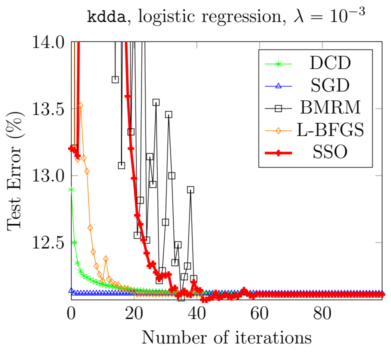
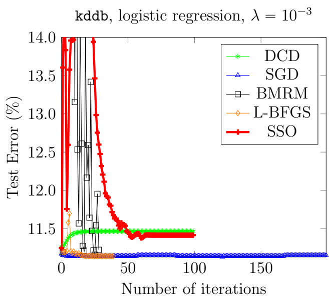


Figure 5.3.: Test error vs iterations for `alpha` and `kdda`.



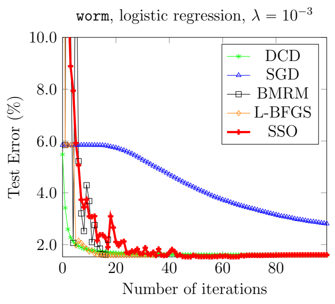


Figure 5.4.: Test error vs iterations for `kddb` and `worm`.

sized datasets. Therefore we will concentrate on the following 4 datasets: `kdda`, `kddb`, `alpha`, `worm`.

On `alpha` the test error oscillates (Figure 5.3), but eventually converges to the optimal value. SGD is faster than SSO on this dataset. On both `kdda` and `kddb` datasets, convergence of SSO is slower than that of SGD. The reason for this is because both these datasets are very sparse and have a very large number of features. Therefore, the number of parameters of SSGD is much higher than that of SGD and other competing algorithms. This makes step size tuning for SSO significantly challenging. On the other hand for the `worm` dataset, where the number of features is small, SSO outperforms SGD and is comparable to BMRM.

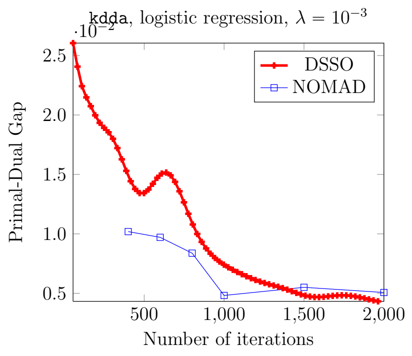


Figure 5.5.: Comparison between synchronous and asynchronous algorithm on `ocr` dataset.

5.5.6 Multi-Machine Experiments

In this section, we are interested in checking the convergence speed of SSO vs other algorithms on medium to large sized datasets but in a distributed memory setting. Therefore we will concentrate on the following 4 datasets: `kdda`, `kddb`, `ocr`, and `dna`. The experiments are run on 8 machines, with 4 cores per machine (total of 32 processors). We also plot test error vs wall-clock time for the parallel experiments. On the `kdda` dataset, even though SSO is competitive in terms of test error vs number of iterations, each update of SSO is slower than that of BMRM and L-BFGS which use numerical linear algebra libraries. On `kddb` and `ocr` we see oscillating behavior which occurs because of the bold driver heuristic. Better step size tuning for SSO is likely to yield competitive results. On `dna`, which is a very heavily imbalanced dataset, all methods converge rather quickly. It is noteworthy that PSGD does not perform well, and often returns very sub-optimal solutions both in terms of test error and primal objective function value.

We also compare the convergence speed of synchronous algorithm (DSSO, Algorithm 1) and asynchronous algorithm (NOMAD, Algorithm 2). This experiment was also run on 8 machines with 4 cores each; as can be seen in Figure 5.5, NOMAD indeed reduces the primal-dual gap more rapidly than DSSO does.

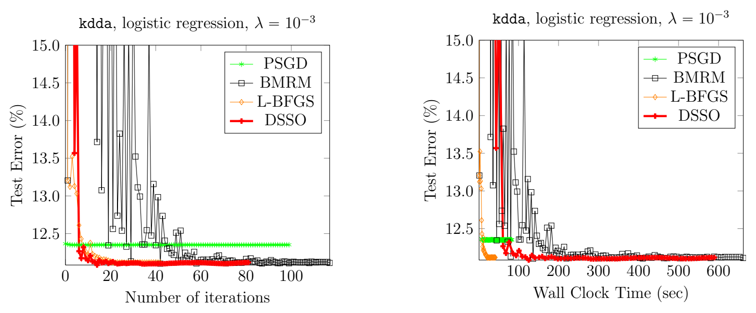


Figure 5.6.: Performances for `kdda` in multi-machine senario.

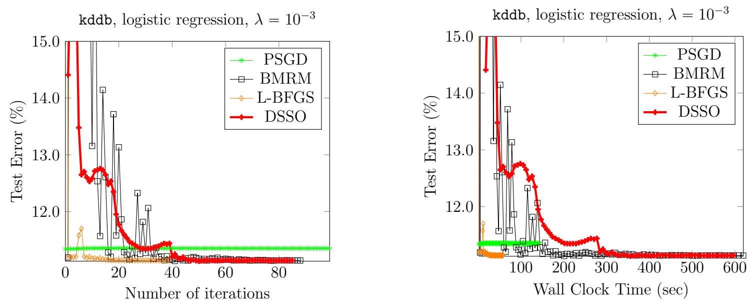


Figure 5.7.: Performances for `kddb` in multi-machine senario.

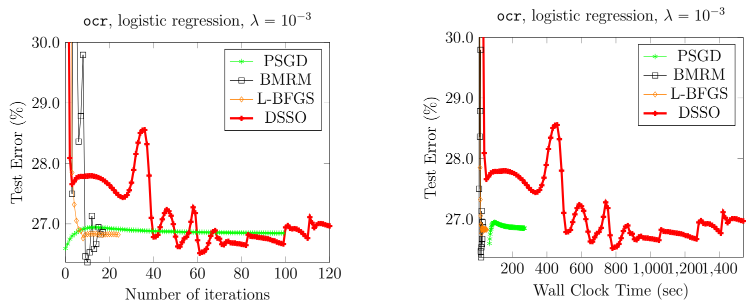


Figure 5.8.: Performances for `ocr` in multi-machine senario.

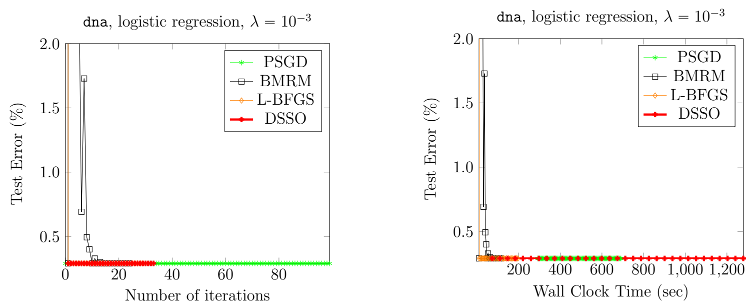


Figure 5.9.: Performances for `dna` in multi-machine senario.

5.6 Discussion and Conclusion

We presented a new equivalent formulation of regularized risk minimization, which has a doubly separable structure and can be exploited to derive an efficient parallel algorithm. Our experimental results are arguably preliminary, and clearly there is a lot of scope for improvement. For instance, using a better step size tuning mechanism will clearly accelerate convergence of our algorithm. We are currently exploring Periodic Step Size adaptation [39] and Stochastic Meta Descent [62]. Also, we believe that using lower values of λ in our experiments would have better brought out the differences between the algorithms; batch algorithms typically converge very fast for strongly convex functions whose eigenvalues are bounded away from zero. Finally, some of the performance difference between the algorithms arises because stochastic optimization algorithms suffer from poor locality of memory access. Nevertheless, discovering the doubly separable nature of the regularized risk gives rise to a naturally parallelizable algorithm, thus settling an open problem in the machine learning community.

6. OTHER EXAMPLES OF DOUBLE SEPARABILITY

For a given statistical model, there can be more than one way to achieve double separability. In this chapter, we briefly introduce multinomial logistic regression and item response theory model, which also belong to regularized risk minimization but allow additional doubly separable formulations different from (5.5).

6.1 Multinomial Logistic Regression

Multinomial logistic regression [33] is a generalization of binary logistic regression which the response variable can have n possible values. The data consists of $(\mathbf{x}_1, y_1)$,$(\mathbf{x}_2, y_2)$,. . .,$(\mathbf{x}_m, y_m)$ where each $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{1, 2, \ldots, n\}$ for $i = 1, 2, \ldots, m$. The model is then parametrized by n number of d-dimensional vectors $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_K \in \mathbb{R}^d$, with the negative log-likelihood being:

$$J(\mathbf{w}_1, \ldots, \mathbf{w}_K) := \sum_{i=1}^{m} -\langle \mathbf{w}_{y_i}, x_i \rangle + \log \left(\sum_{k=1}^{K} \exp\left(\langle \mathbf{w}_i, x_i \rangle\right) \right). \tag{6.1}$$

Gopal and Yang [33] observes that the following property of the logarithm function can be useful in this situation:

$$\log(\gamma) \leqslant a\gamma - \log(a) - 1, \tag{6.2}$$

for any $\gamma > 0$ and $a > 0$. The bound is tight when $a = 1/\gamma$. By introducing auxiliary variables $a_1, a_2, \ldots, a_m$, the objective function can be rewritten as

$$J(a_1, \ldots, a_m, \mathbf{w}_1, \ldots, \mathbf{w}_K) :=$$
$$\sum_{i=1}^{m} \sum_{k=1}^{K} \left(-I\left(y_i = k\right) \cdot \langle \mathbf{w}_k, x_i \rangle + a_i \cdot \exp\left(\langle \mathbf{w}_i, x_i \rangle\right) - \frac{1}{K} \log(a_i) - \frac{1}{K} \right). \tag{6.3}$$

The objective function is now doubly separable. Note that the optimal $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_d$ of (6.3) corresponds to that of (6.1), since the bound (6.2) is tight for optimal a_i values.

6.2 Item Response Theory

Item response theory (IRT) model [21] is an important statistical model in psychometrics to analyze a latent trait of entity measured by an instrument. One of the most popular application is the scoring of tests such as Graduate Record Examination (GRE) or Graduate Management Admission Test (GMAT). Denote latent traits of entities we aim to estimate as $\theta_1, \theta_2, \ldots, \theta_m$, and suppose they are examined by n dichotomous instruments; $y_{ij} = 0, 1$ denotes the measurement of i-th entity by j-th instrument. Then, the negative log-likelihood of IRT model is[1]:

$$J(\theta_1, \theta_2, \ldots, \theta_m, b_1, \ldots, b_n) := \sum_{i=1}^{m} \sum_{j=1}^{n} -y_{ij} \cdot (\theta_i - b_j) + \log\left(1 + \exp\left(\theta_i - b_j\right)\right). \quad (6.4)$$

One can see that the model is readily in doubly separable form.

[1]For brevity of exposition, here we have only introduced the 1PL (1 Parameter Logistic) IRT model, but in fact 2PL and 3PL models are also doubly separable.

7. LATENT COLLABORATIVE RETRIEVAL

7.1 Introduction

Learning to rank is a problem of ordering a set of items according to their relevances to a given context [16]. In document retrieval, for example, a query is given to a machine learning algorithm, and it is asked to sort the list of documents in the database for the given query. While a number of approaches have been proposed to solve this problem in the literature, in this paper we provide a new perspective by showing a close connection between ranking and a seemingly unrelated topic in machine learning, namely, robust binary classification.

In robust classification [40], we are asked to learn a classifier in the presence of outliers. Standard models for classificaion such as Support Vector Machines (SVMs) and logistic regression do not perform well in this setting, since the convexity of their loss functions does not let them give up their performance on any of the data points [48]; for a classification model to be robust to outliers, it has to be capable of sacrificing its performance on some of the data points.

