Enabling and Optimizing Non-linear Feature Interactions in Factorized Linear Algebra

Side Li
University of California, San Diego
s7li@eng.ucsd.edu

Lingjiao Chen
University of Wisconsin, Madison
lchen362@wisc.edu

Arun Kumar
University of California, San Diego
arunkk@eng.ucsd.edu

ABSTRACT

Accelerating machine learning (ML) over relational data is a key focus of the database community. While many real-world datasets are multi-table, most ML tools expect single-table inputs, forcing users to materialize joins before ML, leading to data redundancy and runtime waste. Recent works on "factorized ML" address such issues by pushing ML through joins. However, they have hitherto been restricted to ML models linear in the feature space, rendering them less effective when users construct non-linear feature interactions such as pairwise products to boost ML accuracy. In this work, we take a first step towards closing this gap by introducing a new abstraction to enable pairwise feature interactions in multi-table data and present an extensive framework of algebraic rewrite rules for factorized LA operators over feature interactions. Our rewrite rules carefully exploit the interplay of the redundancy caused by both joins and interactions. We prototype our framework in Python to build a tool we call MorpheusFI. An extensive empirical evaluation with both synthetic and real datasets shows that MorpheusFI yields up to 5x speedups over materialized execution for a popular second-order gradient method and even an order of magnitude speedups over a popular stochastic gradient method.

ACM Reference Format:
Side Li, Lingjiao Chen, and Arun Kumar. 2019. Enabling and Optimizing Non-linear Feature Interactions in Factorized Linear Algebra. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319878

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3319878

1 INTRODUCTION

Understanding and optimizing data-intensive steps in end-to-end ML workflows is a pressing problem for the data management community. While most ML tools expect single-table datasets, most real-world applications with structured data have multiple tables connected by datasets dependencies such as key-foreign key dependencies. This forces data scientists to *materialize* the join output before ML, which introduces redundancy in the data and ML computations, thus wasting memory and runtime. A recent line of work on "factorized ML" avoids such redundancy by pushing ML computations through joins, thus improving efficiency [15, 16, 21, 27]. In particular, [6] generalized this idea to show how *any* ML algorithm expressible in the formal language of *linear algebra* (LA) can be automatically "factorized" using its framework of *algebraic rewrite rules*. Such "factorized LA" rules rewrite LA operations (e.g., matrix-vector multiplication) over the join output's feature matrix into LA operations over the base tables' matrices.

Example (based on [6]). Consider an insurance data scientist using ML to predict customer churn (will a customer move to a competitor?). She joins a table `Customers` with features such as age, income, employer, etc. with another table `Employers` with features about customers' employers such as revenue, city, etc. The feature vectors in the base tables can be viewed as matrices, say, C and E, respectively. The materialized join output can also be viewed as a matrix, say, M. Factorized LA rewrites an LA operation over M, e.g., Mw (w is a parameter vector) into LA operations over C and E. Thus, she writes ML algorithms as if only one table exists, but under the covers, the tool in [6] rewrites the algorithm to operate on C and E.

While factorized LA benefits many ML algorithms, including all generalized linear models (GLMs) [4, 6], it has a key restriction: *linearity over feature vectors*. This reduces its benefits and applicability in cases with a common data preparation step in ML practice: *feature interactions* [2, 10]. For instance, given d features, quadratic (degree 2) interactions create $\binom{d}{2} + d$ extra features by computing pairwise products and individual squares. Such interactions are especially popular for boosting the accuracy of GLMs, since interactions enable such models to represent more functions

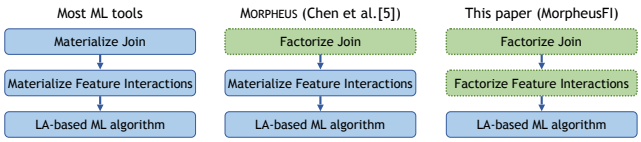

Figure 1: Conceptual comparison of common practice of materializing both joins and feature interactions over normalized data; this creates two forms of redundancy. Recent prior work (MORPHEUS [6]) avoids redundancy caused by joins but not feature interactions. Our work avoids both forms of redundancy.

of the data [2, 18, 28]. Alas, existing factorized LA frameworks force users to *materialize this non-LA step* at least in part, which introduces a new form of redundancy in the data and ML computations and wastes memory and runtimes.

