Universität des Saarlandes

Algorithms for Shared-Memory Matrix Completion and Maximum Inner Product Search

Christina Teflioudi

Dissertation

zur Erlangung des Grades

des Doktors der Ingenieurwissenschaften (Dr.-Ing.)

der Naturwissenschaftlich-Technischen Fakultäten

der Universität des Saarlandes

April 18, 2016

Saarbrücken

Dean	Prof. Dr. Markus Bläser
Colloquium	14.04.2016

Examination Board

Supervisor and 1st Reviewer	Prof. Dr. Rainer Gemulla *Universität Mannheim*
2nd Reviewer	Prof. Dr. Gerhard Weikum *Max-Planck-Institut für Informatik*
3rd Reviewer	Dr. Pauli Miettinen *Max-Planck-Institut für Informatik*
Chairman	Prof. Dr. Gert Smolka *Universität des Saarlandes*
Academic Assistant	Dr. Daria Stepanova *Max-Planck-Institut für Informatik*

Abstract

In this thesis, we propose efficient and scalable algorithms for shared-memory matrix factorization and maximum inner product search. Matrix factorization is a popular tool in the data mining community due to its ability to quantify the interactions between different types of entities. It typically maps the (potentially noisy) original representations of the entities into a lower dimensional space, where the "true" structure of the dataset is revealed. Inner products of the new vector representations are then used to measure the interactions between different entities. The *strongest* of these interactions are usually of particular interest in applications and can be retrieved by solving a maximum inner product search problem.

For large real-life problem instances of matrix factorization and maximum inner product search, efficient and scalable methods are necessary. We first study large-scale matrix factorization in a shared-memory setting and we propose a cache-aware, parallel method that avoids fine-grained synchronization or locking. In more detail, our approach partitions the initial large problem into small, cache-fitting sub-problems that can be solved independently using stochastic gradient descent. Due to the low cache-miss rate and the absence of any locking or synchronization, our method achieves superior performance in terms of speed (up to 60% faster) and scalability than previously proposed techniques.

We then proceed with investigating the problem of maximum inner product search and design a cache-friendly framework that allows for both exact and approximate search. Our approach reduces the original maximum inner product search problem into a set of smaller cosine similarity search problems that can be solved using existing cosine similarity search techniques or our novel algorithms tailored for use within our framework. Experimental results show that our approach is multiple orders of magnitude faster than naive search, consistently faster than alternative methods for exact search, and achieves better quality-speedup tradeoff (up to 3.9x faster for similar recall levels) than state-of-the-art approximate techniques.

Kurzfassung

In dieser Arbeit schlagen wir effiziente und skalierbare Algorithmen für Matrixfaktorisierung und für die Suche nach maximalen Skalarprodukten unter einer gemeinsam genutzten Speicherarchitektur vor. Matrixfaktorisierung ist ein beliebtes Werkzeug in der Data-Mining-Gemeinschaft aufgrund ihrer Fähigkeit, die Interaktionen zwischen verschiedenen Arten von Objekten zu quantifizieren. Sie bildet typischerweise die (möglicherweise verrauschte) originale Darstellung der Objekte auf einen niederdimensionalen Raum ab, wo die wahre Struktur der Daten sichtbar wird. Die Skalarprodukte zwischen den neuen Darstellungen werden dann benutzt, um die Interaktionen zwischen den verschiedenen Objekten zu messen. Die stärksten dieser Interaktionen sind in Anwendungen oft von besonderem Interesse und können durch eine Suche nach maximalen Skalarprodukten abgerufen werden.

Für große, reale Probleme der Matrixfaktorisierung und der Suche nach maximalen Skalarprodukten sind effiziente und skalierbare Methoden notwendig. Zunächst betrachten wir hochskalierbare Matrixfaktorisierung unter einer gemeinsam genutzten Speicherarchitektur und schlagen eine cachebewusste, parallele Methode vor, die feingranulare Synchronisation oder Locking vermeidet. Genauer betrachtet teilt unsere Methode das ursprüngliche, große Problem in kleine, cachepassende Probleme, die unabhänging voneinander durch stochastischen Gradientenabstieg gelöst werden können. Aufgrund der niedrigen Cache Miss Rate und der Abwesenheit von Locking und Synchronisation, erreicht unsere Methode eine verbesserte Leistung in Bezug auf Laufzeit (bis zu 60% schneller) und Skalierbarkeit verglichen mit vorherigen Techniken.

Anschließend erforschen wir das Problem der Suche nach maximalen Skalarprodukten und entwerfen ein cachefreundliches System, das sowohl genaue als auch approximative Suche ermöglicht. Unsere Methode reduziert das ursprüngliche Problem auf eine Reihe von kleineren Problemen der Cosinus-Ähnlichkeitssuche. Diese können durch vorhandene Techniken für Cosinus-Ähnlichkeitssuche oder neue Algorithmen, die eigens für die Benutzung innerhalb unseres Systems gebaut sind, gelöst werden. Die Versuchsergebnisse zeigen, dass unsere genauen Methoden um mehrere Größenordnungen schneller als naive Suche und konstant schneller als alternative Methoden sind, und dass unsere approximativen Techniken einen besseren Qualität-Laufzeit-Trade-Off (bis zu 3.9-Mal schneller für ähnliche Recall-Level) als der moderne Stand der Technik für approximative Suche erreichen.

Acknowledgements

First and foremost, I am deeply grateful to my parents for their support. I would not be where I am today without their prayers, love and encouragement. I would like to express my deepest gratitude to my supervisor, Prof. Rainer Gemulla. I learned a lot from him and, honestly, this work would not have been possible without his help and guidance. Similarly, I feel the need to thank Gerhard Weikum for his support and for the excellent and inspiring environment he ensured in the department. I also want to thank the colleagues and friends with whom we shared so many experiences throughout all these years. I am very grateful to Saskia Metzler, Iulia Bolosteanu and Christine Bocionek for being the proof-readers of my thesis. I also thank the International Max Planck Research School for Computer Science for the financial support that gave me the possibility to pursue a PhD. Finally, I want to thank Christine Bocionek for her encouragement all these years.

Contents

To my parents...

Chapter 1

Introduction

Thesis Scope

Matrix factorization methods, such as singular value decomposition (SVD), non-negative matrix factorization (NMF), or latent-factor models, have recently gained traction in the data mining community. In particular, they have been successfully applied in the context of collaborative filtering in recommender systems [Chen et al., 2012, Das et al., 2010, Hu et al., 2008, Koren et al., 2009, Mackey et al., 2011, Niu et al., 2011, Recht and Ré, 2013, Yu et al., 2012, Zhou et al., 2008], in link prediction [Menon and Elkan, 2011], in global positioning of sensors [Biswas et al., 2006, Singer, 2008], in remote sensing [Schmidt, 1986], in the structure-from-motion problem [Chen and Suter, 2005] in computer vision, and in relation extraction [F. Petroni and Gemulla, 2015, Riedel et al., 2013]. Matrix factorizations are effective tools for analyzing dyadic data in that they discover and quantify the interactions between different entities. In this thesis, we study fast and scalable matrix factorization methods for shared-memory architectures. In addition, we investigate the problem of efficiently identifying the strongest of the discovered interactions between the entities involved in the matrix factorization.

We focus on "matrix completion", a variant of low-rank matrix factorization, in which the input matrix is only partially observed and the observations are potentially noisy. Matrix completion techniques have been shown to be one of the best single approaches used in recommender systems (although in practice ensembles of such methods together with other models are often used). For this reason, we will use a recommender system as an example throughout this thesis.

Recommender systems consider as input a set of users and a set of items (movies, products, songs, books, web-pages etc.). The users provide feedback for the items they

interacted with, e.g., explicit feedback in the form of numerical ratings and time point of rating or implicit feedback, such as page views. This feedback can be represented in terms of a sparse matrix, in which rows correspond to users and columns to items, while the values of the entries are the ratings. A toy example of such a matrix for recommending movies to users can be seen in Fig. 1.1-(a), where we mark with red question-marks the unknown ratings. One of the main tasks of a recommender system is to predict the preferences of each user over items with which she/he did not interact yet, i.e., to predict the values at the positions of the red question-marks in Fig. 1.1-(a).

Matrix completion is a (popular) way to model the user preferences. The initial ratings matrix gets approximated by the product of two new matrices: one representing the users in terms of some latent factors (2 in our example) and one representing the items in terms of the same latent factors (Fig. 1.1-(b)). Although the factors are latent, in that they do not have explicit semantic interpretations, the assumption is that they somehow encode the preferences of the users and the properties of the items. Notice that in our example the first latent factor roughly corresponds to action movies while the second one corresponds to romance movies. The underlying assumption behind the learning process is that users who rate the same items in a similar way should share the same preferences and that items which are rated in a similar way by a user should have similar properties. In our example, the movies "Once" and "Amelie" have similar factor interpretations because the user Debby gave them similar ratings. Similarly, Debby and Charlie have similar preferences (they like romance movies and dislike action movies), because they gave "Once" the same rating. In general, during matrix factorization there is information flow between both entity types (users and movies in our example). We will refer to the task of learning the factor representations of the entities (users/movies) from the input data as the *"matrix completion task"*.

The product of the two factor matrices, created in the matrix completion step, is a completed, dense version of the initial sparse matrix, i.e., it contains predictions for all missing entries (Fig. 1.1-(c)). In the basic case, which we focus on in this thesis, each missing matrix entry is estimated as the inner product of feature vectors for the corresponding user and item[1]. Thus a user is predicted to rate an item highly if features that are important to her/him (i.e., have a large absolute value) match with the features of the item (large value of equal sign). The larger the predicted value for the missing entry, the better the recommendation is. In the example of Fig. 1.1, we consider the top-3 movies for each user. In general, such large entries are of particular interest in matrix factorization applications, because they indicate strong interactions between the entities. We will refer to the task of identifying the pairs of entities with the strongest interactions

[1]In general, the estimation formula can be a complex function of the features, as well as of other data, such as the time stamp of the rating, user bias, implicit feedback, and so on.

(denoted by large entries in the matrix product) as the *"large entry retrieval task"*. Since every entry in the matrix product corresponds to an inner product between the vector representations of two entities (a users and an item in the recommender systems context), the large entry retrieval task is practically equivalent to solving a maximum inner product search (MIPS) problem.

Some applications might involve additional steps after the large entry retrieval step. Recommender systems, for example, retrieve the best recommendations according to the predictive model used (the matrix completion model in our case) and choose some of them to present to their users. The choice depends on application related criteria (e.g., availability, diversity, freshness, profit-margin for the company etc). For example, the recommender system in Fig. 1.1-(d) filters out movies that have already been watched by the user.

In this thesis, we will focus on the first two tasks, namely, the matrix completion and large entry retrieval tasks.

Contributions

In this thesis we present novel, efficient and scalable algorithms for both the matrix completion and the large entry retrieval task.

Real-life applications of matrix factorization may involve matrices with millions of rows (customers) and columns (items) and billions of entries (observed and predicted ratings). At such massive scales, parallel and efficient algorithms are necessary to achieve reasonable performance for both the factorization and retrieval task.

We design algorithms tailored for shared-memory architectures. Nowadays high-end parallel machines routinely ship with multiple terabytes of RAM that can fit very large problem instances of matrix completion. Consider, for example, an extremely large hypothetical problem instance in which the input matrix has 10M rows, 10M columns, and 0.1% of the entries are revealed (for comparison 0.04% of the entries in the Yahoo! Music dataset in the 2011 KDD-Cup were revealed). Assume that 100 features are used per row and per column (i.e., a rank-100 factorization). If for each entry of the sparse matrix (in sparse representation) we need 8 bytes to store the value and another 4+4 bytes to store the row and column indexes of the entry, then the total data and model size is approximately 1.5TB, which can easily fit in such high-end parallel machines. Depending on the application additional compression of the data might be possible. For example, if the ratings are integers between 1-100 (as in Yahoo! Music) then, 1 byte is enough

FIGURE 1.1: Overview of the recommendation process with example

for storing the value of the rating and the overall memory consumption goes down to 0.83TB.

High-end parallel machines usually contain dozens of cores, enough to provide in theory significant speedup. In practice, however, even for embarrassingly parallel problems, algorithms tend to under-utilize the capabilities of multicore systems. The performance of these systems is suffering when: (i) the threads need to communicate (for example when locking) (ii) the threads intensely compete for resources (cache, system bus access). In the latter case, the cache-awareness of the algorithm plays an important role: a high cache-miss rate for a thread means not only more time for this thread to bring data to the cache, but also possibly the eviction from cache of data the other threads need. Similarly, higher cache-miss rates for all the threads means also high competition for the system buses in order to fetch data from main memory.

For these reasons, there is a need for carefully designed, cache-aware algorithms that take

full advantage of the capabilities of multicore systems. In this thesis, we describe such algorithms for the task of matrix factorization and large entry retrieval. Our contributions can be summarized as follows:

- we study cache-aware algorithms for parallel matrix completion in shared-memory architectures
- we propose an efficient, cache-friendly framework for retrieving the large entries in the resulting matrix product.

Matrix Completion

We study matrix completion with stochastic gradient descent (SGD) in a shared-memory setting. SGD is an iterative optimization algorithm that has been shown, in a sequential setting, to be very effective for matrix factorization [Koren et al., 2009]. Unfortunately, the generic SGD algorithm has two main bottlenecks: (i) it usually follows a random memory access pattern, leading to an increased number of cache-misses and (ii) it is not embarrassingly parallel.

While the second point was addressed in many different ways in previously published related work ([Gemulla et al., 2011c, Niu et al., 2011, Recht and Ré, 2013]), the first point remained largely ignored. As a result, the suggested parallel methods only scale for a relatively small number of threads (say up to 8). In Chapter 2, we propose "Cache-aware SGD" (CSGD), a lock-free and cache-friendly algorithm for shared-memory architectures which exhibits superior scalability to existing methods for larger numbers of threads (up to 32 in our experiments). We additionally review existing sequential and shared-memory approaches based on SGD and on alternating-minimizations, and conduct an extensive set of experiments comparing CSGD with alternative existing methods on both real and synthetic datasets.

Large Entry Retrieval (or Maximum Inner Product Search)

Retrieving naively the best recommendations involves first computing the full product of the two factor matrices and then finding the largest entries in the product matrix. This naive solution becomes infeasible in practice, for large problem instances. For example, for our hypothetical problem with the 10M customers and 10M movies, the naive solution will have to conduct 100 trillion inner product computations. If an inner product computation takes about 140 ns on average for vectors with 100 factors (as in

our experimental study), it takes more than 160 days for the naive search to complete (ignoring any other costs such as I/O costs).

In the past, both exact and approximate algorithms (tree-based [Bachrach et al., 2014, Curtin and Ram, 2014, Curtin et al., 2013, Ram and Gray, 2012], based on Locality Sensitive Hashing [Neyshabur and Srebro, 2015, Shrivastava and Li, 2014a,b]) have been proposed for this problem. However, these methods are still relatively slow and they are not designed to be cache-aware.

In Chapter 3, we propose the LEMP framework for finding only the "Large Entries in a Matrix Product", without computing the full matrix product. As already mentioned, since every entry corresponds to an inner product between a user- and an item-vector, LEMP is practically solving a maximum inner product search (MIPS) problem. Our framework decomposes this large MIPS problem instance into a set of small, cache-fitting cosine similarity search problems. For each of these smaller problems, an existing cosine similarity search algorithm can be used, however we additionally propose two new algorithms, tailored for their use within LEMP. Apart from that, LEMP supports approximate retrieval by using more aggressive pruning techniques as well as approximate methods for solving the cosine similarity search problems, such as an adapted version of Locality Sensitive Hashing (LSH, [Gionis et al., 1999]); our methods can trade-off quality and performance and provide approximation guarantees. In addition, in Chapter 3, we compare LEMP with existing methods for exact and approximate MIPS through an extensive set of experiments on real datasets.

Finally, Chapter 4 summarizes the work presented in this thesis and presents directions for future work.

Chapter 2

Shared-Memory Matrix Completion Algorithms

In this chapter[1], we discuss low-rank matrix completion techniques, which have recently received significant attention in the data mining community. At its heart, matrix completion is a variant of low-rank matrix factorization in which the input matrix is only partially observed and the observations are potentially noisy.

Large applications can involve matrices with millions of rows, millions of columns, and billions of entries. For example, Netflix—a company that offers movies for rental and streaming and employs low-rank matrix completion in their recommendation engine—gathered more than five billion ratings for more than 80k movies from its more than 20M customers[2] [Amatriain and Basilico, 2012, Bennett and Lanning, 2007]. Similarly, Yahoo Music! collected billions of user ratings for musical pieces [Dror et al., 2012]. At such massive scales, algorithms for matrix completion must be parallelized to achieve reasonable performance [Das et al., 2007, 2010, Liu et al., 2010, Mackey et al., 2011, Niu et al., 2011, Recht and Ré, 2013, Yu et al., 2012, Zhou et al., 2008].

In this chapter, we study parallel algorithms for matrix completion in a shared-memory environment. Even extremely large matrix completion problem instances can be handled in such environments (see also the example in Chapter 1). The main bottlenecks for scalable processing in shared-memory, multicore architectures are: (i) the communication between the processing units (e.g. when locking is necessary) (ii) the cache-miss rate of the processing units. Algorithms that require often locking or that follow a random

[1] The contents of this chapter have been jointly developed with Rainer Gemulla, Faraz Makari, Peter J. Haas and Yannis Sismanis as in [Gemulla et al., 2011b], [Teflioudi et al., 2012] and [Makari et al., 2015].

[2] status in 2012.

memory access pattern (which implies a high cache-miss rate) tend to have inferior performance, specially when a large number of cores is used.

We propose a cache-aware, lock-free parallel method for matrix completion based on stochastic gradient descent (SGD). SGD is an iterative optimization algorithm that has been shown, in a sequential setting, to be very effective for matrix completion [Koren et al., 2009]. Although many techniques have been developed for parallelizing SGD ([Gemulla et al., 2011c, Niu et al., 2011, Recht and Ré, 2013]), the large majority does not address SGD's high cache-miss rate, caused by the fact that it usually follows a random memory access pattern. The result is that the scalability of these methods suffers significantly when a large number of cores is used.

This chapter is structured as follows: Section 2.1 defines the problem, introduces the notation used in this chapter and provides background information for the matrix completion problem. Section 2.2 discusses gradient descent and stochastic gradient descend (SGD) in a sequential setting. In Sections 2.3 and 2.4, we present our cache-aware SGD-based algorithms for matrix completion, termed SGD++ and CSGD. Related work is discussed in Section 2.5. In Section 2.7, we compare all methods in terms of an extensive set of experiments and summarize our work in Section 2.8.

2.1 The Matrix Completion Problem

To gain understanding about applications of matrix completion, consider the "Netflix problem" [Bennett and Lanning, 2007] of recommending movies to customers. Netflix is a company that offers tens of thousands of movies for rental. The company has more than 20M customers, each of whom can provide feedback about their personal taste by rating movies with 1 to 5 stars. The feedback can be represented in a matrix form, for example

$$\begin{array}{c} \\ \textit{Alice} \\ \textit{Bob} \\ \textit{Charlie} \end{array} \begin{array}{c} \begin{array}{ccc} \textit{Avatar} & \textit{The Matrix} & \textit{Up} \end{array} \\ \begin{pmatrix} ? & 4 & 2 \\ 3 & 2 & ? \\ 5 & ? & 3 \end{pmatrix} \end{array}.$$

Each entry may contain additional data, e.g., the date of rating or other forms of feedback such as click history. The goal of the completion is to predict missing entries (denoted by "?"), so that entries with a high predicted rating can then be recommended to users for viewing. This matrix-completion approach to recommender systems has been successfully

Symbol	Description
$\boldsymbol{V}$	Data matrix
m, n	Number of rows & columns of $\boldsymbol{V}$
Ω	Set of revealed entries in $\boldsymbol{V}$
N	Number of revealed entries in $\boldsymbol{V}$
N_{i*}	Number of revealed entries in row i of $\boldsymbol{V}$
N_{*j}	Number of revealed entries in column j of $\boldsymbol{V}$
r	Rank of the factorization
$\boldsymbol{L}, \boldsymbol{R}$	Factor matrices
$\boldsymbol{E}$	Residual matrix
p	Total number of threads
b	Number of row/column blocks (CSGD)
T	Repetition parameter (CCD++)
s	Number of shufflers (Jellyfish)

TABLE 2.1: Notation

applied in practice; see [Koren et al., 2009] for an excellent discussion of the underlying intuition.

Before we proceed with the problem definition, let us introduce some notation. For a summary of the notation used throughout this chapter please refer to Table 2.1. We denote by the *training set* $\Omega = \{ \omega_1, \ldots, \omega_N \}$ the set of revealed entries in $m \times n$ input matrix $\boldsymbol{V}$, where $\omega_k = (i_k, j_k)$, $k \in [1, N]$, $i_k \in [1, m]$, and $j_k \in [1, n]$. Let also N_{i*} and N_{*j} denote the number of revealed entries in row i and column j, respectively. Finally, we denote by $r \ll \min(m, n)$ a rank parameter. Our goal is to find an $m \times r$ row-factor matrix $\boldsymbol{L}$ and an $r \times n$ column-factor matrix $\boldsymbol{R}$ such that $\boldsymbol{V} \approx \boldsymbol{LR}$, i.e., we aim to approximate $\boldsymbol{V}$ by the low-rank matrix $\boldsymbol{LR}$. The approximation is governed by an application-dependent loss function $L(\boldsymbol{L}, \boldsymbol{R})$ that measures the difference between the revealed entries in $\boldsymbol{V}$ and the corresponding entries in $\boldsymbol{LR}$. (We suppress the dependence on $\boldsymbol{V}$ for brevity.) The matrix completion problem is to find the factor matrices that give rise to the smallest loss, i.e.,

$$(\boldsymbol{L}^*, \boldsymbol{R}^*) = \operatorname*{argmin}_{\boldsymbol{L}, \boldsymbol{R}} L(\boldsymbol{L}, \boldsymbol{R}). \tag{2.1}$$

The matrix $\boldsymbol{L}^* \boldsymbol{R}^*$ is a "completed version" of $\boldsymbol{V}$, and each unrevealed entry $\boldsymbol{V}_{ij}$ is predicted by $[\boldsymbol{L}^* \boldsymbol{R}^*]_{ij}$.

The loss function L may also incorporate user biases, implicit feedback, temporal effects, and confidence levels, as well as regularization terms to prevent over-fitting. The most basic loss is the squared loss:

Loss	Definition (L)	Local loss (L_{ij})
L_{Sl}	$\sum_{(i,j)\in\Omega}(\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij})^2$	$(\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij})^2$
L_{L2}	$L_{\text{Sl}} + \lambda\left(\sum_{ik}\boldsymbol{L}_{ik}^2 + \sum_{kj}\boldsymbol{R}_{kj}^2\right)$	$(\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij})^2 + \lambda\sum_k(N_{i*}^{-1}\boldsymbol{L}_{ik}^2 + N_{*j}^{-1}\boldsymbol{R}_{kj}^2)$
L_{L2w}	$L_{\text{Sl}} + \lambda\left(\sum_{ik}N_{i*}\boldsymbol{L}_{ik}^2 + \sum_{kj}N_{*j}\boldsymbol{R}_{kj}^2\right)$	$(\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij})^2 + \lambda\sum_k(\boldsymbol{L}_{ik}^2 + \boldsymbol{R}_{kj}^2)$

TABLE 2.2: Popular loss functions for matrix completion

$$L_{\text{Sl}}(\boldsymbol{L},\boldsymbol{R}) = \sum_{(i,j)\in\Omega}(\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij})^2$$

Table 2.2 summarizes other popular loss functions. L_{L2} incorporates L2 regularization and is closely related to the problem of minimizing the nuclear norm of the reconstructed matrix [Recht and Ré, 2013]. L_{L2w} incorporates weighted L2 regularization [Zhou et al., 2008], in which the amount of regularization depends on the number of revealed entries. This particular loss function was a key ingredient in the best performing solutions of both the Netflix competition and the 2011 KDD-Cup [Chen et al., 2012, Koren et al., 2009, Zhou et al., 2008].

Our formulation of the matrix completion problem is motivated by its application in data mining settings, where a fixed set of training data and a loss function are given, and the goal is to compute loss-minimizing factor matrices as efficiently as possible. Candes and Recht [2009] discuss the theoretical foundations of the basic minimization problem. There is also a large body of literature that assumes a "true" underlying $\boldsymbol{V}$ matrix together with a stochastic process that generates from $\boldsymbol{V}$ the observed training data. The goal is then to statistically infer the true $\boldsymbol{V}$ matrix from the training data, where the inference algorithm may exploit knowledge about the stochastic process. In one such model, the entries to be revealed are selected randomly and uniformly from the set of all $\boldsymbol{V}$ entries. Many algorithms and supporting theory have been developed for this specific setting; see, e.g., [Mackey et al., 2011].

In the following, we focus on loss functions that admit a summation form. Following Chu et al. [2006], a loss function is in *summation form* if it is written as a sum of *local losses* L_{ij} that occur only at the revealed entries of $\boldsymbol{V}$, i.e.,

$$L(\boldsymbol{L},\boldsymbol{R}) = \sum_{(i,j)\in\Omega} L_{ij}(\boldsymbol{L}_{i*},\boldsymbol{R}_{*j}), \tag{2.2}$$

where $\boldsymbol{L}_{i*}$ and $\boldsymbol{R}_{*j}$ refer to the i-th row of $\boldsymbol{L}$ and j-th column of $\boldsymbol{R}$, respectively. Table 2.2 shows examples of loss functions in summation form together with the corresponding local

losses. We refer to the gradient of a local loss as a *local gradient*. By the linearity of the differentiation operator, the gradient of a loss function having summation form can be represented as a sum of local gradients:

$$L'(\boldsymbol{L}, \boldsymbol{R}) = \sum_{(i,j)\in\Omega} L'_{ij}(\boldsymbol{L}_{i*}, \boldsymbol{R}_{*j}).$$

In the following, we describe popular algorithms based on stochastic gradient descent (Section 2.2) and alternating projections (Section 2.5.6), which have been shown to be effective in the collaborative filtering setting ([Gemulla et al., 2011c, Recht and Ré, 2013, Zhou et al., 2008]).

2.2 Preliminaries

In the following sections, we present sequential and shared-memory algorithms based on stochastic gradient descent; see Table 2.1 for an overview of our notation.

We first describe the basic SGD algorithm. For brevity, we write $L(\boldsymbol{\theta})$ and $L'(\boldsymbol{\theta})$, where $\boldsymbol{\theta} = (\boldsymbol{L}, \boldsymbol{R})$, to denote the loss function and its gradient. Denote by $\nabla_{\boldsymbol{L}} L$ (resp. $\nabla_{\boldsymbol{R}} L$) the $m \times r$ (resp. $r \times n$) matrix of the partial derivatives of L w.r.t. to the entries in $\boldsymbol{L}$ (resp. $\boldsymbol{R}$). Then $L' = (\nabla_{\boldsymbol{L}} L, \nabla_{\boldsymbol{R}} L)$. For example, $[\nabla_{\boldsymbol{L}} L_{\text{SI}}]_{ik} = -2 \sum_{j:(i,j)\in\Omega_{i*}} \boldsymbol{R}_{kj}(\boldsymbol{V}_{ij} - [\boldsymbol{L}\boldsymbol{R}]_{ij})$, where Ω_{i*} denotes the set of revealed entries in row $\boldsymbol{V}_{i*}$.

2.2.1 Gradient Descent

Various gradient-based methods have been explored in the context of matrix completion. Perhaps the simplest algorithm is gradient descent (GD), which iteratively takes small steps in the direction of the negative gradient:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \epsilon_n L'(\boldsymbol{\theta}),$$

where n denotes the step number and $\{ \epsilon_n \}$ is a sequence of decreasing step sizes. Under appropriate conditions, GD has a linear rate of convergence; better rates can be obtained by using a quasi-Newton method, such as L-BFGS-B [Byrd et al., 1995].

2.2.2 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is based on GD, but uses a noisy estimate $\hat{L}'(\boldsymbol{\theta})$ of the gradient $L'(\boldsymbol{\theta})$. SGD starts with some random initial value $\boldsymbol{\theta}_0$ and then iterates the stochastic difference equation

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \epsilon_n \hat{L}'(\boldsymbol{\theta}), \tag{2.3}$$

The gradient estimate is obtained by scaling up just one of the local gradients, i.e., $\hat{L}'(\boldsymbol{\theta}) = NL'_{ij}(\boldsymbol{\theta})$ for some $(i,j) \in \Omega$. The choice of training point (i,j) varies from step to step according to a *training point schedule*; see below. Note that the local gradients at point (i,j) depend only on $\boldsymbol{V}_{ij}$, $\boldsymbol{L}_{i*}$ and $\boldsymbol{R}_{*j}$. Therefore, only a single row $\boldsymbol{L}_{i*}$ and a single column $\boldsymbol{R}_{*j}$ are updated during each SGD step.

For each pass over the training data, SGD performs many "quick-and-dirty" steps whereas gradient descent (or a quasi-Newton method such as L-BFGS) performs a single careful step. For large matrices, the increased number of SGD steps leads to much faster convergence in practice [Gemulla et al., 2011c]. Moreover, the noisy estimation of the descent direction helps keep the algorithm from getting stuck at a local minimum. Batched versions, in which small batches of local losses are averaged, can also be used, although they often perform inferior in practice [Byrd et al., 2012].

The performance of SGD is largely affected by the step size selection scheme and the training point schedule, which we discuss in the following in more detail.

Step size sequence. Step size sequences of the form $\epsilon_n = 1/n^\alpha$ with $\alpha \in (0.5, 1]$ are often used in the literature, since they guarantee asymptotic convergence[3] [Kushner and Yin, 2003]. However, in practice one may want to deviate from the above choice to achieve faster convergence.

All of our SGD implementations use a simple adaptive method for selecting the step size that has worked extremely well in our experiments, even though guarantees of asymptotic convergence have not been formally established.

We refer to one GD step or a sequence of N SGD steps as an *epoch*; an epoch roughly corresponds to a single pass over the data. Exploiting the fact that the current loss can be computed after every epoch, we employ a heuristic called *bold driver* [Battiti, 1989]. Bold driver starts from an initial step size ϵ_0. After each epoch, the algorithm increases the step size by a small percentage (5%) if the loss has decreased during the epoch, and drastically decreases the step size (by 50%) if the loss has increased. Within each epoch,

[3] "Convergence" refers to running an algorithm until some convergence criterion is met; "asymptotic convergence" means that the algorithm converges to the true solution as the runtime increases to $+\infty$.

the step size remains fixed. The initial step size ϵ_0 is obtained by trying different step sizes on a small sample (say, 0.1%) of Ω and picking the one that works best.

More recent work [Chin et al., 2015] on step size selection suggests to make different steps for each $\boldsymbol{L}_{i*}$ and $\boldsymbol{R}_{*j}$ (different step sizes for different users or items). In more detail, for each training point (i, j) processed, the learning rate for $\boldsymbol{L}_{i*}$ will be $\epsilon_0/\sqrt{G_i}$, where G_i monotonically increases (for each training point in the ith row) according to the update rule $G_i = G_i + \frac{\boldsymbol{g}_i^T \boldsymbol{g}_i}{r}$, where by $\boldsymbol{g}_i$ we denote the gradient of the local loss according to $\boldsymbol{L}_{i*}$. In other words, the learning rate depends on the squared sum of past gradient elements. An experimental comparison of the performance of our methods with the step size sequence of Chin et al. [2015] instead of the bold driver remains for future work.

Training point schedule. The sequence in which we choose training points to be processed largely affects the performance of SGD. Intuitively, one desires a sequence that (1) covers a large part of the training points and (2) is as random as possible. Common schedules for an SGD epoch are:

- SEQ: process Ω sequentially in some fixed order. This schedule covers all training points, but it does not involve any randomness.
- WR: sample with replacement from Ω. This schedule creates the most randomness, but it might not cover all training points.
- WOR: sample without replacement from Ω. Unlike the two extreme schedules SEQ and WR, WOR is a compromise: it covers all training points and provides sufficient randomness.

All the above training point selection schedules guarantee convergence, but they converge within a different number of epochs. In practice, WOR often needs less epochs than WR; SEQ requires even more epochs than WR and may converge to an inferior solution. Nevertheless, SEQ epochs are significantly faster than WR or WOR epochs, because they have better memory locality.

In practice, training point selection schedules with high randomness (WR/WOR) are preferred (we used WOR throughout our experiments), because they converge in less epochs (less passes over the data). However, the random memory access patterns of these schedules increases the cache-miss rate of SGD, which can be a major bottleneck for multicore systems. In the following, we propose cache-friendly SGD-based algorithms for parallel matrix completion in shared-memory, multicore systems.

2.3 SGD++: Sequential SGD with Prefetching

In the following, we will study latency-hiding SGD-based algorithms for matrix completion. This section presents SGD++, a sequential method with prefetching, whereas Section 2.4 discusses CSGD, a parallel, cache-aware method.

One way to improve the cache behavior of the WOR schedule is to prefetch the required data into the CPU cache before it is accessed by the SGD algorithm (e.g., using GCC's `__builtin_prefetch` macro). In the beginning of each epoch, we precompute and store a permutation Π of $\{1, \ldots, N\}$ that indicates the order in which training points are to be processed. In the n-th step, the SGD algorithm accesses the values $\boldsymbol{V}_{i_n^\Pi j_n^\Pi}$, $\boldsymbol{L}_{i_n^\Pi *}$, and $\boldsymbol{R}_{* j_n^\Pi}$, whose common index value (i_n^Π, j_n^Π) is determined from the $\Pi(n)$-th entry of Ω. We access Π and then prefetch the index value (i_n^Π, j_n^Π) during SGD step $n-2$ (so that it is in the CPU cache at step $n-1$), and the values $\boldsymbol{V}_{i_n^\Pi j_n^\Pi}$, $\boldsymbol{L}_{i_n^\Pi *}$, and $\boldsymbol{R}_{* j_n^\Pi}$ in SGD step $n-1$ (so that they are in the CPU cache at step n). Note that Π itself is accessed sequentially so that no explicit prefetching is needed. We refer to SGD with prefetching as SGD++; see Algorithm 1.

In general, software prefetching is optimal only if prefetch requests are sent early enough to fully hide memory latency. However, for SGD, this is difficult to be achieved in practice. The reason is that an iteration of the inner loop of Algorithm 1 is computationally inexpensive. When there is not enough computation between prefetching instructions and demand requests, it can be difficult to insert prefetching instructions at the "right time" to provide timely prefetching requests (see [Lee et al., 2012] for a discussion on when prefetching performs suboptimal). The result is that SGD++ manages to only partially overlap computation and data-fetching time. In our experiments, SGD++ was up to 12% faster than SGD (see Section 2.7.3).