We observe that this requirement is very similar to what standard metrics for ranking try to evaluate. Normalized Discounted Cumulative Gain (NDCG) [50], the most popular metric for learning to rank, strongly emphasizes the performance of a ranking algorithm at the top of the list; therefore, a good ranking algorithm in terms of this metric has to be able to give up its performance at the bottom of the list if that can improve its performance at the top.

In fact, we will show that NDCG can indeed be written as a natural generalization of robust loss functions for binary classification. Based on this observation we formulate RoBiRank, a novel model for ranking, which maximizes the lower bound of NDCG. Although the non-convexity seems unavoidable for the bound to be tight

[17], our bound is based on the class of robust loss functions that are found to be empirically easier to optimize [22]. Indeed, our experimental results suggest that RoBiRank reliably converges to a solution that is competitive as compared to other representative algorithms even though its objective function is non-convex.

While standard deterministic optimization algorithms such as L-BFGS [53] can be used to estimate parameters of RoBiRank, to apply the model to large-scale datasets a more efficient parameter estimation algorithm is necessary. This is of particular interest in the context of latent collaborative retrieval [76]; unlike standard ranking task, here the number of items to rank is very large and explicit feature vectors and scores are not given.

Therefore, we develop an efficient parallel stochastic optimization algorithm for this problem. It has two very attractive characteristics: First, the time complexity of each stochastic update is independent of the size of the dataset. Also, when the algorithm is distributed across multiple number of machines, no interaction between machines is required during most part of the execution; therefore, the algorithm enjoys near linear scaling. This is a significant advantage over serial algorithms, since it is very easy to deploy a large number of machines nowadays thanks to the popularity of cloud computing services, e.g. Amazon Web Services.

We apply our algorithm to latent collaborative retrieval task on Million Song Dataset [9] which consists of 1,129,318 users, 386,133 songs, and 49,824,519 records; for this task, a ranking algorithm has to optimize an objective function that consists of $386,133 \times 49,824,519$ number of pairwise interactions. With the same amount of wall-clock time given to each algorithm, RoBiRank leverages parallel computing to outperform the state-of-the-art with a 100% lift on the evaluation metric.

7.2 Robust Binary Classification

We view ranking as an extension of robust binary classification, and will adopt strategies for designing loss functions and optimization techniques from it. Therefore, we start by reviewing some relevant concepts and techniques.

Suppose we are given training data which consists of n data points (x_1, y_1), (x_2, y_2), …, (x_n, y_n), where each $x_i \in \mathbb{R}^d$ is a d-dimensional feature vector and $y_i \in \{-1, +1\}$ is a label associated with it. A linear model attempts to learn a d-dimensional parameter ω, and for a given feature vector x it predicts label $+1$ if $\langle x, \omega \rangle \geqslant 0$ and -1 otherwise. Here $\langle \cdot, \cdot \rangle$ denotes the Euclidean dot product between two vectors. The quality of ω can be measured by the number of mistakes it makes:

$$L(\omega) := \sum_{i=1}^{n} I(y_i \cdot \langle x_i, \omega \rangle < 0). \tag{7.1}$$

The indicator function $I(\cdot < 0)$ is called the 0-1 loss function, because it has a value of 1 if the decision rule makes a mistake, and 0 otherwise. Unfortunately, since (7.1) is a discrete function its minimization is difficult; in general, it is an NP-Hard problem [26]. The most popular solution to this problem in machine learning is to upper bound the 0-1 loss by an easy to optimize function [6]. For example, logistic regression uses the logistic loss function $\sigma_0(t) := \log_2(1 + 2^{-t})$, to come up with a continuous and convex objective function

$$\overline{L}(\omega) := \sum_{i=1}^{n} \sigma_0(y_i \cdot \langle x_i, \omega \rangle), \tag{7.2}$$

which upper bounds $L(\omega)$. It is easy to see that for each i, $\sigma_0(y_i \cdot \langle x_i, \omega \rangle)$ is a convex function in ω; therefore, $\overline{L}(\omega)$, a sum of convex functions, is a convex function as well and much easier to optimize than $L(\omega)$ in (7.1) [15]. In a similar vein, Support Vector Machines (SVMs), another popular approach in machine learning, replace the 0-1 loss by the hinge loss. Figure 7.1 (top) graphically illustrates three loss functions discussed here.

However, convex upper bounds such as $\overline{L}(\omega)$ are known to be sensitive to outliers [48]. The basic intuition here is that when $y_i \cdot \langle x_i, \omega \rangle$ is a very large negative number

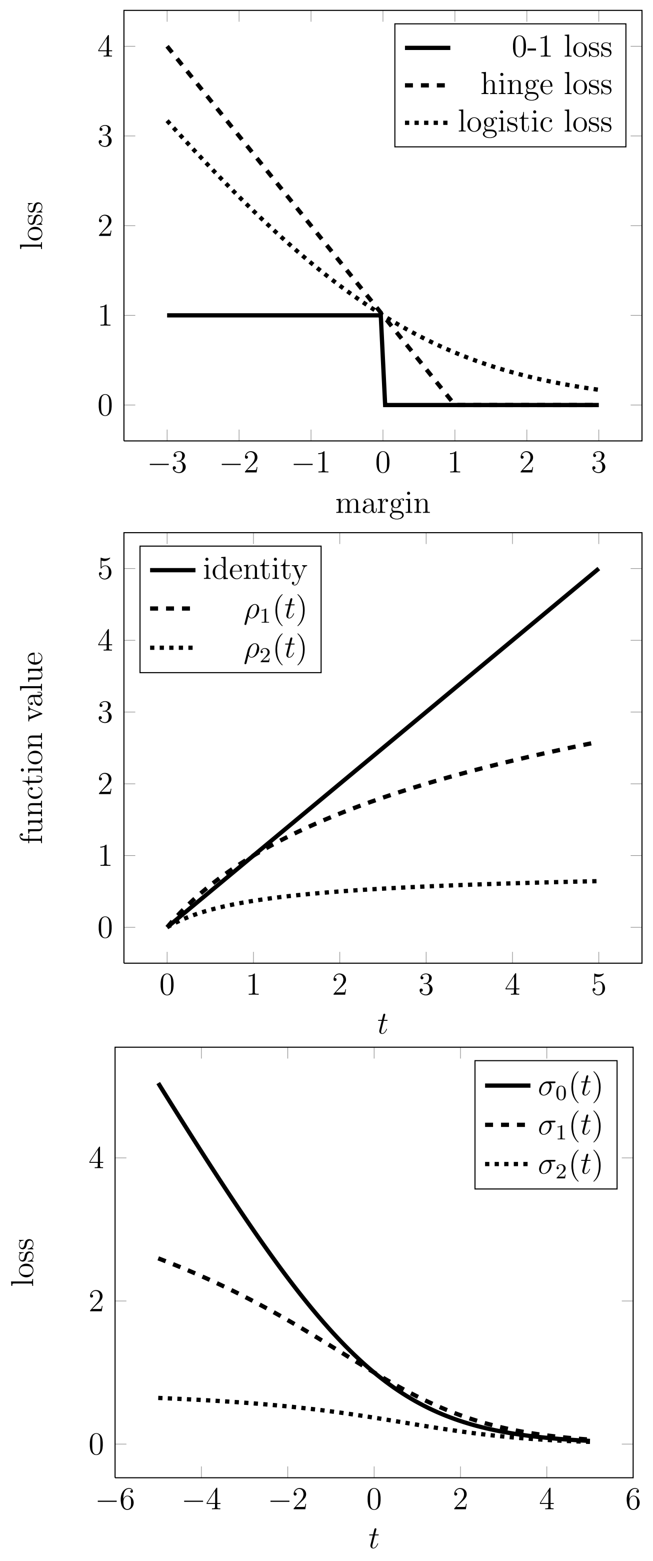


Figure 7.1.: Top: Convex Upper Bounds for 0-1 Loss. Middle: Transformation functions for constructing robust losses. Bottom: Logistic loss and its transformed robust variants.

for some data point i, $\sigma(y_i \cdot \langle x_i, \omega \rangle)$ is also very large, and therefore the optimal solution of (7.2) will try to decrease the loss on such outliers at the expense of its performance on "normal" data points.

In order to construct loss functions that are robust to noise, consider the following two transformation functions:

$$\rho_1(t) := \log_2(t+1), \quad \rho_2(t) := 1 - \frac{1}{\log_2(t+2)}, \tag{7.3}$$

which, in turn, can be used to define the following loss functions:

$$\sigma_1(t) := \rho_1(\sigma_0(t)), \quad \sigma_2(t) := \rho_2(\sigma_0(t)). \tag{7.4}$$

Figure 7.1 (middle) shows these transformation functions graphically, and Figure 7.1 (bottom) contrasts the derived loss functions with logistic loss. One can see that $\sigma_1(t) \to \infty$ as $t \to -\infty$, but at a much slower rate than $\sigma_0(t)$ does; its derivative $\sigma_1'(t) \to 0$ as $t \to -\infty$. Therefore, $\sigma_1(\cdot)$ does not grow as rapidly as $\sigma_0(t)$ on hard-to-classify data points. Such loss functions are called Type-I robust loss functions by Ding [22], who also showed that they enjoy statistical robustness properties. $\sigma_2(t)$ behaves even better: $\sigma_2(t)$ converges to a constant as $t \to -\infty$, and therefore "gives up" on hard to classify data points. Such loss functions are called Type-II loss functions, and they also enjoy statistical robustness properties [22].

In terms of computation, of course, $\sigma_1(\cdot)$ and $\sigma_2(\cdot)$ are not convex, and therefore the objective function based on such loss functions is more difficult to optimize. However, it has been observed in Ding [22] that models based on optimization of Type-I functions are often empirically much more successful than those which optimize Type-II functions. Furthermore, the solutions of Type-I optimization are more stable to the choice of parameter initialization. Intuitively, this is because Type-II functions asymptote to a constant, reducing the gradient to almost zero in a large fraction of the parameter space; therefore, it is difficult for a gradient-based algorithm to determine which direction to pursue. See Ding [22] for more details.

7.3 Ranking Model via Robust Binary Classification

In this section, we will extend robust binary classification to formulate RoBiRank, a novel model for ranking.

7.3.1 Problem Setting

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be a set of contexts, and $\mathcal{Y} = \{y_1, y_2, \ldots, y_m\}$ be a set of items to be ranked. For example, in movie recommender systems $\mathcal{X}$ is the set of users and $\mathcal{Y}$ is the set of movies. In some problem settings, only a subset of $\mathcal{Y}$ is relevant to a given context $x \in \mathcal{X}$; e.g. in document retrieval systems, only a subset of documents is relevant to a query. Therefore, we define $\mathcal{Y}_x \subset \mathcal{Y}$ to be a set of items relevant to context x. Observed data can be described by a set $W := \{W_{xy}\}_{x\in\mathcal{X}, y\in\mathcal{Y}_x}$ where W_{xy} is a real-valued score given to item y in context x.