In this paper, we present a novel and comprehensive framework to support and optimize *quadratic feature interactions within a factorized LA framework*. Our goal is to offer the best of both worlds: benefits of quadratic interactions and generality of factorized LA. We focus on degree 2 interactions, since they are the most common [18] (higher degrees could lead to far too many features with negligible benefits) and they let us study this novel issue in depth.

The core technical challenge is to carefully delineate the double redundancy caused by *two levels of materialization*–denormalization and feature interaction–for LA operators without losing the automation benefits of factorized LA. To tackle this challenge, we augment LA with *two non-linear interaction operators*: *self-interaction* within a matrix and *cross-interaction* between matrices participating in a join. The semantics of our operators allow them to co-exist with LA operators, since their inputs and outputs are also just matrices. Given this extended LA formalism, we devise *a novel and extensive framework of rewrite rules* to convert many LA operators over the output of quadratically interacted denormalized table's matrix to the base tables' matrices. The intuition is to *delay materialization* of interactions in the push down rewrite as far as possible. Such sophisticated rewrites are effectively *impossible* in existing pure LA frameworks.

Since our rewrite rules are more complex than regular factorized LA rewrites, we provide formal proofs of correctness for some rewrites and also quantify the runtime complexity of rewritten LA operators. We then extend our framework to star schema multi-table joins. We find that such joins lead to yet more novel interplays between joins and feature interactions, specifically, between the matrices of the dimension tables. To exploit this opportunity and reduce computational redundancy further, we extend the semantics of our new operators to handle two joins at once.

Factorized LA itself is not always faster than materialized execution–the runtime trade-offs depend on the dimensions of the matrices joined [6]. Our framework also has similar runtime trade-offs, and we perform an in-depth analysis to understand these trade-offs. We find that our framework presents *two novel differences* from the trade-off space of regular factorized LA (no feature interactions). First, *sparsity of feature vectors* plays a more outsized role in our setting in terms of determining where runtime crossovers might occur. Second, our rewrite rules also require an *ordering among dimension tables*; different orderings lead to different runtimes. We explain these issues formally by extending our runtime complexity analyses. We use these analyses to devise a simple and easy-to-check *heuristic decision rule* that helps predict when materialized execution might actually be faster and how to order dimension tables otherwise.

We prototype our framework in the popular ML framework PyTorch and build a tool we call MORPHEUSFI. We perform an extensive experimental evaluation of MORPHEUSFI with both synthetic data and 7 real-world multi-table datasets. We compare with two baselines, materialized execution and factorized LA in MorpheusPy (from the authors of [6]), for various LA operators on synthetic data. MORPHEUSFI yields speedups of up to 10x over both baselines depending on the redundancy present. We also validate that our trade-off analyses accurately predict the trends. We then compare all tools on the real datasets for logistic regression and linear SVM trained using two popular optimization procedures: LA-based LBFGS and non-LA stochastic gradient descent (SGD). On 5 datasets, MORPHEUSFI is up to 5x faster than materialized LBFGS and up to 2x faster than MorpheusPy but slightly slower on the other 2 datasets, as predicted by our heuristic decision rule. Relative to SGD, however, MORPHEUSFI is about 4x faster an average and up to 36x faster overall, while yielding similar accuracy.

Overall, this paper makes the following contributions:

- To the best of our knowledge, this is the first paper to fuse a common data preparation step for linear ML models–feature interactions–within a factorized LA framework to optimize ML over normalized data.
- We augment LA with two non-linear operators to capture quadratic interactions. We devise a novel framework of algebraic rewrite rules to avoid the double redundancy interplay between feature interactions and joins without losing the benefits of factorized LA.
- We extend our framework to support star schema multi-table joins by modifying the semantics of our non-linear operators and reduce computational redundancy further.
- We perform an in-depth analysis of the correctness and time complexity of our rewrite rules. We explain the runtime trade-offs involved and how they differ from factorized LA. We present a simple heuristic decision rule to navigate these trade-offs.

- We present a comprehensive empirical evaluation of our framework, prototyped in Python and named MorpheusFI using both real and synthetic data, comparing it against materialized execution, factorized LA, and a non-LA SGD method. MorpheusFI yields substantial speedups in most cases, including over 10x speedup over SGD in one case.

Outline. Section 2 presents the technical background. Section 3 explains our formal problem setup and basic idea. Section 4 dives into our novel framework of rewrite rules. Section 5 presents deeper analysis and extensions to our framework. Section 6 presents the experiments. We discuss other related work in Section 7 and conclude in Section 8.