In the next section, we discuss CSGD, a method for cache-aware, parallel SGD. Unlike SGD++, CSGD does not rely on prefetching instructions to reduce the cache-miss rate. Instead, it follows a memory access pattern which makes the system's prefetching more effective. CSGD reached in our experiments approximately up to 60% speedup over SGD.

2.4 CSGD: Cache-aware Parallel SGD

In this section, we describe our method for cache-aware parallel SGD (CSGD) for matrix completion. In the following, we focus on different aspects of the method. We will denote by p the number of available threads in the machine.

Algorithm 1 The SGD++ algorithm for matrix completion

Require: Incomplete matrix $\boldsymbol{V}$, initial values $\boldsymbol{L}$ and $\boldsymbol{R}$
  while not converged **do** // *epoch*
    Create random permutation Π of $\{1, \ldots, N\}$ // *WOR schedule*
    for $n = 1, 2, \ldots, N$ **do** // *step*
      Prefetch indexes $(i^{\Pi}_{n+2}, j^{\Pi}_{n+2}) \in \Omega$ for next but one step
      Prefetch data $\boldsymbol{V}_{i^{\Pi}_{n+1} j^{\Pi}_{n+1}}$, $\boldsymbol{L}_{i^{\Pi}_{n+1} *}$, $\boldsymbol{R}_{* j^{\Pi}_{n+1}}$ for next step
      $\boldsymbol{L}'_{i^{\Pi}_{n} *} \leftarrow \boldsymbol{L}_{i^{\Pi}_{n} *} - \epsilon_n N \nabla_{\boldsymbol{L}_{i^{\Pi}_{n} *}} L_{i^{\Pi}_{n} j^{\Pi}_{n}}(\boldsymbol{L}, \boldsymbol{R})$
      $\boldsymbol{R}_{* j^{\Pi}_{n}} \leftarrow \boldsymbol{R}_{* j^{\Pi}_{n}} - \epsilon_n N \nabla_{\boldsymbol{R}_{* j^{\Pi}_{n}}} L_{i^{\Pi}_{n} j^{\Pi}_{n}}(\boldsymbol{L}, \boldsymbol{R})$
      $\boldsymbol{L}_{i^{\Pi}_{n} *} \leftarrow \boldsymbol{L}'_{i^{\Pi}_{n} *}$
    end for
  end while

2.4.1 Parallel processing

Unfortunately, SGD is not embarassingly parallel. In fact, the SGD updates are dependent on each other. To see this, assume two threads p_1 and p_2 running SGD on matrix $\boldsymbol{V}$. Assume p_1 processes entry (i_1, j_1) and updates factors $\boldsymbol{L}_{i_1 *}$ and $\boldsymbol{R}_{* j_1}$, and p_2 similarly processes entry (i_2, j_2). Whenever $i_1 = i_2$ the two threads will try to update the same row-factor $\boldsymbol{L}_{i_1 *}$ causing a dirty write. Similarly, whenever $j_1 = j_2$ the threads will try to update the same column-factor causing a collision on $\boldsymbol{R}_{* j_1}$.

Many different approaches have been proposed to address this issue. In CSGD, we follow the stratification idea proposed by Gemulla et al. [2011c] (see also Section 2.5.1): we partition the input matrix $\boldsymbol{V}$ into $b \times b$ partitions, also called blocks, where b is chosen to be greater than or equal to the number of available threads. The factor matrices are blocked conformingly, i.e., $b \times 1$ for matrix $\boldsymbol{L}$ and $1 \times b$ for matrix $\boldsymbol{R}$. Each thread, is assigned b/p partitions of $\boldsymbol{L}$ and the corresponding row-partitions of $\boldsymbol{V}$. In this way, no collisions on $\boldsymbol{L}$ are possible.

$$\begin{array}{c} \\ \boldsymbol{L}^1 \\ \boldsymbol{L}^2 \\ \vdots \\ \boldsymbol{L}^b \end{array} \begin{array}{c} \begin{array}{cccc} \boldsymbol{R}^1 & \boldsymbol{R}^2 & \cdots & \boldsymbol{R}^b \end{array} \\ \begin{pmatrix} \boldsymbol{V}^{11} & \boldsymbol{V}^{12} & \cdots & \boldsymbol{V}^{1b} \\ \boldsymbol{V}^{21} & \boldsymbol{V}^{22} & \cdots & \boldsymbol{V}^{2b} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{V}^{b1} & \boldsymbol{V}^{b2} & \cdots & \boldsymbol{V}^{bb} \end{pmatrix} \end{array}.$$

Rows and columns are randomly shuffled prior to blocking, so that each block contains N/b^2 training points in expectation. Now observe that when SGD runs on some block $\boldsymbol{V}^{ij}$, it accesses only the matrices $\boldsymbol{L}^i$ and $\boldsymbol{R}^j$. Thus SGD can, for example, be run independently and in parallel on each of the blocks on the main diagonal (i.e., $\boldsymbol{V}^{11}, \ldots, \boldsymbol{V}^{bb}$); the SGD instances refer to disjoint parts of the factor matrices and will hence yield the

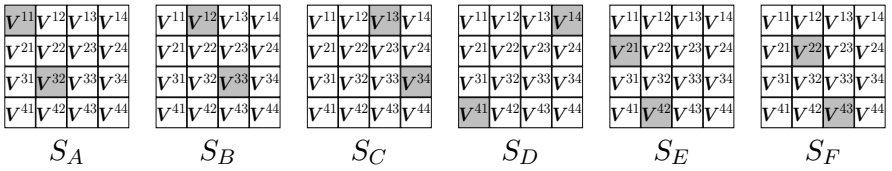


FIGURE 2.1: Examples of strata for a 4×4 blocking of $\boldsymbol{V}$, when $p = 2$ threads are available

same result as processing the main diagonal using sequential SGD. In general, we say that two different blocks $\boldsymbol{V}^{ij}$ and $\boldsymbol{V}^{i'j'}$ are *interchangeable* whenever $i \neq i'$ and $j \neq j'$, i.e., they share neither rows nor columns. We call a set of p pairwise interchangeable blocks a *stratum*, the set of all strata is denoted by $\mathcal{S}$. By the arguments above, the blocks of a stratum can be processed independently and in parallel since they do not share any common rows or columns.

For illustration purposes, let us consider the case when $b = 4$ and $p = 2$. Figure 2.1 shows some of the possible $b!(\frac{b}{p})^p$ strata on a $b \times b$ blocked matrix. It is convenient to view a stratum as a map from a thread k to a block index $(i, j) = S(k)$. For example, $S_B(1) = (1, 2)$, $S_B(2) = (3, 3)$, in the example of Figure 2.1. Stratum S is processed in parallel: thread k processes block $\boldsymbol{V}^{S(k)}$. Continuing the example with $S = S_B$, thread 1 processes block $\boldsymbol{V}^{12}$ and thread 2 processes block $\boldsymbol{V}^{33}$. In what follows, we refer to the processing of a single stratum as a *subepoch* and to a sequence of b^2/p subepochs as an *epoch*. Note that an epoch roughly corresponds to processing N training points: each block contains N/b^2 entries in expectation, we process p blocks per subepoch, and there are b^2/p subepochs per epoch. The CSGD algorithm is described in Algorithm 2: it selects, in each subepoch, p interchangeable blocks (a stratum), processes each block on a separate thread using SGD, and proceeds to the next subepoch.

Algorithm 2 The CSGD algorithm for matrix completion

Require: Incomplete matrix $\boldsymbol{V}$, initial values $\boldsymbol{L}$ and $\boldsymbol{R}$, blocking parameter b, number of threads p
  Block $\boldsymbol{V}$ / $\boldsymbol{L}$ / $\boldsymbol{R}$ into $b \times b$ / $b \times 1$ / $1 \times b$ blocks
  Co-locate points in the same $\boldsymbol{V}$ block in consecutive memory locations
  while not converged **do** *// epoch*
    Pick step size ϵ
    for $s = 1, \ldots, b^2/p$ **do** *// subepoch*
      Pick p blocks $\{\boldsymbol{V}^{S(1)}, \ldots, \boldsymbol{V}^{S(p)}\}$ to form a stratum
      for $k = 1, \ldots, p$ **do** *// in parallel*
        Run SGD on the training points in $\boldsymbol{V}^{S(k)}$ with step size ϵ
      end for
    end for
  end while

More formally, the algorithm decomposes the initial loss $L(\boldsymbol{\theta})$ into a weighted sum of stratum losses

$$L(\boldsymbol{\theta}) = w_1 L_1(\boldsymbol{\theta}) + w_2 L_2(\boldsymbol{\theta}) + \ldots + w_q L_q(\boldsymbol{\theta}), \tag{2.4}$$

where $q = b!(\frac{b}{p})^p$ is the number of possible strata and for the weight w_s of each stratum s, it holds that $0 \leq w_s \leq 1$ and $\sum_{s=1}^{q} w_s = 1$. The stratum weight w_s corresponds to the long-term fraction of steps run on stratum s. In addition, each stratum loss is associated with a stratum-specific constant c_s:

$$L_s(\boldsymbol{\theta}) = c_s \sum_{(i,j)\in\Omega_s} L_{ij}(\boldsymbol{\theta}). \tag{2.5}$$

For Eq. (2.4) and (2.5) to be equivalent to Eq. (2.2) we also require that $w_s c_s = 1$. Therefore, if the algorithm spends more time in a specific stratum s (higher w_s), the corresponding stratum loss will be downweighted (lower c_s). The gradient estimate for the stratum loss becomes:

$$\hat{L}'_s(\boldsymbol{\theta}) = N_s c_s L'_{ij}(\boldsymbol{\theta}). \tag{2.6}$$

Note that, if we set $c_s = N/N_s$ in Eq. (2.6), we fall back to the standard SGD gradient estimate $\hat{L}'_s(\boldsymbol{\theta}) = N L'_{ij}(\boldsymbol{\theta})$.

Since in each subepoch the algorithm takes a step towards the direction of $-\hat{L}'_s(\boldsymbol{\theta})$ and not the correct $-\hat{L}'(\boldsymbol{\theta})$, it is not obvious if it will converge at all. Gemulla et al. [2011a] show that such stratified versions of SGD indeed converge under appropriate regularity conditions on the loss function, the step size sequence and stratum selection sequence (see [Gemulla et al., 2011a] for more details).

2.4.2 Stratum Schedule

Just as the training point schedule of SGD influences its convergence in practice, the stratum schedule influences the convergence properties of CSGD. Formally, a stratum schedule is a (possibly random) sequence $S_1, S_2, \ldots$ of strata from $\mathcal{S}$; We process stratum S_l in the l-th subepoch. In CSGD, we partition matrix $\boldsymbol{V}$ evenly across the p threads, such that each row of blocks is assigned to exactly one core (in Figure 2.1 the first two row-blocks are assigned to thread 1 and the second two to thread 2). The algorithm then picks a stratum to work on. This can happen sequentially (SEQ), e.g., $(S_A, S_B, S_C, \cdots)$, randomly with replacement (WR) e.g., $(S_C, S_B, S_C, \cdots)$ or randomly without replacement (WOR) e.g., $(S_C, S_B, S_F, \cdots)$. We found that in practice WOR achieves the best

results because it randomizes the order of blocks as much as possible while ensuring that every training point is processed in every epoch.

Gemulla et al. [2011a] showed that stratified SGD asymptotically converges to a stationary point of L for similar stratum-selection schedules for the special case, in which $b = p$. We will now generalize for the case that $b > p$. We argue informally, following closely the line of argumentation of Gemulla et al. [2011a]. Gemulla et al. [2011a] showed that stratified SGD converges if the values of w_s and c_s satisfy for each block V^{ij}:

$$\sum_{s:\mathcal{S}\supseteq V^{ij}} w_s c_s = 1. \tag{2.7}$$

We therefore need to show that the SEQ and WR/WOR schedules rely on values of w_s and c_s that satisfy Eq. (2.7). Recall that w_s can be seen as the long-term fraction of steps run on stratum Ω_s or as the probability to select stratum Ω_s and c_s is the weight of the steps that will be taken in stratum Ω_s.

For SEQ, we can set $w_s = N_s/N$, for the b^2/p strata that (if followed sequentially) jointly cover all the training points and $w_s = 0$ for the rest. In other words, each of the strata with non zero weight is chosen equally often with probability proportional to the training points in the stratum. For WR/WOR, let each block V^{ij} appear in d different strata. To have equal probability to choose any of the d strata, we need to set $w_s = N_s/(dN)$ and we take N_s steps while processing stratum s. For the above choices of w_s, it can be shown that Eq. (2.7) holds when $c_s = N/N_s$, and CSGD converges.

2.4.3 Cache-awareness

Cache-awareness is achieved in CSGD by choosing the blocking parameter b, such that each block $\boldsymbol{V}^{ij}$, as well as the corresponding factor matrices $\boldsymbol{L}^i$ and $\boldsymbol{R}^j$, fit into the cache (last level) available per core, i.e., we expect that $b \gg p$. The training points within a block are laid out in consecutive memory locations; if a block is processed using SGD, we expect few cache misses when accessing the data and factor matrices, because of higher locality.

To showcase this, we make a simplified analysis of the cache-miss rates of SGD and CSGD with the WOR training point selection schedule on a single core. Assume the worst case scenario in which a single training point can be stored in one full cache line. Similarly, assume that we need one full cache line to store a row-factor $\boldsymbol{L}_{i*}$ or a column-factor $\boldsymbol{R}_{*j}$. In other words, fetching any of these data into cache implies one cache-miss. Assume that $\boldsymbol{L}$, $\boldsymbol{R}$ and $\boldsymbol{V}$ are all much larger than the available cache size and that the cache is initially empty. SGD will cause one cache miss to bring each one of the N training

points into cache. Also since $\boldsymbol{L}$ and $\boldsymbol{R}$ are very large in comparison to the cache size, the probability to find the currently needed factors into cache is very small. Therefore, we assume it will also cause a cache-miss for each factor. In other words, there will be three cache misses for each training point, i.e. $3N$ in total. On the other hand, each block of CSGD is small enough to fit into cache. This means that we only need to bring the factors into cache once. For all later accesses they will still be in cache. I.e., for each block b with N_b training points, CSGD needs to pay $N_b + m/b + n/b$ cache misses. For the whole matrix $\boldsymbol{V}$, it will cause $\sum_b (N_b + m/b + n/b) = N + m + n$. Hence, the relative cost for SGD vs. CSGD will be:

$$\frac{Cost(SGD)}{Cost(CSGD)} = \frac{3N}{N+m+n} \approx 3.$$

This means that CSGD will have roughly 3x less cache-misses than SGD. Note that this is a worst case analysis. In practice, usually a cache line will fit multiple training points, allowing for prefetching of training points that are highly likely (in the case of CSGD) to be later used.

2.4.4 Synchronization

In order to ensure that no collisions will take place, the original DSGD algorithm [Gemulla et al., 2011c] was forcing the threads to synchronize between subepochs. This means that each subepoch is as slow as the slowest working thread. This is less of a problem when $b = p$ (as in DSGD) and the number of subepochs is relatively small (b subepochs). However, for CSGD, the large number of subepochs (b^2/p), and hence synchronization points, severely limit performance in practice. An alternative procedure is to simply omit synchronization, akin to the PSGD algorithm [Niu et al., 2011] (see Section 2.5.3). In particular, we partition matrix $\boldsymbol{V}$ evenly across the p threads, such that each row of blocks is assigned to exactly one core. This ensures that there will be no inconsistent updates on the row-factor matrix $\boldsymbol{L}$. Each thread then independently processes its blocks in WOR order; inconsistent updates on $\boldsymbol{R}$ may occur, but since $b \gg p$, we expect this to happen rarely.

In our experiments (see Section 2.7) CSGD was the most scalable and best-performing matrix completion algorithm (up to 60% faster than its closest competitor).

2.5 Related Work

In this section, we review related work on parallel matrix completion. We start with SGD-based approaches and continue with methods based on alternating minimizations. Table 2.3 presents an overview of the methods discussed in this section.

2.5.1 Distributed Stratified SGD (DSGD)

As already mentioned in Section 2.4, DSGD was the first method to use stratification for parallelizing SGD in a matrix factorization setting. Although DSGD was originally proposed for shared-nothing ("Distributed") architectures, it is also conceivable for a shared-memory setting. In DSGD the blocking parameter b is hardware dictated and is set to the number of available processing units p. This means that in the general case the blocks are large in comparison to the available cache size and DSGD lacks the cache-awareness exhibited by CSGD. In addition, DSGD (unlike CSGD) synchronizes the threads between subepochs. However this is usually not a problem, since for DSGD (where $b = p$) the number of subepochs per epoch is relatively small (b).

2.5.2 Jellyfish

An alternative approach to DSGD is the Jellyfish algorithm [Recht and Ré, 2013]. As with DSGD, Jellyfish uses a $p \times p$ blocking of the input matrix. Parallel processing is achieved in a manner similar to DSGD: the p available threads work on interchangeable blocks, avoiding fine-grained locking. The main difference to DSGD is that Jellyfish uses a SEQ stratum and training point selection schedule, so that prefetching is very effective. For this reason, it can also be categorized as a cache-aware method. Since sequential stratum and training point selection schedules usually do not lead in fast convergence (in terms of number epochs required to converge), Jellyfish increases the randomness of the SGD by reshuffling the entire data set at the start of each epoch. To speed up data shuffling, Jellyfish maintains s copies of the data, where $s \geq 2$ is a small number. While one copy is being processed, $s - 1$ parallel shuffle threads reorganize the data in the remaining $s - 1$ copies. Jellyfish's disadvantages are that (1) it is memory intensive because multiple copies of the input matrix need to be maintained and (2) parallel shuffling may in practice lead to memory bottlenecks.

2.5.3 Parallel lock-free SGD (PSGD)

As discussed previously, if two SGD steps process training points that lie in the same row i (resp., column j), then both of these steps read and update row $\boldsymbol{L}_{i*}$ (resp., column $\boldsymbol{R}_{*j}$) of the factor matrix. Perhaps the simplest way to parallelize SGD is to partition the training point schedule evenly among the p threads, i.e., each thread runs N/p SGD steps per epoch. To avoid concurrent parameter updates, we lock row i of $\boldsymbol{L}$ and column j of $\boldsymbol{R}$ before processing training point (i, j). This lock-based approach works well when the number p of threads is small (say, $p \leq 8$), but both locking and random memory accesses impede scalability to large numbers of threads. Niu et al. [2011] experimented with a lock-free algorithm, henceforth denoted PSGD, in which no locks are obtained so that inconsistent updates may occur. Since there are usually significantly more rows and columns than available threads (i.e., $m, n \gg p$), it is unlikely that a given row or column is processed by multiple threads at the same time; we thus expect few inconsistent updates. Niu et al. [2011] found virtually no difference between lock-based and lock-free parallel SGD in terms of running time and quality (for matrix completion problems). On the whole, our experiments validate these findings, except that we observed a small performance improvement (of up to 9%) of PSGD over the lock-based approach when using a large number of threads ($p \geq 16$). De Sa, Christopher M and Zhang, Ce and Olukotun, Kunle and Ré, Christopher and Ré, Christopher [2015] recently proposed a lock-free parallel SGD method, termed Buckwild!, that uses low precision arithmetic to achieve lower memory footprint and more effective use of SIMD instructions.

2.5.4 Fast Parallel SGD (FPSGD)

In parallel to CSGD, another stratified method based on DSGD was proposed, termed FPSGD [Zhuang et al., 2013]. FPSGD identifies and addresses the two main problems of DSGD in a shared-memory setting: (i) the random memory access pattern (when a WOR training point schedule is used) (ii) the synchronization between subepochs. The latter is addressed by partitioning the input matrix in at least $(p+1) \times (p+1)$ blocks. In this way, whenever a thread is done with processing a block, there is another "free" block that it can start processing, without causing any collisions. The memory access pattern problem is addressed by using the SEQ training point selection schedule within the blocks. FPSGD uses a random schedule for assigning blocks to threads, however, since within each block it uses the SEQ training point selection schedule, the randomness is relatively limited. Therefore, it usually partitions the input matrix more fine-grained than $(p+1) \times (p+1)$, so that there will be more blocks to choose from and thus, the randomness of the overall process will increase.

Overall, FPSGD and CSGD solve the same problems in different ways: (i) CSGD uses cache-fitting partitions to reduce the cache-misses, while FPSGD uses the SEQ training point selection schedule. (ii) They both employ a fine-grained partitioning of the input matrix, however in CSGD the granularity of the partitioning is dictated by the hardware and the size of the dataset, whereas in FPSGD it has to be tuned, such that the SEQ training point selection schedule will not impede the convergence. An experimental comparison of CSGD and FPSGD remains for future work.

2.5.5 Multi-Level Grid File based Matrix Factorization (MLGF-MF)

MLGF-MF [Oh et al., 2015] is a parallel SGD-based matrix factorization technique that runs on shared-memory and on block-storage devices (like SSD), which makes it appropriate for cases, in which the problem instance does not fit into the memory of the given machine. It uses asynchronous I/O allowing for overlapping of the CPU and I/O processing. Similar to CSGD, the input matrix is partitioned and SGD runs in parallel on interchangeable blocks. However, multi-level grid file (MLGF [Whang et al., 1994]) indexing is used for the partitioning. MLGF partitions each block recursively, if it contains more training points than a pre-specified capacity. This also makes the whole approach more robust, since the threads are expected to spend similar time on each block. CSGD distributes evenly the training points among the blocks by randomly permuting the rows and columns of the input matrix during preprocessing. However, this can be costly, if the input matrix does not fit into memory and one needs to fetch the training points from disk. Therefore, MLGF-MF is a good alternative in this scenario and if only a single machine is available. If the input matrix can fit into memory, Oh et al. [2015] showed experimentally that MLGF-MF performs comparably to FPSGD.

2.5.6 Alternating Least Squares (ALS)

ALS alternates between optimizing for $\boldsymbol{L}$ given $\boldsymbol{R}$, and optimizing for $\boldsymbol{R}$ given $\boldsymbol{L}$. For the loss function L_{Sl}, this amounts to solving a set of least squares problems, one for each row of $\boldsymbol{L}$ and one for each column of $\boldsymbol{R}$:

$$\forall i,\ \boldsymbol{V}_{i*} - \underline{\boldsymbol{L}_{i*}}\boldsymbol{R}^{(n)} = \boldsymbol{0} \tag{2.8}$$

$$\forall j,\ \boldsymbol{V}_{*j} - \boldsymbol{L}^{(n+1)}\underline{\boldsymbol{R}_{*j}} = \boldsymbol{0}, \tag{2.9}$$

where the unknown variable is underlined and $\boldsymbol{R}^{(n)}$ ($\boldsymbol{L}^{(n+1)}$) is $\boldsymbol{R}$ ($\boldsymbol{L}$) during the nth $(n+1)$ iteration. Loss functions L_{L2} and L_{L2w} can also be handled [Zhou et al., 2008].

During the computation of least squares solutions, matrix $\boldsymbol{V}$ is accessed by row when updating $\boldsymbol{L}$ (Eq. (2.8)), and by column when updating $\boldsymbol{R}$ (Eq. (2.9)). For this reason, ALS implementations need to store two (sparse) representations of $\boldsymbol{V}$ in memory: one in row-major order (denoted $\boldsymbol{V}_{\text{r}}$) and one in column-major order (denoted $\boldsymbol{V}_{\text{c}}$). An ALS epoch has time complexity $O(Nr^2 + (m+n)r^3)$.

Parallel ALS (PALS). A shared-memory parallel algorithm for ALS, denoted PALS, is based on the observation that the foregoing least-squares problems can be solved independently [Zhou et al., 2008]. E.g., an update to a row of $\boldsymbol{L}$ does not affect other rows of $\boldsymbol{L}$, so that the processing of rows of $\boldsymbol{L}$ can be partitioned evenly among the available threads. Processing of columns of $\boldsymbol{R}$ can similarly be partitioned among threads. Our implementation of PALS differs from the algorithm of Zhou et al. [2008] only in that it uses multiple threads (which share the same memory space and variables) instead of multiple processes (each with its own address space), which allows PALS to reduce memory consumption.

2.5.7 Cyclic Coordinate Descent (CCD++)

Cyclic coordinate descent (CCD) can also be seen as an alternating minimization method: it optimizes a *single* entry of one of the factor matrices at a time, while keeping all other entries fixed. This leads to a much simpler minimization problem than that of ALS. Practical variants of CCD adopt the approach of "hierarchical" ALS [Cichocki and Phan, 2009] in that they do not operate on the original input matrix, but on the residual matrix $\boldsymbol{E}$ whose entries are $\boldsymbol{V}_{ij} - [\boldsymbol{LR}]_{ij}$ for $(i,j) \in \Omega$.

Recently, a version of CCD, termed CCD++, has been proposed for matrix completion problems [Yu et al., 2012]. Similar to ALS, CCD++ stores two copies of the residual matrix $\boldsymbol{E}$: one in row-major order (to update $\boldsymbol{L}$), denoted $\boldsymbol{E}_{\text{r}}$, and one in column-major order (to update $\boldsymbol{R}$), denoted $\boldsymbol{E}_{\text{c}}$. CCD++ employs a feature-wise sequence of updates, i.e., each iteration loops over all features $f \in [1, r]$. For each feature f, the algorithm executes T update operations, where T is an automatically tuned parameter that is independent of the data size, and each operation updates the fth feature-vector of $\boldsymbol{L}$ (i.e., $\boldsymbol{L}_{*f}$) and then the fth feature-vector of $\boldsymbol{R}$ (i.e., $\boldsymbol{R}_{f*}$). Finally, both copies of the residual matrix are updated and the algorithm continues with feature $f+1$. For each feature, the residual matrix is scanned $2T$ times to update the corresponding feature vectors of $\boldsymbol{L}$ and $\boldsymbol{R}$, and twice to update $\boldsymbol{E}_{\text{r}}$ and $\boldsymbol{E}_{\text{c}}$. An iteration, i.e., the processing of all features, therefore consists of $2r(T+1)$ epochs and has overall time complexity

of $O(TNr)$. Overall, our experiments, as well as results of Yu et al. [2012], indicate that CCD++ is computationally less expensive than ALS and can handle larger ranks efficiently.

Parallel CCD++ (PCCD++). Parallel versions of CCD++ [Yu et al., 2012] are based on ideas similar to those used for parallelizing ALS; in particular, they make use of a similar partitioning of the factor matrices. We refer to the shared-memory variant as PCCD++.

2.5.8 Parallel Collective Matrix Factorization (PCMF)

Parallel Collective Matrix Factorization (PCMF) [Rossi and Zhou, 2015] improves over PCCD++ in a variety of ways. First, it minimizes the time the threads need to wait for each other (i) by removing the synchronization barriers when switching from updating $\boldsymbol{L}_{*f}$ to updating $\boldsymbol{R}_{f*}$ and (ii) by enabling a work-stealing strategy between threads. Apart from the basic matrix factorization, PCMF can be used for joint (collective) matrix factorization of the main input matrix together with matrices carrying additional information. This is useful, for example in a recommender system scenario, when apart from the ratings matrix, we have additional information about the users or items in a matrix form (e.g., user-user friendship matrix or item-item similarity matrix). A comparison of our work with PCMF remains as a direction for future work.

2.5.9 Algorithms for Distributed Matrix Factorization

This thesis focuses on algorithms for shared-memory architectures. However, when the problem instance cannot fit into the memory of single machine or if a single machine does not have enough processing power to provide the necessary speedup, methods for distributed matrix completion can be used. For completeness reasons, we provide here a short overview of such algorithms.

As we already mentioned in Section 2.5.1, Gemulla et al. [2011c] were the first to use stratification for parallel processing in a distributed environment (initially for MapReduce, later also for a small cluster of commodity nodes). Their method, DSGD, stores a partition of $\boldsymbol{L}$ and a row-partition of $\boldsymbol{V}$ locally in each node, whereas the partitions of $\boldsymbol{R}$ are subject to communication (under the assumption that $\boldsymbol{R}$ is smaller than $\boldsymbol{L}$. Otherwise the roles of $\boldsymbol{R}$ and $\boldsymbol{L}$ are reversed). A basic difference to CSGD is that the partitioning scheme in DSGD is dictated by the number of the available processing units. Teflioudi et al. [2012] and Yun et al. [2014] proposed the DSGD++ and NOMAD algorithms, respectively, which were designed to run in-memory on a small cluster of

commodity nodes. Both are based on stratification and exploit thread-level parallelism, asynchronous communication, and overlap communication and computation.

Distributed versions of ALS and CCD++ have also been proposed. The distributed ALS (DALS) and CCD++ (DCCD++) follow closely the rationale of their shared-memory versions. In DALS, each machine stores localy $\boldsymbol{L}$, $\boldsymbol{R}$ and a row-partition of $\boldsymbol{V}$ twice (once in column-major and once in row-major order). When optimizing $\boldsymbol{L}$, each node operates on the part of $\boldsymbol{L}$ that corresponds to the row-partition of $\boldsymbol{V}$ locally stored and, at the end of the iteration, sends it to the other nodes, so that all nodes will have the same $\boldsymbol{L}$ locally stored before the iteration for optimizing $\boldsymbol{R}$ begins. The distribution of CCD++ takes place in a similar way to DALS, with the difference that DCCD++ broadcasts only a single feature-vector instead of the entire factor matrix as in DALS. However, this communication is performed every time a feature-vector is being updated, i.e., $2Tr$ times per iteration.

2.6 Discussion

Table 2.3 summarizes the properties of the parallel methods discussed in this chapter. Regarding the runtime complexity, ALS is generally much more expensive than SGD, because it needs to solve a large number of linear least squares problems. This computational overhead is acceptable, however, when the rank of the factorization is sufficiently small (say, $r \leq 50$). Moreover, both ALS and CCD++ are usually more memory-intensive than SGD, since they need to store the data matrix twice. An exception of an SGD-based method that keeps in memory multiple copies of the input data is Jellyfish. This means that these memory-intensive methods are appropriate for smaller problem instances than SGD-based methods, when only shared-memory architectures are available. An advantage of both ALS and CCD++ over SGD is that the former methods are parameter-free, whereas SGD methods make use of a step size sequence. Our experiments suggest, however, that SGD is the method of choice when the step size sequence is chosen judiciously, e.g., using the bold driver method of Section 2.2.2. Finally, SGD-based methods apply to a wide range of loss functions, whereas ALS and CCD++ target quadratic loss functions. In the next section, we compare these methods in an extensive experimental study.

2.7 Experimental Study

We conducted an experimental study and compared all algorithms along the following dimensions: the time per epoch (excluding loss computation), the number of epochs

Method	Partitioning	Memory consumption	Epochs/iteration	Time/iteration
PALS [Zhou et al., 2008]	$\boldsymbol{V}$ & $\boldsymbol{L}$ by rows, $\boldsymbol{V}$ & $\boldsymbol{R}$ by columns	$2\boldsymbol{V}+\boldsymbol{L}+\boldsymbol{R}$	2	$O(p^{-1}[Nr^2+(m+n)r^3])$
PCCD++ [Yu et al., 2012]	$\boldsymbol{E}$ & $\boldsymbol{L}$ by rows, $\boldsymbol{E}$ & $\boldsymbol{R}$ by columns	$2\boldsymbol{E}+\boldsymbol{L}+\boldsymbol{R}$	$2r(T+1)$	$O(p^{-1}TNr)$
PSGD [Niu et al., 2011]	$\boldsymbol{V}$ & $\boldsymbol{L}$ by rows	$\boldsymbol{V}+\boldsymbol{L}+\boldsymbol{R}$	1	$O(p^{-1}Nr)$
Jellyfish [Recht and Ré, 2013]	$\boldsymbol{V}$ blocked, $\boldsymbol{L}$ by rows, $\boldsymbol{R}$ by columns	$s\boldsymbol{V}+\boldsymbol{L}+\boldsymbol{R}$	1*	$O(p^{-1}Nr)$*
DSGD [Gemulla et al., 2011c]	$\boldsymbol{V}$ blocked, $\boldsymbol{L}$ by rows, $\boldsymbol{R}$ by columns	$\boldsymbol{V}+\boldsymbol{L}+\boldsymbol{R}$	1	$O(p^{-1}Nr)$
CSGD [Makari et al., 2015] (new)	$\boldsymbol{V}$ blocked, $\boldsymbol{L}$ by rows, $\boldsymbol{R}$ by columns	$\boldsymbol{V}+\boldsymbol{L}+\boldsymbol{R}$	1	$O(p^{-1}Nr)$
FPGD [Zhuang et al., 2013] (subsequent)	$\boldsymbol{V}$ blocked, $\boldsymbol{L}$ by rows, $\boldsymbol{R}$ by columns	$\boldsymbol{V}+\boldsymbol{L}+\boldsymbol{R}$	1	$O(p^{-1}Nr)$

*Excluding data shuffling processes.

TABLE 2.3: Overview of shared-memory methods (see Table 2.1 for notation)

	m	n	N	Size	L	λ
Netflix	480k	18k	99M	2.2GB	L_{L2w}	0.05
KDD	1M	625k	253M	5.6GB	L_{L2w}	1
Syn1B-rect	10M	1M	1B	22.3GB	L_{Sl}	-
Syn1B-sq	3.4M	3M	1B	22.3GB	L_{Sl}	-

TABLE 2.4: Summary of datasets

required to converge, and the total time to converge (including loss computations). Recall that an epoch corresponds roughly to a single pass over the input data, so that the number of epochs reflects the number of data scans. When comparing two algorithms A and B in an experiment, we say that A is more *compute-efficient* than B if it needs less time per epoch, more *data-efficient* if needs fewer epochs to converge, and *faster* if it needs less total time.

2.7.1 Overview of Results

In the sequential setting, CCD++ was the fastest method on the relatively small Netflix dataset (up to 3.4x faster than CSGD), but its performance dropped significantly when used on the larger KDD dataset (up to 3.7x slower than CSGD).

In the shared-memory setting, CSGD outperformed all alternative methods on both real and synthetic datasets: It was up to 15.6x faster than PALS, up to 2.5x faster than PSGD, and up to 5.7x faster than PCCD++. CSGD also showed better scalability than PSGD, Jellyfish, and PCCD++ in that we could use more parallel threads before hitting the memory bandwidth. PALS was the most data-efficient but also the least compute-efficient method, whereas PCCD++ was least data-efficient but most compute-efficient. The SGD-based approaches lay in-between.

2.7.2 Experimental Setup

Implementation. We implemented SGD, SGD++, ALS, PSGD, PALS and CSGD in C++. For CCD++ and PCCD++, we used the C++ implementation provided by Yu et al. [2012]. For Jellyfish, we used the C++ implementation of Recht and Ré [2013], but incorporated the bold driver heuristic for step size selection to ensure a fair comparison. In all our experiments, we used $s = 3$ threads (we did not see significant differences for other choices of s). For CSGD, we chose the blocking parameter b such that one partition of the data and the corresponding factors barely fit into the cache available for one core. We used the GNU scientific library (GSL) for solving the least-squares problems

of ALS; in our experiments, GSL was significantly faster than LAPACK and, in contrast to LAPACK, also supports multithreading.

