Knowledge and Information Systems

Ternary Matrix Factorization: Problem Definitions and Algorithms

Samuel Maurus, Claudia Plant

Helmholtz Zentrum München, Technische Universität München, Germany

Abstract. Can we learn from the unknown? Logical data sets of the *ternary* kind are often found in information systems. They contain *unknown* as well as *true/false* values. An unknown value may represent a missing entry (lost or indeterminable) or have meaning, like a *Don't Know* response in a questionnaire. In this paper we introduce algorithms for reducing the dimensionality of logical data (categorical data in general) in the context of a new data mining challenge: Ternary Matrix Factorization (TMF). For a ternary data matrix, TMF exploits ternary logic to produce a *basis* matrix (which holds the major patterns in the data) and a *usage* matrix (which maps patterns to original observations). Both matrices are *interpretable*, and their ternary matrix product approximates the original matrix. TMF has applications in 1) finding targeted structure in ternary data, 2) imputing values through pattern-discovery in highly-incomplete categorical data sets, and 3) solving instances of its encapsulated Binary Matrix Factorization (BMF) problem. Our elegant algorithm FasTer (FASt TERnary Matrix Factorization) has *linear* run-time complexity with respect to the dimensions of the data set and is parameter-robust. A variant of FasTer that exploits useful results from combinatorics provides accuracy bounds for a core part of the algorithm in certain situations. Experiments on synthetic and real-world data sets show that our algorithms are able to outperform state-of-the-art techniques in all three TMF applications with respect to run-time and effectiveness. Finally, convincing speedup and efficiency results on a *parallel* version of FasTer demonstrate its suitability for weak- and strong-scaling scenarios.

Keywords: Three-valued logic; Ternary data; Matrix factorization; Dimensionality reduction; Missing values; Imputation; Parallel algorithms

Received Dec 11, 2014
Revised Mar 02, 2015
Accepted Mar 28, 2015

1. Introduction

Matrix-based dimensionality reduction of noisy discrete data aims to identify a small set of *interpretable* "base patterns" which, given appropriate connectives, can be *combined* to approximately reconstruct each observation in the original data. Binary Matrix Factorization (BMF) (Miettinen et al, 2008) does this for logical data of the binary kind (and hence categorical data by means of simple encoding). Why extend it to Ternary Matrix Factorization (TMF)? The short answer is that ternary data appears frequently in information systems and it is beneficial to process it natively. More colloquially, we argue that "life is full of unknowns" and justify the need for TMF with the following application examples:

1) Patterns in ternary data: Consider voting records from the U.S. House of Representatives[1]. Congressmen voice opinions on 16 key political votes, such as whether the Education and Handicapped Act should be amended to reflect certain needs of disabled infants. The votes are recorded in a ternary way: "yea" (🟩), "nay" (🟥) and uncertainty (⬜). Given voting data, political analysts want to know the general opinions of the congressmen associated with each of the two major political parties (Republican GOP, Democratic Ⓓ), and may be particularly interested in *indecision* (uncertainty).

The analysts might begin with a Singular-Value Decomposition, having preprocessed the ternary data using the arbitrary[2] mappings 🟥 $\mapsto 0$, ⬜ $\mapsto 1$ and 🟩 $\mapsto 2$. The (rounded) basis vectors obtained by considering only the first two singular values in the subsequent decomposition are

Issue:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	-19	-21	-25	-17	-21	-27	-24	-24	-21	-22	-16	-18	-22	-25	-18	-31
	10	-3	16	-17	-19	-14	16	17	16	-1	3	-16	-16	-16	13	5

,

which is clearly difficult to comprehend. Undiscouraged, the analysts look to Non-negative Matrix Factorization (NMF) – its factors do not contain negative values. With the same mappings they get a result similar to

Issue:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	0.24	1.22	0.00	1.8	2.16	2.73	0.00	0.00	0.00	1.13	0.51	1.84	2.18	2.18	2.53	0.00
	1.48	0.83	2.27	0.00	0.00	0.00	2.16	2.19	1.91	0.92	0.95	0.00	0.00	0.00	0.00	1.65

,

which, based on the range of the entries, appears promising. For a clearer picture, the analysts map the values back to their closest ternary value, yielding

Issue:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
GOP	Nay	???	Nay	Yea	Yea	Yea	Nay	Nay	Nay	???	???	Yea	Yea	Yea	Yea	Nay
Ⓓ	???	???	Yea	Nay	Nay	Nay	Yea	Yea	Yea	???	???	Nay	Nay	Nay	Nay	Yea

.

At this point they may well be pleased with the result. After all, they are aware of the democratic tendency for the two major political parties to disagree (🟩 vs. 🟥). That aside, what of the ⬜ values? Are there truly so many issues for which both parties are generally undecided? A brief look at the original data

[1]Data from the UCI Machine-Learning Repository.
[2]Other mapping choices yield analogous results.

reveals that this is *not* the case. In fact, for issues 1, 2, 10 and 11, only 3%, 11%, 2% and 5% of the congressman respectively voted with uncertainty. Why then did NMF produce these *misleading* basis vectors? The answer is that NMF is blind to the semantics of the original ternary data, optimizing the decomposition with respect to the *real-valued addition* used in the classical matrix product for *combining* basis vectors. Otherwise formulated, real-valued addition is the incorrect tool for combining such values because we know that logical data has *nonlinear* combination semantics.

The analysts turn to discrete techniques. After trivially encoding the ternary values into binary – thereby tripling the problem dimensionality – they try the recent Asso algorithm for Boolean Matrix Factorization (Miettinen et al, 2008). Mapping the Asso result[3] back to ternary gives

Issue:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
GOP	Nay	Yea	Nay	Yea	Yea	Yea	Nay	Nay	Nay	Yea	Nay	Yea	Yea	Yea	Nay	Yea
D	Yea	Nay	Yea	Nay	Nay	Nay	Yea	Yea	Yea	Yea	Nay	Nay	Nay	Nay	Yea	Yea

.

The analysts have arrived at the other extreme: there is now *no indication* of which party is the most indecisive. Changing Asso's weightings is futile – it cannot distinguish between the different ternary values in the binary-encoded data.

This paper introduces TMF, which welcomes the requirement to focus on uncertainty to yield

Issue:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
GOP	Nay	Yea	Nay	Yea	Yea	Yea	Nay	Nay	Nay	Nay	Nay	Yea	Yea	Yea	Nay	Yea
D	Yea	Nay	Yea	Nay	Nay	Nay	Yea	Yea	Yea	Yea	Nay	Nay	Nay	Nay	Yea	???

.

Here no pre- or post-processing is required, nor the selection of a sensitive rounding parameter. TMF's novel exploitation of ternary logic directly produces a result which accurately and intuitively divides the congressmen into their associated party. The basis vectors correctly highlight indecision on the final issue – an Export Administration Amendment Act relating to trade with South Africa. This issue is responsible for the largest uncertainty in the data set (almost one quarter of all congressmen), and TMF shows us that it stems from the Democrats. The usage matrix (not shown) additionally includes examples of "conservative" congressmen being explained by neither basis vector, and "progressive" congressmen being explained by the ternary combination of both. We discuss this example further in Section 5.

2) Missing values: In many cases the proposition of *unknown* is understood to mean that the value is *missing* (lost or indeterminable). Categorical data sets with missing values are commonplace, and imputation techniques perform prediction to "fill in the blanks". In contrast, Missing-Value Binary Matrix Factorization (MVBMF) focuses on producing small and tractable factor matrices that encode *interpretable* data patterns based on the information that *is* available. These patterns can in turn be used for imputation if desired. For example, knowing about the core elements of the structure (▀▀) visible to the naked eye in Figure 1 may be useful in an application context, and we can use them to predict

[3] Using $\tau = 0.5$ for the Asso rounding threshold gives the most sensible results here.

the missing (🟥) values if we wish. In this sense, MVBMF is advantageous in that it exposes a succinct set of base patterns in the data *as well* as offering an imputation solution. We show that TMF is an elegant fit for MVBMF, even for data sets which are heavily dominated by missing values (e.g. collaborative filtering approaches to recommender systems, where data sets are usually sparse).

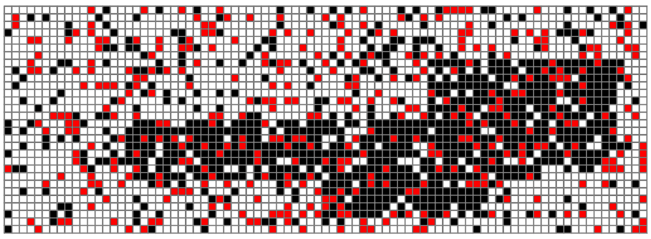

Fig. 1: *Noisy, synthetic binary data (□,■) with missing values (🟥).*

Is imputation the only sensible task to be undertaken for an incomplete categorical matrix? Returning to the concept of semantic ternary data sets, we argue that the missing values might themselves represent structure and thus a level of potentially-valuable information. Consider Figure 2 which depicts an application database lacking data integrity: certain entries which should hold a binary value instead contain an erroneous `NULL` (*unknown*) value. Knowing that `NULL` appears frequently in column a when column b is `TRUE` and column c is `FALSE` would assist application engineers in debugging the problem. In short, TMF can also find patterns that contain missing values.

a	b	c	d	e
...	...	...	...	...
FALSE	FALSE	TRUE	FALSE	TRUE
NULL	TRUE	FALSE	TRUE	TRUE
NULL	TRUE	FALSE	TRUE	FALSE
TRUE	TRUE	TRUE	FALSE	TRUE
NULL	TRUE	FALSE	FALSE	FALSE
...	...	...	...	...

Fig. 2: *A simple application database with erroneous* `NULL` *values. TMF exposes the offending pattern, which can aid in debugging the problem.*

3) Binary Matrix Factorization: We show that the TMF problem inherently encapsulates the problem of Binary Matrix Factorization (BMF). This implies that TMF algorithms, like BMF algorithms, can learn structure from any categorical data set. For the BMF task we show that our algorithm is more efficient and effective than state-of-the-art BMF techniques.

Contributions

– **We introduce Ternary Matrix Factorization (TMF) as a novel data mining challenge**, motivated by the semantics of ternary data which cannot be sufficiently exploited in binary form.

– **We establish a sound optimization goal for TMF** based on ternary logic. This general optimization goal includes BMF as a special case.
– **We provide efficient algorithms for TMF.** The anytime algorithm FasTer effectively solves the NP-Hard TMF problem in *linear time* (with respect to the data set dimensions) and is parameter-robust.
– **We present experiments demonstrating the benefits of using TMF for discovering knowledge through 1) dimensionality reduction of ternary data, 2) missing-value BMF and 3) BMF.** For each type of problem, FasTer outperforms state-of-the-art techniques in terms of effectiveness *and* efficiency.
– **We exploit hardness-of-approximation results in a variant of FasTer,** showing again through systematic experiments that it can significantly improve solution accuracy.
– **We parallelize FasTer** to further demonstrate its scalability. After weighing a number of factors related to high-performance computing, we experiment with three scheduling strategies in weak- and strong-scaling scenarios using data matrices with up to 120 million ternary entries. To the best of our knowledge, these are the first results on parallelism to be published in the area of discrete, logical matrix factorizations.

The initial conference version of this paper introduced TMF as a novel data-mining task (Maurus and Plant, 2014). In this paper, we extend this work in the way described by the final two contributions above. We also enhance the examples and explanations in a number of areas (the new $\oslash_o$ dissimilarity measure in Section 2 is one such example).

Considering the notation presented in Table 1, we continue by discussing ternary logic in Section 2. We formally define the TMF problem and analyze its complexity in Section 3. In Section 4 we present efficient algorithms for TMF approximation, as well as approaches for their parallelization. Experimental results and comparisons are given in Section 5, and Section 6 discusses these findings in light of related work. Section 7 gives concluding remarks. Finally we note that this paper is best viewed in color.

$\|M\|_1$	Entry-wise 1-norm of matrix M (upper case)
m_{ij}, $m_{i\cdot}$, $m_{\cdot j}$	Entry (i, j), row vector i, column vector j from matrix M
$\mathbf{v}_i$	Entry i from a vector $\mathbf{v}$ (bold lower case)
C	Matrix representing the original data set
n, m	Scalars (regular lower case) representing the number of observations (rows) and attributes (columns) in C respectively
k	The rank of the matrix decomposition (equal to the number of rows in B and columns in S)
B	Basis matrix, the k rows of which are the patterns in C
S	Usage matrix, describing each observation in C as a combination of basis vectors from B
$\mathbb{B} = \{\mathfrak{f}, \mathfrak{t}\}$	Set of binary values *false* and *true* (Fraktur typeface)
$\mathbb{T} = \{\mathfrak{f}, \mathfrak{u}, \mathfrak{t}\}$	Set of ternary values *false*, *unknown* and *true*
$\wedge, \vee$	Logical conjunction, disjunction
$\circ, \oplus$	Binary matrix product, binary dissimilarity measure
$\vartriangle, \oslash, \star$	Ternary matrix product, ternary dissimilarity measure, ternary scalar product

Table 1: *Nomenclature, symbols and notation.*

2. Ternary Logic

Ternary logic (Kleene, 1938) extends binary logic by including an additional categorical value "whose logical value of truth or falsity is undefined, undetermined by means of accessible algorithms, or not essential for actual consideration" (Malinowski, 2007). Henceforth we refer to this value as *unknown* and denote it $\mathfrak{u}$ alongside the classical values of $\mathfrak{f}$ and $\mathfrak{t}$ for *false* and *true* respectively. We denote the set of ternary values as $\mathbb{T} = \{\mathfrak{f}, \mathfrak{u}, \mathfrak{t}\}$. With this notation, the classical set of binary values is denoted $\mathbb{B} = \{\mathfrak{f}, \mathfrak{t}\}$. It is clear that $\mathbb{B} \subset \mathbb{T}$.