2 BACKGROUND

2.1 Linear Algebra Tools

Linear algebra (LA) is an elegant formal language to capture linear transformations of matrices. An LA operator converts a matrix (or matrices) to another matrix (or matrices). ML-oriented data scientists and statisticians often specify ML algorithms as LA scripts. Essentially, the training datasets and ML model parameters are both matrices manipulated using LA operators. Common LA operators in ML algorithms include scalar-matrix multiplication, matrix aggregation, matrix-matrix multiplication, and crossproduct (aka Gramian). Popular programming environments and libraries for writing LA scripts include R [24], Python NumPy [26], Matlab, and SAS. Recent tools such as PyTorch[23] and TensorFlow [3] also support LA with NumPy and SciPy [12]. We use Python Numpy integrated with PyTorch for our prototype, but our ideas are generic and orthogonal to the specific LA tool; our ideas can be used with these other LA tools too.

2.2 Feature Interactions

One of the most popular classes of ML models are generalized linear models (GLMs) [1]. They are simple to interpret and efficient to use. However, they suffer from a key restriction: they can only learn hyperplanes over the feature space. ML theory tells us that this can potentially lead to high prediction errors because such models "underfit" the data, especially when the number of training examples is large [28]. To counteract this issue, practitioners routinely use *feature interactions* for statistical ML-based data analysis, especially for GLMs [2, 18]. Given d features, *quadratic interaction* (also called degree 2 interactions) expands the feature vector to $d' = 2d + \binom{d}{2}$ features (d original, d for squares, and $\binom{d}{2}$ for pairs of products). For example, if $d = 100$, $d' = 5,150$. Since the complexity of the feature representation increases, feature interaction often helps mitigate underfitting for GLMs. On the other hand, interactions with degrees higher than 2 are rare, since d' can quickly explode to cause "overfitting," the opposite of underfitting. For example, $d' = 3d + \binom{d}{2} + d(d-1) + \binom{d}{3}$ for degree 3 interactions, which for $d = 100$ becomes $d' = 181,800$ already! Thus, we restrict our focus to quadratic interactions in this paper.

Symbol	Meaning
Y	Target/label feature in table $\mathbf{S}$
S	"Fact" table feature matrix
n or n_S	Number of rows (examples) in S
R_i	"Dimension" table number i feature matrix
n_1	Number of rows in R_i
d_S, d_i	Number of columns in S and R_i resp.
X_i	Expanded dim. table i feature matrix (n_S rows)
K_i	Indicator matrix for joining S and R_i
T	Joined table of S and all R_i
e_S	Sparsity of S
e_i	Sparsity of R_i
$\mathring{M}$	Self-interaction of matrix M
$M_i \otimes M_j$	Cross-interaction on M_i and M_j (same # rows)
$M_i \odot M_j$	Hadamard product of M_i and M_j (same shape)

Table 1: Notation used in this paper.

3 SETUP AND PRELIMINARIES

3.1 Notation

We study the same schema setting as [6] and so, we adopt their notation. Consider a 2-table join schema: $\mathbf{S}(Y, X_s, K_1)$ and $\mathbf{R}_1(\underline{RID_1}, X_1)$. $\mathbf{S}$ is akin to the fact table in OLAP; $\mathbf{R}_1$ is akin to a dimension table. X_s and X_1 are feature vectors, Y is the prediction target, RID_1 is the primary key of $\mathbf{R}$, while K_1 is the foreign key. The projected equi-join is as follows: $\mathbf{T}(Y, [X_s, X_1]) \leftarrow \pi(\mathbf{S} \bowtie_{K_1=RID_1} \mathbf{R}_1)$, wherein $[X_s, X_1]$ is a *concatenation* of the feature vectors from the base tables. For simplicity sake and without loss of generality, assume the join is not selective, i.e., each tuple in $\mathbf{R}$ is referred to at least once in $\mathbf{S}$ and they are the only tuples referred to. So, $\mathbf{T}$ will have the same number of tuples as $\mathbf{S}$. In a general *star schema*, $\mathbf{S}$ can have more foreign keys (say, q): $K_1, K_2, \cdots, K_q$, which correspond to dimension tables $\mathbf{R}_1, \mathbf{R}_2, \cdots, \mathbf{R}_q$, respectively. We focus on star schemas for tractability sake and because they are common in practice (snowflake schemas can also be reduced partially to star schemas). The feature vectors in each table can be viewed as matrices for convenience in LA syntax. In particular, the matrix corresponding to $\mathbf{S}.X_s$ is denoted S; its shape is $n_S \times d_S$. Similarly, we can define R_i and T. Table 1 summarizes our notation.