Hardware. We used two different hardware configurations in our experiments: a machine with 48GB of main memory and an Intel Xeon 2.40GHz processor with 8 cores for the sequential setting and a powerful high-memory server with 512GB of main memory and 4 Intel Xeon 2.40GHz processors with 10 cores each (40 in total) for the shared-memory setting.

Real-world datasets. We used two real-world datasets: Netflix and KDD. The Netflix dataset, which occupies 2.2GB of main memory, consists of roughly 99M ratings of 480k Netflix users for 18k movies; rating values range from 1 to 5. The KDD dataset, which occupies 5.5GB of main memory, corresponds to that of Track 1 of the 2011 KDD-Cup and consists of approximately 253M ratings of 1M Yahoo! Music users for 625k musical pieces. Netflix and KDD differ significantly in the value of $\bar{N}$ (large for Netflix, small for KDD). Detailed statistics for these datasets, as well as for the synthetic datasets described below, are summarized in Table 2.4. For both real-world datasets, we used the official validation sets and focused on L_{L2w} because it performs best in practice [Chen et al., 2012, Koren et al., 2009, Zhou et al., 2008]. We did not tune the regularization parameter λ for varying choices of rank r, but used the values given in Table 2.4 throughout.

Synthetic datasets. For our large-scale experiments, we generated two synthetic datasets that differ in the choice of m and n. We generated each dataset by first creating two rank-50 matrices $\boldsymbol{L}^*_{m\times 50}$ and $\boldsymbol{R}^*_{50\times n}$ with entries sampled independently from the Normal$(0, 10)$ distribution. We then obtained the data matrix by sampling N random entries from $\boldsymbol{L}^*\boldsymbol{R}^*$ and adding Normal$(0, 1)$ noise. Note that the resulting datasets are very structured. We use them here to test the scalability of the various algorithms; the matrices can potentially be factored much more efficiently by exploiting their structure directly. We generated two large datasets with 1B revealed entries and identical sparsity: Syn1B-rect is a tall rectangular matrix, Syn1B-sq is a square matrix. Note that we need to learn more parameters to complete Syn1B-rect (550M) than to complete Syn1B-sq (320M).

Methodology. For all datasets, we centered the input matrix around its mean. To investigate the impact of the factorization rank, we experimented with ranks $r = 50$ and $r = 100$; in practice, values of up to $r = 1000$ can be beneficial [Zhou et al., 2008]. The starting points $\boldsymbol{L}_0$ and $\boldsymbol{R}_0$ were chosen by taking i.i.d. samples from the Uniform$(-0.5, 0.5)$ distribution; the same starting point was used for each algorithm to ensure a fair comparison. Note that, for a given initial point $(\boldsymbol{L}_0, \boldsymbol{R}_0)$ with $\boldsymbol{R}_0 \neq \boldsymbol{0}$, CCD++ needs to compute the residual matrix once at the beginning of the algorithm. In our experiments, the time required for computing the residual matrix was negligible

	Bold driver	Standard(1)	Standard(0.6)
Epochs	40	36	42
Loss (x10^7)	**7.936**	9.267	8.469

TABLE 2.5: SGD step size sequence (Netflix, $r = 50$)

(always less than 0.05% of the total time) and we do not include this overhead in our experimental results. For all SGD-based algorithms, we selected the initial step size based on a small sample of the data (1M entries): 0.0125 for Netflix ($r = 50$), 0.025 for Netflix ($r = 100$), 0.00125 for KDD ($r = 50$, $r = 100$) and 0.000625 for Syn1B-rect and Syn1B-sq. Unless otherwise stated, we used the bold driver heuristic for step size selection with a step-size-increase factor of 5% and a step-size-decrease factor of 50%; step size selection was thus fully automatic. We used the WOR training point schedule and the WOR stratum schedule (unless otherwise stated) throughout our experiments and ran a truncated version of SGD that clipped the entries in the factor matrices to $[-100, 100]$ after every SGD step. Also, unless stated otherwise, all SGD-based algorithms make use of prefetching as in the SGD++ algorithm of Section 2.2.2. For each algorithm, we declared convergence as soon as it reached a point within 2% of the overall best solution.

2.7.3 Sequential Algorithms

We start with a discussion of sequential algorithms, which form a baseline for the parallel methods. Although Jellyfish and CSGD are originally designed for parallel computation, we included them in these experiments in order to investigate whether they can provide significant speed-up over SGD due to their cache-friendliness.

SGD step size sequence. (Table 2.5). In this experiment, we compared the performance of various step size sequences for SGD on the Netflix data for $r = 50$. In Table 2.5, Standard(α) refers to a sequence of form ϵ_0/n^α, where n denotes the epoch and α is a parameter that controls the rate of decay; such sequences are commonly used in stochastic approximation. For this experiment only, we declared SGD as converged if its improvement in loss after one epoch falls below 0.1%. For SGD with the bold driver heuristic, we checked for convergence only in epochs following a drop in step size. We found that the bold driver heuristic significantly outperformed the standard sequences, even though it does not guarantee asymptotic convergence. For example, on Netflix, all step size sequences converged in roughly the same number of epochs, but the bold driver sequence converged to a significantly better factorization.

SGD, SGD++, ALS, CCD++, Jellyfish, CSGD. (Figure 2.2). Overall, on the relatively small Netflix dataset, CCD++ was the best-performing method, but its performance quickly deteriorated for bigger datasets; on KDD, $r = 50$, it needed 2.73h to converge, whereas SGD, the next best method, required only 1.22h. On KDD, CSGD was the fastest method (0.67h and 1.55h for $r = 50$ and $r = 100$, respectively) followed by Jellyfish (0.99h and 1.71h, respectively).

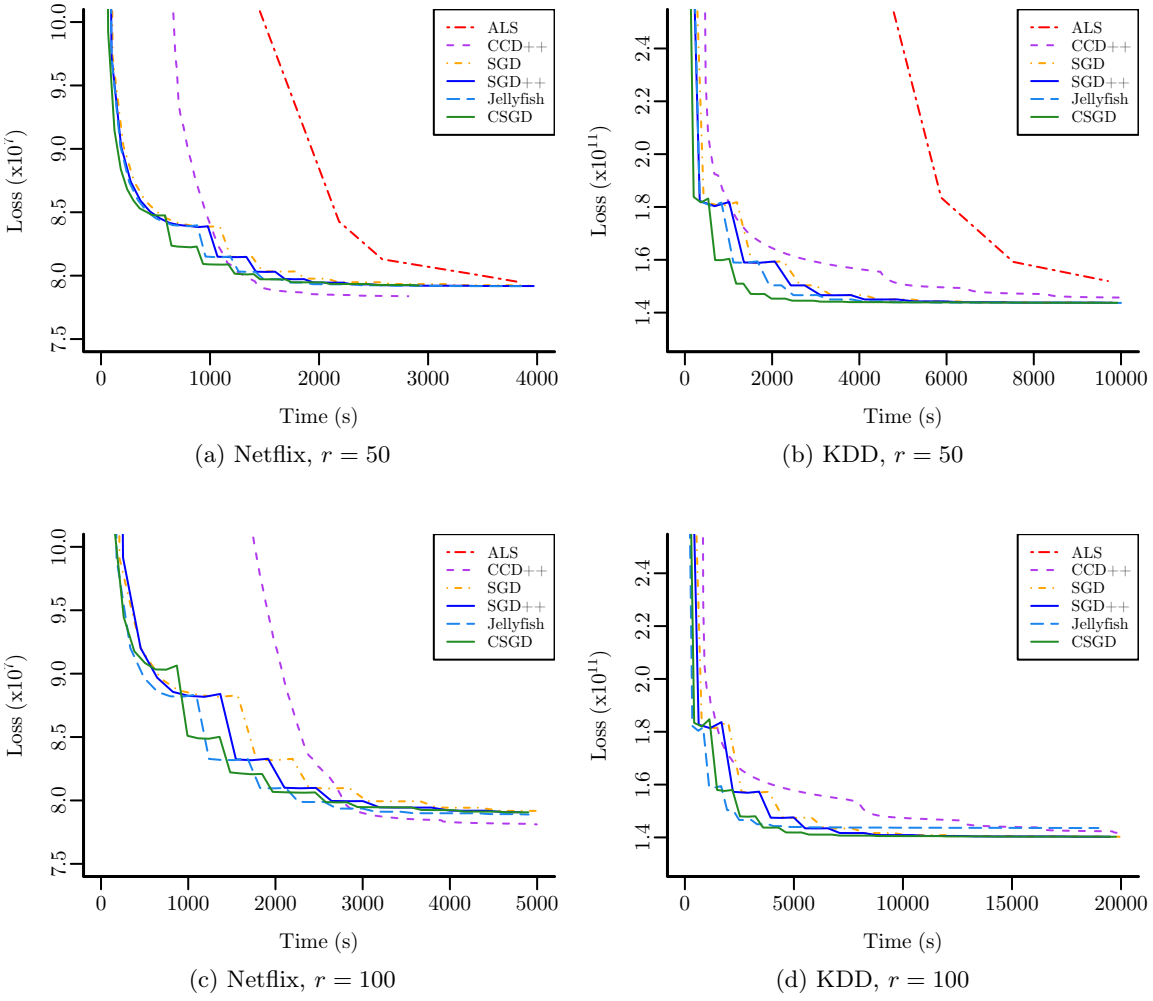
In more detail, in terms of compute-efficiency, we found that SGD++ is up to 12% better than SGD (7.4 vs. 8.4min for KDD, $r = 100$) so that prefetching is beneficial. Jellyfish and CSGD were significantly more compute-efficient than SGD because of their cache-friendly behavior: Jellyfish was 25% (31%) better than SGD on Netflix (KDD) $r = 50$ and CSGD 43% (57%). However, we observed that the stratification used in CSGD decreases the data-efficiency which in turn negatively affects the total time until convergence. This effect diminishes, however, when using a more fine-grained stratification. On the coarse-grained Netflix dataset ($r = 100$) CSGD needed 28 epochs (vs. 18 for SGD), whereas on the more fine-grained KDD ($r = 100$) it needed 16 epochs (vs. 13 for SGD) to converge.

CCD++ was consistently faster than ALS (7x for Netflix and 4.3x for KDD, both $r = 100$). However, CCD++ was less data-efficient than ALS (3136 vs. 5 epochs for Netflix and 4856 vs. 8 epochs for KDD, both $r = 100$). Recall from Section 2.5.7 that CCD++ needs to scan the input matrix $2r(T + 1)$ times in every iteration, i.e., for one full pass over the factors. Therefore, CCD++ requires several epochs, however, every epoch is very inexpensive (each of which takes $O(N)$ time). On the contrary, ALS performs few expensive epochs, each of which requires $O(Nr^2 + (m + n)r^3)$ time. Similarly, SGD++ was less data-efficient than ALS, since it needed up to 4 times more epochs to converge (e.g., 26 vs. 7 epochs for Netflix, $r = 50$). However, SGD++ epochs are more compute-efficient so that SGD++ was faster overall. This effect is strongest when r is high; e.g., SGD++ is ≈5.5x faster for Netflix, $r = 100$ (52.8 vs. 289.8min) but only 2x faster for $r = 50$ (38.2 vs. 78.2min).

2.7.4 Shared-Memory Algorithms on Small and Medium Datasets

In this section, we evaluate our shared-memory methods on datasets of small (Netflix) and medium size (KDD) on our high-memory server. We ran experiments using 8, 16, and 32 threads and refer to these setups as H8, H16, and H32, respectively. The results can be seen in Figure 2.3.

We initially experimented with PSGD both with and without locking. We found that in settings with a large number of threads (H16 and H32), lock-free PSGD was slightly more compute-efficient than PSGD with locking (e.g., 9% for KDD, $r = 100$) without



(a) Netflix, $r = 50$

(b) KDD, $r = 50$

(c) Netflix, $r = 100$

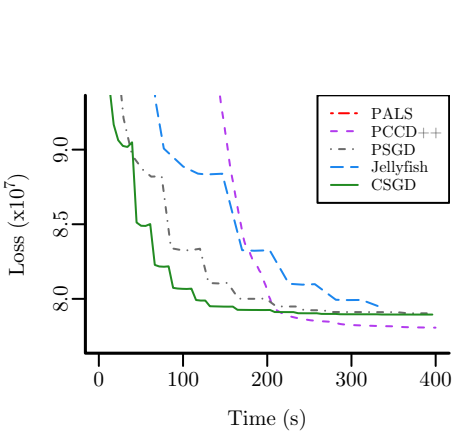
(d) KDD, $r = 100$

FIGURE 2.2: Performance of sequential algorithms

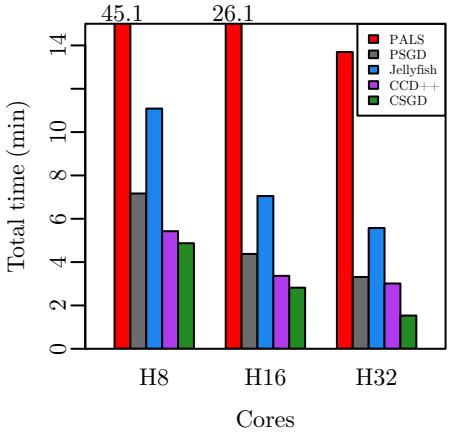
any degradation in the quality of the solution. We presume that this speedup originates from the reduced synchronization costs of lock-free PSGD, which we used throughout our experiments.

Figures 2.3b and 2.3d show the time until convergence on Netflix and KDD, $r = 100$, for various methods and setups. All approaches led to a similar overall loss (all within 1% of each other). Figure 2.3a and 2.3c show examples of the progress of the methods over time. On Netflix, PCCD++ and PALS found a slightly better solution (1% lower loss) than the SGD-based approaches; on KDD, results were almost identical.

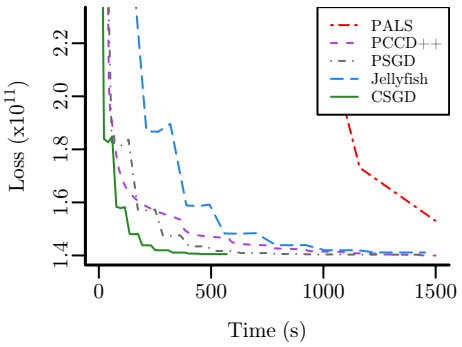
Across all setups and datasets, PALS was the slowest and CSGD the fastest method. The compute-efficiency of PALS was very low so that its epochs were significantly slower than that of alternative methods. In particular, on Netflix, PALS performed only 3 epochs within 400s and was not able to reach the vicinity of the points obtained by the other methods (for this reason, the PALS curve is not visible in Figure 2.3a). PCCD++



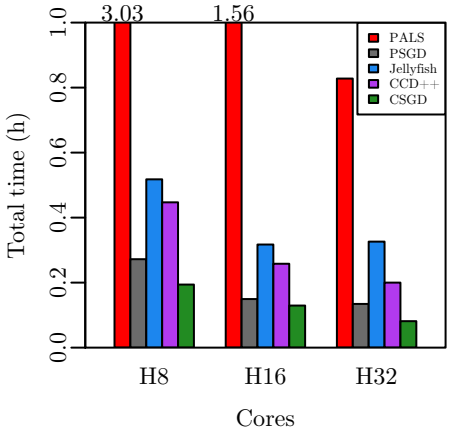
(a) Netflix on H32



(b) Netflix on H8, H16, and H32



(c) KDD on H32



(d) KDD on H8, H16, and H32

FIGURE 2.3: Performance of shared-memory algorithms on real datasets, $r = 100$

performed better than PALS, and it outperformed PSGD on the Netflix dataset (but not on KDD). Jellyfish was slower than PSGD throughout; potentially because worker and shuffle threads compete for memory bandwidth and CPU cache. For this reason, we excluded Jellyfish from our larger-scale experiments. Finally, CSGD was faster than its closest competitors (PCCD++ on Netflix, PSGD on KDD); its cache-conscious blocking thus appears to be effective.[4]

PCCD++ was the least data-efficient, and PALS the most data-efficient method. In more detail, PCCD++ required over 600 times more epochs than PALS to converge (e.g., 4856 vs. 8 epochs on KDD). Nevertheless, PCCD++ was consistently faster than PALS in terms of overall runtime as it is more compute-efficient (see also the complexity discussion in the previous section). Similarly, PSGD and CSGD needed more epochs

[4]Note that our results w.r.t. the relative performance of PSGD and PCCD++ differ from the ones in Yu et al. [2012], presumably due to our use of the bold driver heuristic.

than PALS but fewer than PCCD++ to converge (e.g., 13 epochs for PSGD on KDD); the compute-efficiencies of PSGD and CSGD lie between those of PALS and PCCD++.

Overall, we found that CSGD was the best-performing method on both datasets; CSGD on H16 was faster than any other method on H32. On Netflix, PCCD++ was closest (and slightly better in terms of quality); on KDD, PSGD was closest.

Scalability. (Figures 2.3d, 2.3b). The overall runtime of all algorithms improved significantly as we moved from 8 to 16 threads (1.6x–1.7x speedup on Netflix, 1.5x–1.9x on KDD). When we increased the number of threads further to $t = 32$, all methods still benefited on Netflix, but only PALS and CSGD gave good speedups on KDD (1.6–1.9x, versus $\leq$1.3x speedup for other methods). In general, less CPU-intensive methods such as PCCD++, PSGD and Jellyfish hit the memory bandwidth on KDD. An exception is CSGD (1.6x speedup from H16 to H32), which avoids the memory bottleneck due to better cache utilization. Overall, PALS and CSGD scaled best w.r.t. the number of threads.

Effect of stratification granularity and synchronization on CSGD. (Figure 2.4) To showcase the effect of the stratification granularity and the synchronization on the performance of CSGD, we ran CSGD with different stratification granularities on the KDD dataset ($r = 100$) on a machine with 48GB of main memory and an Intel Xeon 2.40GHz processor with 8 cores. We experimented with the most coarse-grained partitioning for the 8 available threads (8x8), the cache-fitting partitioning (625x625) and an even more fine-grained partitioning (1500x1500). Additionally, we experimented with and without synchronizing the threads between subepochs. Figure 2.4 shows that for coarse-grained partitioning, the compute-efficiency of the synchronized and the non-synchronized versions is similar (63.56s vs. 61.9s per epoch for 8x8), whereas for fine-grained stratification synchronization becomes a bottleneck: for stratification 1500x1500, the synchronized version was 3.7x less compute-efficient than the non-synchronized one (163.18s vs. 43.41s per epoch for 1500x1500). We also see that CSGD with cache-fitting strata (625x625) is significantly more compute-efficient than the simple 8x8 stratification (39.1s vs. 61.9s per epoch on average) and also faster than the 1500x1500 stratification (43.41s). In addition, both the synchronized and non-synchronized versions tend to converge to a solution of similar quality, i.e., collisions in the non-synchronized versions rarely occur or they do not affect the solution in practice.

2.7.5 Shared-Memory Algorithms on Large Datasets

In this section, we evaluate our shared-memory methods on large datasets (Syn1B-sq and Syn1B-rect) on our high-memory server. We ran experiments using 16 and 32 threads

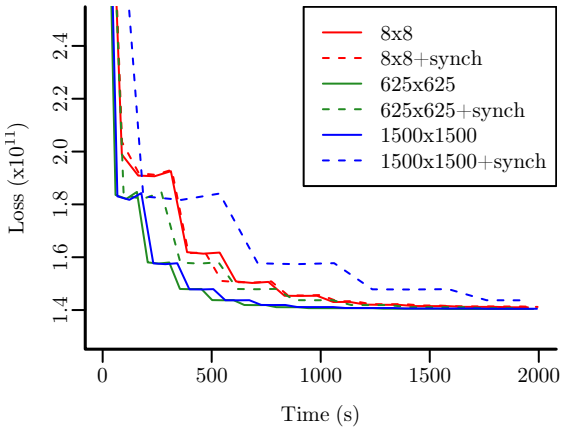
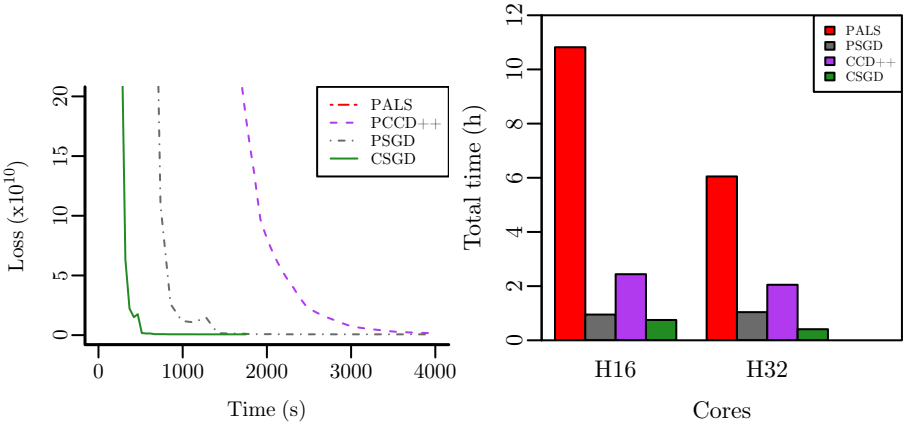


FIGURE 2.4: Impact of stratification granularity and synchronization on CSGD with 8 threads, on KDD $r = 100$



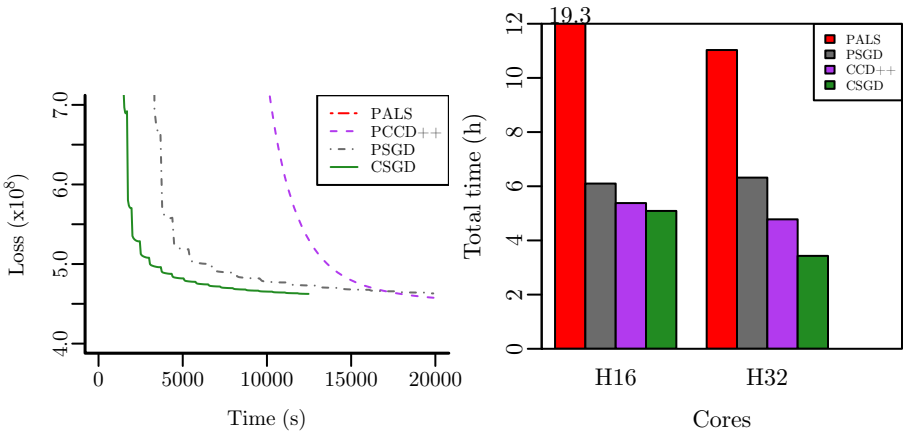
(a) Syn1B-sq on H32 (b) Syn1B-sq

FIGURE 2.5: Performance of shared-memory algorithms on Syn1B-sq

and refer to these setups as H16 and H32, respectively.

Syn1B-sq (Figure 2.5). On the larger Syn1B-sq dataset, CSGD was again the best-performing method. Using 16 (resp., 32) threads, it was 1.3x (resp., 2.5x) faster than PSGD, the second-best-performing method (0.75h vs. 0.95h for H16; 0.41h vs. 1.04h for H32). As with KDD, PSGD was significantly faster than PCCD++. The scalability behavior of the various algorithms was also similar to that for KDD: When moving from 16 to 32 threads, PALS and PCCD++ achieved 1.8x and 1.2x speedup, respectively, PSGD became slower, and CSGD achieved a 1.8x speedup. As before, CSGD with 16 threads was faster than any other method with 32 threads (see Figure 2.5b).

Syn1B-rect (Figure 2.6). On Syn1B-rect, all methods were slower than on Syn1B-sq, presumably because there are more factors to learn. One striking result is that PALS did not converge to an acceptable solution, its loss being four orders of magnitude greater than all other methods. We therefore report the running time of PALS until its loss



(a) Syn1B-rect on H32 (b) Syn1B-rect

FIGURE 2.6: Performance of shared-memory algorithms on Syn1B-rect

changed by less then 0.1% in two consecutive epochs. (Such erratic behavior of PALS was not observed on any other dataset.) As with Netflix, when moving from 16 to 32 threads, only PALS and CSGD yielded good speedups. Specifically, the speedups for PCCD++ and CSGD were 1.13x and 1.5x, respectively, and PSGD actually slowed down. Also as with Netflix, PCCD++ was somewhat faster than PSGD. In general, on Syn1B-rect, PCCD++ behaved better relative to alternative methods than it did on Syn1B-sq. A potential reason for this behavior is that the SGD-based methods are less data-efficient on Syn1B-rect. In particular, we observed that the SGD-based methods required only a small number of epochs to move to the vicinity of the solution, but converged slowly afterwards. Nevertheless, CSGD was the fastest method. It achieved a 25% (57%) speedup in compute-efficiency over PSGD and ran 16.5% (45%) faster than PSGD on H16 (H32).

Overall, CSGD was the fastest method both on KDD and our synthetic datasets. PSGD was the second-fastest method on KDD and Syn1B-sq, while PCCD++ was the second-fastest method on Syn1B-rect and, on this dataset, was competitive with CSGD. PCCD++ was consistently faster (up to 3.5x) than PALS. On the other hand, being compute-intensive and slow, PALS was one of the most scalable methods; its performance always improved by adding more threads.

2.8 Summary

Matrix completion techniques have recently gained attention in the data mining community: they have been successfully applied in a variety of tasks, including recommender systems, open information extraction, latent semantic indexing and link prediction.

In this chapter, we studied parallel algorithms for large-scale matrix completion with millions of rows, millions of columns, and billions of revealed entries for shared-memory architectures. Our method, CSGD, is a stratified version of stochastic gradient descent, which makes use of a fine-grained partitioning that reduces the cache-miss rate during processing. In addition, CSGD is a lock-free approach, i.e., there is no communication imposed between the worker threads. We compared CSGD with a variety of parallel matrix completion algorithms in an extensive set of experiments on both real-world and synthetic datasets of varying sizes. On medium and large datasets CSGD consistently outperforms alternative approaches in terms of speed and scalability and scales better on a large number of processors.

Chapter 3

Exact and Approximate Maximum Inner Product Search

In this chapter[1], we study exact and approximate methods for maximum inner product search (MIPS). Given a large database of real-valued probe vectors as well as a query vector, the MIPS problem is to find all probe vectors that have a large inner product with the query vector. MIPS is a fundamental problem in a number of data mining and information retrieval tasks. Prominent applications include finding good recommendations in recommender systems [Koenigstein et al., 2012, Koren et al., 2009], reasoning about extracted facts in open relation extraction [Riedel et al., 2013], multi-class or multi-label prediction with hundreds of thousands of labels or classes [Dean et al., 2013], and object detection with deformable part models [Dean et al., 2013, Felzenszwalb et al., 2010, Shrivastava and Li, 2014b].

We consider multiple variants of the MIPS problem, which differ in what is considered a large inner product, whether the search is exact or approximate, and whether we are provided with one or multiple query vectors. We focus on settings in which the number of vectors is very large (order of millions) and each vector has comparably low dimensionality (say, 10–500); this is a common setting in many applications that use matrix factorization techniques, like those discussed in chapter 2.

A simple way to solve the MIPS problem is to perform naive search: to compute the inner product between the query vector and all probe vectors. Such an approach is generally computationally infeasible as argued in Chapter 1. To avoid naive search, a number of exact [Curtin and Ram, 2014, Curtin et al., 2013, Ram and Gray, 2012] and approximate [Bachrach et al., 2014, Neyshabur and Srebro, 2015, Shrivastava and Li,

[1]The contents of this chapter have been jointly developed with Rainer Gemulla and Olga Mykytiuk, as in [Teflioudi et al., 2015] and [Teflioudi and Gemulla, 2016].

2014a,b] algorithms for MIPS have been proposed in the literature. Generally, exact methods offer in practice only limited speedup over naive search, whereas approximate methods trade off result quality to achieve much higher speedups.

In this chapter, we introduce LEMP,[2] an efficient algorithm for both exact and approximate MIPS. In contrast to previous methods, which aim to solve the MIPS problem directly, LEMP makes use of the simple observation that both the length and the direction (or cosine similarity) of two vectors influence the value of their inner product. LEMP exploits this observation by grouping the input vectors into *buckets* of similar lengths and subsequently solving a smaller cosine similarity search problem for each bucket. In this way, LEMP (i) exploits vector lengths for early pruning, (ii) is able to choose a suitable search technique individually for each bucket (and query), and (iii) improves cache locality by fitting the small problem instances into cache.

To process buckets, LEMP is able to leverage any existing method for cosine similarity search or MIPS. We consider a number of such methods, including the well-known threshold algorithm (TA, [Fagin et al., 2001]) and techniques for cosine similarity search such as L2AP [Anastasiu and Karypis, 2014] or cover trees [Curtin et al., 2013]. We propose two novel methods for cosine similarity search, termed COORD (for coordinate-based pruning) and ICOORD (for incremental coordinate-based pruning); our new methods are tailored for their use within the LEMP framework and, according to our experimental study, are generally more efficient than previous methods. LEMP supports approximate MIPS by using more aggressive pruning techniques as well as approximate methods for solving the cosine similarity search problems. We propose two novel methods based on ICOORD and an adaptive variant of locality-sensitive hashing (LSH, [Gionis et al., 1999]); our methods can trade-off quality and performance and provide approximation guarantees.

The remainder of this chapter is structured as follows: Section 3.1 describes applications of MIPS and defines multiple variants of the MIPS problem. Section 3.2 introduces the LEMP framework, while Sections 3.3 and 3.4 focus on exact and approximate MIPS using LEMP. Section 3.5 provides some guidance for efficient implementation of LEMP and Section 3.6 discusses parallel versions of LEMP. Section 3.7 summarizes related work. Section 3.8 describes our experimental study and its results and Section 3.9 summarizes our study.

[2]The MIPS problem is equivalent to the problem of finding Large Entries in a Matrix Product; see Section 3.1.

3.1 Preliminaries and Problem Statement

In this section, we introduce the notation used throughout, describe some key applications of MIPS, and formally define multiple variants of the MIPS problem.

3.1.1 Notation

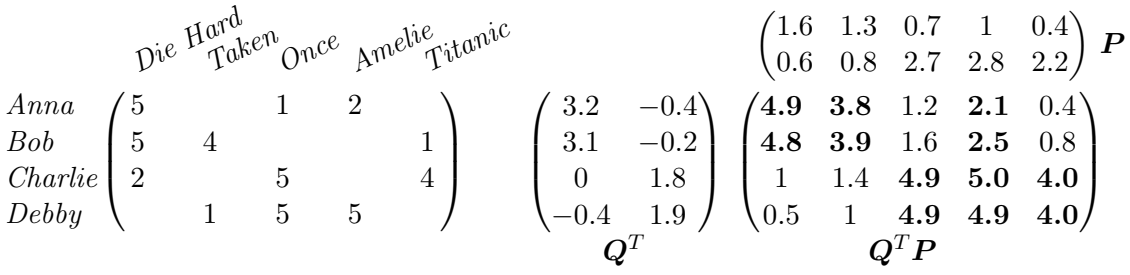
Let $[n] = \{1, \ldots, n\}$. We denote matrices by bold uppercase letters, vectors by bold lowercase letters, and scalars by non-bold lowercase letters. In this chapter, we represent a set of vectors using a matrix in which each column holds one of the vectors. We write $\boldsymbol{M}_j$ for the j-th column of matrix $\boldsymbol{M}$, and $\boldsymbol{v} \in \boldsymbol{M}$ if $\boldsymbol{v}$ is a column of $\boldsymbol{M}$. We denote the i-th element of vector $\boldsymbol{v}$ by v_i.

Denote by $\boldsymbol{Q}$ and $\boldsymbol{P}$ two sets of r-dimensional vectors with cardinality m and n, respectively. We assume throughout that m and n are very large (order of millions) and r is comparably small (say, 10–500). We are interested in finding pairs of vectors, one from $\boldsymbol{Q}$ and one from $\boldsymbol{P}$, with large inner product or, equivalently, the indexes of the large entries in the product matrix $\boldsymbol{Q}^T\boldsymbol{P}$. We refer to $\boldsymbol{Q}$ as the *query matrix* and to $\boldsymbol{P}$ as the *probe matrix*. Similarly, we refer to vectors $\boldsymbol{q} \in \boldsymbol{Q}$ as *query vectors* (or simply *queries*) and to vectors $\boldsymbol{p} \in \boldsymbol{P}$ as *probe vectors*.

3.1.2 Applications of MIPS

The MIPS problem frequently arises in data mining tasks that employ some form of low-rank matrix factorization. Such low-rank matrix factorization methods—e.g., the singular value decomposition (SVD), non-negative matrix factorization (NMF), or latent-factor models—have been successfully applied to a number of prediction tasks (see Skillicorn [2007] as well as Chapter 2). In general, the available data is represented as a matrix in which rows and columns correspond to entities or attributes of interest, and entries to values. Low-rank matrix factorizations are used for dimensionality reduction and to reveal hidden structure in the data. Large entries in the obtained low-rank matrix indicate strong interactions between entities and attributes and are often of particular interest in applications.

In the context of recommender systems, for example, latent-factor models are a popular and powerful approach for predicting the preference of users for items from available feedback; see Koren et al. [2009] for an excellent overview. Figure 3.1a shows a feedback matrix $\boldsymbol{D}$, which contains ratings of a set of users for a set of movies they had watched on a 1–5 star scale. To predict the ratings of the movies users did not yet watch,



(a) Feedback matrix $\boldsymbol{D}$

(b) Factor matrices for users ($\boldsymbol{Q}$) and movies ($\boldsymbol{P}$) as well as corresponding predictions ($\boldsymbol{Q}^T\boldsymbol{P}$)

FIGURE 3.1: Example of a simple matrix factorization model for a recommender system

latent-factor models construct two factor matrices: a user matrix $\boldsymbol{Q}$ and an item matrix $\boldsymbol{P}$, in which columns correspond to users and items, respectively, and rows to latent factors. Figure 3.1b shows an example with $r = 2$ latent factors, which in this case roughly correspond to action and romance. The predicted preference of user i for item j is given by the (i, j) entry of matrix product $\boldsymbol{Q}^T\boldsymbol{P}$ or, equivalently, the inner product $\boldsymbol{q}^T\boldsymbol{p} = \sum_{k=1}^{r} q_k p_k$, where $\boldsymbol{q}$ denotes the i-th column of $\boldsymbol{Q}$ and $\boldsymbol{p}$ the j-th column of $\boldsymbol{P}$. The goal of a recommender system is to recommend to each user the items with a high predicted rating (among other criteria); we thus need to determine which entries are large. In the example of Figure 3.1b, we marked in bold face the top-3 items for each user.[3] In our terminology, user vectors correspond to queries and item vectors to probe vectors. We are interested in finding for each user the top-k items vectors with the largest inner products; we refer to this problem as Top-k-MIPS.