Real-world applications in which ternary data are relevant include:

– **Surveys:** Many data-collection processes involve questions to be answered. *Don't know* ($\mathfrak{u}$) response options are commonly found alongside the binary *Yes* ($\mathfrak{t}$) and *No* ($\mathfrak{f}$) options in surveys and questionnaires (Rubin et al, 1995; Francis and Busch, 1975). Here a ternary value of $\mathfrak{u}$ has an important contextual meaning, such as "I do not know if I am *yea* or *nay* on this issue".

– **Missing values:** Categorical data sets with missing values are common. Here an instance of $\mathfrak{u}$ is understood as a placeholder for an underlying but inaccessible binary value. Depending on the context (Figure 2, for example), the simple fact that the value *is* missing may itself represent a level of information.

– **Database frameworks:** Relational database engines use ternary logic for handling comparisons with `NULL` field content (Codd, 1986). Here `NULL` is equivalent to $\mathfrak{u}$.

Ternary connectives for conjunction ($\wedge$) and disjunction ($\vee$) can be exploited to develop a dimension-reduction technique for ternary data. These connectives, defined in the ternary logic systems of Łukasiewicz, Kleene and Bochvar (Malinowski, 2007), are shown as part of Table 2.

$\wedge$	$\mathfrak{f}$	$\mathfrak{u}$	$\mathfrak{t}$
$\mathfrak{f}$	$\mathfrak{f}$	$\mathfrak{f}$	$\mathfrak{f}$
$\mathfrak{u}$	$\mathfrak{f}$	$\mathfrak{u}$	$\mathfrak{u}$
$\mathfrak{t}$	$\mathfrak{f}$	$\mathfrak{u}$	$\mathfrak{t}$

$\vee$	$\mathfrak{f}$	$\mathfrak{u}$	$\mathfrak{t}$
$\mathfrak{f}$	$\mathfrak{f}$	$\mathfrak{u}$	$\mathfrak{t}$
$\mathfrak{u}$	$\mathfrak{u}$	$\mathfrak{u}$	$\mathfrak{t}$
$\mathfrak{t}$	$\mathfrak{t}$	$\mathfrak{t}$	$\mathfrak{t}$

$\oslash_c$	$\mathfrak{f}$	$\mathfrak{u}$	$\mathfrak{t}$
$\mathfrak{f}$	0	1	1
$\mathfrak{u}$	1	0	1
$\mathfrak{t}$	1	1	0

$\oslash_{mv}$	$\mathfrak{f}$	$\mathfrak{u}$	$\mathfrak{t}$
$\mathfrak{f}$	0	1	1
$\mathfrak{u}$	0	1	0
$\mathfrak{t}$	1	1	0

$\oslash_o$	$\mathfrak{f}$	$\mathfrak{u}$	$\mathfrak{t}$
$\mathfrak{f}$	0	$1/2$	1
$\mathfrak{u}$	$1/2$	0	$1/2$
$\mathfrak{t}$	1	$1/2$	0

Table 2: *Conjunction ($\wedge$), disjunction ($\vee$), unbiased dissimilarity ($\oslash_c$), missing-value dissimilarity ($\oslash_{mv}$) and ordinal dissimilarity ($\oslash_o$) over $\mathbb{T}$. $\oslash_{mv}$ is the only asymmetric table and is oriented such that the top-right entry represents ($\mathfrak{f} \oslash_{mv} \mathfrak{t}$).*

The connectives $\wedge$ and $\vee$ encapsulate their binary counterparts. The other evaluations for $\wedge$ and $\vee$ – those involving at least one $\mathfrak{u}$ value – are intuitively understood: $\mathfrak{f}$ and $\mathfrak{t}$ remain the annihilators for conjunction and disjunction respectively, otherwise the result is semantically defined as *unknown*.

A quantifiable measure for the dissimilarity ("difference") between two arbitrary ternary values is necessary for mining ternary data. For the binary case the classical measure is analogous to the XOR connective: zero for $a = b$ and one for $a \neq b$ (we denote this measure with $\oplus : \mathbb{B} \times \mathbb{B} \mapsto \mathbb{R}_0^+$). In BMF this (optionally-weighted) measure is used in the objective function to penalize incorrect coverage of original data values (Miettinen et al, 2008). An appropriate extension of this measure to the ternary set $\mathbb{T}$ depends on the application context. For example, a value of $\mathfrak{u}$ may be understood as particularly important to preserve (as is the case with the Congressional Voting Records example given earlier). In other situations (e.g. imputation) we may not want *any* $\mathfrak{u}$ values in the reconstructed matrix. We thus proceed with an abstract ternary dissimilarity measure (denoted $\oslash : \mathbb{T} \times \mathbb{T} \mapsto \mathbb{R}_0^+$). In the context of TMF we interpret $(a \oslash b) \in \mathbb{R}_0^+$ as the penalty for covering an original data value $a \in \mathbb{T}$ with value $b \in \mathbb{T}$. We impose that $(c \oslash d) = (c \oplus d)$ for $c, d \in \mathbb{B}$ in order to ensure that $\oslash$ preserves the results of its binary counterpart. Table 2 defines an unbiased, commonly-used concrete version $\oslash_c$ for the case in which all ternary values have equal importance. It also shows a concrete version $\oslash_{mv}$ for missing-value (imputation) problems – here it is clear that a penalty is to be enforced in TMF when covering any original data value with the value $\mathfrak{u}$. The $\oslash_{mv}$ version also shows that reflexivity $[a = b \to (a \oslash b) = 0]$, symmetry $[(a \oslash b) = (b \oslash a)]$ and strictness $[(a \oslash b) = 0 \to a = b]$ are *not* required to hold for $\oslash$ in general. Finally, the table shows the variant $\oslash_o$, which is appropriate if the ternary values are understood to be ordinal in nature (like in a Likert item) and the position of $\mathfrak{u}$ is *between* $\mathfrak{f}$ and $\mathfrak{t}$ (decompositions on ordinal data are investigated further in (Belohlavek, 2013)).

For brevity we abuse the notation and reuse the symbol $\oslash$ for two equally-sized vectors or matrices. Computation is done entry-wise in these cases.

Analogously to the binary case (Miettinen et al, 2008) we define the ternary matrix product $\vartriangle$ as the matrix product over the semiring $\langle \mathbb{T}, \vee, \wedge, \mathfrak{f}, \mathfrak{t} \rangle$ (ternary domain $\mathbb{T}$, "addition" commutative monoid $\vee$ with identity $\mathfrak{f}$, and "multiplication" monoid $\wedge$ with identity $\mathfrak{t}$). Otherwise formulated: given $Q \in \mathbb{T}^{n \times k}$ and $R \in \mathbb{T}^{k \times m}$, the matrix $P \in \mathbb{T}^{n \times m}$ produced from $P = Q \vartriangle R$ has entries calculated using the ternary scalar product $\star$

$$p_{ij} = (Q \vartriangle R)_{ij} = q_{i\cdot} \star r_{\cdot j} = \bigvee_{x=1}^{k} (q_{ix} \wedge r_{xj})\,.$$

Here $\bigvee$ is aggregate ternary disjunction. Note that $\vartriangle$ reduces to $\circ$ (the binary matrix product) when $Q \in \mathbb{B}^{n \times k}$ and $R \in \mathbb{B}^{k \times m}$ because $\vee$ and $\wedge$ preserve the results of their binary counterparts. In such a situation it is clear that $P \in \mathbb{B}^{n \times m}$.

3. The Ternary Matrix Factorization Problem

Before introducing the TMF problem we provide some context by restating the related BMF problem (Miettinen and Vreeken, 2014).

Problem (BMF). *Given a binary data matrix $C \in \mathbb{B}^{n \times m}$ and positive integer k, find a binary usage matrix $S \in \mathbb{B}^{n \times k}$ and basis matrix $B \in \mathbb{B}^{k \times m}$ that minimize*

$$\|C \oplus (S \circ B)\|_1. \qquad (1)$$

Here $\circ$ represents the binary matrix product (Miettinen et al, 2008), defined similarly to $\vartriangle$ except that it uses the binary scalar product which in turn uses the traditional binary connectives. We now introduce TMF as a "logical" extension to BMF.

Problem (TMF). *Given a ternary data matrix $C \in \mathbb{T}^{n\times m}$ and positive integer k, find a binary usage matrix $S \in \mathbb{B}^{n\times k}$ and a ternary basis matrix $B \in \mathbb{T}^{k\times m}$ that minimize*

$$\|C \oslash (S \vartriangle B)\|_1, \tag{2}$$

with the additional restriction that B be binary if C is binary.

Noteworthy is the binary-set restriction for S. Although $\mathfrak{u}$ values in S do not present any difficulties in the matrix product, intuition fails to justify their place. Restricting the usage matrix to binary allows each observation vector in C to be explained by the *presence* or *absence* of each *ternary* basis vector. A $\mathfrak{u}$ value in S would imply that the inclusion of the corresponding basis vector is uncertain. We argue that such behavior deviates from the goal of searching for interpretable, unambiguous structure in the data set and hence scope the TMF problem such that the matrix S is restricted to the set $\mathbb{B}$. Showing that BMF is a special case of TMF is simple.

Theorem 1. *BMF (Miettinen and Vreeken, 2014) is a special case of TMF.*

Proof. For a given *binary* data matrix C, the TMF and BMF solution spaces are identical (S and B must be binary for both problems). The optimization goals are also equivalent, because $(\mathbf{a} \star \mathbf{b}) = (\mathbf{a} \circ \mathbf{b})$ for binary vectors $\mathbf{a}$ and $\mathbf{b}$, and $(a \oslash b) = (a \oplus b)$ for $a, b \in \mathbb{B} \subset \mathbb{T}$. $\square$

The ternary nature of B differentiates TMF from other methods which decompose categorical matrices. Specifically, TMF can natively handle categorical data of the ternary kind. In contrast, BMF mandates that we binary-encode such data beforehand. The drawbacks are that 1) the dimensionality of the problem increases (at least by a factor of two, depending on how the encoding is chosen), 2) the BMF technique may not preserve the integrity of the encoding (resulting in potentially-ambiguous results), and 3) the flexibility to weigh the importance of original data values is lost. By using TMF we avoid these problems for categorical data of the ternary kind, and thus take a definitive step towards the idea of *true* semantic categorical matrix factorization where encoding is not required. We give examples of this on synthetic and real-world data sets in Section 5.

The TMF solution space contains $2^{nk} \cdot 3^{km}$ unique combinations of S and B. In the next section we show that it is computationally useful to consider two related problems with a narrower solution space. We define these as the Ternary Usage Problem (TUP) and the Ternary Basis Problem (TBP).

Problem (TUP). *Given a ternary data matrix $C \in \mathbb{T}^{n\times m}$ and basis matrix $B \in \mathbb{T}^{k\times m}$, find a binary usage matrix $S \in \mathbb{B}^{n\times k}$ that minimizes (2).*

Problem (TBP). *Given a ternary data matrix $C \in \mathbb{T}^{n\times m}$ and binary usage matrix $S \in \mathbb{B}^{n\times k}$, find a ternary basis matrix $B \in \mathbb{T}^{k\times m}$ that minimizes (2), with the additional restriction that B be binary if C is binary.*

Whilst the solution spaces of the TUP and TBP are smaller (2^{kn} and 3^{km} possible solutions respectively), generating a TMF-optimal solution is clearly dependent on optimal inputs for B and S in each respective sub-problem.

Given that the decision version of the BMF problem is NP-Hard (Miettinen et al, 2008), one can trivially show that the decision version of TMF is NP-Hard (BMF is a special case of TMF).

4. A Fast Algorithm for Ternary Matrix Factorization

In Algorithm 1 we present FasTer, an effective heuristic procedure for FASt TERnary Matrix Factorization with a downloadable C++ implementation[4]. FasTer begins by selecting k "representative" rows from C (function `SampleKRows`)[5] as the initial basis matrix. This task is a discrete analog to the column-subset problem (Çivril and Magdon-Ismail, 2009). We approach it in a scalable way by choosing the first row randomly, then subsequently selecting the rows which maximize the aggregate Hamming distance to those already chosen. This diversity approach reduces the likelihood of converging to an undesirable local minimum.

Given this initial basis matrix B_0, the main iteration loop begins by solving the TUP to find an initial usage matrix S_0. Here the optimization can be performed for each row independently. Initially, each observation uses no basis vectors. The optimization for the observation proceeds in a similar way to the classical heuristic for set covering: basis vectors are added in a greedy fashion, one after another, until (2) ceases to decrease. We note that basis vectors may contain noise, however these vectors can still be added if they result in an *overall* error reduction. After this step is complete for all observations, the initial usage matrix S_0 is available.

The algorithm proceeds to solve the TBP based on S_0. Optimization can be performed for each column independently. For each column, a value of $\mathfrak{t}$ or $\mathfrak{u}$ is set in a basis vector if it is profitable considering the observations specified by the corresponding column in S_0. After this step is complete for all columns, the next basis matrix B_1 is available.

As illustrated in Figure 3, these complementary optimization steps alternate in the classical fashion. Solving the TUP based on B_1 yields S_1. Subsequently solving the TBP based on S_1 yields B_2, and so on. The iteration terminates when the global error ceases to decrease. In each refinement step, each sub-problem (i.e. each row in the TUP and each column in the TBP) involves "resetting" the corresponding vector (i.e. setting all entries to $\mathfrak{f}$) and rebuilding incrementally. This simple approach continually filters noise from the basis vectors, keeping them succinct. The usage matrix profits as well: the basis vectors chosen to describe an observation must "earn their keep" in every iteration. To the best of our knowledge, FasTer's alternating approach at this higher level is novel in the context of discrete, logical matrix factorizations.