We adopt a standard problem setting used in the literature of learning to rank. For each context x and an item $y \in \mathcal{Y}_x$, we aim to learn a scoring function $f(x, y) : \mathcal{X} \times \mathcal{Y}_x \to \mathbb{R}$ that induces a ranking on the item set $\mathcal{Y}_x$; the higher the score, the more important the associated item is in the given context. To learn such a function, we first extract joint features of x and y, which will be denoted by $\phi(x, y)$. Then, we parametrize $f(\cdot, \cdot)$ using a parameter ω, which yields the following linear model:

$$f_\omega(x, y) := \langle \phi(x, y), \omega \rangle, \tag{7.5}$$

where, as before, $\langle \cdot, \cdot \rangle$ denotes the Euclidean dot product between two vectors. ω induces a ranking on the set of items $\mathcal{Y}_x$; we define $\mathrm{rank}_\omega(x, y)$ to be the rank of item y in a given context x induced by ω. More precisely,

$$\mathrm{rank}_\omega(x, y) := |\{y' \in \mathcal{Y}_x : y' \neq y, f_\omega(x, y) < f_\omega(x, y')\}|,$$

where $|\cdot|$ denotes the cardinality of a set. Observe that $\mathrm{rank}_\omega(x, y)$ can also be written as a sum of 0-1 loss functions (see e.g. Usunier et al. [72]):

$$\mathrm{rank}_\omega(x, y) = \sum_{y' \in \mathcal{Y}_x, y' \neq y} I\left(f_\omega(x, y) - f_\omega(x, y') < 0\right). \tag{7.6}$$

7.3.2 Basic Model

If an item y is very relevant in context x, a good parameter ω should position y at the top of the list; in other words, $\text{rank}_\omega(x, y)$ has to be small. This motivates the following objective function for ranking:

$$L(\omega) := \sum_{x \in \mathcal{X}} c_x \sum_{y \in \mathcal{Y}_x} v(W_{xy}) \cdot \text{rank}_\omega(x, y), \tag{7.7}$$

where c_x is an weighting factor for each context x, and $v(\cdot) : \mathbb{R}^+ \to \mathbb{R}^+$ quantifies the relevance level of y on x. Note that $\{c_x\}$ and $v(W_{xy})$ can be chosen to reflect the metric the model is going to be evaluated on (this will be discussed in Section 7.3.3). Note that (7.7) can be rewritten using (7.6) as a sum of indicator functions. Following the strategy in Section 7.2, one can form an upper bound of (7.7) by bounding each 0-1 loss function by a logistic loss function:

$$\overline{L}(\omega) := \sum_{x \in \mathcal{X}} c_x \sum_{y \in \mathcal{Y}_x} v\left(W_{xy}\right) \sum_{y' \in \mathcal{Y}_x, y' \neq y} \sigma_0\left(f_\omega(x, y) - f_\omega(x, y')\right). \tag{7.8}$$

Just like (7.2), (7.8) is convex in ω and hence easy to minimize.

Note that (7.8) can be viewed as a weighted version of binary logistic regression (7.2); each (x, y, y') triple which appears in (7.8) can be regarded as a data point in a logistic regression model with $\phi(x, y) - \phi(x, y')$ being its feature vector. The weight given on each data point is $c_x \cdot v(W_{xy})$. This idea underlies many pairwise ranking models.

7.3.3 DCG and NDCG

Although (7.8) enjoys convexity, it may not be a good objective function for ranking. It is because in most applications of learning to rank, it is much more important to do well at the top of the list than at the bottom of the list, as users typically pay attention only to the top few items. Therefore, if possible, it is desirable to *give up* performance on the lower part of the list in order to gain quality at the top. This intuition is similar to that of robust classification in Section 7.2; a stronger connection will be shown in below.

Discounted Cumulative Gain (DCG)[50] is one of the most popular metrics for ranking. For each context $x \in \mathcal{X}$, it is defined as:

$$\mathrm{DCG}_x(\omega) := \sum_{y \in \mathcal{Y}_x} \frac{2^{W_{xy}} - 1}{\log_2 \left(\mathrm{rank}_\omega(x, y) + 2\right)}. \tag{7.9}$$

Since $1/\log(t+2)$ decreases quickly and then asymptotes to a constant as t increases, this metric emphasizes the quality of the ranking at the top of the list. Normalized DCG simply normalizes the metric to bound it between 0 and 1 by calculating the maximum achievable DCG value m_x and dividing by it [50]:

$$\mathrm{NDCG}_x(\omega) := \frac{1}{m_x} \sum_{y \in \mathcal{Y}_x} \frac{2^{W_{xy}} - 1}{\log_2 \left(\mathrm{rank}_\omega(x, y) + 2\right)}. \tag{7.10}$$

These metrics can be written in a general form as:

$$c_x \sum_{y \in \mathcal{Y}_x} \frac{v\left(W_{xy}\right)}{\log_2 \left(\mathrm{rank}_\omega(x, y) + 2\right)}. \tag{7.11}$$

By setting $v(t) = 2^t - 1$ and $c_x = 1$, we recover DCG. With $c_x = 1/m_x$, on the other hand, we get NDCG.

7.3.4 RoBiRank

Now we formulate RoBiRank, which optimizes the lower bound of metrics for ranking in form (7.11). Observe that the following optimization problems are equivalent:

$$\max_\omega \sum_{x \in \mathcal{X}} c_x \sum_{y \in \mathcal{Y}_x} \frac{v\left(W_{xy}\right)}{\log_2 \left(\mathrm{rank}_\omega(x, y) + 2\right)} \Leftrightarrow \tag{7.12}$$

$$\min_\omega \sum_{x \in \mathcal{X}} c_x \sum_{y \in \mathcal{Y}_x} v\left(W_{xy}\right) \cdot \left\{1 - \frac{1}{\log_2 \left(\mathrm{rank}_\omega(x, y) + 2\right)}\right\}. \tag{7.13}$$

Using (7.6) and the definition of the transformation function $\rho_2(\cdot)$ in (7.3), we can rewrite the objective function in (7.13) as:

$$L_2(\omega) := \sum_{x \in \mathcal{X}} c_x \sum_{y \in \mathcal{Y}_x} v\left(W_{xy}\right) \cdot \rho_2 \left(\sum_{y' \in \mathcal{Y}_x, y' \neq y} I\left(f_\omega(x, y) - f_\omega(x, y') < 0\right) \right). \tag{7.14}$$

Since $\rho_2(\cdot)$ is a monotonically increasing function, we can bound (7.14) with a continuous function by bounding each indicator function using logistic loss:

$$\overline{L}_2(\omega) := \sum_{x \in \mathcal{X}} c_x \sum_{y \in \mathcal{Y}_x} v\left(W_{xy}\right) \cdot \rho_2\left(\sum_{y' \in \mathcal{Y}_x, y' \neq y} \sigma_0\left(f_\omega(x,y) - f_\omega(x,y')\right)\right). \tag{7.15}$$

This is reminiscent of the basic model in (7.8); as we applied the transformation function $\rho_2(\cdot)$ on the logistic loss function $\sigma_0(\cdot)$ to construct the robust loss function $\sigma_2(\cdot)$ in (7.4), we are again applying the same transformation on (7.8) to construct a loss function that respects metrics for ranking such as DCG or NDCG (7.11). In fact, (7.15) can be seen as a generalization of robust binary classification by applying the transformation on a *group* of logistic losses instead of a single logistic loss. In both robust classification and ranking, the transformation $\rho_2(\cdot)$ enables models to give up on part of the problem to achieve better overall performance.

As we discussed in Section 7.2, however, transformation of logistic loss using $\rho_2(\cdot)$ results in Type-II loss function, which is very difficult to optimize. Hence, instead of $\rho_2(\cdot)$ we use an alternative transformation function $\rho_1(\cdot)$, which generates Type-I loss function, to define the objective function of RoBiRank:

$$\overline{L}_1(\omega) := \sum_{x \in \mathcal{X}} c_x \sum_{y \in \mathcal{Y}_x} v\left(W_{xy}\right) \cdot \rho_1\left(\sum_{y' \in \mathcal{Y}_x, y' \neq y} \sigma_0\left(f_\omega(x,y) - f_\omega(x,y')\right)\right). \tag{7.16}$$

Since $\rho_1(t) \geqslant \rho_2(t)$ for every $t > 0$, we have $\overline{L}_1(\omega) \geqslant \overline{L}_2(\omega) \geqslant L_2(\omega)$ for every ω.

Note that $\overline{L}_1(\omega)$ is continuous and twice differentiable. Therefore, standard gradient-based optimization techniques can be applied to minimize it.

As in standard models of machine learning, of course, a regularizer on ω can be added to avoid overfitting; for simplicity, we use ℓ_2-norm in our experiments, but other loss functions can be used as well.

7.4 Latent Collaborative Retrieval

7.4.1 Model Formulation

For each context x and an item $y \in \mathcal{Y}$, the standard problem setting of learning to rank requires training data to contain feature vector $\phi(x, y)$ and score W_{xy} assigned on the x, y pair. When the number of contexts $|\mathcal{X}|$ or the number of items $|\mathcal{Y}|$ is large, it might be difficult to define $\phi(x, y)$ and measure W_{xy} for all x, y pairs, especially if it requires human intervention. Therefore, in most learning to rank problems we define the set of *relevant* items $\mathcal{Y}_x \subset \mathcal{Y}$ to be much smaller than $\mathcal{Y}$ for each context x, and then collect data only for $\mathcal{Y}_x$. Nonetheless, this may not be realistic in all situations; in a movie recommender system, for example, for each user *every* movie is somewhat relevant.

On the other hand, implicit user feedback data are much more abundant. For example, a lot of users on Netflix would simply watch movie streams on the system but do not leave an explicit rating. By the action of watching a movie, however, they implicitly express their preference. Such data consist only of positive feedback, unlike traditional learning to rank datasets which have score W_{xy} between each context-item pair x, y. Again, we may not be able to extract feature vector $\phi(x, y)$ for each x, y pair.

In such a situation, we can attempt to learn the score function $f(x, y)$ without feature vector $\phi(x, y)$ by embedding each context and item in an Euclidean latent space; specifically, we redefine the score function of ranking to be:

$$f(x, y) := \langle U_x, V_y \rangle, \tag{7.17}$$

where $U_x \in \mathbb{R}^d$ is the embedding of the context x and $V_y \in \mathbb{R}^d$ is that of the item y. Then, we can learn these embeddings by a ranking model. This approach was introduced in Weston et al. [76] using the name of *latent collaborative retrieval.*

Now we specialize RoBiRank model for this task. Let us define Ω to be the set of context-item pairs (x, y) which was observed in the dataset. Let $v(W_{xy}) = 1$ if

$(x, y) \in \Omega$, and 0 otherwise; this is a natural choice since the score information is not available. For simplicity, we set $c_x = 1$ for every x. Now RoBiRank (7.16) specializes to:

$$\overline{L}_1(U, V) = \sum_{(x,y)\in\Omega} \rho_1 \left(\sum_{y' \neq y} \sigma_0(f(U_x, V_y) - f(U_x, V_{y'})) \right). \tag{7.18}$$

Note that now the summation inside the parenthesis of (7.18) is over all items $\mathcal{Y}$ instead of a smaller set $\mathcal{Y}_x$, therefore we omit specifying the range of y' from now on.

To avoid overfitting, a regularizer term on U and V can be added to (7.18); for simplicity we use the Frobenius norm of each matrix in our experiments, but of course other regularizers can be used.

7.4.2 Stochastic Optimization

When the size of the data $|\Omega|$ or the number of items $|\mathcal{Y}|$ is large, however, methods that require exact evaluation of the function value and its gradient will become very slow since the evaluation takes $O(|\Omega| \cdot |\mathcal{Y}|)$ computation. In this case, stochastic optimization methods are desirable [13]; in this subsection, we will develop a stochastic gradient descent algorithm whose complexity is independent of $|\Omega|$ and $|\mathcal{Y}|$.

For simplicity, let θ be a concatenation of all parameters $\{U_x\}_{x\in\mathcal{X}}$, $\{V_y\}_{y\in\mathcal{Y}}$. The gradient $\nabla_\theta L_1(U, V)$ of (7.18) is

$$\sum_{(x,y)\in\Omega} \nabla_\theta \rho_1 \left(\sum_{y' \neq y} \sigma_0(f(U_x, V_y) - f(U_x, V_{y'})) \right).$$

Finding an unbiased estimator of the above gradient whose computation is independent of $|\Omega|$ is not difficult; if we sample a pair (x, y) uniformly from Ω, then it is easy to see that the following simple estimator

$$|\Omega| \cdot \nabla_\theta \rho_1 \left(\sum_{y' \neq y} \sigma_0(f(U_x, V_y) - f(U_x, V_{y'})) \right) \tag{7.19}$$

is unbiased. This still involves a summation over $\mathcal{Y}$, however, so it requires $O(|\mathcal{Y}|)$ calculation. Since $\rho_1(\cdot)$ is a nonlinear function it seems unlikely that an unbiased stochastic gradient which randomizes over $\mathcal{Y}$ can be found; nonetheless, to achieve standard convergence guarantees of the stochastic gradient descent algorithm, unbiasedness of the estimator is necessary [51].