Another prominent application of matrix factorization models is in the area of open information extraction, which extracts and reasons about statements made in natural language text and other sources. Riedel et al. [2013], for example, construct a *fact matrix*, in which columns correspond to verbal phrases or relations (e.g., "was born in") and rows to (subject, object)-pairs (e.g., ("Einstein", "Ulm")). A nonzero entry indicates that the corresponding fact (a verbal phrase or relation with its subject and object) was observed in the available data. Latent factor models are used to predict additional facts, spot unlikely facts, and reason about verbal phrases [Nickel, 2016]. As in recommender systems, these methods create factor matrices using a suitable model and subsequently determine the large entries in their product; here large entries correspond to facts with a high predicted confidence. We refer to the problem of retrieving all entries above a specified threshold θ as Above-θ-MIPS.

Note that in the above applications, MIPS is applied to the factor matrices obtained from some matrix factorization algorithm. As we have already seen in Chapter 2, fast

[3] In practice, we might ignore movies already watched by the user.

and scalable matrix factorization algorithms have been extensively studied in the literature [Makari et al., 2015, Niu et al., 2011, Recht and Ré, 2013, Teflioudi et al., 2012], so that the factorization itself is usually not a bottleneck (see Section 3.8.1 for some examples). In addition, some of the applications mentioned in this chapter's introduction (e.g., Dean et al. [2013]) do not make use of a prior matrix factorization step. In this chapter, we focus solely on the MIPS problem and are oblivious to how the input matrices have been created (see Chapter 2 for more details on matrix factorization).

3.1.3 Problem Statement

Exact MIPS. We study two variants of exact MIPS. The first one searches for each vector $\boldsymbol{q} \in \boldsymbol{Q}$, the set of k vectors from $\boldsymbol{P}$ with the largest inner product with $\boldsymbol{q}$. Here k is an application-defined parameter. As discussed previously, this problem arises in recommender systems, where we want to retrieve the most relevant items (vectors of $\boldsymbol{P}$) for each user (vector of $\boldsymbol{Q}$).

Definition 3.1 (Top-k-MIPS)**.** Given an integer $k > 0$, find for every $\boldsymbol{q} \in \boldsymbol{Q}$ the set $J \subseteq [n]$ of the k columns of $\boldsymbol{P}$ that attain the k largest values of $\boldsymbol{q}^T\boldsymbol{P}$. Ties are broken arbitrarily.

Note that if $\boldsymbol{Q}$ has only one column (contains a single vector only), the Top-k-MIPS problem is equivalent to top-k scoring with linear scoring function $f(\boldsymbol{p}) = \boldsymbol{q}^T\boldsymbol{p}$ [Fagin et al., 2001]. In the general case, in which $\boldsymbol{Q}$ has multiple columns, Top-k-MIPS is equivalent to multi-query top-k scoring. Usually, MIPS is defined in the literature for a single query, i.e., $\boldsymbol{Q} = (\boldsymbol{q})$. In this work, we focus on the general case in which $\boldsymbol{Q}$ contains multiple query vectors, i.e., the queries may arrive in batches. Our methods can also be used in a streaming setting, in which $\boldsymbol{Q}$ contains a single query vector. By reversing the roles of $\boldsymbol{Q}$ and $\boldsymbol{P}$, we can also find the top-k queries for each probe vector.

The second problem, termed Above-θ-MIPS, asks to retrieve all pairs of vectors with inner product above some application-defined threshold θ. This problem is useful, for example, to determine all high-confidence facts in an open relation extraction scenario.

Definition 3.2 (Above-θ-MIPS)**.** Given a threshold $\theta > 0$, determine the set of large entries

$$\{\, (i,j) \in [m] \times [n] \mid [\boldsymbol{Q}^T\boldsymbol{P}]_{ij} \geq \theta \,\}.$$

A simple solution to the above problems is to first compute $\boldsymbol{Q}^T\boldsymbol{P}$ and then select the entries above the threshold (for Above-θ-MIPS) or the k largest entries per row (for Top-k-MIPS). We refer to this approach as *Naive*; it has time complexity $O(mnr)$ and is

infeasible for large problem instances. Recently, a number of algorithms for exact MIPS have been proposed [Curtin and Ram, 2014, Curtin et al., 2013, Ram and Gray, 2012]; all of these methods are based on suitable tree-based indexes built on $\boldsymbol{P}$ (see Section 3.7).

Approximate MIPS. Exact MIPS methods usually offer only limited speedup compared to naive search. Thus there has been a significant interest in designing methods for approximate MIPS [Bachrach et al., 2014, Neyshabur and Srebro, 2015, Shrivastava and Li, 2014a,b]. Such methods trade off the quality of results in exchange for lower computational costs. In many applications, high-quality approximate results are acceptable. For example, in recommender systems, finding good recommendations fast may be preferable to finding the best recommendations slowly.

There are multiple conceivable ways to measure the quality of the results of an approximate MIPS algorithm with respect to a query $\boldsymbol{q}$. A commonly used metric is *recall*, which corresponds to the fraction of true results—the ones that an exact MIPS algorithm would produce—in the result set produced by the approximate algorithm. Note that for Top-k-MIPS, both approximate and exact methods produce exactly k results, so that recall (fraction of true results overall) and precision (fraction of true results in answer) coincide.

For the Top-k-MIPS problem, recall will indicate how many true results exist in the approximate top-k result for the query. However, the recall value does not give any indication about the quality of the remaining ("false") vectors in the approximate top-k list. To see why this might be of importance, consider again the recommender system scenario. Generally, we prefer methods that give good false results over methods that gives bad false results, and recall does not allow to distinguish these two cases. To formalize this intuition, denote by $s_1, s_2, \ldots, s_k$ the values of the inner products in the exact solution of a Top-k-MIPS problem in decreasing order, and by $\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_k$ the corresponding result of an approximate algorithm. A measure that captures the difference between the result of the exact and the approximate method in absolute terms is the root mean square error (RMSE, [Bachrach et al., 2014]), defined as:

$$RMSE = \sqrt{\frac{1}{k}\sum_{i=1}^{k}(s_i - \hat{s}_i)^2}. \tag{3.1}$$

Alternatively, we can quantify the difference relatively using the average relative error (ARE):

$$ARE = \frac{1}{k}\sum_{i=1}^{k}\left|\frac{s_i - \hat{s}_i}{s_i}\right|. \tag{3.2}$$

We define the recall/RMSE/ARE for a set of queries by taking the average of the recall/RMSE/ARE over all queries.

An approximate MIPS method that provides approximation guarantees takes as input an error bound on either recall, RMSE, or ARE and produces an approximate result that satisfies the specified bound (always or, in some cases, with high probability). Unfortunately, many of the existing approximate methods do not provide such guarantees and proceed in a best-effort manner instead. In Section 3.4, we propose a number of novel approximate methods that do provide error guarantees.

3.2 The LEMP Framework

In this section, we outline the LEMP algorithm for exact and approximate MIPS. For presentation purposes, we initially focus on the Above-θ-MIPS problem and turn to the Top-k-MIPS problem in Section 3.3.5.

3.2.1 Length and Direction

LEMP makes use of the decomposition of an inner product of two vectors $\boldsymbol{q}$ and $\boldsymbol{p}$ into a length and a direction part. Denote by $\|\boldsymbol{v}\| = \sqrt{\sum_f v_f^2}$ the length (Euclidean norm) of vector $\boldsymbol{v} \neq 0$, and by $\bar{\boldsymbol{v}} = \boldsymbol{v}/\|\boldsymbol{v}\|$ its *normalization*, i.e., the unit vector pointing in the direction of $\boldsymbol{v}$. Then

$$\boldsymbol{q}^T\boldsymbol{p} = \|\boldsymbol{q}\|\,\|\boldsymbol{p}\|\cos(\boldsymbol{q},\boldsymbol{p}), \tag{3.3}$$

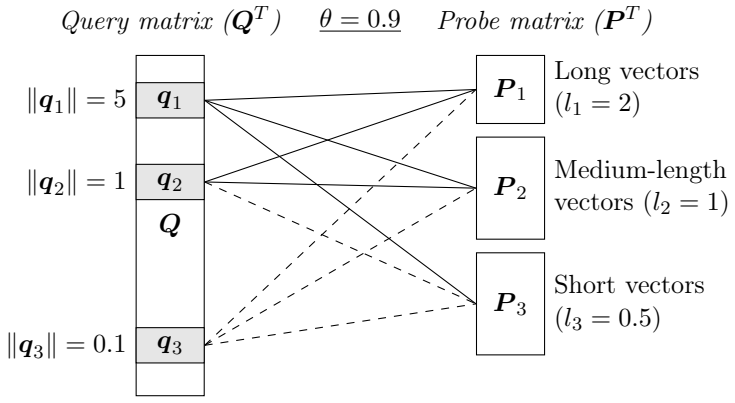
where $\cos(\boldsymbol{q},\boldsymbol{p}) = \bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} \in [-1,1]$ denotes the cosine similarity between $\boldsymbol{q}$ and $\boldsymbol{p}$. As mentioned previously, the inner product coincides with the cosine similarity if $\boldsymbol{q}$ and $\boldsymbol{p}$ have unit length. The problem of cosine similarity search is thus a special case of the MIPS problem.

By rewriting Eq. (3.3), we obtain

$$\boldsymbol{q}^T\boldsymbol{p} \geq \theta \iff \cos(\boldsymbol{q},\boldsymbol{p}) \geq \frac{\theta}{\|\boldsymbol{q}\|\,\|\boldsymbol{p}\|}. \tag{3.4}$$

The inner product thus exceeds threshold θ if and only if the cosine similarity exceeds the modified threshold $\frac{\theta}{\|\boldsymbol{q}\|\,\|\boldsymbol{p}\|}$, which depends on the lengths of $\boldsymbol{q}$ and $\boldsymbol{p}$. Our goal is to find pairs $(\boldsymbol{q},\boldsymbol{p}) \in \boldsymbol{Q} \times \boldsymbol{P}$ such that $\boldsymbol{q}^T\boldsymbol{p} \geq \theta$. From Eq. 3.4, we conclude that:

1. If $\boldsymbol{q}$ and $\boldsymbol{p}$ are short in that $\|\boldsymbol{q}\|\|\boldsymbol{p}\| < \theta$, we cannot have $\boldsymbol{q}^T\boldsymbol{p} > \theta$ since $\cos(\boldsymbol{q},\boldsymbol{p}) \in [-1,1]$ and $\theta/(\|\boldsymbol{q}\|\,\|\boldsymbol{p}\|) > 1$. Such pairs do not need to be considered.



$\theta_b(\boldsymbol{q})$	$\boldsymbol{P}_1$	$\boldsymbol{P}_2$	$\boldsymbol{P}_3$
$\boldsymbol{q}_1$	0.09	0.18	0.36
$\boldsymbol{q}_2$	0.45	0.90	-
$\boldsymbol{q}_3$	-	-	-

Local thresholds

FIGURE 3.2: Illustration of LEMP's bucketization

2. If $\boldsymbol{q}$ and $\boldsymbol{p}$ are of intermediate length in that $\|\boldsymbol{q}\|\|\boldsymbol{p}\| \approx \theta$, then $\boldsymbol{q}^T\boldsymbol{p} > \theta$ if the cosine similarity $\cos(\boldsymbol{q}, \boldsymbol{p})$ is large. Such pairs are best found using a cosine similarity search algorithm.

3. If $\boldsymbol{q}$ and $\boldsymbol{p}$ are long in that $\|\boldsymbol{q}\|\|\boldsymbol{p}\| \gg \theta$, then $\boldsymbol{q}^T\boldsymbol{p} > \theta$ if their cosine similarity is not too small. Such pairs are best found using naive search.

This indicates that vectors of different lengths are best treated in different ways. LEMP exploits this observation as follows. It first groups the vectors of the probe matrix $\boldsymbol{P}$ into a set of small buckets, each consisting of vectors of roughly similar length, and then solves a cosine similarity search problem for each bucket. In particular, we ignore buckets with short vectors, use a suitable cosine similarity search algorithm for buckets with vectors of intermediate lengths, and use (a variant of) naive retrieval for buckets with long vectors. This allows us to prune large parts of the search space and handle the remaining part efficiently.

In more detail, denote by $\boldsymbol{P}_1, \boldsymbol{P}_2, \ldots, \boldsymbol{P}_s$ a set of s buckets and assume that the vectors in each bucket have roughly similar (but not necessarily equal) length. For each bucket $\boldsymbol{P}_b$, $1 \leq b \leq s$, denote by $l_b = \max_{\boldsymbol{p} \in \boldsymbol{P}_b} \|\boldsymbol{p}\|$ the length of its longest vector. Under our assumption, $l_b \approx \|\boldsymbol{p}\|$ for all $\boldsymbol{p} \in \boldsymbol{P}_b$. Figure 3.2 shows an example in which $\boldsymbol{P}$ has been divided into three buckets: $\boldsymbol{P}_1$ holds long vectors (approximate and maximum length 2), $\boldsymbol{P}_2$ medium-length vectors (1), and $\boldsymbol{P}_3$ short vectors (0.5).

Fix some bucket $\boldsymbol{P}_b$. From Eq. (3.4), we obtain that a necessary condition for $\boldsymbol{q}^T\boldsymbol{p} \geq \theta$ for $\boldsymbol{p} \in \boldsymbol{P}_b$ is that

$$\cos(\boldsymbol{q}, \boldsymbol{p}) = \bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} \geq \theta_b(\boldsymbol{q}) \stackrel{\text{def}}{=} \frac{\theta}{\|\boldsymbol{q}\| \, l_b}. \tag{3.5}$$

We refer to $\theta_b(\boldsymbol{q})$ as the *local threshold* of query $\boldsymbol{q}$ for bucket $\boldsymbol{P}_b$. Our goal is thus to find all vectors $\boldsymbol{p} \in \boldsymbol{P}_b$ with a cosine similarity to $\boldsymbol{q}$ of at least $\theta_b(\boldsymbol{q})$. The local threshold allows us to determine how to best process bucket $\boldsymbol{P}_b$, analogous to the discussion above.

Algorithm 3 LEMP for the exact Above-θ-MIPS problem

Require: $\boldsymbol{Q}, \boldsymbol{P}, \theta$
Ensure: $S = \{ (i,j) \mid [\boldsymbol{Q}^T \boldsymbol{P}]_{ij} \geq \theta \}$

1: *// Preprocessing phase*
2: Partition $\boldsymbol{P}$ into buckets $\boldsymbol{P}_1, \ldots, \boldsymbol{P}_s$ of similar length
3: **for all** $b \in 1, 2, \ldots, s$ **do** *// for each bucket*
4: Sort, normalize, and index $\boldsymbol{P}_b$
5: $l_b \leftarrow \max_{\boldsymbol{p} \in \boldsymbol{P}_b} \|\boldsymbol{p}\|$
6: **end for**
7:
8: *// Search phase*
9: $S \leftarrow \emptyset$
10: **for all** $b \in 1, 2, \ldots, s$ **do** *// for each bucket*
11: **for all** $\boldsymbol{q}_i \in \boldsymbol{Q}$ **do** *// for each query*
12: $\theta_b(\boldsymbol{q}_i) \leftarrow \theta \,/\, (\|\boldsymbol{q}_i\| \, l_b)$ *// local threshold*
13: **if** $\theta_b(\boldsymbol{q}_i) \leq 1$ **then** *// prune?*
14: Pick a suitable retrieval alg. A based on $\theta_b(\boldsymbol{q}_i)$
15: Use A to obtain a set of candidates $C_b \supseteq \{ \boldsymbol{p}_j \in \boldsymbol{P}_b \mid \bar{\boldsymbol{q}}_i^T \bar{\boldsymbol{p}}_j \geq \theta_b(\boldsymbol{q}_i) \}$
16: $S \leftarrow S \cup \{ (i,j) \mid \boldsymbol{p}_j \in C_b \text{ and } \boldsymbol{q}_i^T \boldsymbol{p}_j \geq \theta \}$ *// Verify candidates*
17: **end if**
18: **end for**
19: **end for**

If $\theta_b(\boldsymbol{q}) > 1$, we can prune the entire bucket since none of its vectors can potentially pass the threshold. If $\theta_b(\boldsymbol{q}) \approx 1$, we use a suitable cosine similarity search algorithm for the bucket. Finally, if $\theta_b \ll 1$, we use naive retrieval.

Consider again the example of Figure 3.2 and assume a global threshold of $\theta = 0.9$. The figure highlights three query vectors $\boldsymbol{q}_1$, $\boldsymbol{q}_2$, and $\boldsymbol{q}_3$ of decreasing lengths and gives the values of all local thresholds (or "-" if above 1, also indicated by dashed lines). For $\boldsymbol{q}_1$, which is very long, all local thresholds are small so that naive retrieval is well suited for all buckets. For $\boldsymbol{q}_2$, which is shorter, the local threshold is small for bucket $\boldsymbol{P}_1$ (long vectors), large for bucket $\boldsymbol{P}_2$ (medium-length vectors), and above 1 for bucket $\boldsymbol{P}_3$ (short vectors). We use naive retrieval for $\boldsymbol{P}_1$ and a suitable cosine similarity search algorithm for $\boldsymbol{P}_2$. Bucket $\boldsymbol{P}_3$ is pruned. Finally, for $\boldsymbol{q}_3$, which is very short, all local thresholds exceed 1 so that all buckets are pruned.

3.2.2 Algorithm Description

Algorithm 3 summarizes LEMP for exact Above-θ-MIPS. The algorithm consists of a *preprocessing phase* (lines 1–6) and a *search phase* (lines 8–19).

The preprocessing phase groups the columns of $\boldsymbol{P}$ into buckets of similar length (line 2). There are a number of ways to do this, but we chose a simple greedy strategy in our

implementation. In particular, we first sort the columns of $\boldsymbol{P}$ by decreasing length,[4] scan the columns in order, and start a new bucket whenever the length of the current column falls below some threshold (e.g., 90% of l_b). We also make sure that buckets are neither too small nor too large. First, small buckets reduce the efficiency of LEMP due to bucket processing overheads; we thus ensure that buckets contain at least a certain number of vectors (30 in our implementation). The bucket processing overhead of large buckets is negligible. However, when buckets grow larger than the cache size, processing time is negatively affected. For this reason, we select a maximum bucket size that ensures that all relevant data structures fit into the processor cache.

After bucket boundaries have been obtained, we represent each vector $\boldsymbol{p}$ by two separate components: its length $\|\boldsymbol{p}\|$ and its direction $\bar{\boldsymbol{p}}$. We also store the vectors' column number in the original matrix (denoted `id`) and in the bucket (denoted `lid` for "local id"); see Figure 3.4a for an example. This layout allows us to access for each $\boldsymbol{p} \in \boldsymbol{P}_b$ both $\|\boldsymbol{p}\|$ and $\bar{\boldsymbol{p}}$ without further computation. We then create indexes on the contents of each bucket; we defer the discussion of indexing to Section 3.3. For our choice of indexes (Section 3.3.2 and 3.3.3), the overall preprocessing time, including index computation, is $O(rn \log n)$.

The search phase then iterates over buckets and query vectors. For each query, we compute the local threshold $\theta_b(\boldsymbol{q})$ (line 12) and prune buckets based on their length (line 13). For each remaining bucket $\boldsymbol{P}_b$, we select a suitable retrieval algorithm (exact or approximate) based on the local threshold (line 14, cf. Section 3.3). The selected retrieval algorithm computes a set C_b of *candidate vectors*, potentially making use of the index data structures created during the preprocessing phase. We require for exact MIPS that the candidate set contains all vectors in $\boldsymbol{p} \in \boldsymbol{P}_b$ that pass the threshold ($\boldsymbol{q}^T\boldsymbol{p} \geq \theta$), but it may additionally contain a set of *spurious vectors* ($\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} \geq \theta_b(\boldsymbol{q})$ but $\boldsymbol{q}^T\boldsymbol{p} < \theta$). If LEMP is used for approximate MIPS, we lift this requirement, i.e., the candidate set may then miss some vectors that pass the threshold. We instead require the candidate set to contain vectors that allow us to satisfy user-specified bounds on the recall, RMSE or ARE. In both cases, a verification step (line 16) filters out spurious vectors by computing the actual values of the inner products $\boldsymbol{q}^T\boldsymbol{p}$ for all $\boldsymbol{p} \in C_b$.

The order of the two loops in the search phase of Algorithm 3 is chosen to be cache friendly. Since we process probe buckets in the outer loop and since probe buckets are small, their content remains in the cache for the entire inner loop. The inner loop itself scans query vectors sequentially; these vectors may not fit into the cache, but the sequential access pattern makes prefetching effective.

The power of LEMP to prune entire buckets in line 12 depends on the length distribution of the input vectors: generally, the more skewed the length distribution, the more probe

[4] We also sort and normalize query vectors in a manner similar to the bucketization of $\boldsymbol{P}$.

buckets can be pruned. Even if bucket pruning is not particularly effective for a given problem instance, however, the organization of the probe vectors into buckets is still beneficial: it allows cosine similarity search algorithms to be applied and is cache-friendly.

3.3 Exact MIPS

In this section, we propose and discuss a number of exact algorithms for the search phase of LEMP (line 15 of Algorithm 3). Each algorithm takes as input a query vector $\boldsymbol{q} \in \boldsymbol{Q}$ and a bucket $\boldsymbol{P}_b$, and outputs a candidate set $C_b \subseteq \boldsymbol{P}_b$ using some pruning strategy. All algorithms first compute $\|\boldsymbol{q}\|$ and $\bar{\boldsymbol{q}}$; cf. Figure 3.4d.

We discuss two kinds of algorithms: those that make use of only the length information to prune candidate vectors and those that use the normalized vectors as well. For the first category, we propose the LENGTH algorithm (Section 3.3.1), which is a simple variant of the naive algorithm that takes length information into account. Existing cosine similarity search algorithms (e.g., [Bayardo et al., 2007]) as well as TA fall in the second category. Here we additionally present two novel methods, which are specially tailored for vectors of medium dimensionality. The COORD algorithm (Section 3.3.2) applies coordinate-based pruning strategies. The ICOORD algorithm (Section 3.3.3) is based on COORD but uses a more effective (but also more expensive) incremental pruning strategy that also takes length into account.

3.3.1 Length-Based Pruning

Recall that the vectors in bucket $\boldsymbol{P}_b$ are sorted by decreasing length during preprocessing (see also Figure 3.4a). Further, observe from Eq. (3.3) that whenever $\|\boldsymbol{q}\|\,\|\boldsymbol{p}\| < \theta$, so is $\boldsymbol{q}^T\boldsymbol{p}$. Putting both together, LENGTH scans the bucket $\boldsymbol{P}_b$ in order. When processing vector $\boldsymbol{p}$, we check whether $\|\boldsymbol{p}\| \geq \theta/\|\boldsymbol{q}\|$; we precompute $\theta/\|\boldsymbol{q}\|$ to make this check efficient. If $\boldsymbol{p}$ qualifies, we add it to the candidate set C_b. Otherwise, we stop processing bucket $\boldsymbol{P}_b$ and immediately output C_b.

Consider for example a bucket $\boldsymbol{P}_b$ as shown in Figure 3.4a, query vector $\boldsymbol{q} = (1, 1, 1, 1)^T$, and threshold $\theta = 3.8$. We have $\|\boldsymbol{q}\| = 2$ and $\theta/\|\boldsymbol{q}\| = 1.9$, so that we obtain $C_b = \{1, 2, 3\}$. (Here and in the following, we give C_b in terms of local identifiers (`lid`) for improved readability.)

Since LEMP already organizes and prunes buckets by length, we do not expect LENGTH to be particularly effective. In fact, LENGTH degenerates to the naive algorithm in all but one bucket (the "last" bucket that has not been pruned). Nevertheless, since

LENGTH has low overhead and a sequential access pattern, it is an effective method when buckets are small or the local threshold is low (i.e., when coordinate-based pruning is not effective).

3.3.2 Coordinate-Based Pruning

We now proceed to pruning strategies based on the direction (but not length) of the query vector. The key idea is to retain only those vectors from $\boldsymbol{P}_b$ in C_b that point in a similar direction as $\boldsymbol{q}$. In particular, we aim to find all $\boldsymbol{p} \in \boldsymbol{P}_b$ with high cosine similarity to $\boldsymbol{q}$, i.e.,

$$\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}} = \cos(\boldsymbol{q}, \boldsymbol{p}) \geq \theta_b(\boldsymbol{q}). \tag{3.6}$$

Note the usage of normalized vectors here; length information is not taken into account.

Let $\bar{\boldsymbol{q}} = (\bar{q}_1, \ldots, \bar{q}_r)^T$ and $\bar{\boldsymbol{p}} = (\bar{p}_1, \ldots, \bar{p}_r)^T$. Note that $\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}}$ achieves its maximum value for $\bar{\boldsymbol{p}} = \bar{\boldsymbol{q}}$ since then $\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}} = \bar{\boldsymbol{q}}^T \bar{\boldsymbol{q}} = \|\bar{\boldsymbol{q}}\|^2 = 1$. In other words, $\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}}$ is maximized when both vectors agree on all their coordinates. Based on this observation, the key idea of the COORD algorithm is to prune $\bar{\boldsymbol{p}}$ if one of its coordinates deviates too far from the respective coordinate in $\bar{\boldsymbol{q}}$. In more detail, we obtain for each coordinate $f \in [r]$ a lower bound $L_f(\bar{\boldsymbol{q}})$ and an upper bound $U_f(\bar{\boldsymbol{q}})$ on $\bar{p}_f$. If $L_f \leq \bar{p}_f \leq U_f$, we say that $\bar{p}_f$ is *feasible*; otherwise $\bar{p}_f$ is *infeasible*. The bounds are chosen such that whenever a coordinate f of $\boldsymbol{p}$ is infeasible, then $\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}} < \theta_b(\boldsymbol{q})$ so that $\boldsymbol{p}$ can be pruned from the candidate set. Such pruning is particularly effective when $\theta_b(\boldsymbol{q})$ is large or when the query vector is sparse.

In what follows, we provide lower and upper bounds, discuss their effectiveness, and propose the COORD algorithm that exploits them.

Bounding coordinates. Pick some coordinate $f \in [r]$; we refer to f as a *focus coordinate*. Denote by $\bar{\boldsymbol{q}}_{\text{-}f} = \{\, \bar{q}_1, \ldots, \bar{q}_{f-1}, \bar{q}_{f+1}, \ldots, \bar{q}_r \,\}$ the vector obtained by removing coordinate f from $\bar{\boldsymbol{q}}$, similarly $\bar{\boldsymbol{p}}_{\text{-}f}$. Note that $\bar{\boldsymbol{q}}_{\text{-}f}$ and $\bar{\boldsymbol{p}}_{\text{-}f}$ generally have length less than 1. Now we rewrite Eq. (3.6) as follows

$$\begin{aligned}
\theta_b(\boldsymbol{q}) \leq \bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}} &= \textstyle\sum_i \bar{q}_i \bar{p}_i = \bar{q}_f \bar{p}_f + \sum_{i \neq f} \bar{p}_i \bar{q}_i \\
&= \bar{q}_f \bar{p}_f + \bar{\boldsymbol{q}}_{\text{-}f}^T \bar{\boldsymbol{p}}_{\text{-}f} \\
&= \bar{q}_f \bar{p}_f + \|\bar{\boldsymbol{q}}_{\text{-}f}\| \, \|\bar{\boldsymbol{p}}_{\text{-}f}\| \cos(\bar{\boldsymbol{p}}_{\text{-}f}, \bar{\boldsymbol{q}}_{\text{-}f}) \\
&\leq \bar{q}_f \bar{p}_f + \|\bar{\boldsymbol{q}}_{\text{-}f}\| \, \|\bar{\boldsymbol{p}}_{\text{-}f}\| \\
&= \bar{q}_f \bar{p}_f + \sqrt{1 - \bar{q}_f^2} \sqrt{1 - \bar{p}_f^2},
\end{aligned} \tag{3.7}$$

where we used Eq. (3.3), the fact that the cosine similarity cannot exceed 1, and the property $\|\bar{\boldsymbol{q}}\| = \|\bar{\boldsymbol{p}}\| = 1$.

We now solve the resulting inequality $\theta_b(\boldsymbol{q}) \leq \bar{q}_f \bar{p}_f + (1 - \bar{q}_f^2)^{1/2}(1 - \bar{p}_f^2)^{1/2}$ for $\bar{p}_f$ (a full proof can be found at Appendix A) and obtain solutions

$$\bar{p}_f \in [L_f^A, U_f^A] = \begin{cases} [\theta_b(\boldsymbol{q})/\bar{q}_f, 1] & \bar{q}_f > 0 \\ [-1, \theta_b(\boldsymbol{q})/\bar{q}_f] & \bar{q}_f < 0 \end{cases} \tag{3.8}$$

$$\bar{p}_f \in (L_f^B, U_f^B) = \left(\bar{q}_f\theta_b(\boldsymbol{q}) - \sqrt{(1-\theta_b(\boldsymbol{q})^2)(1-\bar{q}_f^2)}, \bar{q}_f\theta_b(\boldsymbol{q}) + \sqrt{(1-\theta_b(\boldsymbol{q})^2)(1-\bar{q}_f^2)}\right) \tag{3.9}$$

$$\bar{p}_f \in [L_f^C, U_f^C] = [-1, 1], \text{ if } \bar{q}_f = 0, \theta_b(\boldsymbol{q}) \leq 0. \tag{3.10}$$

Here Eq. (3.8) is only a valid solution to Eq. (3.7) when $L_f^A \leq U_f^A$, i.e., we ignore it whenever $L_f^A > U_f^A$. In addition Eq. (3.10) provides no pruning power to our algorithm. We therefore directly skip query coordinates of zero value whenever $\theta_b(\boldsymbol{q}) \leq 0$. The feasible region is thus given by:

$$L_f = \begin{cases} \min(L_f^A, L_f^B) & \text{if } L_f^A \leq U_f^A \\ L_f^B & \text{otherwise} \end{cases} \tag{3.11}$$

$$U_f = \begin{cases} \max(U_f^A, U_f^B) & \text{if } L_f^A \leq U_f^A \\ U_f^B & \text{otherwise} \end{cases} \tag{3.12}$$

Note that if the lengths of the vectors within a bucket vary strongly, we are forced to use a low local threshold $\theta_b(\boldsymbol{q})$, which in turn results in looser bounds. This undesirable behavior is avoided by LEMP since it constructs buckets that contain vectors of similar length. The effectiveness of our bounds—and of using normalization and subsequent coordinate-based pruning in general—is thus particularly effective in the context of our LEMP framework.

Effectiveness of bounds. To gain some insight into the effectiveness of our bounds, we plot the *feasible region* $[L_f, R_f]$ for various choices $\theta_b(\boldsymbol{q})$ in Figure 3.3. The x-axis corresponds to the value of $\bar{q}_f$, the y-axis to the lower and upper bounds, and the various oval-shaped gray regions to the feasible regions. Note that $-1 \leq \bar{q}_f, \bar{p}_f \leq 1$.

The pruning power of our bounds depends on both the value of $\theta_b(\boldsymbol{q})$ and on the properties of matrices $\boldsymbol{Q}$ and $\boldsymbol{P}$. First, the larger the local threshold $\theta_b(\boldsymbol{q})$, the smaller the feasible region and the more vectors can be pruned. In fact, for large values of $\theta_b(\boldsymbol{q})$, the

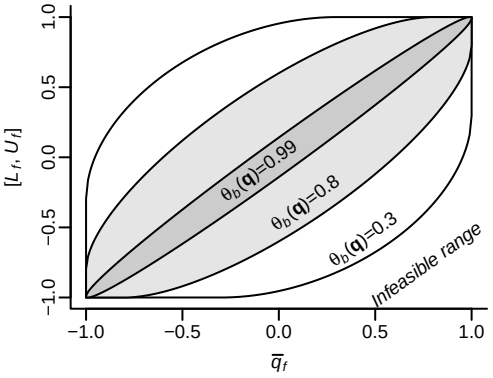


FIGURE 3.3: Feasible regions for various values of $\theta_b(\boldsymbol{q})$

feasible region is small across the entire value range of $\bar{q}_f$. Second, the size of the feasible region decreases as the magnitude of $\bar{q}_f$ increases. This decrease is more pronounced when the local threshold is small. Note that a small feasible region may or may not lead to effective pruning; the value distribution of $\boldsymbol{P}_b$ is also important. Nevertheless, the smaller the feasible region, the more effective the pruning will be.

Based on the observations above, we conclude that our bounds can effectively prune a vector $\bar{\boldsymbol{p}}$ with $\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} < \theta_b(\boldsymbol{q})$ when $\theta_b(\boldsymbol{q})$ is large or when there is some coordinate f for which only one of $\bar{q}_f$ or $\bar{p}_f$ takes a large value. Since all vectors are length-normalized, the latter property holds if $\bar{\boldsymbol{q}}$ or $\bar{\boldsymbol{p}}$ is sufficiently sparse or has a skewed value distribution. If neither holds and $\theta_b(\boldsymbol{q})$ is small, an algorithm such as LENGTH or ICOORD may be a more suitable choice.

Exploiting bounds. The COORD algorithm makes use of the feasible region derived in the previous section to prune unpromising candidates. To do so, LEMP creates indexes for each probe bucket $\boldsymbol{P}_b$ during its preprocessing phase. In the case of COORD, we create r sorted lists $I_1, \ldots, I_r$, one for each coordinate of the vectors in $\boldsymbol{P}_b$. Each entry in list I_f is a (`lid`, $\bar{p}_f$)-pair, where as before `lid` is a bucket-local identifier for the corresponding vector $\bar{\boldsymbol{p}}$. As in Fagin et al.'s threshold algorithm (TA, [Fagin et al., 2001]), from which our index is inspired, the lists are sorted in decreasing order of $\bar{p}_f$. Figure 3.4c shows the sorted-list index for the example bucket given in Figure 3.4a. Although index construction is generally light-weight and fast, LEMP constructs indexes lazily on first use to further reduce computational cost. Buckets with very short vectors, for example, will always be pruned and thus do not need to be indexed.