One limitation of the algorithm is the random selection of the first basis vector from C. To counter this limitation, the algorithm is repeated P times, where P is a number of randomization rounds. We show in Section 5 that P need

[4]dropbox.com/s/znut1tsxutyjfvu/tmf.zip

[5]For imputation problems, values of $\mathfrak{u}$ are replaced with $\mathfrak{f}$ in this selection to ensure that only binary values are present in the initial basis vector.

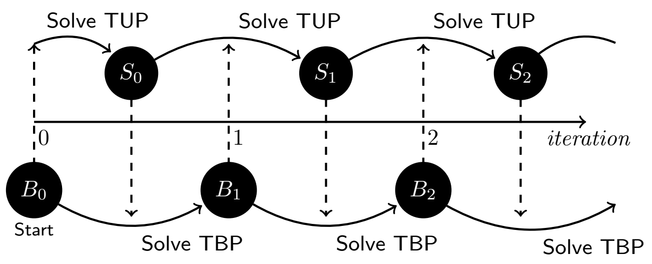

Fig. 3: *FasTer solves for S and B in a leapfrog fashion.*

not be large to significantly reduce the risk of sub-optimal results (i.e. FasTer can use a small default for P).

Theorem 2. *Every randomly-initialized iteration of the* FasTer *algorithm converges to a local TMF optimum.*

Proof. The `while` loop in the function `OptimizeVector` executes $\mathsf{N}_{ov} \leq (k+1)$ iterations (the upper-bound here represents the case in which *all* k entries of the vector $\mathbf{b}$ are set). The first $\mathsf{N}_{ov} - 1$ iterations each make a single-entry modification to $\mathbf{b}$ (the one yielding the largest error-reduction). The final iteration makes no change to $\mathbf{b}$ (no single-entry modifications are profitable). The vector $\mathbf{b}$ returned from `OptimizeVector` is hence the optimal choice with respect to the vector $\mathbf{c}$ and this greedy, local-optimization heuristic. We now focus on the first iteration of the global `while` loop. Here the matrix S_0 is optimally found with respect to the aforementioned strategy and the $\mathbf{c}$ vectors taken from the initial basis matrix B_0. The next-generation basis matrix is then found optimally with respect to the same strategy and the newly-optimized matrix S_0. This process continues until the error reduction Δ becomes nonnegative, which clearly occurs in a finite number of iterations considering line 11 and the fact that $\oslash$ by definition yields nonnegative results. $\square$

4.1. Run-Time Complexity

In FasTer, the frequent error evaluations are done on lines 2 and 9 of the function `OptimizeVector`, where the ternary scalar product $\star$ involves two ternary vectors of length k. We evaluate this product in $\mathcal{O}(1)$ operations by exploiting native hardware bitwise instructions. This is done by transforming each vector into a concatenated binary word using the mappings $\mathfrak{t} \mapsto 11_2$, $\mathfrak{f} \mapsto 00_2$ and $\mathfrak{u} \mapsto 01_2$ (the mapping $\mathfrak{u} \mapsto 10_2$ is also valid and this choice is arbitrary as long as consistency is maintained). For example, the ternary vector $\mathbf{v} = (\mathfrak{t}, \mathfrak{f}, \mathfrak{u}, \mathfrak{t})$ becomes the word 11000111_2. With these mappings, bitwise disjunction (OR) and conjunction (AND) produce the same results as their ternary counterparts. This approach clearly requires that $k \leq b/2$, where b is the hardware word size in bits. On current general-purpose processors where $b = 64$, these efficient bitwise operations can be exploited for all $k \leq 32$ (a generous bound considering that we usually want k to be small). The worst-case complexity of line 9 is hence $\mathcal{O}(m)$ and $\mathcal{O}(n)$ when solving the TUP and TBP respectively. We further note that computation can be aborted as soon as $f \geq f_l$.

```
input  : data matrix $C \in \mathbb{T}^{n\times m}$, rank $k$
output: matrices $S \in \mathbb{B}^{n\times k}$ and $B \in \mathbb{T}^{k\times m}$
$B \leftarrow$ SampleKRows($C$,$k$);
$\Delta \leftarrow -\infty$;                           /* Iterative error change */
$e_b \leftarrow \infty$;                                /* Smallest error so far */
while $\Delta < 0$ do
    for $i \leftarrow 1$ to $n$ do                       /* Solve the TUP */
        $S_{i\cdot} \leftarrow$ OptimizeVector($B^\intercal$,$c_{i\cdot}$,$(\mathfrak{t})$);
    end
    for $i \leftarrow 1$ to $m$ do                       /* Solve the TBP */
        $B_{\cdot i} \leftarrow$ OptimizeVector($S$,$c_{\cdot i}$,$(\mathfrak{u},\mathfrak{t})$);
    end
    $e \leftarrow \|C \oslash (S \vartriangle B)\|_1$;                  /* Global error */
    $\Delta \leftarrow e - e_b$;
    if $\Delta < 0$ then $e_b \leftarrow e$ ;
end
```

Algorithm 1: FasTer

```
$\mathbf{b} \leftarrow (\mathfrak{f},\dots,\mathfrak{f}) \in \{\mathfrak{f}\}^k$;                      /* Start empty */
$f_b \leftarrow \sum_{i=1}^{|\mathbf{c}|} (\mathbf{c}_i \oslash (\mathbf{b} \star \boldsymbol{U}_{i\cdot}))$;           /* Initial error */
$\Delta \leftarrow -\infty$;
while $\Delta < 0$ do
    $\mathbf{l} \leftarrow \mathbf{b}$; $f_l \leftarrow f_b$ ;                           /* Clone optimum */
    for $i \leftarrow 1$ to $k$ where $\mathbf{l}_i = \mathfrak{f}$ do
        for $j \leftarrow 1$ to $|\mathbf{d}|$ do                          /* Modify */
            $\mathbf{y} \leftarrow (\mathbf{b}_1,\dots,\mathbf{b}_{i-1},\mathbf{d}_j,\mathbf{b}_{i+1},\dots,\mathbf{b}_k)$;
            $f \leftarrow \sum_{i=1}^{|\mathbf{c}|} (\mathbf{c}_i \oslash (\mathbf{y} \star \boldsymbol{U}_{i\cdot}))$;
            if $f < f_l$ then                                /* Profitable? */
                $\mathbf{l} \leftarrow \mathbf{y}$; $f_l \leftarrow f$;
            end
        end
    end
    $\Delta \leftarrow f_l - f_b$;
    if $\Delta < 0$ then                                /* Persist optimum */
        $\mathbf{b} \leftarrow \mathbf{l}$; $f_b \leftarrow f_l$;
    end
end
return $\mathbf{b}$;
```

Function OptimizeVector($\boldsymbol{U}$,$\mathbf{c}$,$\mathbf{d}$)

The function `OpimizeVector` involves an inner `for` loop executing k iterations. The error is evaluated in each, and thus each iteration of the outer `while` loop has a worst-case complexity of $\mathcal{O}(km)$ and $\mathcal{O}(kn)$ when solving the TUP and TBP respectively. The largest number of iterations possible for the outer `while` loop in this function is $k+1$ (representing the case where the vector under consideration has each of its k entries set, one after another). The worst-case complexity for the `OptimizeVector` function is thus $\mathcal{O}\left(k^2m\right)$ and $\mathcal{O}\left(k^2n\right)$ for the TUP and TBP respectively.

It is then clear that each iteration of the top-most `while` loop in FasTer has a complexity of $\mathcal{O}\left(k^2nm\right)$. The convergence of the `while` loop requires a number of iterations, which we will denote N. We show empirically in Section 5 that N does not increase with increasing n, m or k (convergence of the `while` loop occurs with $\mathsf{N} < 40$ in all experiments conducted to date). Likewise we show in Section 5 that the number of user-defined iterations P does not need to be increased with increasing n, m or k in order to achieve competitive optimization results. We thus treat both N and P as constants in terms of run-time complexity, resulting in the attractive overall worst-case complexity of $\mathcal{O}\left(k^2nm\right)$ for FasTer. Thus, assuming the aforementioned condition for k holds, FasTer is linear in the dimensions of the data set (generally large) and quadratic in k (small by definition).

FasTer lends itself well to parallelization (Section 4.3) and needs no floating-point arithmetic (Asso (Miettinen et al, 2008), in contrast, operates on an $\mathbb{R}^{m\times m}$ matrix for calculating *association accuracies*).

4.2. Improving Solution Accuracy with FasTer$_{\pm\mathrm{PSC}}$

Can we find an alternative to FasTer's core `OptimizeVector` heuristic for which a bound on the solution quality can be proven? It turns out that the problem of solving for a binary row of the TUP[6] (line 6 of FasTer) is very similar in nature to a known problem from combinatorics. The problem, Positive-Negative Partial Set Cover (±PSC), admits a polynomial-time algorithm with a proven approximation factor (Miettinen, 2008*b*). This hardness result lends credit to the idea that the use of ±PSC in FasTer – a variant we denote with FasTer$_{\pm\mathrm{PSC}}$ – will yield more accurate results. We investigate ±PSC to this end in this section.

To begin, we consider the task of solving for a usage row vector in the TUP (line 6 of FasTer) and restrict ourselves to the case where all values are binary. The problem can be broadly visualized as

$$\begin{bmatrix} c_{i1} & c_{i2} & \cdots & c_{im} \end{bmatrix} \approx \begin{bmatrix} s_{i1} & \cdots & s_{ik} \end{bmatrix} \vartriangle \begin{bmatrix} b_{11} & \cdots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{k1} & \cdots & b_{km} \end{bmatrix}, \tag{3}$$

where we understand that our goal is to find an $s_{i\cdot}$ which minimizes (2). If all values are binary then this matrix formulation is simply a different perspective on what is clearly a kind of set-covering problem (Miettinen, 2008*a*). That is, we are tasked with choosing *sets* (basis vectors) from a *collection* (the basis matrix) such that their union effectively "covers" a certain *universe* (the data vector) with respect to an objective function. Our goal, the TMF objective function, is designed to be robust against noise, and so the classical set-covering problem is too strict. That is, we seldom want *every* universe element (indices of $\mathtt{t}$ values in $c_{i\cdot}$) to be accounted for in a noisy setting – we instead simply want to minimize false positives and false negatives. The aforementioned ±PSC problem provides the appropriate level of relaxation:

[6]We focus on the TUP rather than the TBP because, in the general case, the TBP involves ternary logic and cannot be easily compared with or reduced to combinatorial problems.

Problem ($\pm$PSC). *Given disjoint sets $\mathcal{P}$ and $\mathcal{N}$ ("positive" and "negative" elements respectively) and a collection $\mathfrak{S} = \{\mathcal{S}_1, \dots, \mathcal{S}_k\} \subset 2^{\mathcal{P} \cup \mathcal{N}}$, find a collection $\mathfrak{C} \subset \mathfrak{S}$ which minimizes the cost*

$$\operatorname{cost}_{\pm\text{PSC}}(\mathcal{P}, \mathcal{N}, \mathfrak{C}) = \left| \mathcal{P} \setminus \left(\bigcup_{\mathcal{C} \in \mathfrak{C}} \mathcal{C} \right) \right| + \left| \mathcal{N} \cap \left(\bigcup_{\mathcal{C} \in \mathfrak{C}} \mathcal{C} \right) \right| . \tag{4}$$

In English, the $\pm$PSC objective function (4) simply counts the number of errors (false positives and false negatives) made by a given cover $\mathfrak{C}$. The problem of finding the vector $s_{i\cdot}$ in (3) is then simply the $\pm$PSC instance with $\mathcal{P} = \{j \mid c_{ij} = \mathfrak{t}\}$, $\mathcal{N} = \{j \mid c_{ij} = \mathfrak{f}\}$ and $\mathfrak{S} = \{\{j \mid b_{1j} = \mathfrak{t}\}, \dots, \{j \mid b_{kj} = \mathfrak{t}\}\}$. The $\pm$PSC objective function (4) equally penalizes false positives and false negatives (incorrectly reconstructed values of $\mathfrak{t}$ and $\mathfrak{f}$ respectively), and is thus equivalent to the TMF objective function $\oslash$ for this binary case.

An algorithm for $\pm$PSC with the approximation factor of $2\sqrt{(k + |\mathcal{P}|) \ln |\mathcal{P}|}$ is known (Miettinen, 2008*b*). It involves a reduction to the Red-Blue Set Cover problem (Peleg, 2007), whose approximation factor is proven by a reduction to the classical weighted set-cover problem (Chvatal, 1979).

We now consider the general case of ternary data in C and B. We denote the problem of solving for a row $s_{i\cdot}$ in the TUP as an instance of the Ternary Usage Row (TUR) problem:

Problem (TUR). *Given a vector $\mathbf{c} \in \mathbb{T}^m$ and a matrix $B \in \mathbb{T}^{k \times m}$, find a vector $\mathbf{s} \in \mathbb{B}^k$ which minimizes $\|\mathbf{c} \oslash (\mathbf{s} \vartriangle B)\|_1$.*

The important question of this section is: can we reduce TUR to $\pm$PSC? In order to achieve this, we need to express the ternary vector $\mathbf{c}$ as $\pm$PSC sets $\mathcal{P}$ and $\mathcal{N}$. Likewise, we need to express each of the k ternary basis vectors as $\pm$PSC sets $\mathcal{S}_1 \dots \mathcal{S}_k$. The optimization goal of the resulting $\pm$PSC instance must additionally reflect the optimization goal of the original TUR instance.