We attack this problem by *linearizing* the objective function by parameter expansion. Note the following property of $\rho_1(\cdot)$ [14]:

$$\rho_1(t) = \log_2(t+1) \leqslant -\log_2 \xi + \frac{\xi \cdot (t+1) - 1}{\log 2}. \tag{7.20}$$

This holds for any $\xi > 0$, and the bound is tight when $\xi = \frac{1}{t+1}$. Now introducing an auxiliary parameter ξ_{xy} for each $(x, y) \in \Omega$ and applying this bound, we obtain an upper bound of (7.18) as

$$L(U, V, \xi) := \sum_{(x,y)\in\Omega} -\log_2 \xi_{xy} + \frac{\xi_{xy}\left(\sum_{y'\neq y} \sigma_0(f(U_x, V_y) - f(U_x, V_{y'})) + 1\right) - 1}{\log 2}. \tag{7.21}$$

Now we propose an iterative algorithm in which, each iteration consists of (U, V)-step and ξ-step; in the (U, V)-step we minimize (7.21) in (U, V) and in the ξ-step we minimize in ξ. The pseudo-code of the algorithm is given in the Algorithm 3.

(U, V)-step The partial derivative of (7.21) in terms of U and V can be calculated as:

$$\nabla_{U,V} L(U, V, \xi) := \frac{1}{\log 2} \sum_{(x,y)\in\Omega} \xi_{xy} \left(\sum_{y'\neq y} \nabla_{U,V} \sigma_0(f(U_x, V_y) - f(U_x, V_{y'})) \right).$$

Now it is easy to see that the following stochastic procedure unbiasedly estimates the above gradient:

- Sample (x, y) uniformly from Ω
- Sample y' uniformly from $\mathcal{Y}\setminus\{y\}$
- Estimate the gradient by

$$\frac{|\Omega| \cdot (|\mathcal{Y}| - 1) \cdot \xi_{xy}}{\log 2} \cdot \nabla_{U,V} \sigma_0(f(U_x, V_y) - f(U_x, V_{y'})). \tag{7.22}$$

Algorithm 3 Serial parameter estimation algorithm for latent collaborative retrieval

```
1: $\eta$: step size
2: while convergence in $U, V$ and $\xi$ do
3:     while convergence in $U, V$ do
4:         // $(U, V)$-step
5:         Sample $(x, y)$ uniformly from $\Omega$
6:         Sample $y'$ uniformly from $\mathcal{Y} \backslash \{y\}$
7:         $U_x \leftarrow U_x - \eta \cdot \xi_{xy} \cdot \nabla_{U_x} \sigma_0(f(U_x, V_y) - f(U_x, V_{y'}))$
8:         $V_y \leftarrow V_y - \eta \cdot \xi_{xy} \cdot \nabla_{V_y} \sigma_0(f(U_x, V_y) - f(U_x, V_{y'}))$
9:     end while
10:    // $\xi$-step
11:    for $(x, y) \in \Omega$ do
12:        $\xi_{xy} \leftarrow \frac{1}{\sum_{y' \neq y} \sigma_0(f(U_x, V_y) - f(U_x, V_{y'})) + 1}$
13:    end for
14: end while
```

Therefore a stochastic gradient descent algorithm based on (7.22) will converge to a local minimum of the objective function (7.21) with probability one [58]. Note that the time complexity of calculating (7.22) is independent of $|\Omega|$ and $|\mathcal{Y}|$. Also, it is a function of only U_x and V_y; the gradient is zero in terms of other variables.

ξ-step When U and V are fixed, minimization of ξ_{xy} variable is independent of each other and a simple analytic solution exists:

$$\xi_{xy} = \frac{1}{\sum_{y' \neq y} \sigma_0(f(U_x, V_y) - f(U_x, V_{y'})) + 1}. \tag{7.23}$$

This of course requires $O(|\mathcal{Y}|)$ work. In principle, we can avoid summation over $\mathcal{Y}$ by taking stochastic gradient in terms of ξ_{xy} as we did for U and V. However, since the exact solution is very simple to compute and also because most of the computation time is spent on (U, V)-step rather than ξ-step, we found this update rule to be efficient.

7.4.3 Parallelization

The linearization trick in (7.21) not only enables us to construct an efficient stochastic gradient algorithm, but also makes possible to efficiently parallelize the algorithm across multiple number of machines. The objective function is technically not doubly separable, but a strategy similar to that of DSGD introduced in Chapter 2.3.2 can be deployed.

Suppose there are p number of machines. The set of contexts $\mathcal{X}$ is randomly partitioned into mutually exclusive and exhaustive subsets $\mathcal{X}^{(1)}, \mathcal{X}^{(2)}, \ldots, \mathcal{X}^{(p)}$ which are of approximately the same size. This partitioning is fixed and does not change over time. The partition on $\mathcal{X}$ induces partitions on other variables as follows: $U^{(q)} := \{U_x\}_{x \in \mathcal{X}^{(q)}}$, $\Omega^{(q)} := \left\{(x, y) \in \Omega : x \in \mathcal{X}^{(q)}\right\}$, $\xi^{(q)} := \{\xi_{xy}\}_{(x,y) \in \Omega^{(q)}}$, for $1 \leqslant q \leqslant p$.

Each machine q stores variables $U^{(q)}$, $\xi^{(q)}$ and $\Omega^{(q)}$. Since the partition on $\mathcal{X}$ is fixed, these variables are local to each machine and are not communicated. Now we describe how to parallelize each step of the algorithm: the pseudo-code can be found in Algorithm 4.

Algorithm 4 Multi-machine parameter estimation algorithm for latent collaborative retrieval

```
$\eta$: step size
while convergence in $U, V$ and $\xi$ do
    // parallel $(U, V)$-step
    while convergence in $U, V$ do
        Sample a partition $\left\{\mathcal{Y}^{(1)}, \mathcal{Y}^{(2)}, \ldots, \mathcal{Y}^{(q)}\right\}$
        Parallel Foreach $q \in \{1, 2, \ldots, p\}$
            Fetch all $V_y \in V^{(q)}$
            while predefined time limit is exceeded do
                Sample $(x, y)$ uniformly from $\left\{(x, y) \in \Omega^{(q)}, y \in \mathcal{Y}^{(q)}\right\}$
                Sample $y'$ uniformly from $\mathcal{Y}^{(q)} \setminus \{y\}$
                $U_x \leftarrow U_x - \eta \cdot \xi_{xy} \cdot \nabla_{U_x} \sigma_0(f(U_x, V_y) - f(U_x, V_{y'}))$
                $V_y \leftarrow V_y - \eta \cdot \xi_{xy} \cdot \nabla_{V_y} \sigma_0(f(U_x, V_y) - f(U_x, V_{y'}))$
            end while
        Parallel End
    end while
    // parallel $\xi$-step
    Parallel Foreach $q \in \{1, 2, \ldots, p\}$
        Fetch all $V_y \in V$
        for $(x, y) \in \Omega^{(q)}$ do
            $\xi_{xy} \leftarrow \frac{1}{\sum_{y' \neq y} \sigma_0(f(U_x, V_y) - f(U_x, V_{y'})) + 1}$
        end for
    Parallel End
end while
```

(U, V)-step At the start of each (U, V)-step, a new partition on $\mathcal{Y}$ is sampled to divide $\mathcal{Y}$ into $\mathcal{Y}^{(1)}, \mathcal{Y}^{(2)}, \ldots, \mathcal{Y}^{(p)}$ which are also mutually exclusive, exhaustive and of approximately the same size. The difference here is that unlike the partition on $\mathcal{X}$, a new partition on $\mathcal{Y}$ is sampled for every (U, V)-step. Let us define $V^{(q)} := \{V_y\}_{y \in \mathcal{Y}^{(q)}}$. After the partition on $\mathcal{Y}$ is sampled, each machine q fetches V_y's in $V^{(q)}$ from where it was previously stored; in the very first iteration which no previous information exists, each machine generates and initializes these parameters instead. Now let us define

$$L^{(q)}(U^{(q)}, V^{(q)}, \xi^{(q)}) := \sum_{(x,y) \in \Omega^{(q)}, y \in \mathcal{Y}^{(q)}} -\log_2 \xi_{xy} \\ + \frac{\xi_{xy} \left(\sum_{y' \in \mathcal{Y}^{(q)}, y' \neq y} \sigma_0(f(U_x, V_y) - f(U_x, V_{y'})) + 1 \right) - 1}{\log 2}.$$

In parallel setting, each machine q runs stochastic gradient descent on $L^{(q)}(U^{(q)}, V^{(q)}, \xi^{(q)})$ instead of the original function $L(U, V, \xi)$. Since there is no overlap between machines on the parameters they update and the data they access, every machine can progress independently of each other. Although the algorithm takes only a fraction of data into consideration at a time, this procedure is also guaranteed to converge to a local optimum of the *original* function $L(U, V, \xi)$. Note that in each iteration,

$$\nabla_{U,V} L(U, V, \xi) = q^2 \cdot \mathbb{E}\left[\sum_{1 \leqslant q \leqslant p} \nabla_{U,V} L^{(q)}(U^{(q)}, V^{(q)}, \xi^{(q)}) \right],$$

where the expectation is taken over random partitioning of $\mathcal{Y}$. Therefore, although there is some discrepancy between the function we take stochastic gradient on and the function we actually aim to minimize, in the long run the bias will be washed out and the algorithm will converge to a local optimum of the objective function $L(U, V, \xi)$. This intuition can be easily translated to the formal proof of the convergence; since each partitioning of $\mathcal{Y}$ is independent of each other, we can appeal to the law of large numbers to prove that the necessary condition (2.27) for the convergence of the algorithm is satisfied.

ξ-step In this step, all machines synchronize to retrieve every entry of V. Then, each machine can update $\xi^{(q)}$ independently of each other. When the size of V is very large and cannot be fit into the main memory of a single machine, V can be partitioned as in (U, V)-step and updates can be calculated in a round-robin way.

Note that this parallelization scheme requires each machine to allocate only $\frac{1}{p}$-fraction of memory that would be required for a single-machine execution. Therefore, in terms of space complexity the algorithm scales linearly with the number of machines.

7.5 Related Work

In terms of modeling, viewing ranking problem as a generalization of binary classification problem is not a new idea; for example, RankSVM defines the objective function as a sum of hinge losses, similarly to our basic model (7.8) in Section 7.3.2. However, it does not directly optimize the ranking metric such as NDCG; the objective function and the metric are not immediately related to each other. In this respect, our approach is closer to that of Le and Smola [44] which constructs a convex upper bound on the ranking metric and Chapelle et al. [17] which improves the bound by introducing non-convexity. The objective function of Chapelle et al. [17] is also motivated by ramp loss, which is used for robust classification; nonetheless, to our knowledge the direct connection between the ranking metrics in form (7.11) (DCG, NDCG) and the robust loss (7.4) is our novel contribution. Also, our objective function is designed to specifically bound the ranking metric, while Chapelle et al. [17] proposes a general recipe to improve existing convex bounds.

Stochastic optimization of the objective function for latent collaborative retrieval has been also explored in Weston et al. [76]. They attempt to minimize

$$\sum_{(x,y)\in\Omega} \Phi\left(1 + \sum_{y' \neq y} I(f(U_x, V_y) - f(U_x, V_{y'}) < 0)\right), \tag{7.24}$$

where $\Phi(t) = \sum_{k=1}^{t} \frac{1}{k}$. This is similar to our objective function (7.21); $\Phi(\cdot)$ and $\rho_2(\cdot)$ are asymptotically equivalent. However, we argue that our formulation (7.21) has two major advantages. First, it is a continuous and differentiable function, therefore gradient-based algorithms such as L-BFGS and stochastic gradient descent have convergence guarantees. On the other hand, the objective function of Weston et al. [76] is not even continuous, since their formulation is based on a function $\Phi(\cdot)$ that is defined for only natural numbers. Also, through the linearization trick in (7.21) we are able to obtain an unbiased stochastic gradient, which is necessary for the convergence guarantee, and to parallelize the algorithm across multiple machines as discussed in Section 7.4.3. It is unclear how these techniques can be adapted for the objective function of Weston et al. [76].