COORD is summarized as Algorithm 4. It takes as input a bucket $\boldsymbol{P}_b$, a query $\boldsymbol{q}$, the global and local thresholds $(\theta, \theta_b(\boldsymbol{q}))$, the bucket indexes $I_1, \ldots, I_r$, and a set of focus coordinates $F \subseteq [r]$. We discuss the algorithm using the example of Figure 3.4 with $\theta = 0.9$. Consider the query $\boldsymbol{q}$ shown in Figure 3.4d as well as the corresponding inner products shown in Figure 3.4b. We have $\theta_b(\boldsymbol{q}) = 0.9/(0.5 \cdot 2) = 0.9$, coincidentally

lid	id	$\|\boldsymbol{p}\|$	$\bar{\boldsymbol{p}}$			
1	23	2.0	0.58	0.50	0.40	0.50
2	43	1.9	0.98	0	0	0.20
3	12	1.9	0.53	0	0	0.85
4	54	1.8	0.35	0.93	0	0.10
5	18	1.8	0.58	0.50	0.40	0.50
6	20	1.8	0.30	-0.40	0.81	-0.30

(a) Organization of bucket $\boldsymbol{P}_b$

$\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}}$	$\boldsymbol{q}^T\boldsymbol{p}$
0.97	**0.97**
0.79	0.75
0.80	0.76
0.56	0.52
0.97	0.87
0.26	0.23

(b) Results for query $\boldsymbol{q}$ of (D)

I_1		I_2		I_3		I_4	
lid	$\bar{p}_1$	lid	$\bar{p}_2$	lid	$\bar{p}_3$	lid	$\bar{p}_4$
2	0.98	4	0.93	6	0.81	3	0.85
1	**0.58**	1	0.50	1	0.40	**1**	**0.50**
5	**0.58**	5	0.50	5	0.40	**5**	**0.50**
3	**0.53**	2	0	2	0	**2**	**0.20**
4	**0.35**	3	0	3	0	**4**	**0.10**
6	0.30	6	-0.40	4	0	6	-0.30

(c) Sorted-list index (bold rows show scan range for $\boldsymbol{q}$)

$\|\boldsymbol{q}\|$	$\bar{\boldsymbol{q}}$			
0.5	0.70	0.3	0.4	0.51
$[L_f, U_f]$	$[0.32, 0.94]$	-	-	$[0.09, 0.83]$

(d) Query $\boldsymbol{q}$ and feasible region for focus coordinates

lid	c
1	**2**
2	1
3	1
4	**2**
5	**2**
6	0

$C_b = \{1, 4, 5\}$

(e) CP array

lid	c	$\bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F$	$\|\boldsymbol{p}_F\|^2$	u	$\theta_{\boldsymbol{p}}(\boldsymbol{q})$
1	**2**	**0.66**	**0.59**	**0.32**	**0.9**
2	1	0.10	0.04	0.49	0.95
3	1	0.37	0.28	0.43	0.95
4	2	0.30	0.13	0.47	1
5	2	0.66	0.59	0.32	1
6	0	-	-	-	1

$C_b = \{1\}$

(f) Extended CP array

FIGURE 3.4: Illustration of LEMP as well as the COORD and ICOORD retrieval algorithms for $\theta = 0.9$ and $F = \{1, 4\}$

agreeing with the global threshold. Observe that vectors 1 and 5 pass the local threshold $\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} \geq \theta_b(\boldsymbol{q})$, but only vector 1 additionally passes the global threshold $\boldsymbol{q}^T\boldsymbol{p} \geq \theta$.

COORD does not compute and enforce the bounds for each coordinate, but uses a suitable subset $F \subseteq [r]$ of focus coordinates; see below. For each focus coordinate $f \in F$, COORD computes the feasible region $[L_f, U_f]$ (line 3) and determines the start and end of the corresponding *scan range* in sorted list I_f via binary search for U_f and L_f, respectively (line 4). Vectors outside the scan range violate the bound on coordinate f.

Algorithm 4 The COORD algorithm

Require: $\boldsymbol{q}, \boldsymbol{P}_b, \theta, \theta_b(\boldsymbol{q}), F \subseteq [r], I_1, \ldots, I_r$
Ensure: $C_b \supseteq \{ \boldsymbol{p}_j \in \boldsymbol{P}_b \mid \bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}}_j \geq \theta_b(\boldsymbol{q}) \}$
1: $c \leftarrow$ empty CP array
2: **for all** $f \in F$ **do**
3: Calculate feasible region $[L_f, U_f]$
4: Determine corresponding scan range in sorted list I_f
5: **for all** `lid` in scan range of I_f **do**
6: $c[\texttt{lid}] \leftarrow c[\texttt{lid}] + 1$ *// maintain CP array*
7: **end for**
8: **end for**
9: $C_b = \{ \texttt{lid} \mid c[\texttt{lid}] = |F| \}$ *// filter*

In the example of Figure 3.4, we used $F = \{ 1, 4 \}$. The bounds are shown in Figure 3.4d and the corresponding scan ranges in I_1 and I_4 are shown in bold face in Figure 3.4c.

COORD subsequently scans the scan range of each sorted list I_f, $f \in F$, in sequence (line 5) and maintains a *candidate-pruning array* (CP array, line 6). The CP array contains for each vector $\bar{\boldsymbol{p}} \in \boldsymbol{P}_b$ with local identifier `lid` a counter $c[\texttt{lid}]$ that indicates how often the vector has been seen so far. The CP array of our running example is shown in Figure 3.4e (with an additional `lid` column for improved readability). After completing all scans, COORD includes into C_b all those vectors $\bar{\boldsymbol{p}} \in \boldsymbol{P}_b$ that qualified on all focus coordinates, i.e., for which $c[\texttt{lid}] = |F|$ (line 9). In our example, $C_b = \{ 1, 4, 5 \}$ since only those three vectors occurred in both scan ranges. In particular, vectors 2 and 3 are (correctly) excluded because they appear in only one scan range.

We now turn to the question of how to choose the focus set F. One option is to simply set $F = [r]$. However, processing sorted lists can get expensive if F is large or contains coordinates for which pruning is not effective, i.e., for which a large fraction of the corresponding sorted lists needs to be scanned. We make use of a focus-set size parameter ϕ, typically in the range of 1–5; we discuss the choice of ϕ in Section 3.3.4. COORD then uses the ϕ coordinates of $\bar{\boldsymbol{q}}$ with largest absolute value as focus coordinates. The reasoning behind this choice is that large coordinates will lead to the smallest feasible region (cf. Section 3.3.2); the hope is that they also lead to a small scan ranges and a small candidate set.

To summarize, COORD builds indexes only if needed and uses only a subset of the entries in a subset of the sorted-list indexes. The index scan itself is light-weight; it accesses solely the `lid` part of the lists and increases the counters of the CP array. Also note that the bounds we use for determining the scan range of the lists are simple and relatively cheap to compute. This is important since these bounds need to be computed per query, per bucket, and per focus coordinate. See Section 3.5 for implementation details.

3.3.3 Incremental Coordinate-Based Pruning

COORD scans the sorted-list indexes to find the set of vectors that qualify in each coordinate $f \in F$, i.e., fall in region $[L_f, U_f]$. Other than checking feasibility, the actual values in the scanned lists are ignored. In contrast, the incremental pruning algorithm ICOORD makes use of the $\bar{p}_f$ values as well: It maintains information that allows it to prune additional vectors. Such an approach is generally more expensive than COORD, but the increase in pruning power may offset the costs.

When we derived the bounds of a coordinate f of COORD, we assumed that $\cos(\bar{\boldsymbol{q}}_{\text{-}f}, \bar{\boldsymbol{p}}_{\text{-}f}) = 1$. This is a worst-case assumption; in general, $\cos(\bar{\boldsymbol{q}}_{\text{-}f}, \bar{\boldsymbol{p}}_{\text{-}f})$ will be less (and often much less) than 1. Intuitively, a vector $\bar{\boldsymbol{p}}$ that qualifies barely in all coordinates often does not constitute an actual result. Recall our ongoing example of Figure 3.4. Here vector 4 barely qualifies in both indexes I_1 and I_4 and is thus included into the candidate set of COORD. Vector 4 does not pass the local threshold, however, since $\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}}_4 = 0.56 < 0.9$. COORD is blind to this behavior.

Another potential drawback of COORD is that it does not (and cannot) take into consideration the length distribution of the vectors in each bucket. In the example of Figure 3.4, normalized vectors 1 and 5 are identical and both pass the local threshold. However, since vector 1 is slightly longer than vector 5, only vector 1 passes the global threshold and thus the verification step of LEMP .

Similar to COORD, ICOORD scans the scan ranges of the sorted lists of the focus coordinates. To address the above issues, however, ICOORD additionally maintains a partial inner product for each of the vectors that it encounters. Generalizing our previous notation, denote by $\bar{\boldsymbol{q}}_F$ ($\bar{\boldsymbol{q}}_{\text{-}F}$) the values of the focus coordinates (of all other coordinates) of the query vector; similarly, $\bar{\boldsymbol{p}}_F$ and $\bar{\boldsymbol{p}}_{\text{-}F}$. We obtain

$$\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}} = \bar{\boldsymbol{q}}_F^T \bar{\boldsymbol{p}}_F + \bar{\boldsymbol{q}}_{\text{-}F}^T \bar{\boldsymbol{p}}_{\text{-}F} \leq \bar{\boldsymbol{q}}_F^T \bar{\boldsymbol{p}}_F + \|\bar{\boldsymbol{q}}_{\text{-}F}\| \, \|\bar{\boldsymbol{p}}_{\text{-}F}\|.$$

Since vectors are normalized, the right-hand side can be computed from $\bar{\boldsymbol{q}}_F$ and $\bar{\boldsymbol{p}}_F$ only. Denote the resulting upper bound on the "unseen" part $\bar{\boldsymbol{q}}_{\text{-}F}^T \bar{\boldsymbol{p}}_{\text{-}F}$ of the inner product $\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}}$ by

$$u(\bar{\boldsymbol{q}}_F, \bar{\boldsymbol{p}}_F) = \|\bar{\boldsymbol{q}}_{\text{-}F}\| \, \|\bar{\boldsymbol{p}}_{\text{-}F}\| = \sqrt{1 - \|\bar{\boldsymbol{q}}_F\|^2} \sqrt{1 - \|\bar{\boldsymbol{p}}_F\|^2}.$$

Then $\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}} \leq \bar{\boldsymbol{q}}_F^T \bar{\boldsymbol{p}}_F + u(\bar{\boldsymbol{q}}_F, \bar{\boldsymbol{p}}_F)$. In order to compute this bound, ICOORD uses an *extended CP array*, which maintains for each probe vector, in addition to the frequency counters of COORD (line 6 of Algorithm 4), the quantities $\bar{\boldsymbol{q}}_F^T \bar{\boldsymbol{p}}_F$ and $\|\bar{\boldsymbol{p}}_F\|^2 = \sum_{f \in F} \bar{p}_f^2$. After the extended CP array has been computed, ICOORD includes into the candidate

set only those vectors $\bar{\boldsymbol{p}}$ that satisfy

$$\bar{\boldsymbol{q}}_F^T \bar{\boldsymbol{p}}_F + u(\|\bar{\boldsymbol{q}}_F\|, \|\bar{\boldsymbol{p}}_F\|) \geq \theta_{\boldsymbol{p}}(\boldsymbol{q}) \stackrel{\text{def}}{=} \frac{\theta}{\|\boldsymbol{p}\| \cdot \|\boldsymbol{q}\|}. \tag{3.13}$$

Here $\theta_{\boldsymbol{p}}(\boldsymbol{q})$ is an improved, probe vector-specific local threshold; it holds $\theta_{\boldsymbol{p}}(\boldsymbol{q}) \geq \theta_b(\boldsymbol{q})$. This improved local threshold cannot be used by the COORD algorithm.

Figure 3.4f shows the extended CP array for our running example (to the left of the double vertical lines) as well as the quantities involved in the above pruning condition (to the right; here we write u for $u(\|\bar{\boldsymbol{q}}_F\|, \|\bar{\boldsymbol{p}}_F\|)$). For example, for vector 1, $\bar{\boldsymbol{q}}_F^T \bar{\boldsymbol{p}}_F = 0.58 \cdot 0.70 + 0.50 \cdot 0.51 = 0.66$ and $u = \sqrt{1 - (0.58^2 + 0.50^2)} \cdot \|\bar{\boldsymbol{q}}_{\text{-}F}\|$. The quantity $\|\bar{\boldsymbol{q}}_{\text{-}F}\|$ (not shown in Figure 3.4f) is independent of the probe vectors and thus only computed once. In our example, $\|\bar{\boldsymbol{q}}_{\text{-}F}\| = \sqrt{1 - (0.70^2 + 0.51^2)} = 0.5$. As can be seen in the example, filter condition $\bar{\boldsymbol{q}}_F^T \bar{\boldsymbol{p}}_F + u \geq \theta_{\boldsymbol{p}}(\boldsymbol{q})$ is passed only by vector 1; thus $C_b = \{\, 1 \,\}$. Note that the rows of vector 5 and vector 1 agree in the extended CP array; our improved local threshold (0.9 for vector 1 vs. 1 for vector 5), however, allows us to correctly prune vector 5, but retain vector 1.

3.3.4 Algorithm Selection

Before processing a bucket $\boldsymbol{P}_b$, LEMP needs to decide which retrieval algorithm to use. We have already given some guidance for this choice above: Length-based pruning is suitable for buckets with a skewed length distribution, whereas coordinate-based pruning is suitable for large local thresholds and/or data with a skewed value distribution. In general, the choice of a suitable algorithm is data-dependent.

LEMP uses a simple, pragmatic method for algorithm selection: it samples a small set of query vectors and tests the different methods for each bucket. We observe the wall-clock times obtained by the various methods and select a threshold t_b for each bucket: whenever $\theta_b(\boldsymbol{q}) < t_b$, LEMP will use LENGTH, otherwise it uses coordinate-based pruning. For setting t_b, we simply pick the value that minimizes the runtime on the sampled query vectors.

We proceed similarly to select for each bucket b the parameter ϕ_b for the number of focus coordinates in coordinate-based pruning. We generally explore ϕ_b values in increasing order, i.e., starting with 1 and ending with some upper bound (e.g., 10). To speed up this process, we employ two heuristics. First, we stop exploring larger values for ϕ_b if the performance with the currently tested ϕ value is more than 10% worse than the best performance so far. Second, we use the best ϕ-value of bucket b as a starting point for

tuning bucket $b + 1$. We then start exploring the performance for values of ϕ left and right to this initial value, using the same stopping criterion as above.

The cost of this sample-based profiling step is negligible since the number of query vectors is large; the overall running time is dominated by the time required to process $\boldsymbol{Q}$ in its entirety.

More elaborate approaches for algorithm selection are possible, e.g., some form of reinforcement learning. Our experiments suggest, however, that even the simple selection criterion outlined above gives promising results.

3.3.5 Solving the Top-k-MIPS Problem

Our discussion so far has focused on the Above-θ-MIPS problem; we now proceed to the discussion of the Top-k-MIPS problem. Recall that given a query vector $\boldsymbol{q}$, the Top-k-MIPS problem asks for the vectors $\boldsymbol{p} \in \boldsymbol{P}$ that attain the k largest inner products $\boldsymbol{q}^T\boldsymbol{p}$. Top-$k$-MIPS is often used in recommender systems for retrieving the best k items for each user.

The Top-k-MIPS problem is related to the Above-θ-MIPS problem as follows. Fix a query vector $\boldsymbol{q}$ and denote by θ^* the k-th largest entry in $\boldsymbol{q}^T\boldsymbol{P}$. Given θ^*, the solution of the Top-k-MIPS problem coincides with the solution of the Above-θ-MIPS algorithm with threshold θ^* (assuming no duplicate entries). We do not know θ^*, however, and instead make use of a running lower bound $\hat{\theta} \leq \theta^*$. The value of $\hat{\theta}$ increases as the algorithm proceeds.

In more detail, we take the k longest vectors of $\boldsymbol{P}$ (all located at the beginning of bucket $\boldsymbol{P}_1$) and compute their inner product with $\boldsymbol{q}$. The smallest so-obtained value is our initial choice of $\hat{\theta}$. We then run the Above-θ-MIPS algorithm with threshold $\hat{\theta}$ on the first bucket, determine the top-k answers in the result, and update $\hat{\theta}$ accordingly. In more detail, we set $\hat{\theta}$ to the value of the k-largest inner product found so far. This process is iterated over the subsequent buckets until $\hat{\theta}$ becomes so large that LEMP prunes the next bucket. At this point, we output the current top-k vectors as a result. This strategy is effective because (1) LEMP organizes buckets by decreasing length so that we expect the top-k values to appear in the top-most buckets, and (2) bucket sizes are small (cache-resident) so that the threshold $\hat{\theta}$ is increased frequently. The above algorithm is guaranteed to produce the correct result because $\hat{\theta} \leq \theta^*$: If a bucket contains a vector $\boldsymbol{p}$ with $\boldsymbol{q}^T\boldsymbol{p} \geq \theta^*$, then $\boldsymbol{q}^T\boldsymbol{p} \geq \hat{\theta}$ and we are guaranteed to add $\boldsymbol{p}$ to the candidate set.

Note that the length of $\boldsymbol{q}$ does not affect the result of the Top-k-MIPS problem. We thus simplify the bounds used by our algorithms by normalizing $\boldsymbol{q}$ upfront.

3.4 Approximate MIPS

We now turn attention to using LEMP for approximate MIPS, which allows us to realize further performance gains. We propose three different algorithms. The LEMP-LSHA algorithm (Section 3.4.2) is based on locality-sensitive hashing and offers a probabilistic guarantee on recall for both Above-θ-MIPS and Top-k-MIPS. The LEMP-ABS and LEMP-REL algorithms (Section 3.4.3) offer (non-probabilistic) bounds on the RMSE and ARE, respectively, and can be used with arbitrary exact bucket algorithms.

3.4.1 Locality-Sensitive Hashing for Cosine Similarity Search

We start with a brief, high-level review of locality-sensitive hashing (LSH, [Gionis et al., 1999]) for approximate cosine similarity search. LSH is a popular and highly efficient technique for this problem; we are thus interested in adapting it to process each of LEMP's buckets (Section 3.2).

The key idea of LSH is to use a random hash function to assign probe vectors to bins. The hash function is chosen such that vectors with high cosine similarity are more likely to end up in the same bin than vectors with low cosine similarity. LSH is used as follows: we first group probe vectors into bins according to their hash values in a preprocessing phase. For each query vector $\boldsymbol{q}$, we determine $\boldsymbol{q}$'s bin using the same hash function, and consider as candidates each of the probe vectors in the corresponding bin; all other bins are ignored. Since vectors with high cosine similarity are (more) likely to end up in the same bin, the so-retrieved candidate set is biased towards vectors that are similar to $\boldsymbol{q}$.

In more detail, we use hash functions based on random hyperplanes [Charikar, 2002], which work as follows. We first independently obtain r samples from the standard Normal distribution to form an r-dimensional vector $\boldsymbol{u}$. We view $\boldsymbol{u}$ as the normal vector of a random hyperplane. We then assign each probe vector to a bucket depending on which "side" of the hyperplane it lies, i.e.,

$$h_{\boldsymbol{u}}(\boldsymbol{p}) = \begin{cases} 1 & \boldsymbol{u}^T\boldsymbol{p} \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

One can show that for all pairs of vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^r$ [Charikar, 2002]:

$$Pr_{\boldsymbol{u}}[h_{\boldsymbol{u}}(\boldsymbol{x}) = h_{\boldsymbol{u}}(\boldsymbol{y})] = 1 - \frac{\arccos(\bar{\boldsymbol{x}}^T\bar{\boldsymbol{y}})}{\pi}. \tag{3.14}$$

Note that the right-hand side $s(\boldsymbol{x}, \boldsymbol{y}) = 1 - \arccos(\bar{\boldsymbol{x}}^T \bar{\boldsymbol{y}})/\pi$ is not identical to the cosine similarity $\bar{\boldsymbol{x}}^T \bar{\boldsymbol{y}}$. It is, however, a monotonically increasing function of the cosine similarity, which is sufficient for cosine similarity search.

To make LSH effective, we use l independent hash functions, where $l > 0$ is a parameter. For each vector $\boldsymbol{p}$, we concatenate its hash values into an l-bit binary code, called *signature*. Each of the 2^l potential signatures corresponds to a bin. The parameter l controls both cost and recall: If l is increased, more hash values need to be computed (increasing computational cost), fewer vectors are stored in each bin in expectation (reducing computational costs), and finally two similar vectors are less likely to be mapped to the same bin (reducing recall). To combat the loss in recall, the entire process can be repeated $\mathcal{L}$ times, where $\mathcal{L} > 0$ is another parameter. We then process each query on each of the $\mathcal{L}$ repetitions and union the results. Note that LSH is only effective when the number of probe vectors is larger than $l\mathcal{L}$ (because otherwise we need to compute more inner products to obtain hash values than naive search needs to obtain the exact result).

The most effective combination of l and $\mathcal{L}$ is generally data-dependent. If l is fixed, however, we can determine a suitable value for $\mathcal{L}$ according to the following theorem.

Theorem 3.3. *[Satuluri and Parthasarathy, 2012, Xiao et al., 2011] Let $l > 0$, $\theta > 0$ and $0 < R < 1$. Consider an LSH data structure constructed on $\boldsymbol{P}$ using signatures of length l and*

$$\mathcal{L}(\theta) = \left\lceil \frac{\log(1 - R)}{\log(1 - [1 - \arccos(\theta)/\pi]^l)} \right\rceil \tag{3.15}$$

repetitions. For any query $\boldsymbol{q} \in \mathbb{R}^r$, LSH outputs each probe vector $\boldsymbol{p} \in \boldsymbol{P}$ such that $\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}} \geq \theta$ with probability at least R.

The theorem immediately implies an expected recall of at least R.

Note that LSH has recently been applied to solve the MIPS problem directly [Neyshabur and Srebro, 2015, Shrivastava and Li, 2014a,b]. We discuss these methods in Section 3.7 and study their performance in Section 3.8.

3.4.2 LEMP with Adaptive LSH

In this section, we introduce the LEMP-LSHA algorithm, which makes use of LSH in each of LEMP's buckets. We start with the approximate Above-θ-MIPS problem and then proceed to Top-k-MIPS. In both cases, LEMP-LSHA takes as input a desired recall parameter R and guarantees to output each true result vector with probability at least R. The key idea of LEMP-LSHA is to use an adaptive—i.e., query- and bucket-dependent—number of LSH repetitions to ensure recall R with as low computational cost as possible.

LEMP-LSHA for Above-θ-MIPS. Assume for now that the length l of the hash code is fixed. Recall that LEMP solves many small cosine similarity search problems, one for each of its buckets, and that LEMP uses a query- and bucket-dependent local threshold $\theta_b(\boldsymbol{q})$ for cosine similarity search. Since the local threshold $\theta_b(\boldsymbol{q})$ is not constant, we cannot simply use Eq. (3.15) to determine the number $\mathcal{L}$ of repetitions to use to achieve recall R. The main problem is thus to obtain a suitable choice of $\mathcal{L}$ within the LEMP framework.

The idea of LEMP-LSHA is as follows. We store with each bucket b a number c_b of LSH repetitions; $c_b = 0$ initially for all buckets. When processing query $\boldsymbol{q}$ on bucket b, we compute the local threshold $\theta_b(\boldsymbol{q})$ as before. We then determine the necessary number $\mathcal{L} = \mathcal{L}(\theta_b(\boldsymbol{q}))$ of LSH repetitions needed for this bucket and query according to Eq. (3.15). If $c_b < \mathcal{L}$, we create $\mathcal{L} - c_b$ additional LSH repetitions by reindexing probe vectors and subsequently increase c_b accordingly. In other words, the construction of LSH repetitions is done lazily as needed. After this step, it holds $c_b \geq \mathcal{L}$, i.e., we have a sufficient number of repetitions stored with bucket b. We now pick the first $\mathcal{L}$ of these repetitions to obtain the candidate set using LSH; this step involves computing $l\mathcal{L}$ hash values of the query vector. Note that we may use less than the c_b repetitions stored with bucket b in this step: Since $\mathcal{L}$ repetitions are sufficient to achieve the desired recall, using more than $\mathcal{L}$ repetitions would be wasteful. In addition, we use the same hash functions for all buckets. This allows us to cache and re-use the signatures of the query vector when processing different buckets.

As with most methods for cosine similarity search, LSH is only effective if $\theta_b(\boldsymbol{q})$ is large. Thus we use LSH only when $\theta_b(\boldsymbol{q})$ is large and $\mathcal{L}(\theta_b(\boldsymbol{q}))$ does not exceed a pre-specified space budget. Otherwise, we use the exact LENGTH method. To decide whether or not to use LSH, we use the tuning method described in Section 3.3.4. We subsequently refer to the adaptive version of LSH in combination with LENGTH as LSHA.

The correctness of LEMP-LSHA follows immediately from its construction. We either use LENGTH on each bucket (providing exact results) or use a sufficient number of LSH repetitions (providing recall R).

Theorem 3.4. *Consider the approximate Above-θ-MIPS problem and fix a recall threshold R. For each query $\boldsymbol{q}_i \in \boldsymbol{Q}$ and each probe vector $\boldsymbol{p}_j \in \boldsymbol{P}$ it holds:*

1. *If $\boldsymbol{q}_i^T \boldsymbol{p}_j \geq \theta$, LEMP-LSHA outputs (i, j) with probability at least R.*
2. *Otherwise, if $\boldsymbol{q}_i^T \boldsymbol{p}_j \leq \theta$, LEMP-LSHA does not output (i, j).*

Proof. Fix (i, j). Let b be the bucket that contains $\boldsymbol{p}_j$. Suppose that $\boldsymbol{q}_i^T \boldsymbol{p}_j \geq \theta$. If LEMP uses LENGTH on bucket b, $\boldsymbol{p}_j$ is included into the candidate set because LENGTH is an

exact method. If LEMP uses LSH on bucket b, $\boldsymbol{p}_j$ is included with probability at least R since we use sufficiently many repetitions according to Th. 3.3. This establishes the first assertion. The second assertion holds because LEMP verifies all candidate vectors, i.e., it outputs (i, j) only if $\boldsymbol{q}_i^T \boldsymbol{p}_j \geq \theta$. □

It remains to select the length parameter l of the hash code as well as the space budget for storing repetitions. Parameter l is usually tuned in a dataset-specific way. In our setting, however, buckets contain few vectors by construction (so that they fit into the cache). Since computing hash values requires l inner products per LSH repetition, we cannot afford to use a large value of l; otherwise, LENGTH would be more efficient than LSH. We thus keep l small. For similar reasons and to keep space consumption acceptable, we set the per-bucket budget of LSH repetitions to a relatively small value. In particular, our implementation fixes $l = 8$ and uses a budget of 200 repetitions; these choices provided good results across all datasets in our experimental study.

LEMP-LSHA for Top-k-MIPS. Recall from Section 3.3.5 that Top-k-MIPS for a given query vector $\boldsymbol{q}$ is equivalent to Above-θ^*-MIPS, where θ^* is the k-th largest value in $\boldsymbol{q}^T \boldsymbol{P}$ (we assume in this section that all values of $\boldsymbol{q}^T \boldsymbol{P}$ are distinct). Thus θ^* depends on both $\boldsymbol{q}$ and $\boldsymbol{P}$ and may vary wildly across queries. When LSH is used for top-k search, θ^* is unknown, which poses severe difficulties. One way to support top-k processing is to perform a grid search to select suitable values of l and $\mathcal{L}$ empirically. Another way is to use a sequence of LSH structures with decreasing threshold values; the last LSH structure should use a threshold smaller than the smallest top-k inner product value for any query.[5] The first option does not provide any quality guarantees and is generally cumbersome and inefficient (esp. when queries have wildly varying values of θ^*). This problem has also been observed in our experimental study; see Section 3.8. The second approach is costly because the cost of signature construction is determined by the worst-case query. There is also an inherent risk of constructing too many (high preprocessing cost) or too few (lower recall than desired) of these LSH structures.

In the context of LEMP, we can avoid the problem mentioned above and derive an efficient LSH-based algorithm for Top-k-MIPS. Our algorithm uses the techniques of Section 3.3.5 on LEMP for Top-k-MIPS, but employs LSHA instead of an exact search method in each bucket. In more detail, we maintain a top-k list of the probe vectors with the largest inner products found so far; this top-k list also allows us to obtain a lower bound $\hat{\theta}$ on θ^*. The top-k list is initialized with the k longest probe vectors (all at the start of bucket $\boldsymbol{P}_1$). We then process buckets in order of decreasing length. We use $\hat{\theta}$ to obtain the local threshold $\hat{\theta}_b(\boldsymbol{q})$ for the next bucket b; this local threshold is then used to obtain the candidate set from bucket b with LSHA. After candidates have

[5] http://www.mit.edu/~andoni/LSH/manual.pdf

been obtained, we update the top-k list and $\hat{\theta}$ and proceed to the next bucket (now with the modified value of $\hat{\theta}$). Using this approach, LEMP-LSHA avoids the problems of the plain LSH methods by frequently estimating and updating the threshold value to use.

Theorem 3.5. *Consider the approximate Top-k-MIPS problem and fix a recall threshold R. For each query $\boldsymbol{q}_i \in \boldsymbol{Q}$ and each probe vector $\boldsymbol{p}_j \in \boldsymbol{P}$ such that $\boldsymbol{q}_i^T \boldsymbol{p}_j$ is among the k largest values in $\boldsymbol{q}_i^T \boldsymbol{P}$, LEMP-LSHA outputs (i, j) with probability at least R.*

Proof. Fix (i, j) and suppose that $\boldsymbol{q}_i^T \boldsymbol{p}_j$ is among the k largest values in $\boldsymbol{q}_i^T \boldsymbol{P}$. We know that $\boldsymbol{q}_i^T \boldsymbol{p}_j \geq \theta^*$ by definition of θ^*. Let b be the bucket that contains $\boldsymbol{p}_j$, and let $\hat{\theta}$ be the (random) threshold value that LEMP-LSHA uses to run LSHA on bucket b. Since $\hat{\theta}$ is based on the (approximate) top-k list found so far, we must have $\hat{\theta} \leq \theta^*$. By Th. 3.4, LEMP-LSHA then includes (i, j) into the candidate set with probability at least R. Since $\boldsymbol{q}_i^T \boldsymbol{p}_j$ is among the k largest values in $\boldsymbol{q}_i^T \boldsymbol{P}$, (i, j) will be immediately added to the top-k list and not be removed later on. □

3.4.3 LEMP-ABS and LEMP-REL for Approximate Top-k-MIPS

In this section, we describe the LEMP-ABS and LEMP-REL for Approximate Top-k-MIPS, which provide RMSE and ARE quality guarantees, respectively. Both LEMP-ABS and LEMP-REL can be used with any exact bucket algorithm.

To see how we can use LEMP for approximate Top-k-MIPS, recall the recommender system use case and fix a user (query vector). We are interested in retrieving the top-k recommended items (probe vectors) for that user. Suppose that the ratings of these items (inner products) lie on a scale from 1 (bad) to 5 (great). The key idea of our algorithms is as follows: if the best top-k list of the user contains items with ratings between, say, 4.9–5, then an approximate top-k list with items rated, say, 4.8–5 is almost as good. If the approximate list can be retrieved significantly faster, then this small loss in quality is acceptable: a fast good result may be preferable to a slow perfect result. This observation is exploited by LEMP-ABS and LEMP-REL: Both algorithms augment the threshold computation of LEMP so that LEMP retrieves good, but not perfect, results. In more detail, we use threshold values that are larger than the ones needed for exact Top-k-MIPS, which in turn leads to faster processing times.

The difference between LEMP-ABS and LEMP-REL lies in how the augmentation of the threshold is performed. Recall that LEMP maintains a running lower bound $\hat{\theta}$ on the threshold θ^* for Top-k-MIPS. Before processing each bucket, we augment $\hat{\theta}$ based on an *error parameter* $\epsilon \geq 0$:

- LEMP-ABS augments $\hat{\theta}$ by an additive error term, i.e., we set

$$\hat{\theta}_{abs}(\epsilon) = \hat{\theta} + \epsilon. \tag{3.16}$$

- LEMP-REL augments $\hat{\theta}$ by a relative error term, i.e., we set

$$\hat{\theta}_{rel}(\epsilon) = \begin{cases} \hat{\theta}/(1-\epsilon) & \hat{\theta} \geq 0 \\ \hat{\theta} & \hat{\theta} < 0, \end{cases} \tag{3.17}$$

  for $0 \leq \epsilon < 1$. Note that we do not augment $\hat{\theta}$ if it is negative (in which case the ARE may not be a meaningful measure).

We then compute the local thresholds based on $\hat{\theta}_{abs}(\epsilon)$ or $\hat{\theta}_{rel}(\epsilon)$, respectively, and proceed as in exact LEMP for Top-k-MIPS. Note that each augmented threshold value is larger than the non-augmented threshold when $\epsilon > 0$; both values are equal when $\epsilon = 0$.

The error parameter directly corresponds to quality guarantees on the obtained result. The following theorem establishes that for LEMP-ABS, ϵ is an upper bound on the RMSE of the approximate result.

Theorem 3.6. *For any query $\boldsymbol{q} \in \boldsymbol{Q}$, the RMSE (Eq. (3.1)) of LEMP-ABS for Top-k-MIPS with error parameter $\epsilon \geq 0$ is at most ϵ.*

Proof. Denote as before by $s_1, s_2, \ldots, s_k$ the the values of the inner products in the exact solution of the Top-k-MIPS problem for $\boldsymbol{q}$ in decreasing order, and by $\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_k$ the inner products obtained by LEMP-ABS, again in decreasing order. Assume for the moment that

$$\hat{s}_i + \epsilon \geq s_i \tag{3.18}$$

for $1 \leq i \leq k$. Then

$$RMSE = \sqrt{\frac{1}{k}\sum_{i=1}^{k}(s_i - \hat{s}_i)^2} \leq \sqrt{\frac{1}{k}\sum_{i=1}^{k}((\hat{s}_i + \epsilon) - \hat{s}_i)^2} = \epsilon,$$

as desired.