We start by considering the form of the simple case where $m = 1$,

$$[c_{i1}] \approx [s_{i1} \quad \cdots \quad s_{ik}] \vartriangle \begin{bmatrix} b_{11} \\ \vdots \\ b_{k1} \end{bmatrix},$$

and take $\oslash_o$ as an example concrete dissimilarity measure. We are now faced with choosing "positive" $\mathcal{P}$, "negative" $\mathcal{N}$ and "basis vector" $\mathcal{S}$ sets for $\pm$PSC that represent each possible ternary value $c_{i1} \in \mathbb{T}$. To achieve this, we can map ternary values to sets (using arbitrary elements α and β) in the following way:

Case	$\mathcal{P}$	$\mathcal{N}$
$c_{i1} = \mathfrak{f}$	$\{\}$	$\{\alpha, \beta\}$
$c_{i1} = \mathfrak{u}$	$\{\alpha\}$	$\{\beta\}$
$c_{i1} = \mathfrak{t}$	$\{\alpha, \beta\}$	$\{\}$

Case	$\mathcal{S}_j$ (for $j = 1 \dots k$)
$b_{j1} = \mathfrak{f}$	$\{\}$
$b_{j1} = \mathfrak{u}$	$\{\alpha\}$
$b_{j1} = \mathfrak{t}$	$\{\alpha, \beta\}$

Using this choice, the set-union of "basis vector" sets $\mathcal{S}_j$ preserves the ternary disjunction operation as required. We can also observe that our objective functions are equivalent with respect to optimization. For example, if we obtained a solution with cover $\{\}$ (ternary value $\mathfrak{f}$) to a $\pm$PSC instance with $\mathcal{P} = \{\alpha, \beta\}$ (ternary value $\mathfrak{t}$) and $\mathcal{N} = \{\}$, the objective function (4) states that our cost is two. If the solution's cover were instead $\{\alpha\}$ (ternary value $\mathfrak{u}$), the same objective function prescribes a cost of one. We recognize that the objective function is in agreement with (a constant scaling of) the TMF objective function that uses $\oslash_o$, and it is trivial to show that this result extends to the case of $m > 1$. Thus, we can successfully reduce TUR to ±PSC when $\oslash_o$ is our dissimilarity measure.

Does the result extend to arbitrary concrete versions of $\oslash$? In analyzing the case of $\oslash_c$ in a similar way, we arrive at the following theorem:

Theorem 3. *It is not possible to reduce TUR to* ±PSC *such that the* $\oslash_c$ *dissimilarity measure and ternary disjunction semantics are preserved.*

Proof. It suffices to consider a TUR instance with $m = 1$. We seek mappings $\mathfrak{f} \mapsto \mathcal{F}$, $\mathfrak{u} \mapsto \mathcal{U}$ and $\mathfrak{t} \mapsto \mathcal{T}$ from ternary values to sets for use in the ±PSC. Ternary disjunction (Table 2) mandates conditions such as $\mathcal{F} \cup \mathcal{U} = \mathcal{U}$ and $\mathcal{U} \cup \mathcal{T} = \mathcal{T}$, and from these conditions we recognize that $\mathcal{F} \subset \mathcal{U} \subset \mathcal{T}$. This implies that

$$\mathcal{F} \setminus \mathcal{U} = \mathcal{F} \setminus \mathcal{T} = \mathcal{U} \setminus \mathcal{T} = \{\}\,. \tag{5}$$

To preserve $\oslash_c$ (to within a constant factor) we require a constant set-difference size between each pairwise combination of sets, that is

$$|(\mathcal{T} \setminus \mathcal{U}) \cup (\mathcal{U} \setminus \mathcal{T})| = |(\mathcal{T} \setminus \mathcal{F}) \cup (\mathcal{F} \setminus \mathcal{T})| = |(\mathcal{U} \setminus \mathcal{F}) \cup (\mathcal{F} \setminus \mathcal{U})|\,,$$

which upon substitution with (5) reduces to

$$|(\mathcal{T} \setminus \mathcal{U})| = |(\mathcal{T} \setminus \mathcal{F})| = |(\mathcal{U} \setminus \mathcal{F})|\,.$$

Clearly there are no sets for which this holds under the condition $\mathcal{F} \subset \mathcal{U} \subset \mathcal{T}$. □

Assuming we wish to use the unbiased dissimilarity measure $\oslash_c$, it follows that it is not possible to exploit the ±PSC approximation factor by reducing TUR to ±PSC. For FasTer$_{\pm\text{PSC}}$ in general, this implies that the strict reducibility of TUR to ±PSC is dependent on our choice of $\oslash$. The reduction succeeds for the ordinal dissimilarity measure $\oslash_o$. In the general case, the ±PSC algorithm simply becomes an alternate heuristic.

The run-time complexity of the ±PSC algorithm is higher than FasTer's `OptimizeVector` heuristic. To see why, we note that $m + k$ sets are created out of our k basis vector sets in the worst case during the reduction of ±PSC to Red-Blue Set Cover (Miettinen, 2008*b*). Solving a weighted set-cover instance is the most computationally-significant part of solving Red-Blue Set Cover. By using a priority queue and an inverted index, solving a weighted set-cover instance has a worst-case run-time in $\mathcal{O}\left(|\mathcal{U}| \cdot |\mathfrak{S}| \cdot \log |\mathfrak{S}|\right)$, where $|\mathcal{U}|$ is the number of elements in the universe and $|\mathfrak{S}|$ the number of sets in the weighted set-cover instance (Cormode et al, 2010). In our worst-case scenario, the number of universe elements is $|\mathcal{U}| = m$ and the number of sets $|\mathfrak{S}| = m + k$, so the worst-case run-time is in $\mathcal{O}\left(m \cdot (m + k) \cdot \log(m + k)\right)$. Therefore, unlike the `OptimizeVector` heuristic, the run-time of the ±PSC variant is super-linear in m.

Considering the results presented here, Section 5 directly compares FasTer and FasTer$_{\pm\text{PSC}}$ for BMF and TMF instances where the minimization of $\oslash_c$ is the goal.

4.3. Efficiently parallelizing FasTer

Practitioners often look to parallelism for processing massive data sets and/or improving response time. Although an aggressive optimization of FasTer's parallel performance on the various kinds of TMF problems is not within the scope of this paper, we offer here a discussion on a number of the relevant factors. Through this discussion, we justify our choice of the shared-memory (OpenMP) approach for parallelizing FasTer's TUP and TBP loops (used in our experiments on parallelization in Section 5.8).

Typically, parallelizing a program serves to overcome memory and/or computational bottlenecks. We choose to focus on computational rather than memory bottlenecks because 1) storage of the discrete ternary values $\{\mathfrak{f}, \mathfrak{u}, \mathfrak{t}\}$ required for computation (e.g. the matrix C) can be done concisely and can exploit sparsity, and 2) FasTer does no floating-point arithmetic and requires no large intermediate storage.

If we assume that k is small, computational bottlenecks arise in FasTer if the data matrix is large or if a higher solution accuracy is sought (through the selection of a larger number of randomization rounds P). In the latter case, the improvement in accuracy is not proportional to P (the experiments in Section 5.7 highlight this "diminishing returns" behavior). Striving for higher *accuracy* through more processors is hence rather inefficient, and so we focus instead on strong- and weak-scaling scenarios for a fixed P.

In a strong-scaling scenario ("solving a given problem faster"), we fix the size of the TMF instance and increase the number of processors available for sharing the work. The wall-clock execution time decreases by a factor proportional to the number of processors in the ideal case. In a weak-scaling scenario ("solving a larger problem in the same time"), the size of the problem is increased proportionally to the number of processing units so that each processing unit is always responsible for a fixed amount of work. The wall-clock execution time remains constant in the ideal case.

In FasTer, the convergence-iteration loop on line 4 is not a good candidate for parallelization due to the data ("flow") dependency between each iteration and its predecessor. We instead investigate the options for parallelizing the TUP and TBP loops on lines 5 and 8. Considering that our focus is on computational rather than memory bottlenecks, we choose with OpenMP (OpenMP Architecture Review Board, 2005) a shared-memory, "fork-and-join" approach rather than a distributed-memory approach.

A means by which loop iterations are assigned to processing units – a *scheduling strategy* – needs to be selected. Here we recognize that the work done by `OptimizeVector` is not constant for every loop iteration. For example, searching for and choosing a single basis vector may suffice to exactly explain the first data observation when solving for the first usage matrix row in the TUP, whereas many more may be needed to approximate the second. For relatively small data sets, we might therefore expect that a *static* scheduling strategy (which equally divides loop iterations between processing units in advance) yield in performance to a *dynamic* strategy (which assigns smaller chunks of work to each processing unit in a just-in-time manner). For the more practical case of larger data sets, however, we expect that variations in `OptimizeVector` work would tend to balance out across processors.

Also to be considered is the additional overhead of *dynamic* scheduling strategies – they require more time to delegate work to processing units. We note that OpenMP offers two strategies for dynamic scheduling. The first is simply named *dynamic* and involves just-in-time allocation of chunks of *constant* size. The second is named *guided* and involves just-in-time allocation of chunks of *exponentially-decreasing* size. The latter invests in a higher allocation overhead (due to the larger number of chunks typically involved) in order to further homogenize the completion time of each processing unit.

Although not treated by the experiments in this paper, we note that a distributed-memory approach is an option if memory bottlenecks are encountered. In such an approach, each distributed node would always work on the same subset of TUP-row and TBP-column indices respectively, and thus not be required to read the entire data matrix. If bottlenecks involved both memory *and* computation, a hybrid approach (a mixture of distributed- and shared-memory programming) is an option. In such a setup, each distributed processing unit would work on a fixed subset of TUP rows and TBP columns, and would further distribute this work among processing units in its shared-memory environment.

Based on this discussion, Section 5.8 presents speedup and efficiency results for strong- and weak-scaling scenarios based on TUP- and TBP-parallelization using each of the three OpenMP scheduling strategies *static*, *dynamic* and *guided.*

5. Experiments

In this section we compare FasTer and FasTer$_{\pm\text{PSC}}$ to state-of-the-art algorithms using synthetic and real-world data. All measurements are based on 20 trials unless otherwise indicated. A plotted data point represents the mean measurement value from these trials, and the corresponding error bar spans one standard deviation in each direction. To reduce clutter we omit the display of error bars for every second point in a data series. FasTer and FasTer$_{\pm\text{PSC}}$ use $\mathsf{P} = 20$ randomization iterations in all cases (based on the observations in Section 5.7).

5.1. Synthetic Data Generation

For experiments on synthetic data, a new data set is generated for each of the 20 trials. Unless otherwise stated, our generation approach is analogous to (Miettinen, 2009). It involves producing "ground truth" basis $\tilde{B} \in \mathbb{T}^{k \times m}$ and usage $\tilde{S} \in \mathbb{B}^{n \times k}$ matrices. The "noiseless" data matrix $\tilde{C}$ is obtained with $\tilde{C} = \tilde{S} \vartriangle \tilde{B}$, after which a specified amount of uniform noise is simulated by randomly modifying values in $\tilde{C}$ (note that no values of $\mathfrak{u}$ are created in this step for the binary-only experiments). The resulting noisy matrix C is then the data matrix for each algorithm. The parameters of interest in generating the synthetic data sets are 1) the dimensions n, m and k, 2) the percentage η of noise, 3) the average count λ of $\mathfrak{t}$ values per row of $\tilde{S}$ and 4) the average density $\rho_{\mathfrak{t}}$ and $\rho_{\mathfrak{u}}$ of $\mathfrak{t}$ and $\mathfrak{u}$ values in a row of $\tilde{B}$ respectively. We offer the C++ tools used to generate C matrices based on these parameters[4].

All experiments involve systematically varying one parameter over a range, with non-varying parameters being assigned a default value. Ranges and default values are given in Table 3. For consistency we choose the defaults and ranges identically[7] to those used by Miettinen in the BMF experiments (Miettinen, 2009). The defaults are sensible: the combination of $k = 16$, $\lambda = 4$ and $\rho_t = 10\%$ strikes a balance between tractability (a digestible *count* of base concepts), variation (a non-trivial number of base concepts *combined* to generate observations), and density (a binary matrix with these defaults is typically 57% dense). The default noise value of $\eta = 10\%$ is large enough to be realistic whilst not disadvantaging comparison methods like GreEss which, as a result of its strict coverage requirements and "from-below approximation" approach, tend to be less robust against noise (as we will see in Figure 4).

The extreme values of the λ, ρ_t and ρ_u ranges correspond to reasonable sparsity/saturation limits for the corresponding data matrix (for the binary case, a value of $\rho_{\mathrm{t}} = 30\%$ yields a matrix with an average density of 65%, for example). Heavy distortion of the binary signal is simulated with the maximum noise η value of 40%. Noteworthy is Miettinen's choice of 28 for k's upper-bound: it corresponds to a data set with so great a complexity that it hedges the chances for understanding the "big picture" of its model (irrespective of how accurately it describes the data). In line with our emphasis on *interpretability*, we therefore follow Miettinen and focus on tractable ("small") values of k.

n	m	k	η	λ	ρ_{t}	ρ_{u}
150	80	16 $[8, 28]$	10% $[0, 40]$	4 $[2, 6]$	10% $[5, 30]$	10% $[5, 30]$

Table 3: *Default parameter values (and ranges $[a, b]$) for synthetic data generation.*

5.2. Comparison Methods

We compare to state-of-the-art algorithms for each problem type: TMF, Missing-Value BMF (MVBMF) and BMF. For TMF and BMF problems, FasTer is compared to Asso (Miettinen et al, 2008), PaNDa (Lucchese et al, 2010)[8] and GreEss (Belohlavek, 2013)[9]. For MVBMF, FasTer is compared to Asso$^{\mathrm{MV}}$ (Yadava and Miettinen, 2012)[10] and MMMF (Srebro et al, 2004)[11]. We compare FasTer$_{\pm\mathrm{PSC}}$ directly to FasTer for TMF and BMF problems in order to investigate the behavior of the two heuristics discussed in Section 4.2.