Note that Weston et al. [76] proposes a more general class of models for the task than can be expressed by (7.24). For example, they discuss situations in which we have side information on each context or item to help learning latent embeddings. Some of the optimization techniqures introduced in Section 7.4.2 can be adapted for these general problems as well, but is left for future work.

Parallelization of an optimization algorithm via parameter expansion (7.20) was applied to a bit different problem named multinomial logistic regression [33]. However, to our knowledge we are the first to use the trick to construct an unbiased stochastic gradient that can be efficiently computed, and adapt it to stratified stochastic gradient descent (SSGD) scheme of Gemulla et al. [31]. Note that the optimization algorithm can alternatively be derived using convex multiplicative programming framework of Kuno et al. [42]. In fact, Ding [22] develops a robust classification algorithm based on this idea; this also indicates that robust classification and ranking are closely related.

7.6 Experiments

In this section we empirically evaluate RoBiRank. Our experiments are divided into two parts. In Section 7.6.1, we apply RoBiRank on standard benchmark datasets from the learning to rank literature. These datasets have relatively small number of relevant items $|\mathcal{Y}_x|$ for each context x, so we will use L-BFGS [53], a quasi-Newton algorithm, for optimization of the objective function (7.16). Although L-BFGS is designed for optimizing convex functions, we empirically find that it converges reliably to a local minima of the RoBiRank objective function (7.16) in all our experiments. In Section 7.6.2 we apply RoBiRank to the million songs dataset (MSD), where stochastic optimization and parallelization are necessary.

Name	$\lvert\mathcal{X}\rvert$	avg.	Mean NDCG			Regularization Parameter		
		$\lvert\mathcal{Y}_x\rvert$	RoBiRank	RankSVM	LSRank	RoBiRank	RankSVM	LSRank
TD 2003	50	981	0.9719	0.9219	0.9721	10^{-5}	10^{-3}	10^{-1}
TD 2004	75	989	0.9708	0.9084	0.9648	10^{-6}	10^{-1}	10^{4}
Yahoo! 1	29,921	24	0.8921	0.7960	0.871	10^{-9}	10^{3}	10^{4}
Yahoo! 2	6,330	27	0.9067	0.8126	0.8624	10^{-9}	10^{5}	10^{4}
HP 2003	150	984	0.9960	0.9927	0.9981	10^{-3}	10^{-1}	10^{-4}
HP 2004	75	992	0.9967	0.9918	0.9946	10^{-4}	10^{-1}	10^{2}
OHSUMED	106	169	0.8229	0.6626	0.8184	10^{-3}	10^{-5}	10^{4}
MSLR30K	31,531	120	0.7812	0.5841	0.727	1	10^{3}	10^{4}
MQ 2007	1,692	41	0.8903	0.7950	0.8688	10^{-9}	10^{-3}	10^{4}
MQ 2008	784	19	0.9221	0.8703	0.9133	10^{-5}	10^{3}	10^{4}

Table 7.1.: Descriptive Statistics of Datasets and Experimental Results in Section 7.6.1.

7.6.1 Standard Learning to Rank

We will try to answer the following questions:

- What is the benefit of transforming a convex loss function (7.8) into a non-covex loss function (7.16)? To answer this, we compare our algorithm against RankSVM [45], which uses a formulation that is very similar to (7.8), and is a state-of-the-art pairwise ranking algorithm.
- How does our non-convex upper bound on negative NDCG compare against other convex relaxations? As a representative comparator we use the algorithm of Le and Smola [44], mainly because their code is freely available for download. We will call their algorithm LSRank in the sequel.
- How does our formulation compare with the ones used in other popular algorithms such as LambdaMART, RankNet, etc? In order to answer this question, we carry out detailed experiments comparing RoBiRank with 12 different algorithms. In Figure 7.2 RoBiRank is compared against RankSVM, LSRank, InfNormPush [60] and IRPush [5]. We then downloaded RankLib [1] and used its default settings to compare against 8 standard ranking algorithms (see Figure7.3) - MART, RankNet, RankBoost, AdaRank, CoordAscent, LambdaMART, ListNet and RandomForests.
- Since we are optimizing a non-convex objective function, we will verify the sensitivity of the optimization algorithm to the choice of initialization parameters.

We use three sources of datasets: LETOR 3.0 [16] , LETOR 4.0[2] and YAHOO LTRC [54], which are standard benchmarks for learning to rank algorithms; Table 7.1 shows their summary statistics. Each dataset consists of five folds; we consider the first fold, and use the training, validation, and test splits provided. We train with different values of the regularization parameter, and select a parameter with the best NDCG value on the validation dataset. Then, performance of the model with this

[1]http://sourceforge.net/p/lemur/wiki/RankLib

[2]http://research.microsoft.com/en-us/um/beijing/projects/letor/letor4dataset.aspx

parameter on the test dataset is reported. For a fair comparison, every algorithm follows exactly the same protocol and uses the same split of data. All experiments in this section are conducted on a computing cluster where each node has two 2.1 GHz 12-core AMD 6172 processors with 48 GB physical memory per node. We used an optimized implementation of the L-BFGS algorithm provided by the Toolkit for Advanced Optimization (TAO)[3] for estimating the parameter of RoBiRank. For the other algorithms, we used the implementations provided by the authors. Our main goal is to compare the performance of the models, and not the speed of parameter estimation. However, we note that the training time is very comparable for all three algorithms, with RoBiRank being at most two to three times slower than other algorithms on some datasets.

We use values of NDCG at different levels of truncation as our evaluation metric [50]; see Figure 7.2. While RoBiRank outperforms its competitors on most of the datasets, of particular interest is the result on TD 2004 dataset. The performance of RankSVM is a bit insensitive to the level of truncation for NDCG. On the other hand, RoBiRank, which uses non-convex loss function to concentrate its performance at the top of the ranked list, performs much better especially at low truncation levels. It is also interesting to note that the NDCG@k curve of LSRank is similar to that of RoBiRank, but RoBiRank consistently outperforms at each level. RobiRank dominates Inf-Push and IR-Push on all datasets except TD 2003 and OHSUMED where IRPush seems to fare better at the top of the list. Compared to the 8 standard algorithms, again RobiRank either outperforms or performs comparably to the best algorithm except on two datasets (TD 2003 and HP 2003), where MART and Random Forests overtake RobiRank at few values of NDCG. We present a summary of the NDCG values obtained by each algorithm in Table 7.1.

We also investigated the sensitivity of parameter estimation to the choice of initial parameter. We initialized ω randomly with 10 different seed values. Blue lines in Figure 7.4 show mean and standard deviation of NDCG values at different levels of