It remains to show that (3.18) holds for all i. To see this, observe that for each of its buckets, LEMP-ABS uses a threshold $\hat{\theta}$ that satisfies $\hat{\theta} \leq \hat{s}_k$. This is because LEMP-ABS takes $\hat{\theta}$ to be the lowest inner product value in the current top-k list, which is upper bounded by its final value $\hat{s}_k$. This implies that $\hat{\theta}_{abs}(\epsilon) = \hat{\theta} + \epsilon \leq \hat{s}_k + \epsilon$ for all

buckets. Denote by $j_1, \ldots, j_k$ the exact result of Top-k-MIPS; we have $s_i = \boldsymbol{q}^T \boldsymbol{p}_{j_i}$. Let $u \leq k$ be the largest index such that $s_u > \hat{s}_k + \epsilon$ (if such an index exists). Pick any j_i, $1 \leq i \leq u$, and denote by b the bucket that contains $\boldsymbol{p}_{j_i}$. Since $\hat{\theta}_{abs}(\epsilon) \leq \hat{s}_k + \epsilon$ for all buckets, including bucket b, and since $\boldsymbol{q}^T \boldsymbol{p}_{i_j} > \hat{s}_k + \epsilon$, LEMP-ABS will include j_i into its candidate list when processing bucket b. Since $\boldsymbol{q}^T \boldsymbol{p}_{j_i}$ is among the k-th largest inner product values overall, j_i will subsequently be added to the top-k list and not be evicted later on. Thus all indexes $j_1, \ldots, j_u$ are included in the final top-k list of LEMP-ABS. For $i \leq u$, we thus have $s_i = \hat{s}_i$ so that Eq. (3.18) holds. Now consider any index $i > u$. We have $s_i \leq \hat{s}_k + \epsilon$ by our choice of u. Since $\hat{s}_k \leq \hat{s}_i$, it follows that $s_i \leq \hat{s}_i + \epsilon$, i.e., Eq. (3.18) holds. □

The error bound on the RMSE obtained by LEMP-ABS is absolute, i.e., it does not depend on the scale of the values in the actual result. In cases where the top-k values can differ wildly across different queries, it may be more appropriate to use relative error bounds instead. This means that we require small error for results with small top-k values, but allow for larger error for results with large top-k values. Such bounds are achieved by LEMP-REL.

Theorem 3.7. *For any query $\boldsymbol{q} \in \boldsymbol{Q}$, the ARE (Eq. (3.2)) of LEMP-REL for Top-k-MIPS with error parameter $0 \leq \epsilon < 1$ is at most ϵ.*

Proof. Using the notation above, suppose that $\hat{s}_k < 0$ when LEMP-REL terminates. Then for all buckets, we must have had $\hat{\theta} < 0$ (since $\hat{\theta} \leq \hat{s}_k$ by definition) and thus $\hat{\theta}_{rel}(\epsilon) = \hat{\theta}$. This implies that whenever $\hat{s}_k < 0$, LEMP-REL did not augment the threshold and thus produced exact results. The ARE is thus 0 and the assertion holds.

Now suppose that $\hat{s}_k \geq 0$. Then we can show using arguments as in the proof of Th. 3.6 that

$$\hat{s}_i/(1-\epsilon) \geq s_i \tag{3.19}$$

for $1 \leq i \leq k$. The ARE satisfies

$$ARE = \frac{1}{k} \sum_{i=1}^{k} \left| \frac{s_i - \hat{s}_i}{s_i} \right| = \frac{1}{k} \sum_{i=1}^{k} \left| 1 - \frac{\hat{s}_i}{s_i} \right| \leq \frac{1}{k} \sum_{i=1}^{k} \left| 1 - \frac{\hat{s}_i}{\hat{s}_i/(1-\epsilon))} \right| = \epsilon \tag{3.20}$$

as asserted. □

To further improve the performance of LEMP-ABS and LEMP-REL, we use the augmented thresholds $\hat{\theta}_{abs}(\epsilon)$ and $\hat{\theta}_{rel}(\epsilon)$ only for candidate generation but not during verification. That is, we update the top-k list by taking into the consideration all candidate vectors.

3.5 Implementation Details

In this section, we give some guidance on how to implement the COORD, ICOORD, and LSHA algorithms efficiently.

COORD. In our implementation, we store the sorted-list indexes column-wise to reduce memory bandwidth: the data values are accessed only during binary search to determine the scan range, and the local identifiers are accessed only during the actual scan phase. For efficiency reasons, we also avoid clearing the CP array when moving from one query vector to the next. Instead, we keep the array uninitialized and proceed as follows. When scanning the first sorted list, we set to 1 instead of incrementing the corresponding entry of the CP array and increment while scanning the remaining sorted lists. After all lists have been scanned, we scan the first sorted list again and only consider the corresponding entries of the CP array for inclusion into the candidate set. Since the first sorted list is scanned twice (for CP array initialization and filtering), we take the focus coordinate with the smallest scan range as the first one.

ICOORD. Since ICOORD needs access to both coordinate values and local identifiers during scanning, we store the sorted lists row-wise. The extended CP array is initialized and accessed in the same way as the CP array of COORD. In order to reduce memory bandwidth and avoid excessive checking, we do not keep the counter information of COORD in the extended CP array: the filtering condition of Eq. (3.13) is usually pruning vectors more aggressively than the simple check of COORD. Since Eq. (3.13) contains expensive floating-point operations (such as divisions and square roots), we rewrite the conditions and accept a vector $\bar{\boldsymbol{p}}$ if:

$$\bar{\boldsymbol{q}}_F^T \bar{\boldsymbol{p}}_F \|\boldsymbol{p}\| > \theta / \|\boldsymbol{q}\|,$$

for which the right-hand side needs to be computed only once. If this test fails, we accept $\bar{\boldsymbol{p}}$ if and only if:

$$\|\boldsymbol{p}\|^2 \|\boldsymbol{q}\|^2 (1 - \|\bar{\boldsymbol{p}}_F\|^2)(1 - \|\bar{\boldsymbol{q}}_F\|^2) \geq (\theta - \bar{\boldsymbol{q}}_F^T \bar{\boldsymbol{p}}_F \|\boldsymbol{p}\| \|\boldsymbol{q}\|)^2.$$

ICOORD's strength lies into accumulating partial inner products from many lists. If we decide to use $\phi_b = 1$ for some bucket b, ICOORD and COORD will produce the same candidate set, but COORD does so faster. We thus use COORD instead of ICOORD whenever $\phi_b = 1$.

LSHA. To create the random vectors $\boldsymbol{u}$ from the standard Normal distribution, we follow the approach of Satuluri and Parthasarathy [2012], which allows for compressed

storage. In addition, using Eq. (3.15) for each query-bucket pair can be expensive. In our implementation, we precompute and cache the smallest local threshold that corresponds to each one the 200 signatures in the budget. During query processing, we perform binary search on these values to find $\mathcal{L}$ for each $\theta_b(\boldsymbol{q})$.

3.6 Parallelizing LEMP

In this section, we show how to take advantage of multithreading and instruction level parallelism for MIPS. When multiple queries are considered, the MIPS problem becomes embarrassingly parallel: queries can be partitioned among threads; each thread can run its own instance of the problem. All algorithms described in this chapter (LEMP, naive, cover trees [Curtin et al., 2013], TA [Fagin et al., 2001], simpleLSH [Neyshabur and Srebro, 2015], PCA-tree [Bachrach et al., 2014], etc.) can be parallelized in this way. Here we show how to apply such ideas to LEMP in an efficient way that scales to many processors.

Multithreading. When multithreading is used, it is important for scalability to a large number of cores to avoid synchronization and cache misses to the extent possible. We avoid cache misses by enforcing during bucketization the restriction that each bucket should fit into the available cache per core. Synchronization, on the other hand, can in general take place in LEMP in three places: (i) when a thread writes its results to memory, (ii) when a thread needs to obtain the next query to work on, and (iii) when a thread needs to access an index which is not yet created. To handle (i), we assign to each thread a separate memory area to write its results. The final result is then given by the union of these memory areas. To handle (ii), we partition the queries among the threads during the preprocessing phase. Thus each thread knows upfront which queries it is responsible for and no further synchronization is needed. To ensure a similar workload among threads, queries are assigned to partitions randomly. We handle (iii) as follows. When a thread needs to access an index which is not yet created during the search phase, it needs to obtain exclusive access to the index in order to build it. While the index is being built, all other threads that try to read this index need to wait; we want to minimize such waiting times. For the Above-θ-MIPS problem, in which we know θ upfront, we compute in advance which buckets can contribute to the result in the worst case (longest query vector). The indexes of these buckets are created in parallel during the preprocessing phase. Thus no synchronization is needed during the search phase. For the Top-k-MIPS, for which the set of required indexes is not known upfront, synchronization is inevitable. However, we can reduce synchronization overhead during search phase as follows. Recall that during tuning, we run a sample of queries against

the probe buckets, so that their performance w.r.t. the LENGTH algorithm and other direction-based algorithms (for which we need indexes) is assessed. After we assess the performance of the sample w.r.t. LENGTH, we have a first crude estimate of how many buckets are contributing to the result. At this point, we pause the tuning phase, build the estimated number of required indexes for these buckets in parallel and then continue tuning. In this way, a large part of the required indexes is created before the search phase starts. If additional indexes are needed during the search phase, we build them on demand and require synchronization. Our experiments suggest that the combination of these techniques allows LEMP to scale almost linearly to a large number of processors.

Instruction-level parallelism. LEMP's performance (as well as the one of some other methods) can benefit from the use of instruction-level parallelism. As a proof of concept, we extended LEMP to use SSE instructions to speedup (i) the inner-product calculation during the verification phase, and (ii) the maintenance of the extended CP array. Recall that while ICOORD is scanning the sorted lists, it maintains for each encountered probe vector both a partially seen inner product and an upper bound on the remaining unseen part. Maintaining these quantities involves two multiplications, which we parallelize using SIMD instructions.

3.7 Related Work

A number of existing methods for and related to the MIPS problem have been proposed in the literature. We first review existing algorithms for exact MIPS and subsequently turn attention to cosine similarity search algorithms. Finally, we review approximate methods for MIPS. In general, LEMP differs from existing methods in that it separates the length and direction of the input vectors, prefers inexpensive pruning strategies over more aggressive, expensive ones, selects suitable search methods dynamically, and provides approximation guarantees for approximate MIPS.

3.7.1 Exact Methods

Algorithms for MIPS. To the best of our knowledge, Ram and Gray [2012] were the first to pose and address the problem of Top-k-MIPS. They organize the probe vectors in a *metric tree* in which each node is associated with a sphere that covers the probe vectors below the node. Given a query vector, the spheres are exploited to avoid processing subtrees that cannot contribute to the result. The metric tree itself is constructed by repeatedly splitting the set of probe vectors into two partitions (based on Euclidean distances). In subsequent work [Curtin et al., 2013], the metric tree is

replaced by a *cover tree* [Beygelzimer et al., 2006]. Both approaches effectively prune the search space, but they suffer from high tree-construction costs and from random memory access patterns during tree traversal. The latter problem was investigated more closely by Curtin and Ram [2014], who proposed a *dual-tree algorithm* that additionally arranges query vectors in a cover tree and processes queries in batches. The dual-tree methods loosens the bounds for pruning the search space, however, and was found to be ineffective in practice (confirmed also in our experimental study).

LEMP differs from these tree-based techniques in that it separates length and direction information, makes use of multiple, light-weight indexing and search methods, and has more favorable memory access patterns. Note that the single-tree approach can also be used within the LEMP framework as a bucket algorithm, which directly solves the MIPS problem. We expect that such a combination will have positive effect w.r.t. indexing time and cache locality. This was also confirmed in our experimental study.

An alternative approach is taken by Zhang et al. [2014] in the context of recommender systems: the matrix factorization method used to produce the input matrices is modified, such that all vectors are (approximately) unit vectors and the inner product of user and item vectors can be approximated by standard cosine similarity search. However, this modification may affect the quality of the recommendations and is not suitable for all applications. In contrast, LEMP makes no assumption on the source or method used to compute the input matrices.

Threshold algorithm. Some of our indexing techniques are inspired by the popular threshold algorithm (TA) of Fagin et al. [2001] for top-k query processing for monotonic functions. TA arranges the values of each coordinate of the probe vectors in a sorted list, one per coordinate. Given a query, TA repeatedly selects a suitable list (e.g., round robin or heap-based), retrieves the next vector from the top of the list, and maintains the set of the top-k results seen so far. While scanning the lists, TA maintains an upper bound on the score of any unseen item, which is monotonically reduced with each step the algorithm makes. Whenever this upper bound falls below the threshold θ (or the score of the kth element in the top-k results), TA can safely terminate. If TA is able to stop early, it can be very efficient. This early termination depends on the data and the list selection schedule. Note that TA usually focuses on vectors of low dimensionality (say up to 10), whereas we focus on vectors of medium sizes (say 10 to 500). TA can be used for finding vectors with large inner products almost as is; the only difference is that sorted lists need to be processed bottom-to-top when the respective coordinate of the query vector is negative.

LEMP improves over TA in multiple ways: First, bucket pruning eliminates early all short probe vectors, which otherwise TA would have to consider. Second, TA scans

lists from top-to-bottom, whereas LEMP considers only the feasible region. Third, TA immediately computes the inner product of each vector selected from one of the lists in the index, i.e., candidate verification is triggered by individual coordinates. LEMP does not immediately calculate an inner product when it encounters a vector: it first scans multiple lists and prunes the vectors before verification, based on the so-obtained information. Finally, index scan and verification are interleaved in TA, resulting in a random memory access pattern and a potentially high cache-miss rate. LEMP ensures that all bucket-related data (original vectors and indexes) fits into cache, thereby reducing the cache-miss rate.

In our experimental study we investigated the performance of TA in comparison to LEMP. We also experimented with TA in combination with LEMP, i.e., we used TA as a bucket algorithm. This addresses the first and the final point in the discussion above. Our experimental results indicate that a combination of TA and LEMP can be up to 17x faster than just using TA. Generally, LEMP can improve TA's performance for top-k problems with linear scoring functions (i.e., inner products).

Algorithms for fast cosine similarity search. Exact cosine similarity search algorithms, like all-pairs similarity search (APSS, [Bayardo et al., 2007, Chaudhuri et al., 2006, Lee et al., 2010, Xiao et al., 2008]), cannot be used directly for the MIPS problem. However, these methods can be used (with some modifications) as search methods for LEMP's buckets.

Typical APSS algorithms and applications involve sparse vectors of high dimensionality (tens or hundreds of thousands of coordinates). In such settings, sparsity must be retained during indexing to keep the index size manageable. Thus APSS algorithms generally index only the non-zero values of each coordinate (in contrast to LEMP). In addition, coordinates are often permuted such that dense coordinates (called prefix) appear before sparser coordinates (suffix); only the suffix is indexed. The index is used to obtain candidate vectors, which are further pruned based on properties of prefixes and suffixes [Anastasiu and Karypis, 2014, Lee et al., 2010, Xiao et al., 2008]. Finally, full similarity scores are computed for each candidate.

L2AP [Anastasiu and Karypis, 2014] is a state-of-the-art APSS algorithm for exact cosine similarity search; it exploits the Euclidean norms of suffixes and prefixes for index compression and candidate filtering. L2AP can be used as a bucket algorithm for LEMP after a few modifications. In particular, we create a separate L2AP index for each bucket. In L2AP, like in most APSS algorithms, a lower bound on the cosine similarity threshold needs to be fixed a priori. In our setting, we pick the lower bound $\theta_b(\boldsymbol{q}_{max})$, where $\boldsymbol{q}_{max}$ is the query vector with the largest length.

L2AP follows a similar pruning technique to ICOORD during candidate generation and verification: it accumulates $\bar{\boldsymbol{q}}_F^T \bar{\boldsymbol{p}}_F$ and precomputes $u(\bar{\boldsymbol{q}}_F, \bar{\boldsymbol{p}}_F)$. ICOORD differs in the following ways: (i) L2AP scans all indexed lists corresponding to non-zero query coordinates, whereas ICOORD scans only ϕ of them and only their feasible regions, (ii) L2AP uses sophisticated filtering conditions both during and after scanning. These filtering techniques eliminate the majority of the candidates, but are generally expensive. In contrast, ICOORD filters candidates only once and after index scanning, which is cheap but may result in a larger number of candidates. See Section 3.8 for an experimental comparison of the two methods.

3.7.2 Approximate Methods for MIPS

Koenigstein et al. [2012] approached the approximate Top-k-MIPS problem by clustering the query vectors and solving the Top-k-MIPS problem only for the cluster centroids. The results for the centroids are taken as approximate results for all the queries in the respective cluster. The authors derive also relative error bounds (ARE) on the results, based on the cosine similarity of the query and the centroid. If this bound is larger than a desired value, the algorithm falls back to exact search for that specific query. Such a method can be directly applied in combination with LEMP. We do not consider such an approach here because it was outperformed by PCA-trees in previous studies (see below). Moreover, the clustering phase may be expensive and the method's performance heavily depends on the quality of the clusters (and the number-of-clusters parameter). Finally, the clustering approach is not suitable for online processing since all queries need to be known in advance.

Recently, a number of novel methods have been proposed that perform transformations of the query and/or probe vectors such that MIPS is reduced to nearest neighbor search (NN) in Euclidean space [Bachrach et al., 2014, Shrivastava and Li, 2014b] or to cosine similarity search [Neyshabur and Srebro, 2015, Shrivastava and Li, 2014a] on the transformed vectors. The existence of such transformations is promising because they enable the direct use of existing methods for NN or cosine similarity search. All transformations slightly increase the dimensionality of the data vectors, either by one [Bachrach et al., 2014, Neyshabur and Srebro, 2015] or by two [Shrivastava and Li, 2014a,b]. The additional coordinates generally hold information related to the length of the vector. LEMP differs from these methods in that it exploits length information directly via bucketization, rather than indirectly via transformation. This allows LEMP to perform quick initial length-based pruning for many buckets and to select suitable search algorithms for the remaining buckets. In the remainder, we discuss the transformation methods in more detail.

Bachrach et al. [2014] showed how to reduce the Top-k-MIPS to an equivalent nearest-neighbor problem in Euclidean space by introducing the following asymmetric transformations for the query and probe vectors, respectively:

$$t_{query}(\boldsymbol{q}) = (0, \boldsymbol{q}),$$
$$t_{probe}(\boldsymbol{p}) = (\sqrt{(\max_i \|\boldsymbol{p}_i\|)^2 - \|\boldsymbol{p}\|^2}, \boldsymbol{p}).$$

Note that the added coordinate is large (and often dominant) for short probe vectors and small for long probe vectors. The authors then build a so-called PCA-tree on the transformed probe vectors. The PCA-tree is a binary tree of depth $d \ll r + 1$, which is formed by splitting probe vectors based on their first d principal components. Probe vectors are thus partitioned across the leaves of the tree. During query processing, the tree is traversed to find a set of d leaves (and the corresponding probe vectors) which best match the transformed query. The input parameter d controls the trade-off between speedup and quality: The larger d, the fewer total number of candidates, the larger the speedup, and the lower the quality of the result. The PCA-tree method does not provide any error bounds, but was empirically shown to outperform the clustering method of Koenigstein et al. [2012].

It is known that LSH cannot be used to solve MIPS on the original vectors [Neyshabur and Srebro, 2015, Shrivastava and Li, 2014b]. Shrivastava and Li [2014b] derived a transformation, which allows the use of LSH methods for Euclidean distances. In their later work, Shrivastava and Li [2014a] proposed an alternative asymmetric transformation for LSH for cosine similarity search, which provided better results. Neyshabur and Srebro [2015] proposed a transformation similar to the one of Bachrach et al. [2014] described above; they also use this transformation to employ LSH for cosine similarity search. The resulting simpleLSH scheme outperformed the methods of Shrivastava and Li [2014a,b] in their experimental study. In Section 3.8, we compare the performance of PCA-tree and simpleLSH with our methods. More recent work of Ballard et al. [2015], uses *diamond sampling* to solve the Row-Top-k problem. Intuitively, with diamond sampling, probe vectors whose high-valued coordinates coincide with the high-valued coordinates of the query have higher chances to be sampled (and the more coordinates they agree on the higher the chances of being sampled). In the end, the probe vectors that are more frequently sampled are the best candidates for having large inner product to the query. A comparison of LEMP with diamond sampling remains for future work.

3.8 Experimental Study

We conducted an extensive experimental study using multiple real-world datasets. The goals and results of our experimental study are summarized below:

- We investigated the performance of various state-of-the-art methods for exact MIPS: LEMP, naive search (Naive, Section 3.7), the threshold algorithm (TA), and the single and dual cover tree approaches (Tree, D-Tree). We found that LEMP consistently outperformed alternative exact methods and was the best-performing method overall. In particular, LEMP was up to multiple orders of magnitude faster than Naive and between 2x and 20x faster than the best-performing alternative method.
- We studied the relative performance of different bucket methods for exact MIPS, including COORD, ICOORD, TA, cover trees, and L2AP. We found that a combination of LENGTH and ICOORD was the most efficient bucket-search method overall.
- We investigated the effect of the dimensionality r of the input vectors on each algorithm's performance for exact MIPS. Our results suggest that LEMP maintains its performance advantage across all dimensionalities we considered.
- For approximate MIPS, we compared LEMP with the state-of-the-art methods PCA-tree and simpleLSH. We found that LEMP outperformed the alternative methods and provided a better speed-quality tradeoff. For example, LEMP was up to 3.9x faster than the best alternative method at similar recall levels.
- Finally, we studied the scalability of our parallel LEMP variant. Our method had near linear speedups up to 32 processors (the largest number considered in our experiments) and became up to 23% faster when SIMD instructions were used.

3.8.1 Experimental Setup

All datasets and our source code can be found at `http://dws.informatik.uni-mannheim.de/en/resources/software/lemp`.

Hardware. Our experiments were run on a machine with 48 GB RAM and an Intel Xeon 2.40GHz processor. Unless stated otherwise, our experiments were carried out on a single thread and no SIMD instructions were used.

Dataset	m	n	r	CoV of lengths $\boldsymbol{Q}$	CoV of lengths $\boldsymbol{P}$	% Non-Zero	Naive (min)
IE-NMF	771K	132K	10	2.05	5.49	28.2	27.4
			50	1.56	5.53	36.2	112.0
			100	1.34	4.45	50.8	280.5
IE-SVD	771K	132K	10	2.04	5.46	100	29.1
			50	1.51	4.44	100	113.0
			100	1.28	3.64	100	249.5
Netflix	480K	17K	10	0.12	0.16	100	1.5
			50	0.16	0.22	100	5.6
			100	0.19	0.22	100	17.5
KDD	1000K	624K	51	0.38	0.40	100	838.3

TABLE 3.1: Overview of datasets

Datasets. We used real-world datasets from collaborative filtering and information extraction applications (cf. Section 3.1.2). Table 3.1 summarizes our datasets. The table gives the sizes of the input data and for various choices of rank r, the coefficient of variation (CoV) of the lengths of the input vectors, the percentage of non-zero entries and the time required by Naive (see the discussion in Section 3.8.2)

For our experiments with collaborative filtering data, we used factorizations of the popular Netflix [Bennett and Lanning, 2007] and KDD [Dror et al., 2012] datasets.[6] Both datasets consist of ratings of users for movies (Netflix) or musical pieces (KDD). For Netflix, we performed a plain matrix factorization with DSGD++ using L2 regularization with regularization parameter $\lambda = 50$, as in Teflioudi et al. [2012]. For KDD, we used the factorization of Koenigstein et al. [2011],[7] which incorporates the music taxonomy, temporal effects, as well as user and item biases; this dataset has been used in previous studies of the Top-k-MIPS problem. Since we were ultimately interested in retrieving the top-k movies/songs for each user, we used the collaborative filtering datasets to study the performance of the various methods for the Top-k-MIPS problem.

For the open information extraction scenario, we extracted around 16M subject-pattern-object triples from the New York Times corpus,[8] which contains news articles, using the methods described in Nakashole et al. [2012]. We removed infrequent arguments and patterns, and constructed a binary argument-pattern matrix: An entry in the matrix was set to 1 if the corresponding argument (subject-object pair) occurred with the corresponding pattern; otherwise, the entry was set to 0. We factorized this binary matrix using the singular-value decomposition (SVD) and non-negative matrix factorization (NMF); we denote the resulting datasets as IE-SVD and IE-NMF, respectively. For SVD,

[6] The KDD (Yahoo! Music) dataset corresponds to Track 1 of the 2011 KDD-Cup.
[7] We zeroed out all subnormal numbers.
[8] http://catalog.ldc.upenn.edu/LDC2008T19

which produces factorization $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$, we set $\boldsymbol{Q}^T = \boldsymbol{U}\sqrt{\boldsymbol{\Sigma}}$ and $\boldsymbol{P} = \sqrt{\boldsymbol{\Sigma}}\boldsymbol{V}^T$. For the IE datasets, we studied Above-θ-MIPS and Top-k-MIPS, which are both relevant in applications. Above-θ-MIPS aims to find all high-confidence facts, whereas Top-k-MIPS retrieves the k most probable arguments of a pattern (as in Riedel et al. [2013]). For the latter problem, we make use of the transposed matrices IE-SVD^T and IE-NMF^T.

We factorized Netflix, IE-SVD and IE-NMF with ranks 10, 50 and 100. Unless stated otherwise, we use rank $r = 50$. We investigate the effect of other choices in Section 3.8.3. As stated previously, fast and scalable matrix factorization algorithms have been proposed in the literature so that the time for matrix factorization is often not a bottleneck in applications. For example, we obtained IE-SVD ($r = 50$) and IE-NMF ($r = 50$) in less than four minutes each using Matlab. As another example, in chapter 2 we factorized the KDD dataset ($r = 50$) with CSGD in roughly seven minutes. In all three cases, the factorization time is significantly larger than the time required to perform MIPS using Naive so that MIPS is the main bottleneck.

Algorithms. We implemented LEMP and TA in C++, and used the C++ code of Tree and D-Tree provided by the authors of Curtin et al. [2013] and Curtin and Ram [2014].[9] For L2AP, we adjusted the publicly available C code.[10] For PCA-trees [Bachrach et al., 2014] and simpleLSH [Neyshabur and Srebro, 2015], we created our own C++ implementation. SimpleLSH uses the LSH implementation that we use for LEMP-LSHA with minor modifications.

We ran six "pure" versions of LEMP for exact MIPS, in which only one method was used within a bucket. We denote these methods as LEMP-X, where X is: L for LENGTH, C for COORD, I for ICOORD, TA for TA, L2AP for L2AP and Tree for cover tree. We also ran the two mixed versions LEMP-LC (LENGTH and COORD) and LEMP-LI (LENGTH and ICOORD), in which the appropriate search method is chosen as described in Section 3.3.4. Unless stated otherwise, we use LEMP-LI (and denote it LEMP). For approximate MIPS, we ran LEMP-LSHA, LEMP-REL, and LEMP-ABS.

For TA, we experimented with two different list-selection schedules: a round robin (RR) on the lists corresponding to non-zero query coordinates and one that selects the sorted list i that maximized $q_i p_i$, where p_i refers to the next coordinate value in list i. The latter strategy selects the "most-promising" coordinate; we implemented it efficiently using a max-heap. We additionally improve the performance of TA by allowing multiple steps on the same list if all these steps access probe vectors that are already explored. In this way, we reduce the stopping threshold faster, while incurring almost no overhead (no

[9] http://mlpack.org/
[10] http://glaros.dtc.umn.edu/gkhome/l2ap/overview

candidate verification takes place, since the probe vectors are already explored). Finally, we report the best results achieved by TA (RR or heap-based).

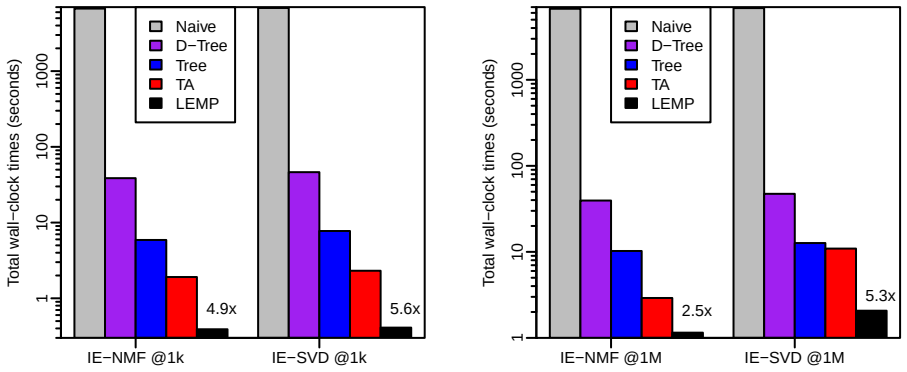
Methodology. We compare all algorithms for both the Above-θ-MIPS and the Top-k-MIPS problems. For Top-k-MIPS, we experimented with $k \in \{1, 5, 10, 50\}$. For the Above-θ-MIPS problem, we selected θ such that we retrieve the top-10^3, -10^4, -10^5, -10^6 and -10^7 entries in the entire product matrix $\boldsymbol{Q}^T\boldsymbol{P}$. We subsequently refer to this number of results as *retrieval level*.

Unless otherwise stated, we compare all methods in terms of overall wall-clock time, which includes preprocessing, tuning, and retrieval time. Preprocessing involves the construction of indexes (cover trees for Tree and D-Tree; sorted lists for LEMP, TA and L2AP; LSH signatures for LEMP-LSHA and simpleLSH; tree for PCA-tree) and, for LEMP only, the time required for the normalization, sorting, and bucketization of the input vectors. Tuning refers to the time required to automatically select suitable values for the parameters ϕ and t_b of LEMP.

Choice of parameters. LEMP's parameters (ϕ and t_b) were tuned on a small sample of the datasets as explained in Section 3.3.4. The base parameter of the cover trees was set to 1.3 as suggested by Curtin and Ram [2014]. For all LEMP algorithms, we used a fine-grained bucketization such that all data structures of a bucket fit into the available processor cache. For LEMP-L2AP, we used the same combination of filters and bounds that Anastasiu and Karypis [2014] report as most efficient w.r.t. execution time. For LEMP-LSHA, we set the signature length to 8 bits and maximum number of signatures to be used to 200.

3.8.2 Exact MIPS

In this section, we compare LEMP with previous methods for exact MIPS. Figures 3.5 and 3.6 show the relative performance of LEMP (using the LI bucket algorithm), TA, Tree and D-Tree for the Above-θ-MIPS and Top-k-MIPS problems, respectively. The speedup of LEMP with respect to the best-performing method other than LEMP is marked in the figures. We use Naive as a baseline; its running time is independent of θ for Above-θ-MIPS and only slightly affected by k for Top-k-MIPS. To keep our study manageable, we only ran Naive for the Top-1-MIPS problem; this is a fair comparison because running times for larger k may be slightly above, but not below the times reported here. The wall-clock times for this and additional experiments, as well as average candidate set sizes, can be found in Tables B.1 and B.2 in the appendix.



(a) Above-θ-MIPS @1K (b) Above-θ-MIPS @1M

FIGURE 3.5: Total wall-clock times (incl. indexing and tuning) for exact Above-θ-MIPS @1K and @1M on different datasets

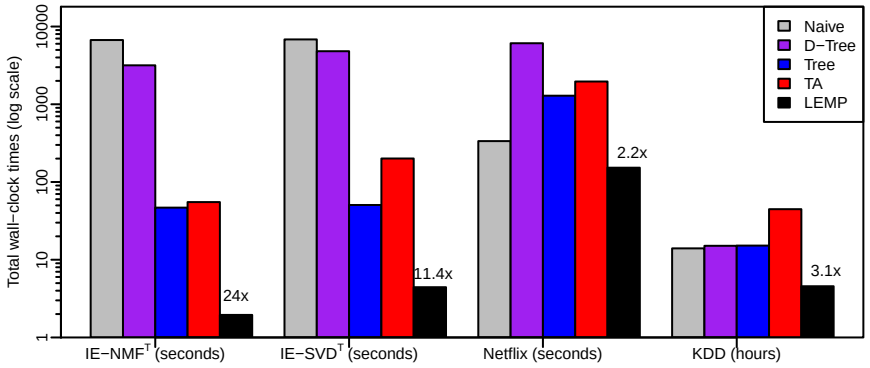


FIGURE 3.6: Total wall-clock times (incl. indexing and tuning) for exact Top-1-MIPS on different datasets

In the following, we discuss the performance of the algorithms in terms of overall running time, preprocessing time and pruning power.

Overall performance. In general, LEMP was the fastest method, reaching up to 17000x speedup over the Naive baseline and up to 24x speedup over the next best method. The second fastest method in the majority of cases was Tree, followed by TA and D-Tree. LEMP, Tree, and TA appear to have best performance on datasets with large skew in their length distribution, like the IE datasets (high CoV in Table 3.1) and also on datasets with sparse vectors (IE-NMF). On datasets with little skew in their length distribution, like Netflix and KDD, all methods had difficulties in providing large speedups over Naive. However, for KDD, some methods were still able to offer significant savings in terms of running time: e.g., LEMP took 9.4 hours less time than Naive. Tables B.1 and B.2 show that the performance of all methods but Naive deteriorates as the result size or k increases (θ decreases), since the output size increases and pruning opportunities decrease. Generally, there is a break-even point at which any method will

Dataset	LEMP	TA	Tree	Dual Tree
IE-NMF	0.78	0.46	5.33	38.5
IE-SVD	1.18	0.76	7.1	46.2
IE-NMF^T	0.84	2.98	31.84	38.5
IE-SVD^T	1.51	4.88	37.5	46.4
Netflix	1.63	0.10	1.10	3510.9
KDD	63.32	4.47	208.7	2880.0

TABLE 3.2: Maximum preprocessing times (in seconds) including indexing and tuning

be slower than Naive. This is the case, for example, for all methods other than LEMP on Netflix/KDD for $k \geq 1$.

Preprocessing time. Table 3.2 shows the preprocessing time for the different datasets and methods. For Tree and D-Tree, we give the wall-clock time of producing the cover tree(s) and for TA the time to create the sorted lists. The preprocessing costs of these methods are fixed and depend on the size of probe matrix (and additionally of the query matrix for D-Tree). For LEMP, we report the sum of maximum indexing and the maximum tuning time (normally the preprocessing times vary from problem to problem since LEMP constructs indexes lazily). On the one hand, LEMP suffers from tuning overhead, but on the other hand, it benefits from lazy index construction, especially for datasets with skewed length distribution. The larger the length skew and the size of the probe matrix, the larger the preprocessing savings of LEMP over the other methods and the higher the chances of outweighing the tuning overhead. For example, for IE-NMF^T, which has $n = 771K$, LEMP needed 0.84s vs. 2.98s for TA and 31.84s for Tree. The highest costs appeared for the Tree and D-Tree methods. Preprocessing costs can be one of the major bottlenecks for these methods. In fact, Tables B.1 and B.2 show that preprocessing can be a large part of the overall running time, specially for datasets with large length skew. For example, D-Tree needed more time to create its trees for Netflix (tree construction was 80% of the overall time) than Naive needed to retrieve the Row-Top-1 entries. Similarly, for the IE datasets (retrieval level $\leq 10^6$, $k \leq 10$), LEMP (and, in some cases, also TA) terminated before the Tree method finished preprocessing.