Asso and Asso$^{\mathrm{MV}}$ require an input parameter $0 \leq \tau \leq 1$, used as a threshold for rounding purposes. Optimal selection of this parameter is difficult (Miettinen and Vreeken, 2014), so we report the best result from $\tau = 0, 0.1, \ldots, 1$. For PaNDa the author's recommendations (Lucchese et al., 2010) were followed (20 randomization rounds, frequency based sorting, prefix-tree data structure for transaction storage, default row- and column-tolerance ratios), with the top k detected patterns taken to form the basis matrix.

For TMF and BMF problems we include results from modified versions of Singular Value Decomposition (SVD) and Non-negative Matrix Factorization (NMF). Here the classical methods are used in a way similar to that described in (Miettinen, 2009). Firstly, the data matrix $C \in \mathbb{T}^{n \times m}$ has its entries mapped

[7] Aside from ρ_{u} which is not applicable in BMF.
[8] The publicly-available implementation was used.
[9] We thank Radim Belohlavek for sharing an up-to-date implementation.
[10] We thank Pauli Miettinen for sharing an up-to-date implementation.
[11] We use the author's original implementation with YALMIP and SDPT3 as the SPD solver.

to real values using a mapping $m_r : \mathbb{T} \mapsto \mathbb{R}$. Given this input, the algorithm produces real-valued factors which are subsequently multiplied using the classical matrix product to generate a reconstructed data matrix $\hat{C} \in \mathbb{R}^{n \times m}$. $\hat{C}$ is then transformed back to the original categorical space in an entry-wise fashion using a mapping $m_t : \mathbb{R} \mapsto \mathbb{T}$ and compared with C (using $\oslash_c$) to calculate the error as normal. For BMF experiments with $C \in \mathbb{B}^{n \times m}$, the mappings are clear and the corresponding methods are labeled SVD^{01}and NMF^{01}. The mappings are less obvious for TMF experiments with $C \in \mathbb{T}^{n \times m}$ – here we compare two different mapping types and denote the corresponding methods with SVD^{012}, NMF^{012}, SVD^{021}and NMF^{021}. The mappings are summarized in Table 4. We stress that we provide SVD and NMF results for *reference only* – it is clear from our Introduction that their factors cannot be reliably interpreted in a logical context.

	$m_r(a)$ with $a \in \mathbb{T}$	$m_t(a)$ with $a \in \mathbb{R}$
SVD^{01}/ NMF^{01}	1 if $a = \mathfrak{t}$, else 0	$\mathfrak{f}$ if $a < 0.5$, else $\mathfrak{t}$
SVD^{012}/ NMF^{012}	2 if $a = \mathfrak{t}$, else 1 if $a = \mathfrak{u}$, else 0	$\mathfrak{t}$ if $a \geq 1.5$, else $\mathfrak{f}$ if $a < 0.5$, else $\mathfrak{u}$
SVD^{021}/ NMF^{021}	2 if $a = \mathfrak{u}$, else 1 if $a = \mathfrak{t}$, else 0	$\mathfrak{u}$ if $a \geq 1.5$, else $\mathfrak{f}$ if $a < 0.5$, else $\mathfrak{t}$

Table 4: *Mappings for modified* SVD *and* NMF.

5.3. Ternary Matrix Factorization

Figure 4 shows the results of deploying FasTer to synthetic TMF problems. For Asso and PaNDa, we first transform each ternary data value into a binary triple using the mappings $\mathfrak{f} \mapsto (0, 0, 1)$, $\mathfrak{u} \mapsto (0, 1, 0)$ and $\mathfrak{t} \mapsto (1, 0, 0)$, thereby tripling the matrix dimension m. After running each algorithm, the binary triples from the reconstructed data matrix are transformed back to ternary ($\mathfrak{t}$ if the first triple entry is 1, else $\mathfrak{u}$ if the second triple entry is 1, otherwise $\mathfrak{f}$) for calculating the error. For the GreEss algorithm we use the scale $L = \{0, 0.5, 1\}$ in accordance with the notation in (Belohlavek, 2013). The run-time complexity of the GreEss algorithm was a prohibiting factor in our experiments, hence we only perform a comparison on smaller data sets ($n = m = 50$). We also note that the effectiveness of PaNDa is typically measured using a different objective function which favors *succinct* patterns to avoid overfitting, however we argue that the current comparison is warranted because the model-order k is always known (overfitting is hence not a factor).

FasTer consistently outperforms Asso, PaNDa and GreEss. The GreEss algorithm is competitive for the zero noise case, however quickly falls away when noise is added (a manifestation of its strict-coverage requirements and "from-below approximation" (Belohlavek, 2013)). With respect to Asso and PaNDa, FasTer's superior performance is partially explained in Section 4.2. That is, there is no binary encoding scheme we can choose that permits the optimization goals of Asso and PaNDa to be equivalent to the $\oslash_c$-based TMF objective function.

Of particular mention in Figure 4 is that there are many cases where FasTer even outperforms SVD and NMF (real-valued methods that are not suitable for interpretation in a logical context). Recalling the first example from our Introduction, we suggest that FasTer's success here lies in its exploitation of

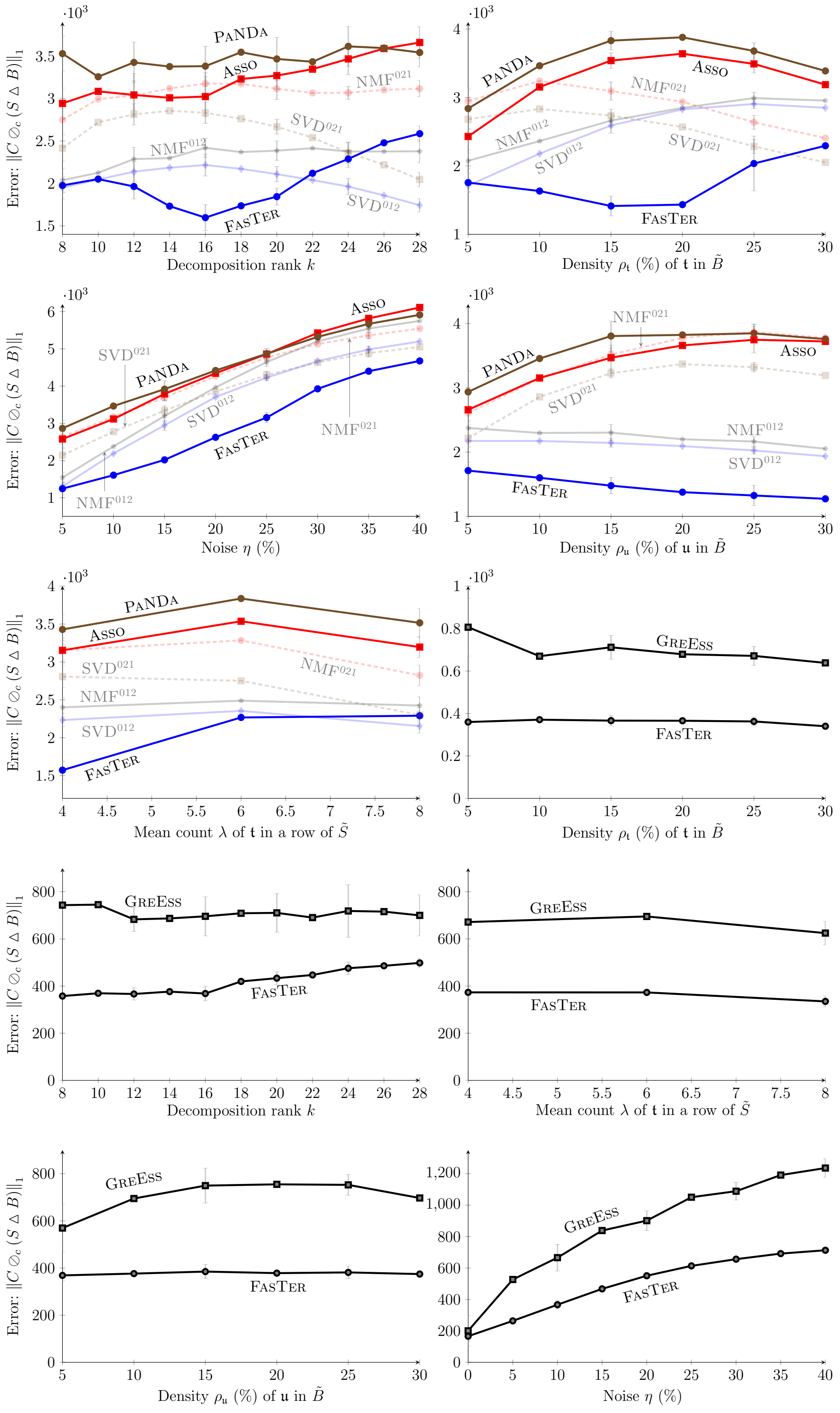

Fig. 4: *Ternary Matrix Factorization on synthetic data with varying data-generation parameters k, ρ_t, η, ρ_u and λ.* GreEss *comparisons use $n = m = 50$.*

the nonlinear ternary connectives in place of real-valued connectives in the matrix product, enabling it to describe ternary data sets more concisely.

5.3.1. Real-world: Congressional Voting Records

We now demonstrate FasTer on three open, real-world ternary data sets. The first target (from the Introduction) is the **Congressional Voting Records** data set from the UCI Machine Learning Repository. Here the $n = 435$ observations of $m = 16$ attributes record the opinions of congressmen on key voting issues. Values of $\mathfrak{t}$ and $\mathfrak{f}$ indicate that the congressman voted "yea" and "nay" on the issue respectively. A value of $\mathfrak{u}$ implies that the position of the congressman was *Don't know* – it does *not* imply that the data value is missing. FasTer's usage matrix (not shown) correctly assigns 344 (79%) of the congressmen to their party, and its basis vectors ($k = 2$) represent the "party base opinions" which intuitively disagree on most issues:

Issue:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
GOP	Nay	Yea	Nay	Yea	Yea	Yea	Nay	Nay	Nay	Nay	Nay	Yea	Yea	Yea	Nay	Yea
D	Yea	Nay	Yea	Nay	Nay	Nay	Yea	Yea	Yea	Yea	Nay	Nay	Nay	Nay	Yea	Yea

Interestingly, the basis vectors are both in $\mathbb{B}$, indicating that the congressmen are generally steadfast. This is confirmed in the original data by the relatively low occurrence of $\mathfrak{u}$ values. As discussed earlier however, the $\mathfrak{u}$ values (*indecision*) are our focus, so we vary $w_{\oslash} = (\mathfrak{u} \oslash \mathfrak{t}) = (\mathfrak{u} \oslash \mathfrak{f})$ to penalize incorrect coverage of $\mathfrak{u}$ values (i.e. emphasize their importance). We see below that it is then the Democratic basis vector for which uncertainty is exposed, first for an Export Administration Amendment Act relating to trade with South Africa (Issue 16), and then for Issue 2 which revolves around the level of federal cost-sharing for water projects. This result cannot be reproduced with BMF techniques – the ability to weigh the importance of certain ternary values is unique to TMF.

D	Yea	Nay	Yea	Nay	Nay	Nay	Yea	Yea	Yea	Yea	Nay	Nay	Nay	Nay	Yea	???	$w_{\oslash} = 3$
D	Yea	???	Yea	Nay	Nay	Nay	Yea	Yea	Yea	Yea	Nay	Nay	Nay	Nay	Yea	???	$w_{\oslash} = 5$

5.3.2. Real-world: Stack Overflow

Considering our focus on *interpretation at scale*, we proceed to analyze FasTer's output for a *larger* data set. We sourced the **Stack Overflow** (SO) data ($n =$ 1.02 million) from a public API[12] and, having generously been granted permission to do so, provide it (alongside a description) for reuse[4]. SO is a popular programming question-and-answer site. An observation in this data set consists of $m = 12$ basic attributes of an *answer* and its corresponding *question* (e.g. measures on structure, formatting and "usefulness"). The natural model order here is apriori unclear, however our analysis of the error curve (omitted for brevity) with varying k suggests (through a clear "kink") the choice of $k = 6$. This rank corresponds to a richer model than that for the Congressional Voting Records, yet still permits the digestible presentation of FasTer's results below. We note

[12]api.stackexchange.com

that the sequential execution time was less than two minutes for $\mathsf{P} = 20$ (machine/implementation details given in Section 5.8), and that the default $\oslash_c$ was used.

	Answer metadata						Corresponding *Question* metadata					
	💡	🔗	</>	≣	🗨	✅	💡	🔗	</>	≣	🗨	✅
1.	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗	✓	✓
2.	✓	✗	✓	✗	✓	✗	✗	✗	✓	✓	✓	✓
3.	?	✗	✓	✓	✗	✗	✗	✗	✓	✓	✓	✓
4.	?	✗	✗	✗	✓	✗	?	✗	✗	✓	✗	✓
5.	✓	✓	✗	✓	✗	✗	?	✗	✗	✓	✗	✓
6.	?	✓	✗	✓	✗	✓	?	✗	✓	✓	✗	✓

We offer brief comments on the results. In a general sense, it is clear that *all* questions 1) are answered to the asker's satisfaction (✅ = ✓) and 2) refrain from using references (🔗 = ✗). This second point is interesting – it echoes SO's concern that questions are often submitted despite inadequate research[13]. Questions judged *valuable* by the community (💡 = ✓) are often short (no more than one text paragraph, ≣ = ✗) and do not include code samples (</> = ✗). A suggestion for a user wanting to ask a highly-useful question might then be: keep the question *succinct* and *general*[14]. The first basis vector also indicates that *answers* to such valuable questions 1) *do* include a code sample and 2) are themselves valuable.