[3]http://www.mcs.anl.gov/research/projects/tao/index.html

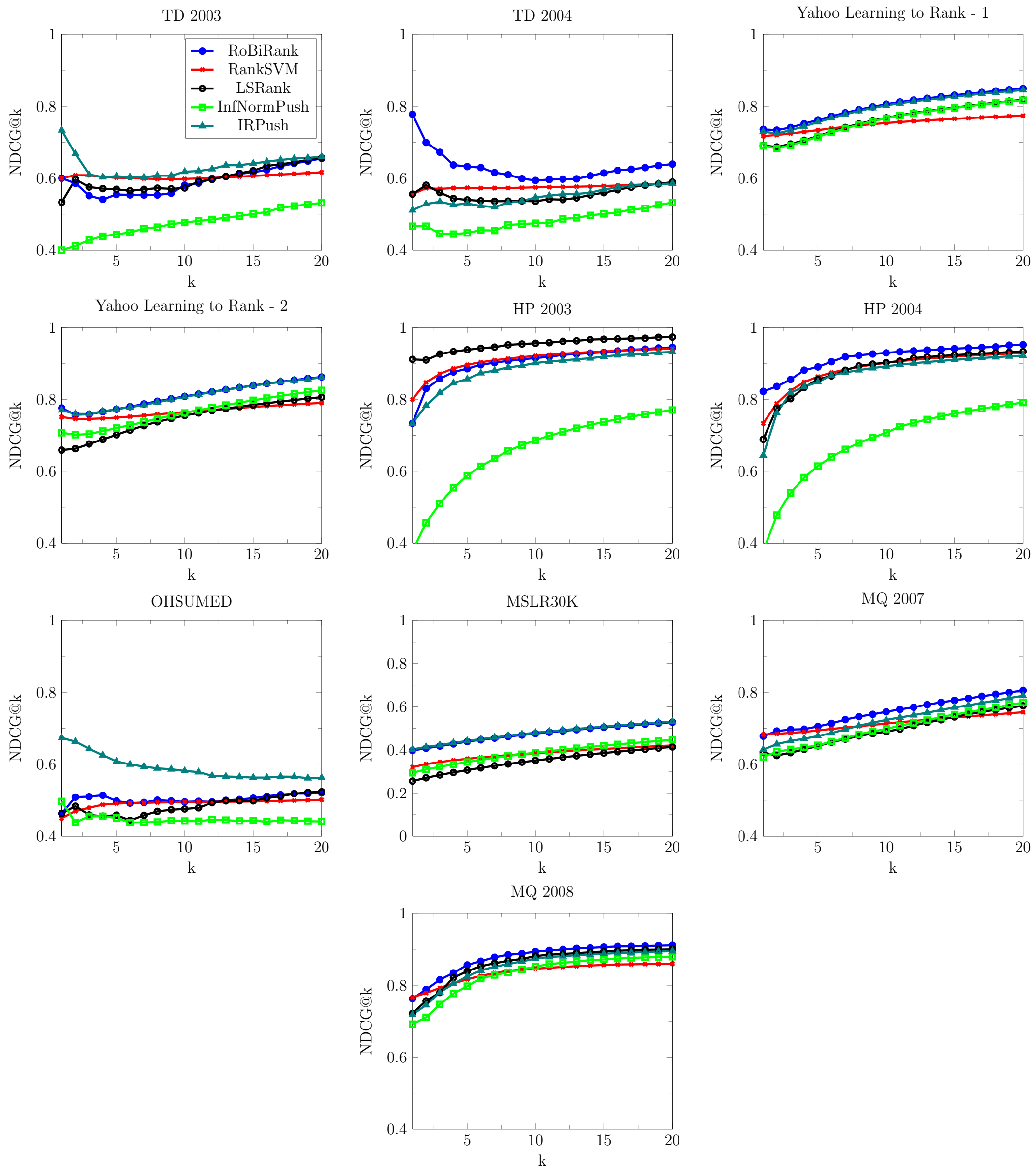


Figure 7.2.: Comparison of RoBiRank, RankSVM, LSRank [44], Inf-Push and IR-Push

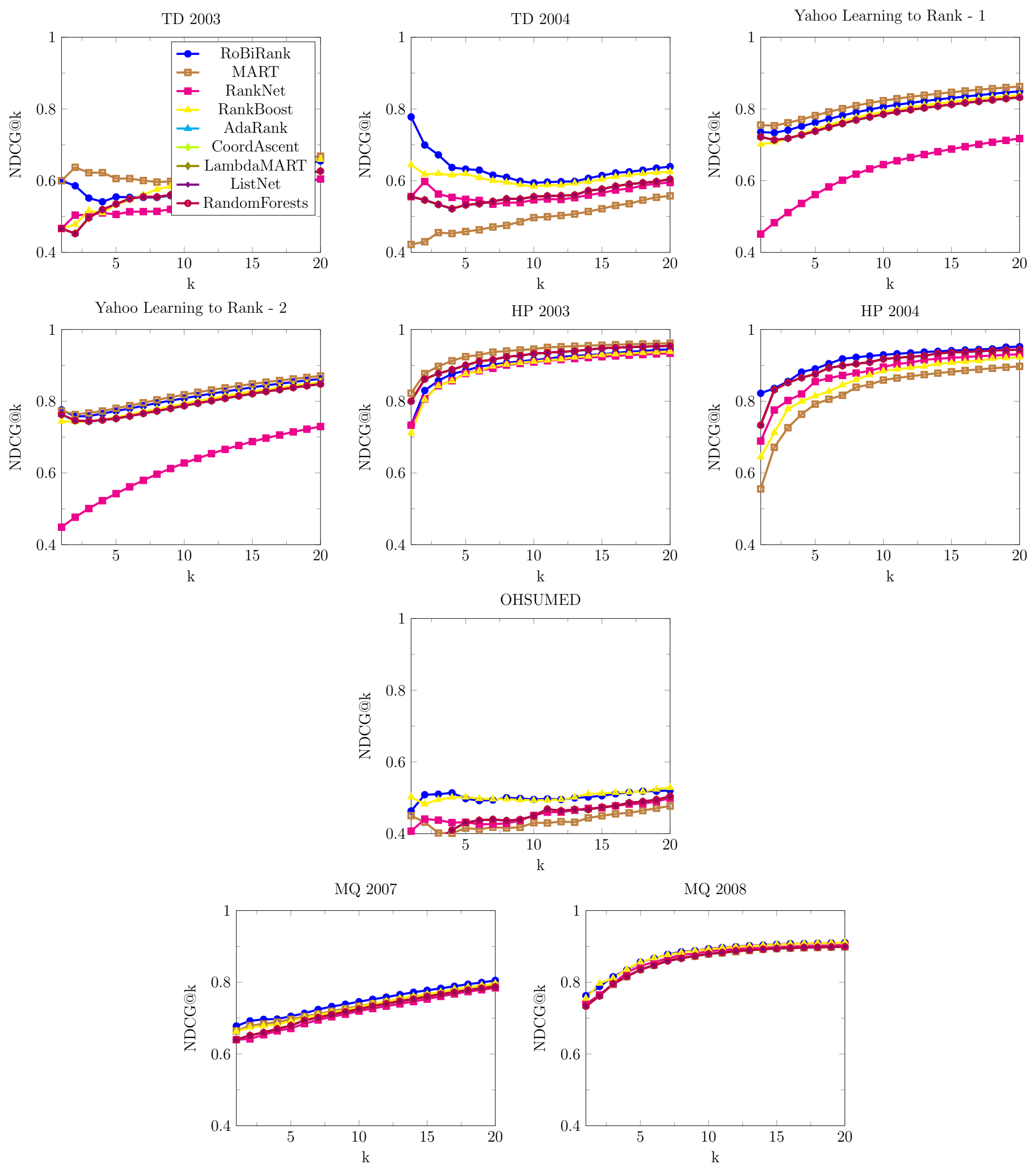


Figure 7.3.: Comparison of RoBiRank, MART, RankNet, RankBoost, AdaRank, CoordAscent, LambdaMART, ListNet and RandomForests

truncation; as can be seen, even though our objective function is non-convex, L-BFGS reliably converges to solutions with similar test performance. This conclusion is in line with the observation of Ding [22]. We also tried two more variants; initialization by all-zeroes (red line) and the solution of RankSVM (black line). In most cases it did not affect the quality of solution, but on TD 2003 and HP 2004 datasets, zero initialization gave slightly better results.

7.6.2 Latent Collaborative Retrieval

In this subsection, we ask the following question: Given large amounts of computational resources, what is the best latent collaborative retrieval model (in terms of predictive performance on the test dataset) that one can produce within a given wall-clock time? Towards this end, we work with the parallel variant of RoBiRank described in Section 7.4.3. As a representative dataset we use the Million Song Dataset (MSD) [9], which consists of 1,129,318 users ($|\mathcal{X}|$), 386,133 songs ($|\mathcal{Y}|$), and 49,824,519 records ($|\Omega|$) of a user x playing a song y in the training dataset. The objective is to predict the songs from the test dataset that a user is going to listen to[4].

Since explicit ratings are not given, NDCG is not applicable for this task; we use precision at 1 and 10 [50] as our evaluation metric. Squared frobenius norm of matrices U and V were added to the objective function (7.16) for regularization, and the entries of U and V were independently sampled uniformly from 0 to $1/\sqrt{d}$. We performed a grid-search to find the best step size parameter.

This experiment was run on a computing cluster where each machine is equipped with 2 Intel Xeon E5 processors (16 cores) and 32GB of RAM. Our algorithm is implemented in C++ and uses Intel Thread Building Blocks (TBB) to handle thread-level parallelization, and MVAPICH2 was used for machine-to-machine communication. Due to a limitation of the job scheduler on the cluster all experiments had to be stopped after 100,000 seconds.

[4]the original data also provides the number of times a song was played by a user, but we ignored this in our experiment.

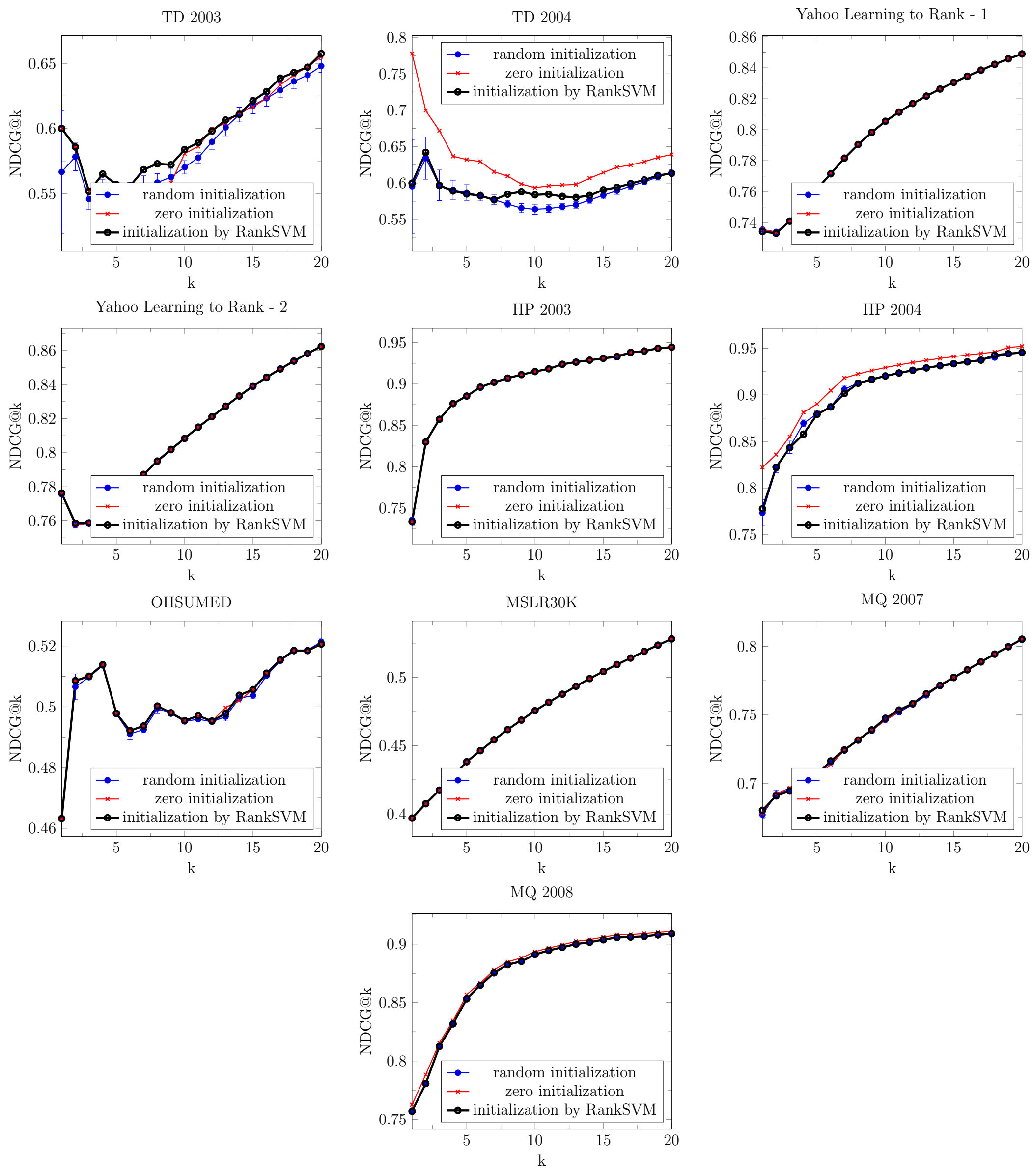


Figure 7.4.: Performance of RoBiRank based on different initialization methods

In our first experiment we study the scaling behavior of RoBiRank as a function of number of machines. RoBiRank p denotes the parallel version of RoBiRank which is distributed across p machines. In Figure 7.5 (left) we plot mean Precision@1 as a function of the number of machines $\times$ the number of seconds elapsed; this is a proxy for CPU time. If an algorithm linearly scales across multiple processors, then all lines in the figure should overlap with each other. As can be seen RoBiRank exhibits near ideal speed up when going from 4 to 32 machines[5].

In our next experiment we compare RoBiRank with a state of the art algorithm from Weston et al. [76], which optimizes a similar objective function (7.24). We compare how fast the quality of the solution improves as a function of wall clock time. Since the authors of Weston et al. [76] do not make available their code, we implemented their algorithm within our framework using the same data structures and libraries used by our method. Furthermore, for a fair comparison, we used the same initialization for U and V and performed an identical grid-search over the step size parameter for both algorithms.

Figure 7.5 (center, right) shows the results of the experiment. It can be seen that on a single machine the algorithm of Weston et al. [76] is very competitive and outperforms RoBiRank. The reason for this might be the introduction of the additional ξ variables in RoBiRank, which slows down convergence. However, RoBiRank training can be distributed across processors, while it is not clear how to parallelize the algorithm of Weston et al. [76]. Consequently, RoBiRank 32 which uses 32 machines for its computation can produce a significantly better model within the same wall clock time window.

7.7 Conclusion

In this chapter, we developed RoBiRank, a novel model on ranking, based on insights and techniques from the literature of robust binary classification. Then, we

[5]The graph for RoBiRank 1 is hard to see because it was run for only 100,000 CPU-seconds.