Pruning power. Tables B.1 and B.2 show how many candidates remain on average after pruning for each of the different methods. For the Top-k-MIPS problem, LEMP had the highest pruning power for the IE datasets and Netflix. Note that LEMP was the only method outperforming Naive on Netflix. In fact, it is difficult to improve on Naive on this dataset: Netflix has the smallest length skew, which makes pruning less effective, and a relatively small probe matrix, which makes Naive perform reasonably well. TA ranked often third in terms of pruning power. Especially for datasets with low length skew, TA tended to perform poorly. For example, for Netflix, $k = 1$, TA had almost no pruning power (16K candidates per query, out of a total of 17.7K). We also see the

effect of TA's random memory access pattern here. Although TA verified almost the same number of candidates as Naive, it was 5.8x slower (1961.8s vs. 335.8s). Also note that sparsity affects the behavior of TA: It checked 3.2x less candidates for the sparse IE-NMF^T dataset, $k = 1$, than for IE-SVD^T (1899 vs. 6090 candidates per query). The main reason for the relatively low pruning power of TA for dense datasets is that it is length-oblivious, i.e., it checks short probe vectors if they have a single, sufficiently large coordinate; these vectors are discarded by LEMP. On the other hand, for sparse datasets, large values for individual coordinates correlate well with the length of the vectors so that essentially TA explores long vectors first. We expect that a combination of LEMP and TA can address the problems of length-obliviousness and random memory accesses; see Section 3.8.5.

For the D-Tree, given a fixed, high θ value (as in the Above-θ-MIPS problem), the grouping of queries helps to reduce the frequency of visits of the probe-tree nodes (and thus the candidate checking). D-Tree was actually able to prune more candidates than all other methods for this problem. For the top-k case, the bounds for a group of queries depend on the worst running lower bound $\hat{\theta}$ among all queries of the group. Thus, for the top-k problem, D-Tree had usually looser bounds and, therefore, less pruning power than Tree.

Cache exploitation. Recall that LEMP does not create buckets that exceed the cache size. To study the effect of this approach, we experimented with a cache-oblivious version of LEMP in which bucket sizes were unrestricted. We found that for datasets with large length skew, runtime differences were marginal: LEMP creates small buckets anyway when lengths are skewed. For datasets with less length skew, such as KDD, there was a significant difference in runtime: LEMP created more than 15x more buckets than its cache-oblivious version (26 vs. 403), and was almost 40% faster (7.9h vs. 4.56h).

3.8.3 Influence of Dimensionality

In our next experiment, we investigated the impact of the dimensionality r of the input vectors (the rank of the factorization) on the performance of exact MIPS algorithms.[11] We experimented with ranks 10, 50 and 100. The properties of the resulting datasets are summarized in Table 3.1.

Figure 3.7 shows the performance (in log scale) of Tree, D-Tree, TA, LEMP (again, using LEMP-LI) and Naive for the Top-k-MIPS problem for $r \in \{10, 50, 100\}$ and various values of k. We omit results for each method when it performed worse than Naive. As

[11] We omit the KDD dataset from this set of experiments because it was only available to us with $r = 51$.

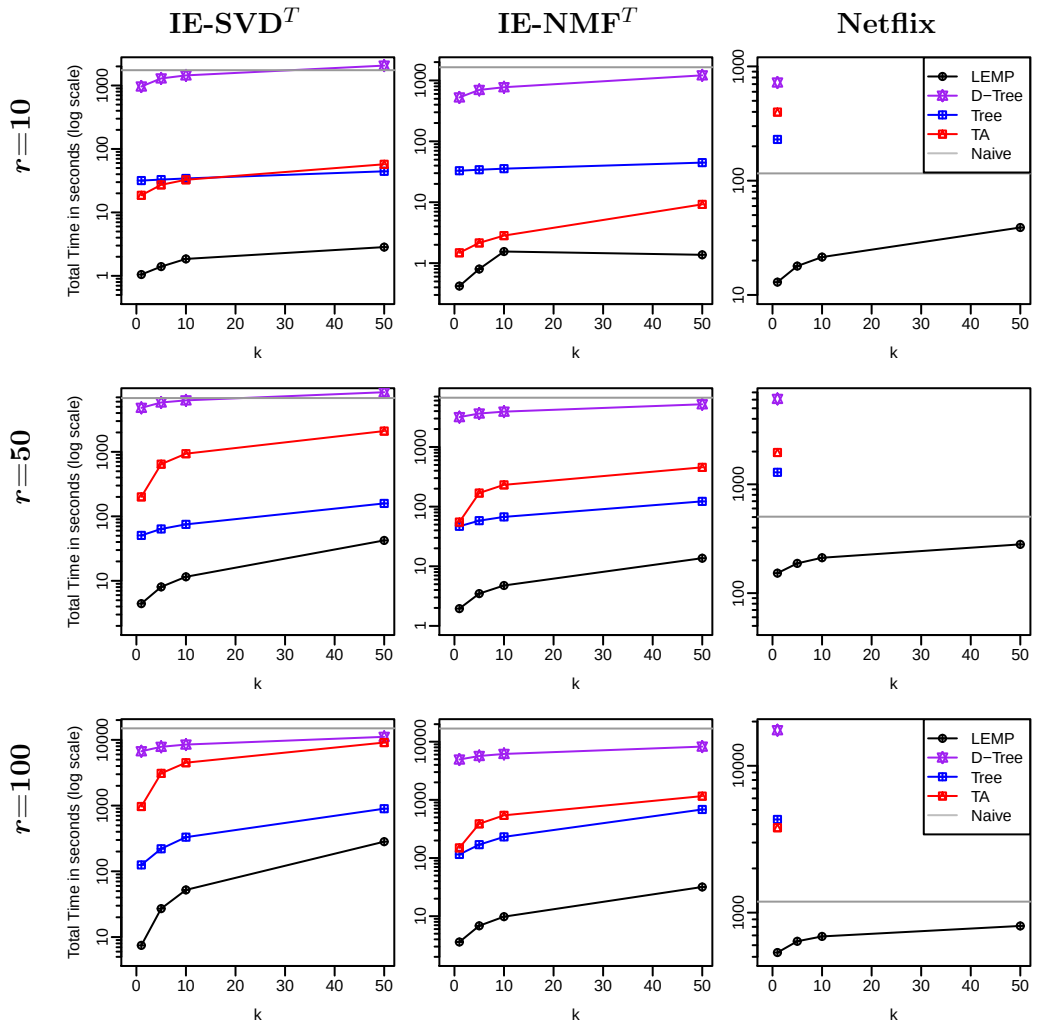


FIGURE 3.7: Total wall-clock times for different ranks for exact Top-k-MIPS

expected, all methods became slower when the rank was increased. Also note that the lower the rank the more competitive Naive becomes. This is because lower rank implies less expensive inner product computations and thus less work for Naive. LEMP was the best performing method for IE-SVD^T and IE-NMF^T regardless of the rank. TA behaved better for low ranks than for larger ranks (recall that TA is designed for vectors of low dimensionality), especially for sparse datasets (IE-NMF^T). Tree ranked almost always in between LEMP and TA, whereas D-Tree was the worst performing method.

For Netflix, all methods perform poorly, mainly because the Netflix dataset is relatively small. However, LEMP was the only method able to offer speedup over Naive.

3.8.4 Approximate MIPS

In this section, we compare various methods for approximate MIPS: PCA-tree, simpleLSH, LEMP-LSHA, LEMP-ABS, and LEMP-REL. The latter two methods use LI

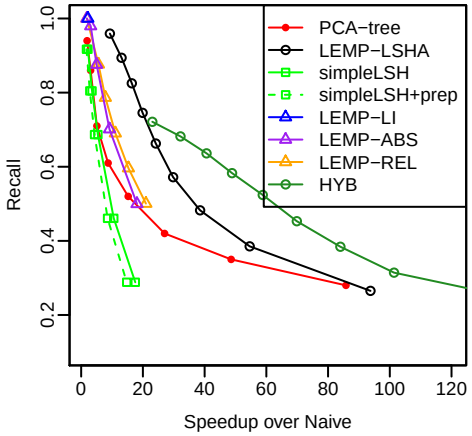
as their exact bucket algorithm (see Section 3.8.5). Figure 3.8 shows results for KDD and Netflix for Top-10-MIPS. We present the performance of simpleLSH both without preprocessing time (simpleLSH) and with preprocessing time (simpleLSH+prep).

Each considered method provides parameters to control the speed-quality tradeoff. Each data point in Figure 3.8 corresponds to one setting of these parameters and indicates the resulting speedup over Naive (x-axis) as well as the value of one of our error measures (y-axis). Note that the values of simpleLSH and simpleLSH+prep are superimposed for Netflix. For PCA-trees, we varied the depth parameter d in a range from 3–10. For simpleLSH, we varied the number $\mathcal{L}$ of signatures and signature length l. To ensure a fair comparison, we present the simpleLSH results for the best combination of l and $\mathcal{L}$ per dataset and recall value. To keep our study manageable, we only considered signature lengths that are multiples of 8 (a byte). For LEMP-LSHA, we varied recall parameter R in a range from 0.9–0.05 for Netflix and from 0.9–0.1 for KDD. For LEMP-ABS, we set ϵ between 0.5–4 for Netflix (rating scale 1–5) and between 10–50 for KDD (rating scale 1–100). Finally, for LEMP-REL, we varied ϵ in the range 0.16–0.5 for Netflix and 0.33–0.47 for KDD.

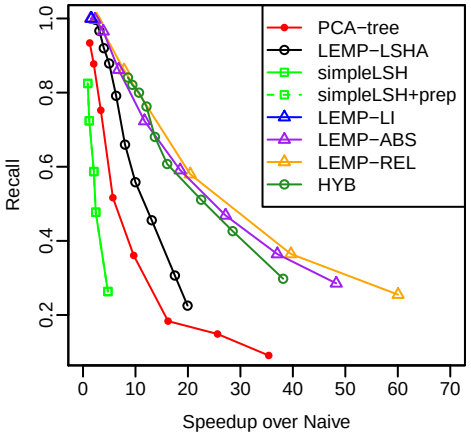
KDD. Figures 3.8a, 3.8c, and 3.8e show the results for KDD using the recall, RMSE and ARE error measures, respectively. First note that LEMP-LSHA offered the best performance-quality tradeoff with respect to all error measures. This indicates that LEMP's bucketization along with our adaptive variant of LSH is very effective. Also note that LEMP-LSHA satisfied the recall bound in all cases (the left-most point corresponds to $R = 0.9$, the right-most point to $R = 0.1$). In fact, in many cases, LEMP-LSHA gave better results than guaranteed by the error bound; e.g., for $R = 0.9$, we obtained recall 0.96. This is because LEMP-LSHA uses the exact LENGTH methods on buckets where it considers LSH too expensive. For such buckets, exact results are obtained, which increases recall. Note that LEMP-LSHA with $R = 0.9$ and a resulting recall of 0.96 was 4x faster than the best exact method (LEMP-LI, also shown in Figure 3.8a). Finally, note that LEMP-LSHA also performed well with respect to RMSE and ARE, although no error guarantees are provided.

LEMP-REL and LEMP-ABS were the next best methods and performed similarly to each other. For both methods, the obtained RMSE or ARE is much lower than guaranteed by error parameter ϵ. For example, we use RMSE bounds $\epsilon \in [10, 50]$ for LEMP-ABS, but obtain RMSE values in $(0, 4]$. The reason for this behavior is that our error analysis of LEMP-ABS and LEMP-REL is based on the worst case, which often does not occur in practice.

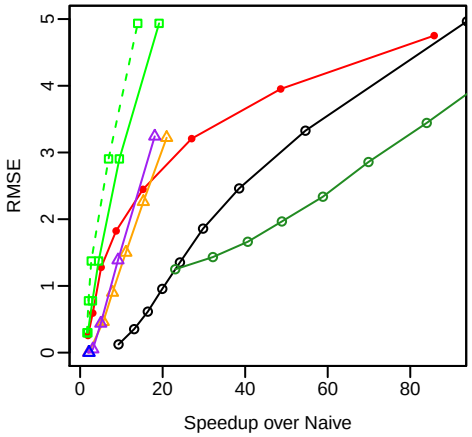
PCA-trees offered little speedup for small depths, which correspond to high recall (the smaller the depth, the more vectors each leaf holds). Since there is no pruning mechanism



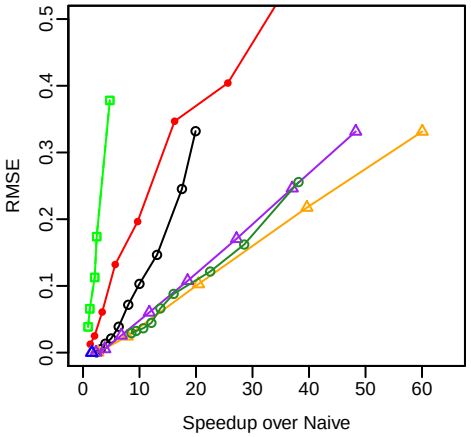
(a) Recall on KDD (higher is better)



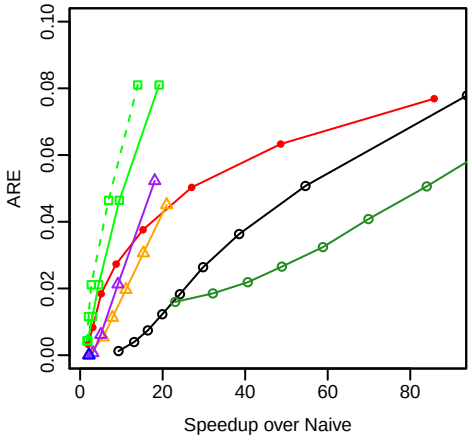
(b) Recall on Netflix (higher is better)



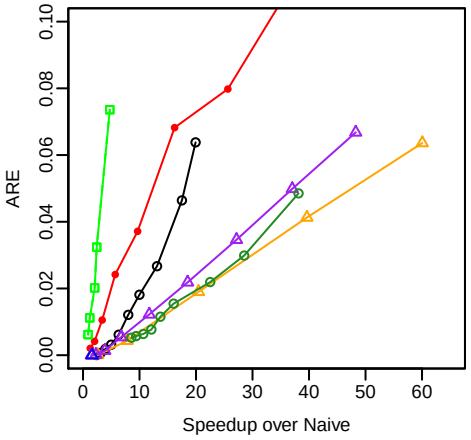
(c) RMSE on KDD (lower is better)



(d) RMSE on Netflix (lower is better)



(e) ARE on KDD (lower is better)



(f) ARE on Netflix (lower is better)

FIGURE 3.8: Speedup over Naive of approximate Top-10-MIPS on KDD and Netflix.

within each leaf, the runtime was large. For larger depths (and lower recall), however, PCA-tree offered significant speedup and reached similar speedups to those of LEMP-LSHA.

SimpleLSH was the worst performing method: it behaved similarly to PCA-tree for high recall levels, but was soon outperformed by all other methods, for lower recall levels. Note that although simpleLSH and LEMP-LSHA are both based on LSH, their performance differs significantly. One reason is that LEMP-LSHA monitors the change in the scores of the top-k list when moving from bucket to bucket. This allows LEMP-LSHA to choose the optimal number $\mathcal{L}$ of signatures per query and bucket, whereas simpleLSH uses a global number of signatures. Moreover, the transformation used by simpleLSH causes the new "length coordinate" to be significantly larger than the other coordinates for all short vectors. In such cases, the bin associated with each vector is often determined by the length coordinate. This implies that shorter probe vectors are clustered in a couple of bins. If the query falls into one of those bins, many probe vectors need to be considered, which is expensive. If the query does not fall into such a bin, then few vectors are considered so that processing is fast but comes with a drop in recall.

Netflix. Figures 3.8b, 3.8d, and 3.8f show our results on Netflix. SimpleLSH was the least-effective method overall. First, as described before, simpleLSH is negatively affected by length skew in the probe vectors. Moreover, Netflix is a smaller dataset with much fewer probe vectors than KDD. Methods that perform expensive per-query operations (such as computing simpleLSH's query hashes) are thus not expected to perform well on Netflix. PCA-tree performs slightly better than simpleLSH in terms of recall and significantly better in terms of RMSE.

The best performing method in this dataset is LEMP-REL, closely followed by LEMP-ABS, whereas LEMP-LSHA ranks third. All LEMP methods perform significantly better than simpleLSH and PCA-tree, e.g., for around 15x speedup over Naive, the recall obtain by LEMP-REL is close to 70%, whereas PCA-tree achieves around 20% recall. As another example, LEMP-REL is 3.9x faster than PCA-tree for RMSE values close to 0.35. Note that all methods achieve smaller RMSE for Netflix than for KDD due to the different scales of these datasets. As on KDD and for the same reasons, the guaranteed error bounds on RMSE (and ARE) were much larger than the errors actually obtained by the algorithms.

Finally, LEMP-REL, as in the KDD dataset, behaved better than LEMP-ABS in terms of ARE. This is no surprise, since LEMP-ABS is oblivious to the magnitude of the true top-k values. It, thus, allows larger relative errors for small top-k values than LEMP-REL does.

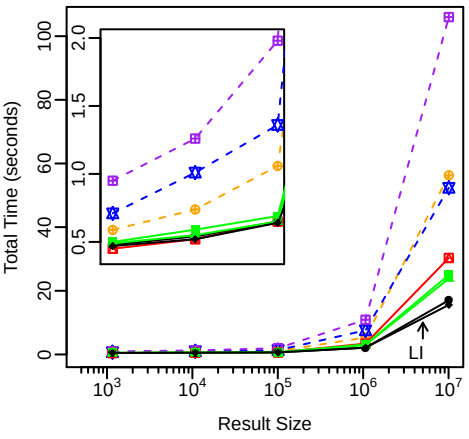
The reason for which the augmented threshold methods performed so well (and better than LEMP-LSHA) on Netflix is the following: in Netflix all the true results are found in roughly the first half of the probe set (largest lengths), however by the time LEMP has retrieved the true results, it has also entered a "plateau" in the length distribution of the probe set that does not allow it to prune buckets easily. This means that LEMP, although it has found the results, cannot terminate early. In such a situation, augmenting the local thresholds will lead to early termination without severe losses in the quality of the results. LEMP-LSHA, on the other hand, might offer fast within-bucket processing, but it still has no way to prune these "excessive" buckets. Therefore the relative performance of the augmented threshold methods and LEMP-LSHA depends on the properties of the dataset. The above observation also suggests that the performance of LEMP-LSHA can possibly be improved if combined with threshold augmentation.

Discussion. To summarize, LEMP-LSHA was the best-performing method on KDD, whereas LEMP-ABS and LEMP-REL were the best-performing methods on Netflix. This indicates that no single algorithm is best in all settings. One option to decide which algorithm to use on a given dataset is to perform algorithm selection in the tuning phase of LEMP (see Section 3.3.4).

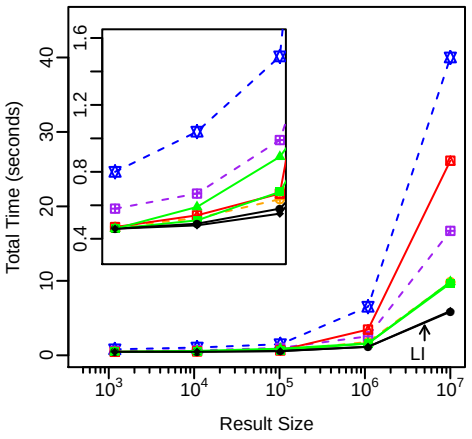
3.8.5 Relative Performance of Bucket Algorithms

In the preceding experiments, we used LI as the bucket algorithm for LEMP because it provided the best overall performance. In this section, we consider and compare various alternative choices. Our results for exact MIPS are summarized in Figure 3.9. Wall-clock times for all experiments and average candidate set sizes can be found in Tables B.3, B.4 and B.5 in Appendix B. In the following, we discuss the performance of each algorithm in turn.

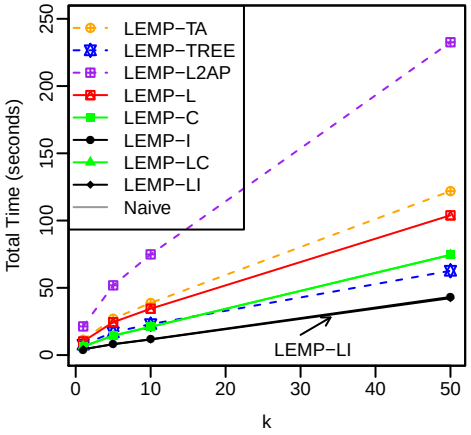
LEMP-L. For the IE datasets, LEMP-L was able to reduce the average candidate set size around 98% (13211 candidates per query vs. 771611 for Naive, IE-SVD^T, $k = 50$), whereas for datasets with less length skew the reduction ranged between 40% and 64% (Netflix) and 14% and 24% (KDD). Overall, LEMP-L was able to provide significant speedup over Naive: up to 17000x (670x) for IE-SVD and 15900x (440x) for IE-NMF for Above-θ-MIPS (Top-k-MIPS). In fact, the simple LEMP-L method outperformed all other methods for the IE datasets and small result sizes. I.e., bucket pruning was very effective for the datasets with large length skew. This indicates that LEMP's separate treatment of short and long vectors is beneficial. The performance of LEMP-L acts as a baseline for the performance of other bucket algorithms: LEMP-L's main filtering mechanism is bucket-level pruning, which is common to all LEMP methods.



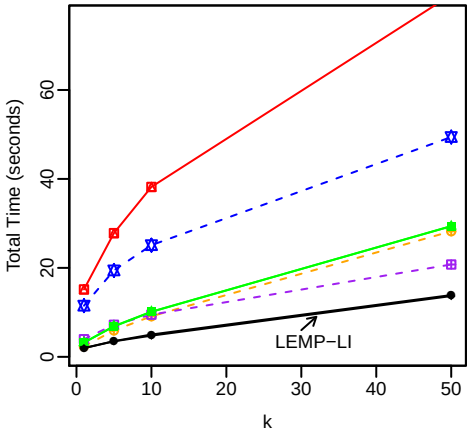
(a) Above-θ-MIPS IE-SVD



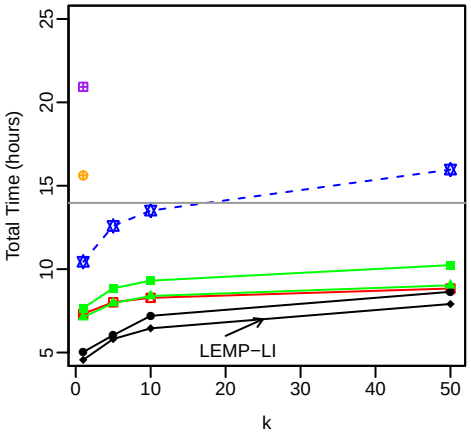
(b) Above-θ-MIPS IE-NMF



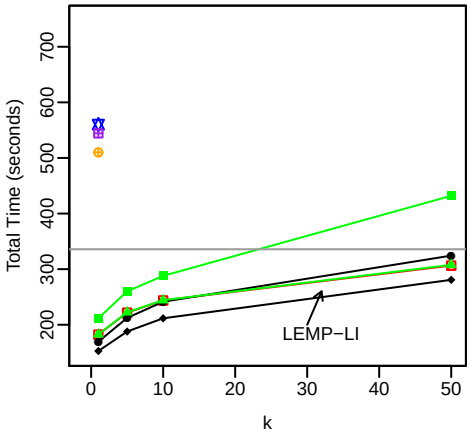
(c) Top-k-MIPS IE-SVD^T



(d) Top-k-MIPS IE-NMF^T



(e) Top-k-MIPS KDD



(f) Top-k-MIPS Netflix

FIGURE 3.9: Comparison of LEMP bucket-algorithms in terms of total wall-clock times (incl. indexing and tuning) for exact MIPS

LEMP-C, LEMP-I. COORD created up to 7x less candidates per query than LEMP-L (e.g., 271 vs. 1915 for IE-NMF^T, $k = 1$) and its speedup over LEMP-L ranged between 2.7x and 4.7x. ICOORD reduced the candidates even further (34 candidates for IE-NMF^T, $k = 1$, 46x less than LEMP-L), with up to 7x speedup. The difference in the pruning power of COORD and ICOORD was more prevalent in the case of the KDD dataset (145K vs. 377K candidates per query). In the absence of large length skew or sparsity, ICOORD accumulates as much information as possible for the probe vectors. COORD, on the other hand, is not able to take full advantage of all the available information. For this reason, ICOORD was the best performing method (when LEMP was used together with only 1 bucket-algorithm) in terms of running time for the majority of datasets and configurations.

LEMP-LI. As discussed above, LEMP-L was the best performing method for datasets with high length skew on small retrieval levels. On the other hand, LEMP-I showed superior behavior in all other cases. LEMP-LI, for a small extra tuning cost, combines the strong points of both methods. In the majority of cases, it was the fastest method overall. In the remaining cases, the performance of LEMP-LI was similar to that of the best-performing method.

LEMP-TA. LEMP-TA was also able to offer speedup over LEMP-L for the sparse datasets: up to 3.5x for the Above-θ-MIPS (@ retrieval level 10M) and up to 6x for Top-1-MIPS. However, it was usually outperformed by COORD and ICOORD: e.g., LEMP-I was up to 3x faster. The reason for ICOORD's superior behavior is that TA is usually not possible to identify good candidates by observing the value of only one coordinate. ICOORD avoids this problem by gathering information about the vectors from multiple coordinates (lists); based on this information, it prunes as many candidates as possible before actually calculating an inner product. Also note that LEMP-TA was significantly faster (up to 17x for IE-SVD^T, $k = 50$) than the standard TA algorithm, since the length-obliviousness and cache-misses problems are addressed by LEMP. This indicates that a method like LEMP might improve the performance of TA when linear scoring functions are used.

LEMP-L2AP. LEMP-L2AP was the method with the most aggressive pruning for all datasets (e.g., only 18 candidates per query for KDD, $k = 1$). However, this extensive pruning has a high cost: L2AP scans all the lists in the index that correspond to non-zero query coordinates and checks the filtering conditions during and after scanning. Also, the actual threshold used when querying the index can be far away from the lower bound used during index creation, which affects scanning time. For these reasons, ICOORD consistently outperformed L2AP (1.3x to 6.2x faster). Actually, L2AP was slower than Naive for both Netflix and KDD.

	$k = 1$	$k = 5$	$k = 10$	$k = 50$
No SIMD	273.3	348.8	386.9	474.5
SIMD-verify	239.6	294.5	318.6	383.7
SIMD-verify&scan	230.9	283.5	309.4	364.2

TABLE 3.3: Performance (in terms of total wall-clock times) of LEMP-LI with and without using SIMD instructions on the KDD dataset for exact Top-k-MIPS. Time in minutes.

LEMP-Tree. LEMP-Tree creates one tree per bucket (lazy construction), instead of one tree for the entire probe dataset. This explains why LEMP-Tree had much better performance than Tree (up to 10x faster) for the datasets for which preprocessing was Tree's bottleneck (see Above-θ-MIPS experiments, small result sizes). In terms of pruning power, LEMP-Tree did not have a consistent behavior w.r.t. Tree. For datasets with large length skew (IE-NMF^T, IE-SVD^T), LEMP-Tree checked less candidates per query, whereas for datasets for small skew (e.g., KDD) it checked more. However, even in these cases, LEMP-Tree was faster than Tree, due to the better cache utilization provided by the bucketization.

3.8.6 Parallel LEMP

In our final set of experiments, we investigated the performance and scalability of our parallel versions of LEMP.

Instruction-level parallelism. Table 3.3 shows the performance of LEMP-LI on our largest dataset (KDD) without SIMD, with SIMD in verification, and with SIMD in verification and scanning. We observed that for large values of k, for which less opportunities for pruning exist (low value of $\hat{\theta}$) and more candidates need to be verified, the speedup due to SIMD in verification reaches 19%. For small values of k, the speedup is lower (12% for $k = 1$). Using instruction-level-parallelism for scanning in addition, further improved the runtime by 6%.

Multi-threading. Figure 3.10 shows the performance of LEMP-LI in terms of wall-clock time for a number of processors varying between 1 and 32. For each setting, we also give the speedup compared to sequential processing. The figure shows results for Top-1-MIPS on the KDD dataset on an Intel(R) Xeon(R) CPU E7-4870 @ 2.40GHz with 40 cores and 512GB RAM. Note that even for embarrassingly parallel problems such as multi-query MIPS, linear speedups are rarely achieved when a large number of processors is used. This is mainly because memory bandwidth and synchronization quickly become a bottleneck. Figure 3.10 shows that LEMP was able to achieve near linear speedups even for large numbers of processors. This indicates that LEMP's careful cache utilization and avoidance of synchronization is effective.

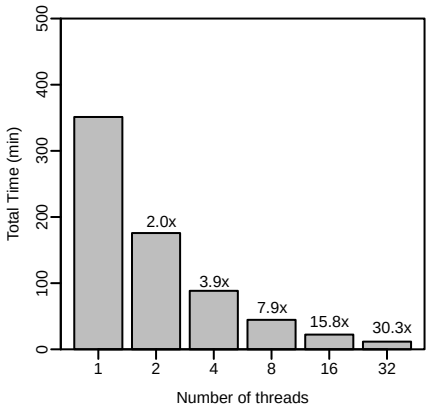


FIGURE 3.10: Scalability of LEMP-LI for KDD, exact Top-1-MIPS

3.9 Summary

The problem of maximum inner product search (MIPS) arises in a number of data mining and information retrieval tasks, such as finding good recommendations in recommender systems, reasoning about extracted facts in open relation extraction, multi-class or multi-label prediction with hundreds of thousands labels or classes, and object detection with deformable part models. Given its broad applications and the fact that naive search is infeasible in practice, many approaches have been recently developed both for exact and approximate MIPS.

In this chapter, we introduced LEMP, a cache-friendly framework for exact and approximate MIPS. Our approach reduces the original MIPS problem into a set of smaller cosine similarity search problems that can be solved using existing techniques or our novel COORD and ICOORD algorithms. We also proposed three methods for approximate MIPS within the LEMP framework and derived quality guarantees. Finally, we compared our approach with previously proposed methods in an extensive set of experiments. Our results showed that LEMP is up to multiple orders of magnitude faster than naive retrieval and that, in combination with our ICOORD method, it is consistently faster than alternative MIPS methods across all our datasets. In addition, our approximate methods offer better quality-speedup tradeoffs (up to 3.9x faster for similar recall levels) than state-of-the-art approximate MIPS techniques. In terms of scalability, LEMP was able to achieve near linear speedups on up to 32 processors.

Chapter 4

Conclusion and Future Work

In this thesis, we presented efficient and scalable algorithms for matrix completion and for maximum inner product search. We believe that these techniques could be successfully applied in a variety of real-life data mining tasks and applications. We also hope that our results will contribute to future research in the fields of scalable matrix completion and maximum inner product search.

Matrix Completion

We studied the problem of large-scale matrix completion, i.e., given a large input matrix with millions of rows and columns and only partially observed and potentially noisy entries to recover the values of the unobserved entries. In the context of recommender systems, in which matrix completion techniques have been very successful, the rows (columns) of the matrix correspond to users (items) and the entries to ratings expressing the user preferences for the items. The goal is to predict the values of the missing entries, i.e., predict the preferences of users for items they have not seen before. We proposed CSGD, a shared-memory algorithm, which allows SGD-based matrix completion to scale on a large number of cores. To achieve that, our method avoids fine-grained locking or synchronization and follows a low cache-miss rate memory access pattern . Experiments on medium and large datasets have shown that our approach is up to 60% faster than state-of-the-art alternative methods.

Shared-memory matrix completion is an exciting topic of research and there exists already a plethora of promising results on this field. In our opinion, it would be very interesting to investigate the behavior of existing techniques when combined with each other, in order to produce hybrid approaches that combine the strong points of many

methods. For example, it might worth exploring enabling low-precision arithmetic (like in Buckwild! [De Sa, Christopher M and Zhang, Ce and Olukotun, Kunle and Ré, Christopher and Ré, Christopher, 2015]) to a cache-aware SGD-based method (like CSGD or FPSGD [Zhuang et al., 2013]). In addition, we have seen in our experiments that there are cases in which SGD-based methods move very fast towards the vicinity of the solution, but then they converge very slowly. I.e., one could use SGD to move fast close to the solution and then switch to a different method that can take more precise steps towards the solution (e.g., L-BFGS or ALS).

We also believe that designing efficient shared-memory matrix completion algorithms can also improve distributed processing. For example, algorithms, like DSGD++ [Teflioudi et al., 2012] and NOMAD [Yun et al., 2014] have been shown to be particularly effective in a distributed setting since they overlap computation and communication between the different machines. It would be interesting to explore if such algorithms can further benefit from the cache-local processing of a CSGD-like method when running SGD locally.

Finally, while matrix completion has been found to be successful in analyzing dyadic data, in many use cases the data are not dyadic. In such cases, more general models, like tensors [Karatzoglou et al., 2010] or factorization machines [Rendle, 2012] are usually used. In addition, models that incorporate side information [Natarajan and Dhillon, 2014, Rossi and Zhou, 2015, Xu et al., 2013] or implicit feedback [Koren, 2008] in the factorization introduce more dependencies between the parameters. The parallelization of SGD for such models is a challenging direction for future work.

Maximum Inner Product Search

We investigated the problem of Maximum Inner Product Search (MIPS), i.e. the problem of finding pairs of vectors with large inner product, given two sets of vectors. In the context of recommender systems, the two sets of vectors correspond to users and items. Such vector representations can be derived through an earlier matrix factorization step. MIPS is, then, used to retrieve the best recommendations for each user.

To solve this problem we proposed LEMP, a cache-friendly framework that reduces the MIPS problem to a set of smaller cosine similarity search problems. To do so, our method is using the geometrical interpretation of vectors and their decomposition into length and direction. The main advantage of LEMP is that, since it solves a set of subproblems (instead of a single large problem), it can use different methods for each of them. In particular, it is able to choose the most appropriate method depending on the properties of the subproblem it tries to solve. Approximate search is also possible within

the LEMP framework. In this thesis, we proposed three techniques for approximate MIPS that offer guarantees on the quality of the result. LEMP was found to be consistently faster than alternative exact MIPS approaches and to offer a better speed-quality tradeoff than state-of-the-art approximate MIPS methods in an extensive empirical study on real datasets.

Exact and approximate MIPS is an emerging topic of research that recently received lots of interest in the scientific community. In our work so far, we incorporated in LEMP previously proposed approaches for MIPS, like the cover trees method of Curtin et al. [2013] or TA [Fagin et al., 2001], and observed that the performance of the methods improved. In general, we would like to see LEMP as an "umbrella" framework, within which also future methods will be used.

In addition, we would like to explore methods for vectors of very high dimensionality (so far we focused on methods for medium dimensionality vectors). MIPS can be used, apart from the matrix factorization scenario, for multi-class or multi-label prediction with hundreds of thousands of labels or classes [Dean et al., 2013], and object detection with deformable part models (e.g., Dean et al. [2013], Felzenszwalb et al. [2010]). Such use cases involve inner products between vectors with thousands of coordinates.

Another possible direction for future work is to explore distributed versions of LEMP. In our work, we showed how to use LEMP in a multi-threaded environment. However, if the probe matrix cannot fit into the memory of a single machine, distributed processing is necessary. Here the interesting case is the Top-k-MIPS problem, in which the nodes might need to broadcast the top-k lists through the network.