Questions deemed *non-valuable* (💡 = ✗) have typically been discussed (🗨 = ✓). We might interpret this to mean that the community has tried to clarify the question or attempted to highlight its shortcomings through comments. Despite this, valuable answers can still be found (perhaps those that use whatever useful information the question contains to identify and address the latent question) that are also discussed.

Questions for which the usefulness is not known (💡 = ?) are seldom discussed (🗨 = ✗). Perhaps the question is too esoteric for the community to offer an opinion. Along this line of thought, we see that answers to such questions often reference an external source (🔗 = ✓) rather than providing a code sample (</> = ✗). Noteworthy for such a question is that there are many cases where it is also unknown if its *accepted* answer is valuable (✅ = ?), echoing SO's sentiment that "if you ask a vague question, you'll get a vague answer".

The corresponding accuracy results for the Stack Overflow and Congressional Voting data sets are given in Table 5. In this table we also report accuracy results for a third, questionnaire data set (**Africa Religion**). This $n = 22601$ data set was sourced from the Pew Research Center's survey on "Tolerance and Tension: Islam and Christianity in Sub-Saharan Africa" (instructions for obtaining the data are in the supplementary material[4]). Here a survey participant's "yes", "no" and "don't know" responses form an $m = 44$ ternary observation. The choice of $k = 2$ is natural given the survey's clear focus on comparing the opinions of Muslims and Christians (Pew Research Center, 2010).

[13] "Search, and research" is the first point of advice on SO's *How to Ask* page.

[14] One of the highest-rated questions on the site, for example, is simply "What is a plain English explanation of Big-O?"

	$\|C \oslash_c (S \vartriangle B)\|_1$: Mean (± SD)		
	Asso	PaNDa	FasTer (using $\oslash_c$)
Congr. Voting	1.731 (±0) K	1.732 (±0.019) K	**1.638** (±0) K
Stack Overflow	3.093 (±0) M	3.692 (±0.107) M	**2.124** (±0.065) M
Africa Religion	0.294 (±0) M	0.295 (±0.002) M	**0.280** (±0.001) M

Table 5: *Accuracy results for ternary real-world data sets (k chosen as discussed). K and M are shorthand for 10^3 and 10^6.*

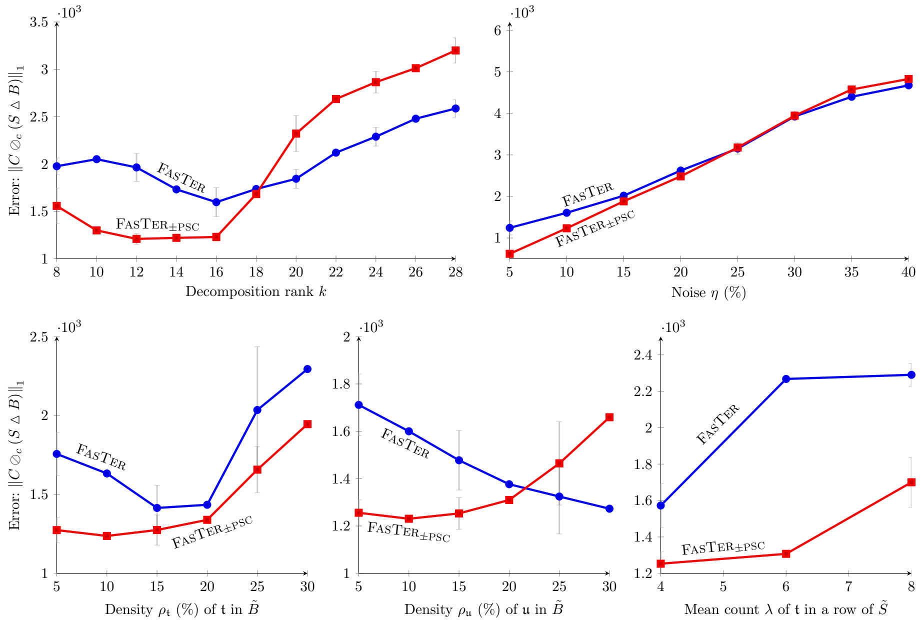

Fig. 5: *Ternary Matrix Factorization for* FasTer$_{\pm\text{PSC}}$ *on synthetic data with varying data-generation parameters k, η, ρ_t, ρ_u and λ.*

5.4. Using the ±PSC solver in FasTer

Figure 5 compares FasTer$_{\pm\text{PSC}}$ directly with FasTer on the TMF synthetic data. As discussed in Section 4.2, FasTer$_{\pm\text{PSC}}$ employs the approximation algorithm for Positive-Negative Partial Set Covering in order to solve the TUP, whereas FasTer uses the scalable greedy heuristic.

The plots show that FasTer$_{\pm\text{PSC}}$ can give very competitive results. For example, the results are close to the optimal solution (which has an average error of 1200 in the case of 10% noise) when k has a small-to-moderate value. For a number of parameter combinations, however, FasTer$_{\pm\text{PSC}}$ delivers poorer results than FasTer. Section 4.2 helps to explain why: although strictly not applicable in the ternary case, the approximation factor for the ±PSC algorithm grows with k as well as the density of the matrix C. We see that the plots reflect this: the results become weaker for increasing k and increasing ρ_u, ρ_t and λ (which increase the density of C), yet remain stable in comparison to FasTer for increasing noise (which has no effect on the approximation factor).

The results for the binary case are given in Figure 6. In this case, the approximation factor is strictly applicable, and we again see the results weakening for increasing k, $\rho_{\mathfrak{t}}$ and λ.

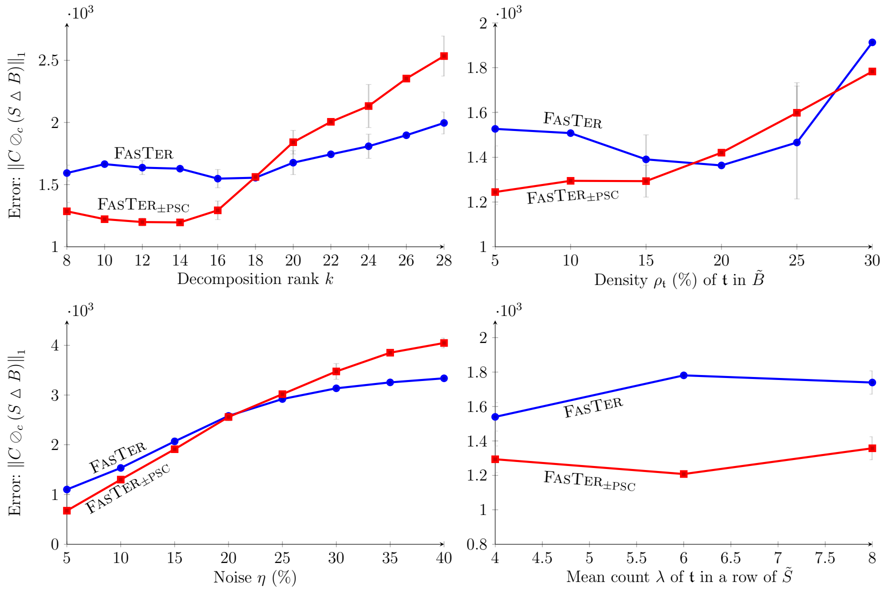

Fig. 6: *Binary Matrix Factorization for* FasTer$_{\pm\text{PSC}}$ *on synthetic data with varying data-generation parameters k, $\rho_{\mathfrak{t}}$, η and λ.*

5.5. Missing-Value BMF

Figure 7 shows the results of deploying FasTer to synthetic MVBMF problems. Our focus here is the MCAR case (missing completely at random, i.e. the fact that a data value is missing is independent of the observation's values) (Vreeken and Siebes, 2008). The parameter $\rho_{\mathfrak{u}}$ was set to zero in the data-generation process. Missing values $\mathfrak{u}$ were injected into the noisy matrix $C \in \mathbb{B}^{n\times m}$ at a density of ρ_m (default value 50%), thereby transforming C into a ternary matrix. FasTer uses the missing-value version $\oslash_{mv}$ of the dissimilarity operator.

The FasTer and Asso$^{\text{MV}}$ curves in Figure 7 are sometimes visually close. As an additional statistical quantifier for FasTer's performance, we hence provide p-values above each plot. These p-values correspond to the result of a Wilcoxon signed-rank test[15] applied to 20 pairs of FasTer and Asso$^{\text{MV}}$ results for each parameter value. Almost all results are significant with respect to the typical 0.05 threshold.

Like SVD and NMF, the real-valued technique MMMF has more freedom in selecting its factor matrices and is able to impute values more effectively. Its factors, however, are equally misleading on interpretation. FasTer, which *does* produce interpretable factors, clearly outperforms Asso$^{\text{MV}}$ and surprisingly

[15]A non-parametric test that does not assume the population to be normally distributed

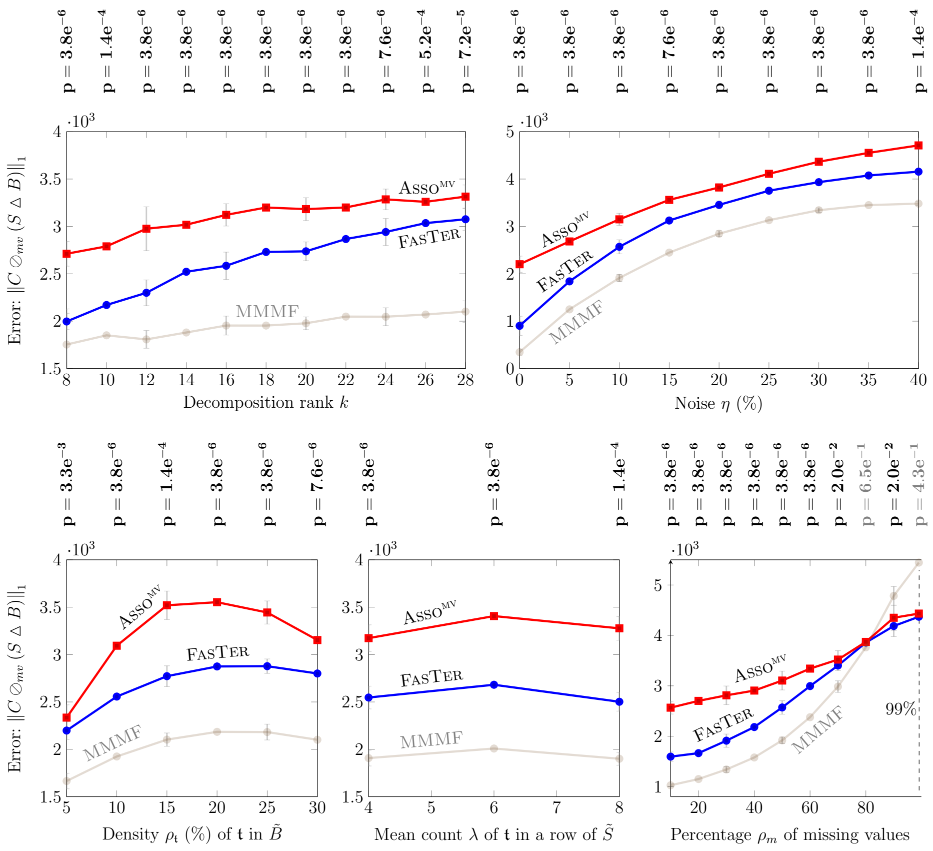

Fig. 7: *Experimental results for synthetic Missing-Value Binary Matrix Factorization (MVBMF) with varying k, η, $\rho_{\mathfrak{t}}$, λ and ρ_m. Note that MMMF produces real-valued factors that do not lend themselves well to interpretation in a logical context. Wilcoxon sign-rank test p-values corresponding to the comparison between* ASSO$^{\text{MV}}$ *and* FASTER *are shown above the respective plots (e.g.* $\mathbf{p = 3.8e^{-6}}$*) for each point. Non-significant results with respect to a threshold of 0.05 are shown in a lighter gray.*

outperforms MMMF for cases in which the data set is dominated by missing values.