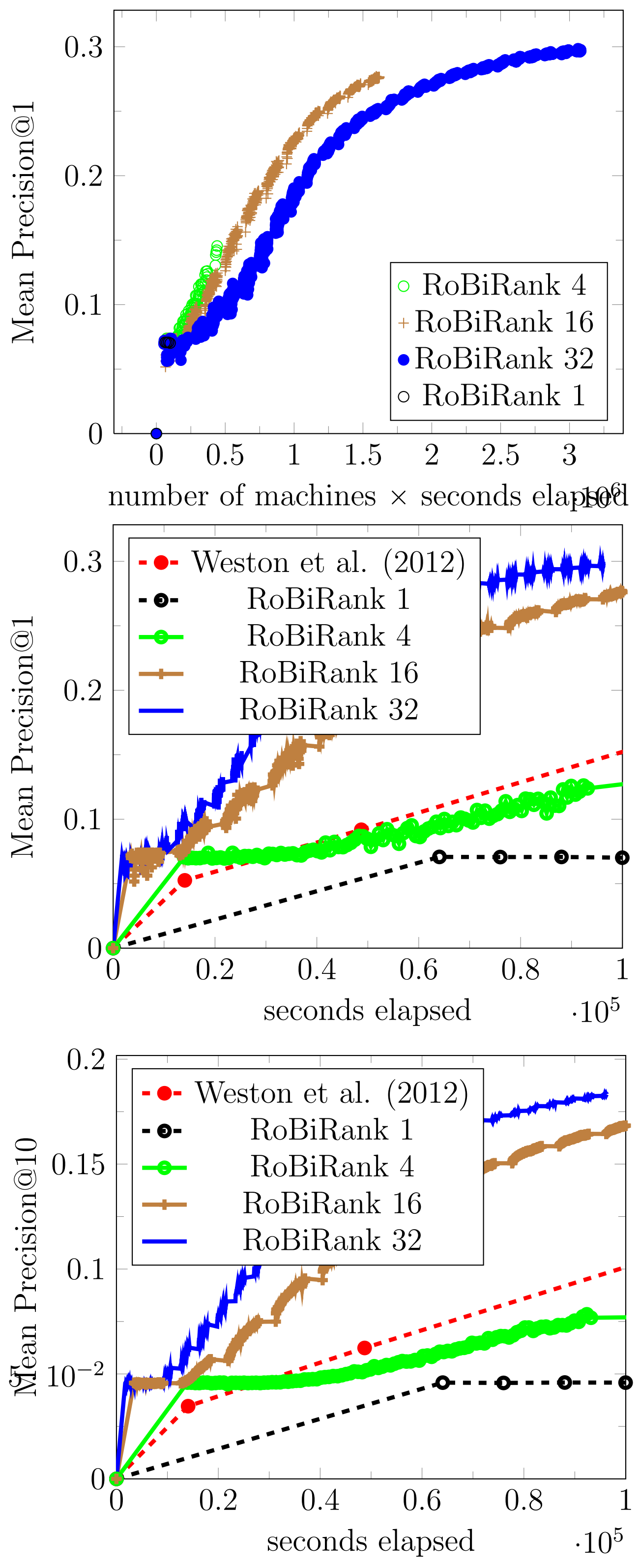


Figure 7.5.: Top: the scaling behavior of RoBiRank on Million Song Dataset. Middle, Bottom: Performance comparison of RoBiRank and Weston et al. [76] when the same amount of wall-clock time for computation is given.

proposed a scalable and parallelizable stochastic optimization algorithm that can be applied to the task of latent collaborative retrieval which large-scale data without feature vectors and explicit scores have to take care of. Experimental results on both learning to rank datasets and latent collaborative retrieval dataset suggest the advantage of our approach.

We are currently investigating how to extend our method to the more general context of collaborative retrieval tasks in which additional side information is available, as discussed in Weston et al. [76].

8. SUMMARY

8.1 Contributions

We provide a summary of our contributions and discuss directions for future research.

Optimization of Doubly Separable Functions We have identified double separability as a useful property of a function that can be used to efficiently parallelize stochastic optimization algorithms. We have proposed DSSO, an adaptation of DSGD for saddle-point problems which has similar convergence guarantees, and NOMAD, an asynchronous algorithm that utilizes fine-grained partitioning of the problem for efficient scaling in massive distributed computing environment.

Reformulating Existing Statistical models as Doubly Separable Functions We have shown that a large class of statistical models can be formulated as doubly separable functions. In the case of matrix completion and item response theory, the original formulations were readily in doubly separable form; on the other hand, for regularized risk minimization and multinomial logistic regression, we had to introduce different parameter expansion techniques to achieve double separability.

Formulation of Ranking as Robust Binary Classification We have argued that metrics for information retrieval such as NDCG can be understood as generalization of robust classification, and proposed RoBiRank, a novel algorithm for learning to rank. Then, we have identified that when RoBiRank is applied for latent collaborative retrieval, a feature-free version of the ranking problem, it can be efficiently parallelized with a simple extension of techniques for doubly separable functions.

8.2 Future Work

The idea of double separability can be further generalized for more than two dimensions. For example, one may define triple separability, or even separability of an arbitrary order. Tensor factorization problems will be naturally fit into this extended framework, but are there any other statistical models which can be formulated in this extended notion of separability? Also, can optimization algorithms for doubly separable functions be generalized for this more general setting? These are interesting questions that remain unanswered yet.

Furthermore, although we have confined our interest to optimization problems in this thesis, double separability might be found useful in Bayesian models as well to efficiently parallelize MCMC sampling or variational inference algorithms. We are currently investigating how the NOMAD framework can be used for Latent Dirichlet Allocation (LDA).

LIST OF REFERENCES

LIST OF REFERENCES

[1] Apache hadoop, 2009. http://hadoop.apache.org/core/.

[2] Graphlab datasets, 2013. http://graphlab.org/downloads/datasets/.

[3] Intel thread building blocks, 2013. https://www.threadingbuildingblocks.org/.

[4] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford. A reliable effective terascale linear learning system. *CoRR*, abs/1110.4198, 2011.

[5] S. Agarwal. The infinite push: A new support vector ranking algorithm that directly optimizes accuracy at the absolute top of the list. In *SDM*, pages 839–850. SIAM, 2011.

[6] P. L. Bartlett, M. I. Jordan, and J. D. McAuliffe. Convexity, classification, and risk bounds. *Journal of the American Statistical Association*, 101(473):138–156, 2006.

[7] R. M. Bell and Y. Koren. Lessons from the netflix prize challenge. *SIGKDD Explorations*, 9(2):75–79, 2007. URL `http://doi.acm.org/10.1145/1345448.1345465`.

[8] M. Benzi, G. H. Golub, and J. Liesen. Numerical solution of saddle point problems. *Acta numerica*, 14:1–137, 2005.

[9] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[10] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, 1989.

[11] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1997.

[12] C. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[13] L. Bottou and O. Bousquet. The tradeoffs of large-scale learning. *Optimization for Machine Learning*, page 351, 2011.

[14] G. Bouchard. Efficient bounds for the softmax function, applications to inference in hybrid models. 2007.

[15] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, England, 2004.

[16] O. Chapelle and Y. Chang. Yahoo! learning to rank challenge overview. *Journal of Machine Learning Research-Proceedings Track*, 14:1–24, 2011.

[17] O. Chapelle, C. B. Do, C. H. Teo, Q. V. Le, and A. J. Smola. Tighter bounds for structured estimation. In *Advances in neural information processing systems*, pages 281–288, 2008.

[18] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2:199–222, 1969.

[19] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In B. Schölkopf, J. Platt, and T. Hofmann, editors, *Advances in Neural Information Processing Systems 19*, pages 281–288. MIT Press, 2006.

[20] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008. URL `http://doi.acm.org/10.1145/1327452.1327492`.

[21] C. DeMars. *Item response theory.* Oxford University Press, 2010.

[22] N. Ding. *Statistical Machine Learning in T-Exponential Family of Distributions.* PhD thesis, PhD thesis, Purdue University, West Lafayette, Indiana, USA, 2013.

[23] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer. The yahoo! music dataset and kdd-cup'11. *Journal of Machine Learning Research-Proceedings Track*, 18: 8–18, 2012.

[24] R.-E. Fan, J.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, Aug. 2008.

[25] J. Faraway. *Extending the Linear Models with R.* Chapman & Hall/CRC, Boca Raton, FL, 2004.

[26] V. Feldman, V. Guruswami, P. Raghavendra, and Y. Wu. Agnostic learning of monomials by halfspaces is hard. *SIAM Journal on Computing*, 41(6):1558–1590, 2012.

[27] V. Franc and S. Sonnenburg. Optimized cutting plane algorithm for support vector machines. In A. McCallum and S. Roweis, editors, *ICML*, pages 320–327. Omnipress, 2008.

[28] J. Friedman, T. Hastie, H. Höfling, R. Tibshirani, et al. Pathwise coordinate optimization. *The Annals of Applied Statistics*, 1(2):302–332, 2007.

[29] A. Frommer and D. B. Szyld. On asynchronous iterations. *Journal of Computational and Applied Mathematics*, 123:201–216, 2000.

[30] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 69–77. ACM, 2011.

[31] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Conference on Knowledge Discovery and Data Mining*, pages 69–77, 2011.

[32] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2012)*, 2012.

[33] S. Gopal and Y. Yang. Distributed training of large-scale logistic models. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 289–297, 2013.

[34] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, New York, 2 edition, 2009.

[35] J.-B. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms, I and II*, volume 305 and 306. Springer-Verlag, 1996.

[36] T. Hoefler, P. Gottschling, W. Rehm, and A. Lumsdaine. Optimizing a conjugate gradient solver with non blocking operators. *Parallel Computing*, 2007.

[37] C. J. Hsieh and I. S. Dhillon. Fast coordinate descent methods with variable selection for non-negative matrix factorization. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining(KDD)*, pages 1064–1072, August 2011.

[38] C. J. Hsieh, K. W. Chang, C. J. Lin, S. S. Keerthi, and S. Sundararajan. A dual coordinate descent method for large-scale linear SVM. In W. Cohen, A. McCallum, and S. Roweis, editors, *ICML*, pages 408–415. ACM, 2008.

[39] C.-N. Hsu, H.-S. Huang, Y.-M. Chang, and Y.-J. Lee. Periodic step-size adaptation in second-order gradient descent for single-pass on-line structured learning. *Machine Learning*, 77(2-3):195–224, 2009.

[40] P. J. Huber. *Robust Statistics*. John Wiley and Sons, New York, 1981.

[41] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009. URL `http://doi.ieeecomputersociety.org/10.1109/MC.2009.263`.

[42] T. Kuno, Y. Yajima, and H. Konno. An outer approximation method for minimizing the product of several convex functions on a convex set. *Journal of Global Optimization*, 3(3):325–335, September 1993.

[43] J. Langford, A. J. Smola, and M. Zinkevich. Slow learners are fast. In *Neural Information Processing Systems*, 2009. URL `http://arxiv.org/abs/0911.0491`.

[44] Q. V. Le and A. J. Smola. Direct optimization of ranking measures. Technical Report 0704.3359, arXiv, April 2007. `http://arxiv.org/abs/0704.3359`.

[45] C.-P. Lee and C.-J. Lin. Large-scale linear ranksvm. *Neural Computation*, 2013. To Appear.

[46] D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(3):503–528, 1989.

[47] J. Liu, W. Zhong, and L. Jiao. Multi-agent evolutionary model for global numerical optimization. In *Agent-Based Evolutionary Search*, pages 13–48. Springer, 2010.

[48] P. Long and R. Servedio. Random classification noise defeats all convex potential boosters. *Machine Learning Journal*, 78(3):287–304, 2010.

[49] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. In *PVLDB*, 2012.

[50] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008. URL `http://nlp.stanford.edu/IR-book/`.

[51] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM J. on Optimization*, 19(4): 1574–1609, Jan. 2009. ISSN 1052-6234.

[52] J. Nocedal. Updating quasi-Newton matrices with limited storage. *Mathematics of Computation*, 35:773–782, 1980.

[53] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Series in Operations Research. Springer, 2nd edition, 2006.

[54] T. Qin, T.-Y. Liu, J. Xu, and H. Li. Letor: A benchmark collection for research on learning to rank for information retrieval. *Information Retrieval*, 13(4):346–374, 2010.

[55] S. Ram, A. Nedic, and V. Veeravalli. Distributed stochastic subgradient projection algorithms for convex optimization. *Journal of Optimization Theory and Applications*, 147:516–545, 2010.

[56] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In P. Bartlett, F. Pereira, R. Zemel, J. Shawe-Taylor, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701, 2011. URL `http://books.nips.cc/nips24.html`.

[57] P. Richtarik and M. Takac. Distributed coordinate descent method for learning with big data. Technical report, 2013. URL `"http://arxiv.org/abs/1310.2059"`.

[58] H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.

[59] R. T. Rockafellar. *Convex Analysis*, volume 28 of *Princeton Mathematics Series*. Princeton University Press, Princeton, NJ, 1970.

[60] C. Rudin. The p-norm push: A simple convex ranking algorithm that concentrates at the top of the list. *The Journal of Machine Learning Research*, 10: 2233–2271, 2009.

[61] B. Schölkopf and A. J. Smola. *Learning with Kernels*. MIT Press, Cambridge, MA, 2002.

[62] N. N. Schraudolph. Local gain adaptation in stochastic gradient descent. In *Proc. Intl. Conf. Artificial Neural Networks*, pages 569–574, Edinburgh, Scotland, 1999. IEE, London.