Finally, we need to investigate algorithms for more complex models. So far, we considered finding strong interactions within a matrix factorization model (e.g., good recommendations in a recommender system), that captures relationships between dyadic data. More complicated models, like tensor decomposition methods [Karatzoglou et al., 2010] and factorization machines [Rendle, 2012] can model interactions between more than two entities. It would be interesting to investigate if some of the LEMP approaches could be transferred to identify strong interactions between sets of entities when such models are used.

Appendix A

Derivation of Feasible Region Bounds

In this section, we solve Eq. (3.7):

$$\theta_b(\boldsymbol{q}) \leq \bar{q}_f \bar{p}_f + \sqrt{(1-\bar{q}_f^2)(1-\bar{p}_f^2)}$$

for $\bar{p}_f$ and show how we obtained the bounds in Eq. (3.11) and Eq. (3.12).

First, notice that, since $\bar{p}_f$ is a normalized coordinate, it is bounded within the $[-1, 1]$ region, i.e, $-1 \leq \bar{p}_f \leq 1$. According to Eq. (3.7), a probe vector $\bar{\boldsymbol{p}}$ can only qualify if:

$$\begin{aligned} \theta_b(\boldsymbol{q}) &\leq \bar{q}_f \bar{p}_f + \sqrt{(1-\bar{q}_f^2)(1-\bar{p}_f^2)} \\ \theta_b(\boldsymbol{q}) - \bar{q}_f \bar{p}_f &\leq \sqrt{(1-\bar{q}_f^2)(1-\bar{p}_f^2)}. \end{aligned} \tag{A.1}$$

To solve inequality Eq. (A.1), we will distinguish the following cases:

- A) $\theta_b(\boldsymbol{q}) - \bar{q}_f \bar{p}_f \leq 0$ and $\bar{q}_f \neq 0$. This implies that $\theta_b(\boldsymbol{q}) \leq \bar{q}_f \bar{p}_f$, which also triggers the following cases:
  - A1) if $\bar{q}_f > 0$, then all $\bar{\boldsymbol{p}}$ with $\bar{p}_f \geq \theta_b(\boldsymbol{q})/\bar{q}_f$ are in the feasible region, i.e., $[L_f^A, U_f^A] = [\theta_b(\boldsymbol{q})/\bar{q}_f, 1]$.
  - A2) if $\bar{q}_f < 0$, then all $\bar{\boldsymbol{p}}$ with $\bar{p}_f \leq \theta_b(\boldsymbol{q})/\bar{q}_f$ are in the feasible region, i.e., $[L_f^A, U_f^A] = [-1, \theta_b(\boldsymbol{q})/\bar{q}_f]$.

The above analysis can be summarized as:

$$\bar{p}_f \in [L_f^A, U_f^A] = \begin{cases} [\theta_b(\boldsymbol{q})/\bar{q}_f, 1] & \bar{q}_f > 0 \\ [-1, \theta_b(\boldsymbol{q})/\bar{q}_f] & \bar{q}_f < 0 \end{cases} \tag{A.2}$$

Here Eq. (A.2) is only a valid solution to Eq. (3.7) when $L_f^A \leq U_f^A$, i.e., we can ignore it whenever $L_f^A > U_f^A$.

- B) $\theta_b(\boldsymbol{q}) - \bar{q}_f\bar{p}_f > 0$ and $\bar{q}_f \neq 0$. In this case, we square both left and right hand side of Eq. (A.1). Then,

$$(\theta_b(\boldsymbol{q}) - \bar{q}_f\bar{p}_f)^2 < (1-\bar{q}_f^2)(1-\bar{p}_f^2)$$
$$\bar{p}_f^2 - 2\theta_b(\boldsymbol{q})\bar{q}_f\bar{p}_f + \theta_b(\boldsymbol{q})^2 + \bar{q}_f^2 - 1 < 0,$$

which has roots: $\bar{p}_{f_{1,2}} = \bar{q}_f \cdot \theta_b(\boldsymbol{q}) \pm \sqrt{(1-\bar{q}_f^2)(1-\theta_b(\boldsymbol{q})^2)}$. The bounds in this case become:

$$(L_f^B, U_f^B) = (\bar{q}_f \cdot \theta_b(\boldsymbol{q}) - \sqrt{(1-\bar{q}_f^2)(1-\theta_b(\boldsymbol{q})^2)}, \bar{q}_f \cdot \theta_b(\boldsymbol{q}) + \sqrt{(1-\bar{q}_f^2)(1-\theta_b(\boldsymbol{q})^2)}). \tag{A.3}$$

- C) $\bar{q}_f = 0$. In this case, Eq. (A.1) can be rewritten as:

$$\theta_b(\boldsymbol{q}) \leq \sqrt{(1-\bar{p}_f^2)}.$$

Here again we will separate two cases:

  - C1) $\theta_b(\boldsymbol{q}) > 0$. In this case, we have two roots which happen to be the same as in case B.
  - C2) $\theta_b(\boldsymbol{q}) \leq 0$. In this case, Eq. (A.1) is satisfied for any value of $\bar{p}_f$, i.e., $[L_f^C, U_f^C] = [-1, 1]$. Note that in this case we get no extra pruning information. Thus, our algorithm will simply skip query coordinates of zero value, whenever $\theta_b(\boldsymbol{q}) \leq 0$.

We now need to combine the regions derived from cases A, B and C. If $\bar{q}_f = 0$, only case C will give an interval for $\bar{p}_f$. Let us examine the case in which $\bar{q}_f > 0$ (the analysis is similar also for $\bar{q}_f < 0$). According to case A we have one feasible region $[L_f^A, U_f^A] = [\theta_b(\boldsymbol{q})/\bar{q}_f, 1]$. Obviously, if $L_f^A > U_f^A$ this region is empty and we only need to consider the feasible region according to case B: $(L_f^B, U_f^B) = \Big(\bar{q}_f \cdot \theta_b(\boldsymbol{q}) - \sqrt{(1-\bar{q}_f^2)(1-\theta_b(\boldsymbol{q})^2)}, \bar{q}_f \cdot \theta_b(\boldsymbol{q}) +$

$\sqrt{(1-\bar{q}_f^2)(1-\theta_b(\boldsymbol{q})^2)}\Big)$. If feasible region A is non-empty, we need to take the union of both region A and B. We will later show that those two regions can only be overlapping, i.e., their union is a single interval. In this case, the lower bound will be the lower of $\{L_f^A, L_f^B\}$ and the upper bound will be the larger of $\{U_f^A, U_f^B\}$. This brings us to the formulas of Eq. (3.11) and Eq. (3.12).

Now, we will prove by contradiction that region A and region B cannot be disjoint. Let us assume that they are disjoint. Then, it holds that:

$$\begin{aligned} U_f^B &< L_f^A \\ \bar{q}_f \cdot \theta_b(\boldsymbol{q}) + \sqrt{(1-\bar{q}_f^2)(1-\theta_b(\boldsymbol{q})^2} &< \frac{\theta_b(\boldsymbol{q})}{\bar{q}_f} \\ 0 \leq \sqrt{(1-\bar{q}_f^2)(1-\theta_b(\boldsymbol{q})^2)} &< \theta_b(\boldsymbol{q})\frac{(1-\bar{q}_f^2)}{\bar{q}_f}. \end{aligned} \tag{A.4}$$

Notice that for the right hand side of Eq. (A.4) to be larger than 0, we require also that:

$$\theta_b(\boldsymbol{q}) \geq 0. \tag{A.5}$$

Since both sides of Eq. (A.4) are positive, we can square them:

$$\begin{aligned} 0 \leq (1-\bar{q}_f^2)(1-\theta_b(\boldsymbol{q})^2) &< \theta_b^2(\boldsymbol{q})\frac{(1-\bar{q}_f^2)^2}{\bar{q}_f^2} \\ \frac{1-\theta_b(\boldsymbol{q})^2}{\theta_b(\boldsymbol{q})^2} &< \frac{1-\bar{q}_f^2}{\bar{q}_f^2}. \end{aligned} \tag{A.6}$$

Let us now examine if the Eq. (A.6) is possible or not. We will now try to construct the same inequality starting from the fact that feasible region A is non empty, i.e.,

$$-1 \leq \frac{\theta_b(\boldsymbol{q})}{\bar{q}_f} \leq 1. \tag{A.7}$$

For the derivation, keep in mind that $\bar{q}_f > 0$ (initial assumption) and that $\theta_b(\boldsymbol{q}) \geq 0$ (Eq. (A.5)). Then we can transform Eq. (A.7) into:

$$\begin{aligned} \theta_b(\boldsymbol{q}) &\leq \bar{q}_f \\ \theta_b^2(\boldsymbol{q}) &\leq \bar{q}_f^2 \\ -\theta_b^2(\boldsymbol{q}) &\geq -\bar{q}_f^2 \\ 1-\theta_b^2(\boldsymbol{q}) &\geq 1-\bar{q}_f^2. \end{aligned} \tag{A.8}$$

By dividing side by side with Eq. (A.8), we get:

$$\frac{1-\theta_b(\boldsymbol{q})^2}{\theta_b(\boldsymbol{q})^2} \geq \frac{1-\bar{q}_f^2}{\bar{q}_f^2},$$

which means that Eq. (A.6) is not possible (contradiction).

Appendix B

Additional Experimental Results for MIPS

Tables B.1 - B.5 show running times for exact Above-θ-MIPS and Top-k-MIPS experiments for different retrieval levels and values of k (including those presented in the figures of Sec. 3.8).

Dataset	Algorithm	@1K		@10K		@100K		@1M		@10M		Preprocessing
		Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	time [%]
IE-SVD	Naive	6825	*(132K)*	-	*(132K)*	-	*(132K)*	-	*(132K)*	-	*(132K)*	-
	Tree	7.74	*(2.1)*	7.96	*(3.1)*	8.36	*(4.9)*	12.67	*(30.8)*	50.74	*(236.8)*	15% – 95%
	D-Tree	46.33	***(0.2)***	46.36	***(0.4)***	46.47	***(0.7)***	47.30	***(5.9)***	57.94	***(66.2)***	80% – 99%
	TA	2.31	*(3.1)*	2.71	*(6.9)*	3.30	*(12.6)*	10.92	*(81.54)*	98.84	*(818.8)*	<1% – 32%
	LEMP-LI	**0.41**	*(1.03)*	**0.47**	*(0.44)*	**0.56**	*(1.06)*	**2.07**	*(17.47)*	**16.20**	*(217.49)*	8% – 90%
IE-NMF	Naive	6703.2	*(132K)*	-	*(132K)*	-	*(132K)*	-	*(132K)*	-	*(132K)*	-
	Tree	5.89	*(2.3)*	6.10	*(3.3)*	6.50	*(5.2)*	10.24	*(29.4)*	39.41	*(185.4)*	14% – 93%
	D-Tree	38.60	*(0.2)*	38.62	***(0.3)***	38.72	***(0.7)***	39.46	***(5.1)***	47.06	***(46.6)***	81% – 99%
	TA	1.91	*(0.9)*	1.93	*(1.2)*	2.02	*(2.1)*	2.91	*(11.4)*	17.50	*(139.8)*	2.5% –23%
	LEMP-LI	**0.39**	***(0.1)***	**0.44**	*(1.14)*	**0.58**	*(2.4)*	**1.15**	*(7.24)*	**6.11**	*(51.03)*	14% – 88%

TABLE B.1: Comparison of LEMP with state-of-the-art algorithms for the exact Above-θ-MIPS problem w.r.t. wall-clock time (in seconds). We give the average candidate set size per query in parentheses. The last column gives the preprocessing time as minimum/maximum percentage of the overall time (occurs at large/small result sizes, resp.).

Dataset	Algorithm	k=1		k=5		k=10		k=50		Preprocessing
		Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	time [%]
IE-SVD^T	Naive	6825	*(771K)*	-	*(771K)*	-	*(771K)*	-	*(771K)*	-
	Tree	50.6	*(357)*	63.7	*(772)*	75.1	*(1119)*	158.5	*(3213)*	25% – 77%
	D-Tree	4766.8	*(24K)*	5826.8	*(29K)*	6285.1	*(30K)*	8408.5	*(34K)*	<1%
	TA	200.60	*(6090)*	644.71	*(19482)*	936.62	*(28036)*	2100.08	*(62037)*	<1% –2.4%
	LEMP-LI	**4.43**	***(54)***	**8.05**	***(285)***	**11.55**	***(512)***	**42.32**	***(1772)***	3.7% – 15.4%
IE-NMF^T	Naive	6703.2	*(771K)*	-	*(771K)*	-	*(771K)*	-	*(771K)*	-
	Tree	46.8	*(465)*	58.2	*(845)*	67.3	*(1107)*	122.5	*(2591)*	27% – 70%
	D-Tree	3169.8	*(16K)*	3642.6	*(18K)*	3915.2	*(19K)*	5203.1	*(21K)*	<1% –1.2%
	TA	55.15	*(1899)*	169.04	*(5514)*	232.10	*(7552)*	455.99	*(14721)*	<1% –5.4%
	LEMP-LI	**1.95**	***(33)***	**3.48**	***(100)***	**4.76**	***(136)***	**13.66**	***(417)***	7% – 23%
Netflix	Naive	335.8	*(17.7K)*	-	*(17.7K)*	-	*(17.7K)*	-	*(17.7K)*	-
	Tree	1289.1	*(9279)*	>Naive	-	>Naive	-	>Naive	-	<1%
	D-Tree	6095.6	*(11K)*	>Naive	-	>Naive	-	>Naive	-	<57%
	TA	1961.8	*(16K)*	>Naive	-	>Naive	-	>Naive	-	<1%
	LEMP-LI	**152.72**	***(4108)***	**187.78**	***(4485)***	**211.59**	***(5889)***	**280.74**	***(8151)***	<1%
KDD	Naive	14h	*(624K)*	-	*(624K)*	-	*(624K)*	-	*(624K)*	-
	Tree	15.2h	*(86K)*	>Naive	-	>Naive	-	>Naive	-	<1%
	D-Tree	15.1h	***(72K)***	>Naive	-	>Naive	-	>Naive	-	<6
	TA	44.7h	*(567K)*	>Naive	-	>Naive	-	>Naive	-	<1%
	LEMP-LI	**4.56h**	*(85K)*	**5.81h**	***(149K)***	**6.45h**	***(187K)***	**7.91h**	***(277K)***	<1%

TABLE B.2: Comparison of LEMP with state-of-the-art algorithms for the exact Top-k-MIPS problem w.r.t. wall-clock time (in seconds, unless stated otherwise). We give the average candidate set size per query in parentheses. The last column gives the preprocessing time as minimum/maximum percentage of the overall time (occurs at large/small k, resp.).

Dataset	Algorithm	@1K		@10K		@100K		@1M		@10M	
		Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$
IE-SVD	LEMP-L2AP	1.02	***(0.09)***	1.33	***(0.14)***	2.07	***(0.39)***	11.38	***(1.49)***	107.69	***(14.16)***
	LEMP-C	0.65	*(1.61)*	0.52	*(2.89)*	0.62	*(4.66)*	2.87	*(43.37)*	23.08	*(432.21)*
	LEMP-I	0.43	*(0.18)*	0.47	*(0.42)*	0.57	*(1.06)*	2.14	*(16.18)*	17.18	*(227.36)*
	LEMP-L	**0.40**	*(1.21)*	**0.46**	*(2.64)*	0.61	*(5.47)*	3.61	*(71.36)*	32.86	*(716.15)*
	LEMP-LC	0.41	*(1.09)*	0.48	*(2.15)*	0.59	*(3.8)*	2.78	*(42.73)*	22.45	*(431.96)*
	LEMP-LI	0.41	*(1.03)*	0.47	*(0.44)*	**0.56**	*(1.06)*	**2.07**	*(17.47)*	**16.20**	*(217.49)*
	LEMP-TA	0.55	*(1.84)*	0.80	*(2.98)*	1.01	*(5.35)*	5.27	*(30.74)*	55.60	*(453.64)*
	LEMP-TREE	0.67	*(1.71)*	0.92	*(3.14)*	1.39	*(5.8)*	7.79	*(46.58)*	57.01	*(320.44)*
IE-NMF	LEMP-L2AP	0.77	*(0.1)*	0.84	***(0.17)***	1.03	*(0.44)*	2.53	***(1.51)***	15.96	***(14.11)***
	LEMP-C	0.42	*(0.75)*	0.45	*(1.21)*	0.56	*(2.59)*	1.46	*(15.92)*	9.44	*(147.85)*
	LEMP-I	0.40	***(0.09)***	**0.44**	***(0.17)***	**0.50**	***(0.43)***	**1.10**	*(4.26)*	**5.92**	*(44.38)*
	LEMP-L	0.42	*(1.5)*	0.48	*(2.88)*	0.62	*(5.76)*	3.55	*(70.02)*	34.29	*(614.01)*
	LEMP-LC	0.42	*(0.77)*	0.44	*(1.23)*	0.51	*(2.6)*	1.39	*(15.96)*	8.93	*(148.67)*
	LEMP-LI	**0.39**	*(0.1)*	0.44	*(1.14)*	0.58	*(2.4)*	1.15	*(7.24)*	6.11	*(51.03)*
	LEMP-TA	0.43	*(0.55)*	0.47	*(0.85)*	0.59	*(1.73)*	1.65	*(4.94)*	9.72	*(62.34)*
	LEMP-TREE	0.78	*(2.7)*	1.04	*(4.37)*	1.51	*(7.55)*	6.82	*(43.58)*	43.63	*(251.14)*

TABLE B.3: Comparison of LEMP bucket algorithms for the exact Above-θ-MIPS problem w.r.t. wall-clock time (in seconds). We give the average candidate set size per query in parentheses.

Dataset	Algorithm	k=1		k=5		k=10		k=50	
		Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$
IE-SVD^T	LEMP-L2AP	21.30	***(15)***	51.84	***(24)***	74.86	***(35)***	232.57	***(129)***
	LEMP-C	6.06	*(541)*	14.63	*(1418)*	20.99	*(2182)*	74.41	*(7823)*
	LEMP-I	**4.17**	*(61)*	8.13	*(261)*	11.76	*(565)*	43.18	*(2325.19)*
	LEMP-L	10.16	*(1272)*	24.44	*(3102)*	34.47	*(4386)*	103.90	*(13211)*
	LEMP-LC	6.68	*(685)*	14.23	*(1436)*	20.97	*(2190)*	74.52	*(7803)*
	LEMP-LI	4.43	*(54)*	**8.05**	*(285)*	**11.55**	*(512)*	**42.32**	*(1772)*
	LEMP-TA	11.41	*(578)*	27.04	*(1448)*	38.75	*(2092)*	121.91	*(6521)*
	LEMP-TREE	7.75	*(213)*	16.95	*(536)*	23.00	*(744)*	62.67	*(2034)*
IE-NMF^T	LEMP-L2AP	3.96	***(11)***	7.21	***(24)***	9.48	***(33)***	20.76	***(126)***
	LEMP-C	3.20	*(271)*	6.83	*(659)*	10.11	*(1001)*	29.40	*(3039)*
	LEMP-I	1.97	*(34)*	3.53	*(89)*	5.02	*(146)*	13.89	*(343)*
	LEMP-L	15.16	*(1915)*	27.81	*(3541)*	38.20	*(4869)*	81.39	*(10319)*
	LEMP-LC	3.20	*(274)*	6.90	*(664)*	10.15	*(1026)*	29.36	*(3041)*
	LEMP-LI	**1.95**	*(33)*	**3.48**	*(100)*	**4.76**	*(136)*	**13.66**	*(417)*
	LEMP-TA	2.35	*(89)*	5.93	*(328)*	9.12	*(540)*	28.23	*(1757)*
	LEMP-TREE	11.50	*(345)*	19.43	*(640)*	25.11	*(851)*	49.42	*(1693)*

TABLE B.4: Comparison of LEMP bucket algorithms for the exact Top-k-MIPS problem on the IE datasets w.r.t. wall-clock time (in seconds, unless stated otherwise). We give the average candidate set size per query in parentheses.

Dataset	Algorithm	k=1		k=5		k=10		k=50	
		Time	$\|C\|/q$	Time	$\|C\|/q$	Time	$\|C\|/q$	Time	$\|C\|/q$
KDD	LEMP-L2AP	20.93h	***(18)***	>Naive	-	>Naive	-	>Naive	-
	LEMP-C	7.67h	*(377K)*	8.85h	*(437K)*	9.31h	*(461K)*	10.24h	*(507K)*
	LEMP-I	5.03h	*(145K)*	6.04h	*(175K)*	7.20h	*(281K)*	8.64h	*(366K)*
	LEMP-L	7.32h	*(445K)*	8.02h	*(488K)*	8.28h	*(504K)*	8.84h	*(537K)*
	LEMP-LC	7.12h	*(375K)*	7.96h	*(437K)*	8.39h	*(460K)*	9.03h	*(506K)*
	LEMP-LI	**4.56**	*(85K)*	**5.81h**	*(149K)*	**6.45h**	*(187K)*	**7.91h**	*(277K)*
	LEMP-TA	15.62h	*(412K)*	>Naive	-	>Naive	-	>Naive	-
	LEMP-TREE	10.44h	*(196K)*	12.59h	*(237K)*	13.51h	*(258K)*	15.97h	*(307K)*
Netflix	LEMP-L2AP	544.30	***(32)***	>Naive	-	>Naive	-	>Naive	-
	LEMP-C	211.59	*(6112)*	260.11	*(7465)*	287.93	*(8242)*	431.95	*(10291)*
	LEMP-I	169.57	*(3908)*	212.29	*(4601)*	241.39	***(5709)***	324.31	***(8120)***
	LEMP-L	182.39	*(6486)*	221.81	*(7839)*	243.82	*(8593)*	306.43	*(10587)*
	LEMP-LC	182.70	*(6284)*	222.37	*(7688)*	244.12	*(8483)*	307.74	*(10565)*
	LEMP-LI	**152.72**	*(4108)*	**187.78**	***(4485)***	**211.59**	*(5889)*	**280.74**	*(8151)*
	LEMP-TA	510.04	*(7284)*	>Naive	-	>Naive	-	>Naive	-
	LEMP-TREE	559.97	*(6475)*	>Naive	-	>Naive	-	>Naive	-

TABLE B.5: Comparison of LEMP bucket algorithms for the exact Top-k-MIPS problem on KDD, Netflix w.r.t. wall-clock time (in seconds, unless stated otherwise). We give the average candidate set size per query in parentheses.

List of Figures

List of Tables

Bibliography

Amatriain, X. and Basilico, J. (2012). Netflix Recommendations: Beyond the 5 Stars (Part 1). http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html.

Anastasiu, D. C. and Karypis, G. (2014). L2AP: Fast Cosine Similarity Search With Prefix L-2 Norm Bounds. In *ICDE*.

Bachrach, Y., Finkelstein, Y., Gilad-Bachrach, R., Katzir, L., Koenigstein, N., Nice, N., and Paquet, U. (2014). Speeding Up the Xbox Recommender System Using a Euclidean Transformation for Inner-product Spaces. In *RecSys*.

Ballard, G., Pinar, A., Kolda, T. G., and Seshadri, C. (2015). Diamond Sampling for Approximate Maximum All-pairs Dot-product (MAD) Search. In *ICDM*.

Battiti, R. (1989). Accelerated Backpropagation Learning: Two Optimization Methods. *Complex Systems*.

Bayardo, R. J., Ma, Y., and Srikant, R. (2007). Scaling Up All Pairs Similarity Search. In *WWW*.

Bennett, J. and Lanning, S. (2007). The Netflix Prize. In *Proceedings of the KDD Cup Workshop*.

Beygelzimer, A., Kakade, S., and Langford, J. (2006). Cover Trees for Nearest Neighbor. In *ICML*.

Biswas, P., Liang, T., Wang, T., and Ye, Y. (2006). Semidefinite programming based algorithms for sensor network localization. *ACM Transactions on Sensor Networks*.

Byrd, R., Chin, G., Nocedal, J., and Wu, Y. (2012). Sample size selection in optimization methods for machine learning. *Mathematical Programming*.

Byrd, R. H., Lu, P., Nocedal, J., and Zhu, C. (1995). A Limited Memory Algorithm for Bound Constrained Optimization. *SIAM Journal on Scientific Computing*.

Candes, E. J. and Recht, B. (2009). Exact Matrix Completion via Convex Optimization. *Foundations of Computational Mathematics*.

Charikar, M. S. (2002). Similarity Estimation Techniques from Rounding Algorithms. In *ACM Symposium on Theory of Computing*. ACM.

Chaudhuri, S., Ganti, V., and Kaushik, R. (2006). A Primitive Operator for Similarity Joins in Data Cleaning. In *ICDE*.

Chen, P. and Suter, D. (2005). Recovering the missing components in a large noisy low-rank matrix: application to SFM source. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Chen, P., Tsai, C., Chen, Y., Chou, K., Li, C., Tsai, C., Wu, K., Chou, Y., Li, C., Lin, W., Yu, S., Chiu, R., Lin, C., Wang, C., Wang, P., Su, W., Wu, C., Kuo, T., McKenzie, T., Chang, Y., Ferng, C., Ni, C., Lin, H., Lin, C., and Lin, S. (2012). A Linear Ensemble of Individual and Blended Models for Music Rating Prediction. In *Proceedings of KDD Cup 2011 competition*.

Chin, W.-S., Zhuang, Y., Juan, Y.-C., and Lin, C.-J. (2015). A Learning-Rate Schedule for Stochastic Gradient Methods to Matrix Factorization. In *Advances in Knowledge Discovery and Data Mining*, Lecture Notes in Computer Science. Springer International Publishing.

Chu, C. T., Kim, S. K., Lin, Y. A., Yu, Y. Y., Bradski, G., Ng, A. Y., and Olukotun, K. (2006). Map-Reduce for Machine Learning on Multicore. In *NIPS*.

Cichocki, A. and Phan, A. H. (2009). Fast Local Algorithms for Large Scale Nonnegative Matrix and Tensor Factorizations. *IEICE Transactions on Fundamentals of Electronics*.

Curtin, R. R. and Ram, P. (2014). Dual-tree fast exact max-kernel search. *Statistical Analysis and Data Mining*.

Curtin, R. R., Ram, P., and Gray, A. G. (2013). Fast Exact Max-Kernel Search. In *SDM*.

Das, A. S., Datar, M., Garg, A., and Rajaram, S. (2007). Google News Personalization: Scalable Online Collaborative Filtering. In *WWW*.

Das, S., Sismanis, Y., Beyer, K. S., Gemulla, R., Haas, P. J., and McPherson, J. (2010). Ricardo: Integrating R and Hadoop. In *SIGMOD*.

De Sa, Christopher M and Zhang, Ce and Olukotun, Kunle and Ré, Christopher and Ré, Christopher (2015). Taming the wild: A unified analysis of hogwild-style algorithms. In *NIPS*.

Dean, T., Ruzon, M., Segal, M., Shlens, J., Vijayanarasimhan, S., and Yagnik, J. (2013). Fast, Accurate Detection of 100,000 Object Classes on a Single Machine. In *CVPR*.

Dror, G., Koenigstein, N., Koren, Y., and Weimer, M. (2012). The Yahoo! Music Dataset and KDD-Cup '11. *Journal of Machine Learning Research: Proceedings Track*.

F. Petroni, L. D. C. and Gemulla, R. (2015). CORE: Context-Aware Open Relation Extraction with Factorization Machines. In *EMNLP*.

Fagin, R., Lotem, A., and Naor, M. (2001). Optimal Aggregation Algorithms for Middleware. In *PODS*.

Felzenszwalb, P., Girshick, R., McAllester, D., and Ramanan, D. (2010). Object Detection with Discriminatively Trained Part-Based Models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Gemulla, R., Haas, P. J., Nijkamp, E., and Sismanis, Y. (2011a). Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent. Technical Report RJ10481, IBM Almaden Research Center.

Gemulla, R., Haas, P. J., Sismanis, Y., Teflioudi, C., and Makari, F. (2011b). Large-scale Matrix Factorization with Distributed Stochastic Gradient Descent. In *NIPS Workshop on Parallel and Large-Scale Machine Learning (Big Learning)*.

Gemulla, R., Nijkamp, E., Haas, P. J., and Sismanis, Y. (2011c). Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*.

Gionis, A., Indyk, P., and Motwani, R. (1999). Similarity Search in High Dimensions via Hashing. In *VLDB*.

Hu, Y., Koren, Y., and Volinsky, C. (2008). Collaborative Filtering for Implicit Feedback Datasets. In *ICDM*.

Karatzoglou, A., Amatriain, X., Baltrunas, L., and Oliver, N. (2010). Multiverse Recommendation: N-dimensional Tensor Factorization for Context-aware Collaborative Filtering. In *RecSys*.

Koenigstein, N., Dror, G., and Koren, Y. (2011). Yahoo! Music Recommendations: Modeling Music Ratings with Temporal Dynamics and Item Taxonomy. In *RecSys*.

Koenigstein, N., Ram, P., and Shavitt, Y. (2012). Efficient retrieval of recommendations in a matrix factorization framework. In *CIKM*.

Koren, Y. (2008). Factorization Meets the Neighborhood: A Multifaceted Collaborative Filtering Model. In *KDD*.

Koren, Y., Bell, R., and Volinsky, C. (2009). Matrix Factorization Techniques for Recommender Systems. *IEEE Computer*.

Kushner, H. and Yin, G. (2003). *Stochastic Approximation and Recursive Algorithms and Applications*. Springer.

Lee, D., Park, J., Shim, J., and Lee, S.-g. (2010). An Efficient Similarity Join Algorithm with Cosine Similarity Predicate. In *DEXA: Part II*.

Lee, J., Kim, H., and Vuduc, R. (2012). When Prefetching Works, When It Doesn&Rsquo;T, and Why. *ACM Transactions on Architecture and Code Optimization*.

Liu, C., Yang, H.-c., Fan, J., He, L.-W., and Wang, Y.-M. (2010). Distributed Nonnegative Matrix Factorization for Web-Scale Dyadic Data Analysis on Mapreduce. In *WWW*.

Mackey, L., Talwalkar, A., and Jordan, M. (2011). Divide-and-Conquer Matrix Factorization. In *NIPS*.

Makari, F., Teflioudi, C., Gemulla, R., Haas, P., and Sismanis, Y. (2015). Shared-memory and shared-nothing stochastic gradient descent algorithms for matrix completion. *Knowledge and Information Systems*.

Menon, A. and Elkan, C. (2011). Link Prediction via Matrix Factorization. In *Machine Learning and Knowledge Discovery in Databases*. Springer Berlin Heidelberg.

Nakashole, N., Weikum, G., and Suchanek, F. (2012). PATTY: A Taxonomy of Relational Patterns with Semantic Types. In *EMNLP-CoNLL*.

Natarajan, N. and Dhillon, I. S. (2014). Inductive matrix completion for predicting gene-disease associations. *Bioinformatics*.

Neyshabur, B. and Srebro, N. (2015). On Symmetric and Asymmetric LSHs for Inner Product Search. In *ICML*.

Nickel, M. (2016). A Review of Relational Machine Learning for Knowledge Graphs. *Proceedings of IEEE (to appear)*.

Niu, F., Recht, B., Ré, C., and Wright, S. J. (2011). Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS*.

Oh, J., Han, W.-S., Yu, H., and Jiang, X. (2015). Fast and Robust Parallel SGD Matrix Factorization. In *KDD*.

Ram, P. and Gray, A. G. (2012). Maximum inner-product search using cone trees. In *KDD*.

Recht, B. and Ré, C. (2013). Parallel Stochastic Gradient Algorithms for Large-Scale Matrix Completion. *Mathematical Programming Computation*.

Rendle, S. (2012). Factorization Machines with libFM. *ACM Transactions on Intelligent Systems and Technology*.

Riedel, S., Yao, L., Marlin, B. M., and McCallum, A. (2013). Relation Extraction with Matrix Factorization and Universal Schemas. In *HLT-NAACL*.

Rossi, R. and Zhou, R. (2015). Scalable relational learning for large heterogeneous networks. In *DSAA*.

Satuluri, V. and Parthasarathy, S. (2012). Bayesian Locality Sensitive Hashing for Fast Similarity Search. *Proceedings of the VLDB Endowment*.

Schmidt, R. (1986). Multiple emitter location and signal parameter estimation. *IEEE Transactions on Antennas and Propagation*.

Shrivastava, A. and Li, P. (2014a). An Improved Scheme for Asymmetric LSH. *CoRR*, abs/1410.5410.

Shrivastava, A. and Li, P. (2014b). Asymmetric LSH (ALSH) for Sublinear Time Maximum Inner Product Search (MIPS). In *NIPS*.

Singer, A. (2008). A remark on global positioning from local distances. *Proceedings of the National Academy of Sciences*.

Skillicorn, D. (2007). *Understanding complex datasets: data mining with matrix decompositions*. Taylor & Francis Ltd.

Teflioudi, C. and Gemulla, R. (2016). Exact and Approximate Maximum Inner Product Search with LEMP (under review). *ACM Transactions on Database Systems (TODS)*.

Teflioudi, C., Gemulla, R., and Mykytiuk, O. (2015). LEMP: Fast Retrieval of Large Entries in a Matrix Product. In *SIGMOD*.

Teflioudi, C., Makari, F., and Gemulla, R. (2012). Distributed Matrix Completion. In *ICDM*.

Whang, K. Y., Kim, S. W., and Wiederhold, G. (1994). Dynamic Maintenance of Data Distribution for Selectivity Estimation. *The VLDB Journal*.

Xiao, C., Wang, W., Lin, X., and Yu, J. X. (2008). Efficient Similarity Joins for Near Duplicate Detection. In *WWW*.

Xiao, C., Wang, W., Lin, X., Yu, J. X., and Wang, G. (2011). Efficient Similarity Joins for Near-duplicate Detection. *ACM Transactions on Database Systems*.

Xu, M., Jin, R., and Zhou, Z. (2013). Speedup Matrix Completion with Side Information: Application to Multi-Label Learning. In *NIPS*.

Yu, H.-F., Hsieh, C.-J., Si, S., and Dhillon, I. S. (2012). Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems. In *ICDM*.

Yun, H., Yu, H.-F., Hsieh, C.-J., Vishwanathan, S., and Dhillon, I. S. (2014). NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion. In *VLDB*.

Zhang, Z., Wang, Q., Ruan, L., and Si, L. (2014). Preference Preserving Hashing for Efficient Recommendation. In *SIGIR*.

Zhou, Y., Wilkinson, D., Schreiber, R., and Pan, R. (2008). Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *AAIM*.

Zhuang, Y., Chin, W.-S., Juan, Y.-C., and Lin, C.-J. (2013). A Fast Parallel SGD for Matrix Factorization in Shared Memory Systems. In *RecSys*.