Table 6 shows MVBMF results for ten UCI data sets. All are categorical and have a known model-order (used for the k parameter). Attributes with missing values were removed during pre-processing to ensure completeness. For each trial, information-loss was then simulated by randomly replacing categorical values with missing values ($\rho_m = 0, 10, \ldots, 90, 99$) before encoding in ternary form. As an encoding example, consider the Hayes-Roth observation vector $(3, 4, 4, 4)$ which loses information to become $(3, ?, 4, 4)$. The corresponding ternary encoded form is $(\mathfrak{f}, \mathfrak{f}, \mathfrak{t}, \mathfrak{u}, \mathfrak{u}, \mathfrak{u}, \mathfrak{u}, \mathfrak{f}, \mathfrak{f}, \mathfrak{f}, \mathfrak{t}, \mathfrak{f}, \mathfrak{f}, \mathfrak{f}, \mathfrak{t})$. The given dimension m is the number of attributes *after* encoding. The error reported is the mean and standard deviation over 20 trials, each measuring for $\rho_m = 0, 10, \ldots, 90, 99$ the *average* reconstruction error against the encoded form of the complete data. Figure 8 visualizes the expanded results for two of the data sets (the remaining plots are

			Error (described in caption): Mean (± SD)	
Data set	n	m	Asso$^{\text{MV}}$	FasTer
Breast Cancer	277	51	1.965 (±0.020) K	**1.915** (±0.011) K
Congr. Voting	435	48	4.573 (±0.087) K	**4.003** (±0.036) K
Hayes-Roth	160	15	592 (±11)	**544** (±5)
Lenses	24	9	77 (±1)	**70** (±1)
Lymphography	148	59	1.879 (±0.020) K	**1.739** (±0.012) K
Promoter	106	228	6.080 (±0.032) K	**6.012** (±0.018) K
Soybean	47	102	757 (±18)	**666** (±13)
SPECT Heart	267	44	1.202 (±0.014) K	**1.089** (±0.009) K
Tic-tac-toe	958	27	8.209 (±0.034) K	**8.170** (±0.017) K
Trains	10	71	146 (±8)	**139** (±2)

Table 6: *MVBMF for ten UCI data sets (mean and standard deviation over 20 trials, each measuring the average error over $\rho_m = 0, 10, \ldots, 90, 99$). In each case k was chosen as the known number of classes in the data set. K is shorthand for 10^3.*

given as a supplement[4]). In general, whilst Asso$^{\text{MV}}$ is competitive and sometimes marginally more effective, FasTer shows convincing results over the full ρ_m range at a lower run-time complexity.

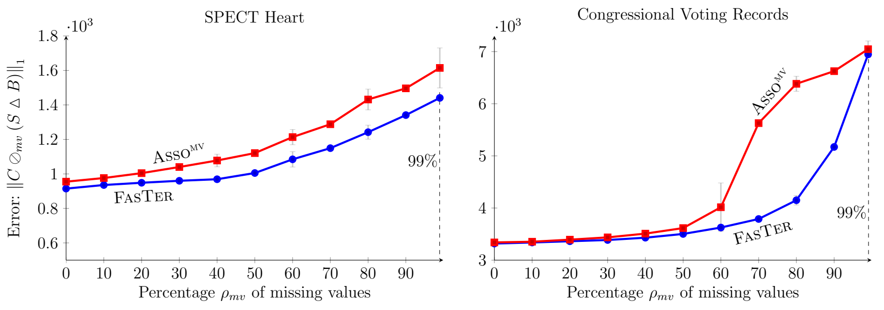

Fig. 8: *Error from Missing-Value Binary Matrix Factorization (MVBMF) on the SPECT Heart (left) and Congressional Voting Records (right) data (complement to the summary in Table 6).*

5.6. Binary Matrix Factorization

Although not the core focus of this work, Figure 9 shows the results of deploying FasTer to synthetic BMF problems. The parameter ρ_u was set to zero in the data-generation process to ensure that $C \in \mathbb{B}^{n \times m}$.

The results for the real-valued methods are no surprise (Miettinen, 2009). They have greater freedom in the selection of their factor matrices and generate optimal rank-k decompositions to explain the data. With reference to our Introduction, however, we again stress that these results can be misleading if interpreted. Of the interpretable methods, FasTer produces generally-superior results (we confess that Asso is the method of choice for BMF if the data rank is very high, however in such cases we again argue that its model's interpretation would be a demanding task). We note in particular that FasTer is density-robust: the reconstruction quality is more stable for increasing ρ_t.

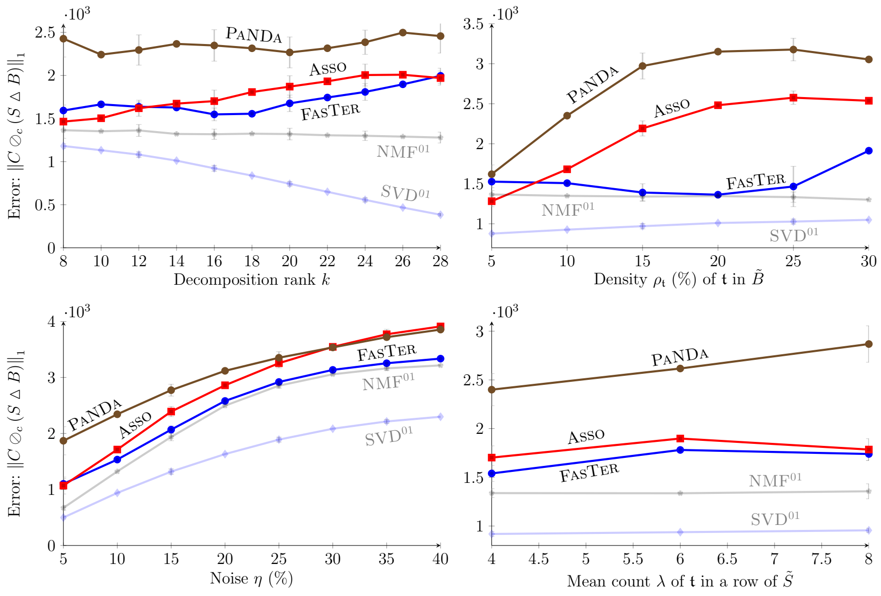

Fig. 9: *Binary Matrix Factorization on synthetic data with varying data-generation parameters k, ρ_t, η and λ. Note that 1) SVD and NMF produce factors that are not reliably interpretable, and 2) the density of C is already approximately 65% when $\rho_t = 30\%$.*

5.7. Run-time Complexity

Our experiments confirm the run-time growth of FASTER to be linear in n and m and quadratic in k. Figure 10 shows the significant differences in run-time for varying m. Here FASTER's linear growth is superior to that of ASSO (quadratic) and PANDA (tending quadratic). We note that each algorithm was run independently on a single processing unit of the hardware described in Section 5.8. C/C++ implementations were used.

Figure 10 also shows that the number of internal iterations N required for FASTER's convergence does not increase with n, m or k. Here N is recorded as the number of iterations required by the best approximation from $\mathsf{P} = 20$ randomization rounds.

Finally, Figure 10 shows that only a small number of randomization rounds P are required for achieving competitive results, even with large n, m and k. Based on these observations we selected $\mathsf{P} = 20$ for all other experiments. The figure also shows that FASTER solutions remain accurate when the data dimensionality is high (e.g. $m = 25000$). Solutions for a high observation-count (e.g. $n = 50000$) are comparatively weaker (likely caused by the stochastic initialization strategy).

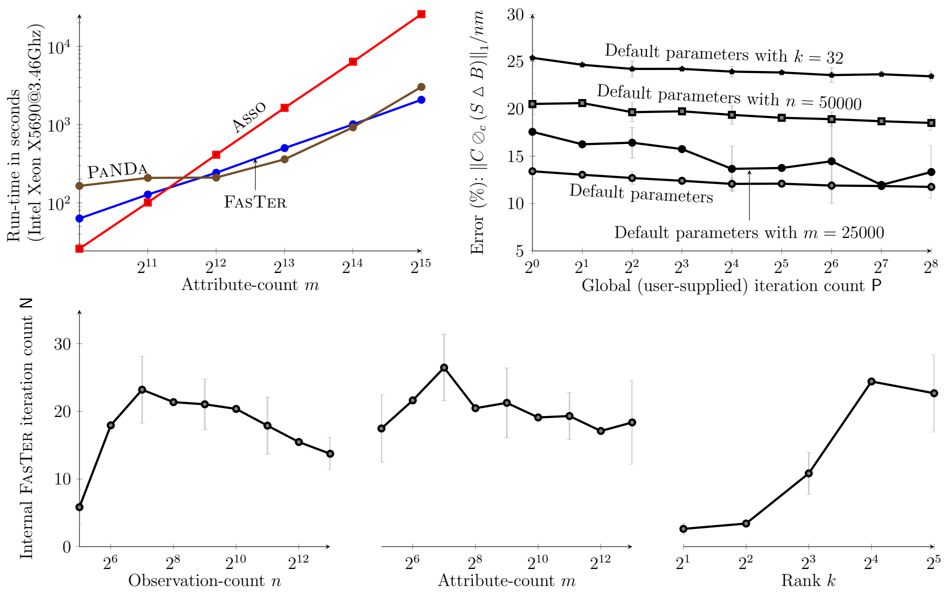

Fig. 10: ***(Top Left)*** *Run-time comparison for varying m (*FasTer *measurements are for* P $= 20$ *global iterations).* ***(Top Right)*** *Affect on the error for varying global-iteration count* P. ***(Below)*** *Count* N *of internal* FasTer *convergence iterations for varying n, m and k.*

5.8. Parallelization

Considering the discussion from Section 4.3, we parallelized FasTer using the C++ directives and functions of OpenMP (version 2.5). Work-sharing constructs were used to divide the TUP and TBP among multiple threads. The effect of the three OpenMP scheduling strategies *static*, *dynamic* and *guided* were explored for this division of work. All experiments in this section were performed on an IBM x3650 M3 with two Intel Xeon X5690 6-core processors (3.46 GHz) and hyper-threading enabled, giving 24 virtual cores in total.

Figure 11 shows the speedup and efficiency curves for up to 24 processing units in a strong-scaling setting. In this setting, we are interested in how the solution time varies with the number of processors for a *fixed* TMF problem size. For the fixed problem we used a single synthetic data set with the default parameters from Table 3 and dimensions $n = 15000$ and $m = 8000$ ($n \cdot m = 120$ million).

The speedup results are near-ideal for up to eight processors. Afterwards, the overhead incurred from OpenMP's management of the threads is more noticeable. The speedup factor reaches a maximum of near 13 using 22 processors and dynamic scheduling. Overall, static and dynamic scheduling have very similar performance. Guided scheduling almost always performs slightly worse. On this large data set it allocates chunks of exponentially-decreasing size, resulting in a higher overhead compared to normal dynamic scheduling which allocates fewer constant-sized chunks. The results show that the variation in the amount of work done by each TUP and TBP loop iteration is not enough to warrant the choice of a guided strategy for such a large data set.

We witnessed super-linear speedup for static scheduling with two and four processors. Super-linear speedup is sometimes seen in parallel computing, and in this case is not surprising given the nature of OpenMP's *static* scheduling strategy in the context of FasTer. When using this strategy with a fixed number of processors, OpenMP guarantees that each processing unit will receive identical iteration ranges for the TUP and TBP loops. For example, if $p = 4$ processing units are available to work on a data set with $n = 100$ rows, *static* scheduling would always allocate the first 25 TUP-loop iterations to the first processing unit, the second 25 to the second processing unit, and so on. This implies that each processor only ever needs to operate on a subset of the input data, and can hence exploit lower-level CPU caches (such as the L1 cache) to a higher extent during FasTer's N convergence iterations than if a single processor were used.

Finally, we remark that we have presented these results in the form of speedup and efficiency metrics (common in high-performance computing literature). The wall-clock execution-time metric is however also valuable to get a practical feel for FasTer's performance on this large data set (120 million entries). We therefore note that the maximum wall-clock execution time was 66507 seconds (18 hours) for $p = 1$ processor, and the minimum execution time was 5243 seconds (1.5 hours) using $p = 22$ processors and dynamic scheduling. For modern-day laptops and desktops having $p = 4$ processing units, our measured static-scheduling time of 17223 seconds (4.8 hours) can be taken as an estimate for the expected runtime.

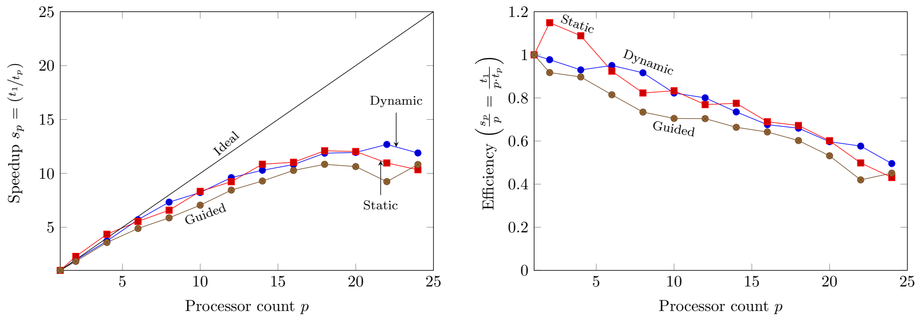

Fig. 11: *Speedup (**left**) and efficiency (**right**) for varying* OpenMP *scheduling strategies in a strong-scaling TMF scenario (where the solution time varies with the number of processors for a fixed, large TMF problem).* t_1 *and* t_p *are the wall-clock solution times for* 1 *and* p *processors respectively.*

Figure 12 presents results for a weak-scaling scenario. In this scenario, the problem size (here the number of data matrix rows n) is varied proportionally to the number of processors. We are interested in how the solution time varies with the number of processors for a fixed problem size *per processing unit.* The data sets used for these experiments were again generated using default parameters, except that m was fixed to 800 and n was varied as shown. 20 independent trials were performed in each case.

Guided and dynamic scheduling strategies have approximately equal performance. As expected from the discussion in Section 4.3, static scheduling is noticeably weaker and less stable (higher result variability) on these smaller data sets. For $p = 20$ processing units, for example, static scheduling sometimes yields comparable results to the dynamic strategies, and sometimes notably worse results. This highlights the *non-adaptive* nature of the static scheduling strategy

– a weakness in this case as it is not able to respond to variations in the work required to solve the TUP and TBP across processing units.