[63] S. Shalev-Shwartz and N. Srebro. Svm optimization: Inverse dependence on training set size. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 928–935, 2008.

[64] S. Shalev-Shwartz, Y. Singer, and N. Srebro. Pegasos: Primal estimated sub-gradient solver for SVM. In *Proc. Intl. Conf. Machine Learning*, 2007.

[65] A. J. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *Very Large Databases (VLDB)*, 2010.

[66] S. Sonnenburg, V. Franc, E. Yom-Tov, and M. Sebag. Pascal large scale learning challenge. 2008. URL `http://largescale.ml.tu-berlin.de/workshop/`.

[67] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, editors, *Conference on World Wide Web*, pages 607–614. ACM, 2011. URL `http://doi.acm.org/10.1145/1963405.1963491`.

[68] M. Tabor. *Chaos and integrability in nonlinear dynamics: an introduction*, volume 165. Wiley New York, 1989.

[69] C. Teflioudi, F. Makari, and R. Gemulla. Distributed matrix completion. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 655–664. IEEE, 2012.

[70] C. H. Teo, S. V. N. Vishwanthan, A. J. Smola, and Q. V. Le. Bundle methods for regularized risk minimization. *Journal of Machine Learning Research*, 11: 311–365, January 2010.

[71] P. Tseng and C. O. L. Mangasarian. Convergence of a block coordinate descent method for nondifferentiable minimization. *J. Optim Theory Appl*, pages 475–494, 2001.

[72] N. Usunier, D. Buffoni, and P. Gallinari. Ranking with ordered weighted pairwise classification. In *Proceedings of the International Conference on Machine Learning*, 2009.

[73] A. W. Van der Vaart. *Asymptotic statistics*, volume 3. Cambridge university press, 2000.

[74] S. V. N. Vishwanathan and L. Cheng. Implicit online learning with kernels. *Journal of Machine Learning Research*, 2008.

[75] S. V. N. Vishwanathan, N. Schraudolph, M. Schmidt, and K. Murphy. Accelerated training conditional random fields with stochastic gradient methods. In *Proc. Intl. Conf. Machine Learning*, pages 969–976, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-383-2.

[76] J. Weston, C. Wang, R. Weiss, and A. Berenzweig. Latent collaborative retrieval. *arXiv preprint arXiv:1206.4603*, 2012.

[77] G. G. Yin and H. J. Kushner. *Stochastic approximation and recursive algorithms and applications*. Springer, 2003.

[78] H.-F. Yu, C.-J. Hsieh, S. Si, and I. S. Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In M. J. Zaki, A. Siebes, J. X. Yu, B. Goethals, G. I. Webb, and X. Wu, editors, *ICDM*, pages 765–774. IEEE Computer Society, 2012. ISBN 978-1-4673-4649-8.

[79] Y. Zhuang, W.-S. Chin, Y.-C. Juan, and C.-J. Lin. A fast parallel sgd for matrix factorization in shared memory systems. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 249–256. ACM, 2013.

[80] M. Zinkevich, A. J. Smola, M. Weimer, and L. Li. Parallelized stochastic gradient descent. In nips23e, editor, *nips23*, pages 2595–2603, 2010.

APPENDIX

A. SUPPLEMENTARY EXPERIMENTS ON MATRIX COMPLETION

A.1 Effect of the Regularization Parameter

In this subsection, we study the convergence behavior of NOMAD as we change the regularization parameter λ (Figure A.1). Note that in Netflix data (left), for non-optimal choices of the regularization parameter the test RMSE increases from the initial solution as the model overfits or underfits to the training data. While NOMAD reliably converges in all cases, on Netflix the convergence is notably faster with higher values of λ; this is expected because regularization smooths the objective function and makes the optimization problem easier to solve. On other datasets, the speed of convergence was not very sensitive to the selection of the regularization parameter.

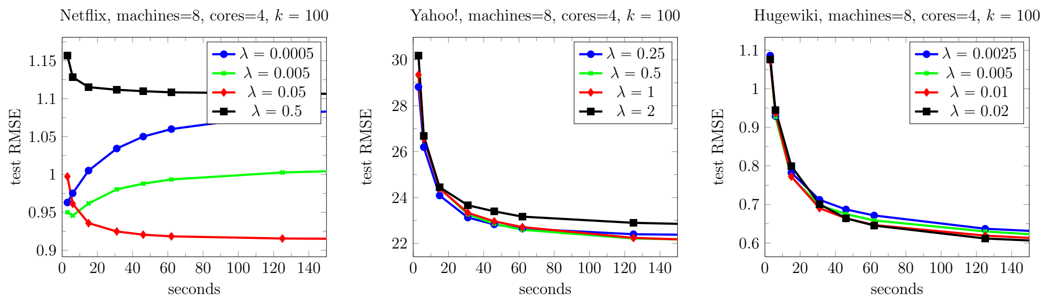


Figure A.1.: Convergence behavior of NOMAD when the regularization parameter λ is varied.

A.2 Effect of the Latent Dimension

In this subsection, we study the convergence behavior of NOMAD as we change the dimensionality parameter k (Figure A.2). In general, the convergence is faster for smaller values of k as the computational cost of SGD updates (2.21) and (2.22) is linear to k. On the other hand, the model gets richer with higher values of k, as its parameter space expands; it becomes capable of picking up weaker signals in the data, with the risk of overfitting. This is observed in Figure A.2 with Netflix (left) and Yahoo! Music (right). In Hugewiki, however, small values of k were sufficient to fit the training data, and test RMSE suffers from overfitting with higher values of k. Nonetheless, NOMAD reliably converged in all cases.

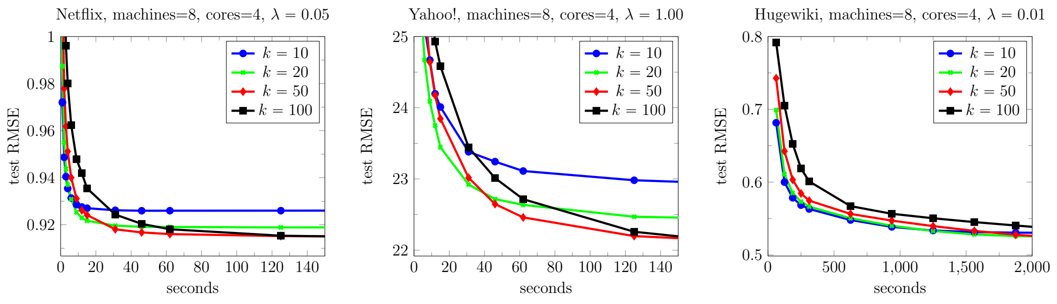


Figure A.2.: Convergence behavior of NOMAD when the latent dimension k is varied.

A.3 Comparison of NOMAD with GraphLab

Here we provide experimental comparison with GraphLab of Low et al. [49]. GraphLab PowerGraph 2.2, which can be downloaded from `https://github.com/graphlab-code/graphlab` was used in our experiments. Since GraphLab was not compatible with Intel compiler, we had to compile it with `gcc`. The rest of experimental setting is identical to what was described in Section 4.3.1.

Among a number of algorithms GraphLab provides for matrix completion in its collaborative filtering toolkit, only Alternating Least Squares (ALS) algorithm is suitable for solving the objective function (4.1); unfortunately, Stochastic Gradient Descent (SGD) implementation of GraphLab does not converge. According to private conversations with GraphLab developers, this is because the abstraction currently provided by GraphLab is not suitable for the SGD algorithm. Its `biassgd` algorithm, on the other hand, is based on a model different from (4.1) and therefore not directly comparable to NOMAD as an optimization algorithm.

Although each machine in HPC cluster is equipped with 32 GB of RAM and we distribute the work into 32 machines in multi-machine experiments, we had to tune `nfibers` parameter to avoid out of memory problems, and still was not able to run GraphLab on Hugewiki data in any setting. We tried both synchronous and asynchronous engines of GraphLab, and report the better of the two on each configuration.

Figure A.3 shows results of single-machine multi-threaded experiments, while Figure A.4 and Figure A.5 shows multi-machine experiments on HPC cluster and commodity cluster respectively. Clearly, NOMAD converges orders of magnitude faster than GraphLab in every setting, and also converges to better solutions. Note that GraphLab converges faster in single-machine setting with large number of cores (30) than in multi-machine setting with large number of machines (32) but small number of cores (4) each. We conjecture that this is because the locking and unlocking of a variable has to be requested via network communication in distributed memory setting; on the other hand, NOMAD does not require a locking mechanism and thus scales better with the number of machines.

Although GraphLab `biassgd` is based on a model different from (4.1), for the interest of readers we provide comparisons with it on commodity hardware cluster. Unfortunately, GraphLab `biassgd` crashed when we ran it on more than 16 machines, so we had to run it on only 16 machines and assumed GraphLab will linearly scale up to 32 machines, in order to generate plots in Figure A.5. Again, NOMAD was orders of magnitude faster than GraphLab and converges to a better solution.

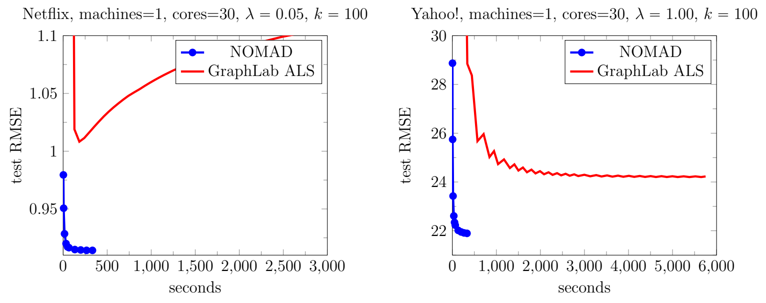


Figure A.3.: Comparison of NOMAD and GraphLab on a single machine with 30 computation cores.

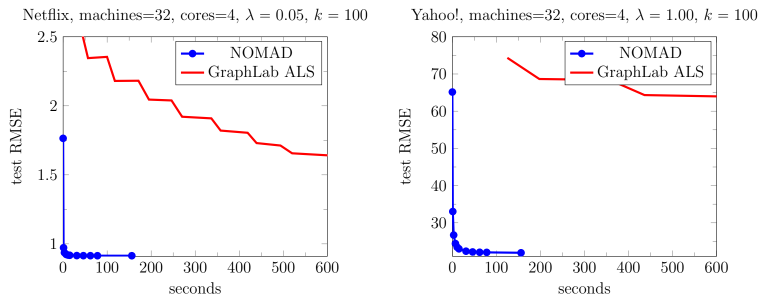


Figure A.4.: Comparison of NOMAD and GraphLab on a HPC cluster.

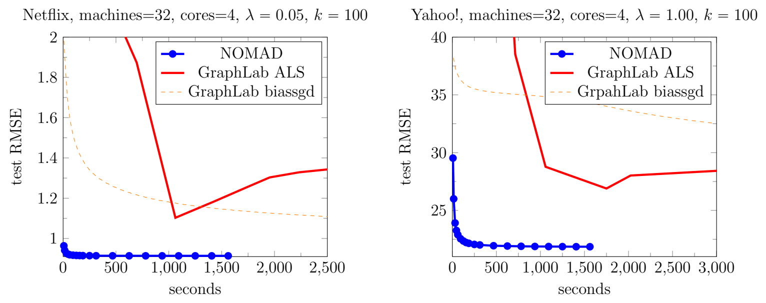


Figure A.5.: Comparison of NOMAD and GraphLab on a commodity hardware cluster.

VITA

VITA

Hyokun Yun was born in Seoul, Korea, on February 6, 1984. He was a software engineer in Cyram(c) from 2006 to 2008, and he received bachelor's degree in Industrial & Management Engineering and Mathematics at POSTECH, Republic of Korea in 2009. Then, he joined graduate program of Statistics at Purdue University in US; under supervision of Prof. S.V.N. Vishwanathan, he earned master's degree in 2013 and doctoral degree in 2014. His research interests are statistical machine learning, stochastic optimization, social network analysis, recommendation systems and inferential model.