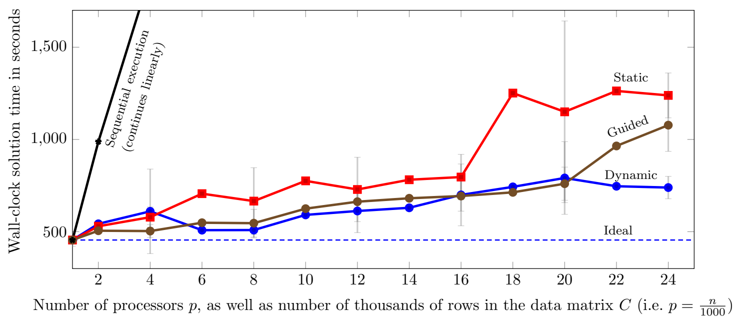

Fig. 12: *Variation in wall-clock solution time by* OpenMP *scheduling strategy as the number of processors increases proportionally to the data-matrix dimension* n*. The constant horizontal line represents ideal weak-scaling. The steeply-increasing series on the left represents a part of the line for the corresponding sequential solution times (which behaves linearly).*

At a high level, the strong- and weak-scaling results presented in this section show that it is useful and efficient to parallelize FasTer in the ways discussed. Additional "fine-tuning" may yield better performance, and extensions could be investigated such as the parallelization of 1) input file parsing, and 2) initial basis matrix B_0 selection.

6. Related Work and Discussion

Too numerous to mention are the techniques for factorizing complete real-valued matrices (see (Miettinen, 2009) for an overview). Although such methods (SVD, NMF) accurately reconstruct the data matrix, we have shown that they fail to reliably identify meaningful (interpretable) structure in categorical data sets.

The work done by Miettinen on Binary Matrix Factorization (BMF) (Miettinen et al, 2008), along with subsequent investigations (e.g. (Belohlavek and Vychodil, 2010; Lucchese et al., 2010; Lu et al, 2008)), has received considerable attention in the community. BMF approximates a data matrix $C \in \mathbb{B}^{n \times m}$ through the binary matrix product of a usage matrix $S \in \mathbb{B}^{n \times k}$ and basis matrix $B \in \mathbb{B}^{k \times m}$. These matrices *are* interpretable in the data domain. TMF differentiates itself by considering the more general problem of ternary data, however the techniques intersect in the sense that BMF is encapsulated by TMF. Our experiments have shown that FasTer outperforms both Asso and PaNDa (efficiency *and* effectiveness) in a BMF context.

The Positive-Negative Partial Set Cover problem (Miettinen, 2008*b*) is a generalization of the classical set cover problem in combinatorics. It has been studied in the context of BMF and exploited in a variation thereof (Miettinen, 2008*a*). In this paper we have shown that its use for TMF and BMF in FasTer can yield very competitive results, despite the fact that ±PSC's approximation factor is not always strictly applicable in a mathematical sense. The trade off is speed – FasTer loses its linear run-time complexity if we use ±PSC in place of the normal heuristic.

Methods for factorizing *incomplete* binary data sets have been developed. Asso$^{\text{MV}}$ is presented in (Yadava and Miettinen, 2012) and focuses particularly on highly-incomplete (e.g. 99% missing values) data sets. Our experiments show that FasTer produces generally-superior results to Asso$^{\text{MV}}$. Asso$^{\text{MV}}$ also has quadratic run-time complexity in m (because it is based on Asso) and hence is not as efficient as FasTer (linear run-time complexity).

A related imputation method by Vreeken (Vreeken and Siebes, 2008) exploits information theory. The "simple completion" approach produces promising results for data matrices with up to 24% missing values (Vreeken and Siebes, 2008; Yadava and Miettinen, 2012). The method is based on the KRIMP algorithm which, designed primarily for transactional databases, 1) produces a large set of (typically small) itemsets rather than a succinct set of broad basis vectors, and 2) requires filtering a set of pre-mined candidates which is exponentially costly with increasing m and controlled by sensitive parameters (e.g. minimum itemset support) (Akoglu, Tong, Vreeken and Faloutsos, 2012).

Maximum-Margin Matrix Factorization (MMMF) (Srebro et al., 2004) involves low-norm decompositions and supports incomplete matrices. Like SVD and NMF, MMMF produces real-valued factors and thus has greater freedom for optimizing imputation. Unlike TMF, however, the MMMF factors do not lend themselves to interpretation in a logical context. Despite the advantage that MMMF has in choosing its factors, we have even shown that FasTer outperforms it for the case where missing values heavily dominate the data set ($\rho_{mv} > 80\%$).

Finally, GreEss (Belohlavek, 2013) uses many-valued logic for factorizing an *ordinal* data matrix. We show in Section 5.3 that it is possible to use GreEss for TMF. Here our results show that FasTer produces factorization results that are superior to GreEss. In addition, GreEss 1) produces a non-binary usage matrix S (more difficult to interpret), 2) has superlinear run-time complexity in the data set dimensions, 3) cannot be applied to imputation (the technique requires $(a \oslash b) = 0 \leftrightarrow a = b$), and 4) is less flexible in weighting the importance of values (due to the one-dimensional ordinal scale).

Investigations into parallelism for classical matrix factorization techniques like SVD and NMF exist in the literature (e.g. (Lin, 2007), (Luk, 1985)). Similar contributions in the field of association-rule mining have also been surveyed (Zaki, 1999). To the best of our knowledge, investigations into parallelism do not yet exist in the area of discrete, logical matrix factorizations (e.g. Binary and Ordinal Matrix Factorizations). Our shared-memory approach to parallelizing FasTer for TMF problems shows that we can efficiently solve given problems faster (strong scaling) as well as larger problems in a near-constant amount of time (weak scaling). As a reference point for our parallel results, we refer the reader to one of the most influential contributions in the area of high-performance, discrete-data mining: the parallel version of the Apriori algorithm (Agrawal and Shafer, 1996). Although a direct comparison is not our aim, we simply remark that both results share similar efficiency characteristics. For example, our speedup curves in Figure 11 show similar performance to those in Figure 5 of the Apriori paper (both have efficiency of above 90% for eight processors, for instance). Also of note is that the parallel Apriori likewise experienced superlinear speedup for four processors, perhaps hinting that this phenomenon is not too uncommon on modern hardware.

7. Conclusion

We have presented algorithms for the dimensionality reduction of discrete, logical data (categorical data in general) in the context of the new Ternary Matrix Factorization (TMF) problem. TMF is a novel, multi-purpose data-mining challenge based on three-valued logic. It provides a new perspective on "unknown" values in data sets. We have shown that TMF yields useful decompositions in a number of general scenarios. Firstly, TMF supports the (optionally-guided) search for structure in ternary data sets where "unknown" has contextual meaning, like a *don't know* answer to a questionnaire item with *yes*, *no* and *don't know* options. Secondly, TMF can perform Missing-Value Binary Matrix Factorization (MVBMF) to discover structure in incomplete categorical data sets where "unknown" is understood as a placeholder for a lost or indeterminable binary value, thereby offering an elegant imputation solution. Finally, TMF is a generalization of Binary Matrix Factorization (BMF) and can be used to find patterns in binary data sets (and hence categorical data sets through the use of simple encoding).

We have shown that FasTer outperforms state-of-the-art methods on TMF, MVBMF and BMF problems with respect to an intuitive objective function. Perhaps more importantly, the anytime FasTer algorithm scales *linearly* with the dimensions of the data set and is parameter-, noise- and density-robust. With run-time as a trade-off, it is also possible to significantly increase effectiveness if we exploit the known Positive-Negative Partial Set Cover algorithm from combinatorics.

Through parallelization, FasTer can overcome the computational and/or memory bottlenecks often encountered in practical applications. Our experiments have shown that a shared-memory parallel version of FasTer exhibits comparable efficiency characteristics to other high-performance data-mining algorithms in strong- and weak- scaling scenarios.

This paper has not addressed the issue of model-order selection (the decomposition rank k is a parameter of the algorithm), nor "online" situations involving dynamic data. These points, along with an investigation into alternate initialization strategies for FasTer, remain open topics for future work on TMF and its algorithms.

Acknowledgements. This work is supported by the Helmholtz Young Investigators Groups programme.

References

Agrawal, R. and Shafer, J. C. (1996), 'Parallel mining of association rules', *IEEE Transactions on knowledge and Data Engineering* **8**(6), 962–969.

Akoglu, L., Tong, H., Vreeken, J. and Faloutsos, C. (2012), 'Fast and reliable anomaly detection in categorical data', *ACM International Conference on Information and Knowledge Management*, pp. 415–424.

Belohlavek, R. (2013), 'Beyond Boolean matrix decompositions: Toward factor analysis and dimensionality reduction of ordinal data', *IEEE International Conference on Data Mining* pp. 961–966.

Belohlavek, R. and Vychodil, V. (2010), 'Discovery of optimal factors in binary data via a novel method of matrix decomposition', *Journal of Computer and System Sciences* **76**(1), 3–20.

Chvatal, V. (1979), 'A greedy heuristic for the set-covering problem', *Mathematics of operations research* **4**(3), 233–235.

Çivril, A. and Magdon-Ismail, M. (2009), 'On selecting a maximum volume sub-matrix of a matrix and related problems', *Theoretical Computer Science* **410**(4749), 4801–4811.

Codd, E. F. (1986), 'Missing information in relational databases', *ACM Sigmod Record* **15**(4), 53–53.

Cormode, G., Karloff, H. and Wirth, A. (2010), 'Set cover algorithms for very large datasets', *Proceedings of the 19th ACM international conference on Information and knowledge management*, pp. 479–488.

Francis, J. D. and Busch, L. (1975), 'What we know about "I don't knows"', *The Public Opinion Quarterly* **39**(2), 207–218.

Kleene, S. C. (1938), 'On notation for ordinal numbers', *The Journal of Symbolic Logic* **3**(4), 150–155.

Lin, C.-J. (2007), 'Projected gradient methods for nonnegative matrix factorization', *Neural computation* **19**(10), 2756–2779.

Lu, H., Vaidya, J. and Atluri, V. (2008), 'Optimal boolean matrix decomposition: Application to role engineering', *IEEE International Conference on Data Engineering*, pp. 297–306.

Lucchese, C., Orlando, S. and Perego, R. (2010), 'Mining top-k patterns from binary datasets', *SIAM International Conference on Data Mining* , Vol. 10, pp. 165–176.

Luk, F. T. (1985), 'A parallel method for computing the generalized singular value decomposition', *Journal of Parallel and Distributed Computing* **2**(3), 250–260.

Malinowski, G. (2007), 'Many-valued logic and its philosophy', *in* 'Handbook of the History of Logic', Vol. 8, North-Holland, pp. 13–94.

Maurus, S. and Plant, C. (2014), 'Ternary matrix factorization', *IEEE International Conference on Data Mining* (Best Paper Award).

Miettinen, P. (2008*a*), 'The Boolean column and column-row matrix decompositions', *Data Mining and Knowledge Discovery* **17**(1), 39–56.

Miettinen, P. (2008*b*), 'On the positivenegative partial set cover problem', *Information Processing Letters* **108**(4), 219–221.

Miettinen, P. (2009), 'Matrix Decomposition Methods for Data Mining', PhD thesis, University of Helsinki.

Miettinen, P., Mielikainen, T., Gionis, A., Das, G. and Mannila, H. (2008), 'The discrete basis problem', *IEEE Transactions on Knowledge and Data Engineering* **20**(10), 1348–1362.

Miettinen, P. and Vreeken, J. (2014), 'MDL4BMF: Minimum description length for boolean matrix factorization', *ACM Transactions on Knowledge Discovery from Data* **8**(4), 1–31.

OpenMP Architecture Review Board (2005), 'OpenMP application program interface'.

Peleg, D. (2007), 'Approximation algorithms for the label-CoverMAX and red-blue set cover problems', *Journal of Discrete Algorithms* **5**(1), 55–64.

Pew Research Center (2010), 'Executive summary: Tolerance and tension: Islam and christianity in sub-saharan Africa', Technical report, Pew Research Center.

Rubin, D. B., Stern, H. S. and Vehovar, V. (1995), 'Handling "Don't know" survey responses', *Journal of the American Statistical Association* **90**(431), 822–828.

Srebro, N., Rennie, J. and Jaakkola, T. S. (2004), 'Maximum-margin matrix factorization', *Advances in neural information processing systems* , pp. 1329–1336.

Vreeken, J. and Siebes, A. (2008), 'Filling in the blanks-krimp minimisation for missing data', *IEEE International Conference on Data Mining*, pp. 1067–1072.

Yadava, P. and Miettinen, P. (2012), 'BMF with missing values', Master's Thesis, University of Saarland.

Zaki, M. J. (1999), 'Parallel and distributed association mining: A survey', *IEEE Concurrency* **7**(4), 14–25.

Author Biographies

Samuel Maurus received B.Eng. and B.Sc. degrees from Swinburne University of Technology in Melbourne, Australia (2009) and a M.Sc. degree from the Technical University of Munich, Germany (2012). He is currently a Ph.D. student at the Technical University of Munich (based at Helmholtz Zentrum München) where he is working on matrix-factorization techniques for data-mining applications.

Claudia Plant received her PhD in 2007 and is currently leading the group iKDD (integrative Knowledge Discovery in Databases) at Helmholtz Zentrum München and Technische Universität Mnchen (Munich, Germany). Her research focuses on database-related data mining, especially clustering, parameter-free data mining and integrative mining of heterogeneous data. She has contributed to top-level database and data mining conferences like KDD, SIGMOD, ICDM and ICDE, as well as application-related journals such as Bioinformatics, Cerebral Cortex and Water Research.

Correspondence and offprint requests to: Samuel Maurus, Helmholtz Zentrum München, D-85764 Neuherberg, Germany. Email: samuel.maurus@helmholtz-muenchen.de