3.2 Prior Work: Normalized Matrix

To understand the ideas proposed in our paper, we need to understand prior work on factorized LA, especially the so-called

"normalized matrix" abstraction introduced by [6]. It is a logical matrix data type to represent multi-table data using existing data types in LA tools. In our notation, the normalized matrix for the 2-table join is the triple (S, K_1, R_1). Here, K_1 is an indicator (0/1) matrix encoding the primary key-foreign key dependency. Assume the ordering of tuples in **S** and **R** are fixed and their respective rows are numbered sequentially. Then, we have $K_i[i, j] = 1$ iff the i^{th} row of **S** refers to the j^{th} row of $\mathbf{R}_1$. All other entries of K_i are 0, i.e., it is highly sparse. Now, it is clear that $T = [S, K_1R_1]$, wherein $[,]$ denotes column-wise matrix stitching, since the multiplication K_1R_1 replicates tuples of **R** as per the join/denormalization semantics. Overall, the normalized matrix abstraction is an elegant way of representing the join in LA syntax. It has a generalization to star schemas as well [6].

Factorized LA is a framework of *algebraic rewrite rules* that use this abstraction to "push down" LA operations through joins by rewriting an LA operation over T to LA operations over S, K_1, and R_1. As an illustration, consider the LA operation left-matrix multiplication (LMM), which is common in LA-based ML: Tw. Here, T has the shape $n \times d$ ($d = d_S + d_1$), while w is the model parameter vector, e.g., the weights of logistic regression, with the shape $d \times 1$. The rewrite rule using the normalized matrix (S, K_1, R_1) is as follows:

$$Tw \rightarrow Sw_S + K_1(R_1w_1)$$

In the above, $w \triangleq [w_S, w_1]$ splits w s.t. w_S is of shape $d_S \times 1$, while w_1 is $d_1 \times 1$. The portion R_1w_1 pre-computes partial inner products to yield an intermediate vector of shape $n_1 \times 1$. Its multiplication with K_1 "expands" that intermediate vector to length n_S as per the primary key-foreign key dependency. Since LA-based ML algorithms are just a series of LA operations, factorized LA effectively automates the push down for any ML algorithm expressible in LA.

3.3 Formalizing Quadratic Interactions

Note that the data processing operation of feature interactions is *non-linear* and thus, it is outside the scope of prior work on factorized LA. Before presenting our new framework that combines feature interactions and factorized LA, we need to formally define 3 non-linear operators: *self-interaction* and *cross-interaction*,

Definition 3.1. ***Self-interaction:*** *Given a matrix M of shape $n \times d$,* self-interaction *of M, denoted $\mathring{M}$ is a non-linear unary operator that outputs a matrix M' of shape $n \times d'$, wherein $d' = d + \binom{d}{2} = \binom{d+1}{2}$. The entries of the i^{th} row of M' are defined by the following binomial enumeration: $\forall a = 1, 2, \cdots, d, b = 1, 2, \cdots, a$, and $k = \binom{a}{2} + b$, we have $\mathring{M}[i, k] = M[i, a] \cdot M[i, b]$.*

Definition 3.2. ***Interaction:*** *Given a matrix M,* interaction *of M, is a non-linear unary operator that is denoted as inter(M) and defined as inter$(M) \triangleq [M, \mathring{M}]$.*

Definition 3.3. ***Cross-interaction:*** *Given two matrices M_1 of shape $n \times d_1$ and M_2 of shape $n \times d_2$,* cross-interaction *of M_1 and M_2, denoted $M_1 \otimes M_2$ is a non-linear binary operator that outputs a matrix M' of shape $n \times d'$, wherein $d' = d_1d_2$. The entries of the i^{th} row of M' are defined by the following pairwise enumeration: $\forall k = 1, 2, \cdots, d_1d_2$, letting $a = \lceil k/d_2 \rceil$ and $b = k \mod d_2$, we have $M'[i, k] = M_1[i, a] \cdot M_2[i, b]$ (where $M_2[i, 0] \triangleq M_2[i, d_2]$).*

Here, self-interaction captures pairwise interactions within a matrix (avoiding duplicate pairs), while full interaction includes the original d features. Cross-interaction captures pairwise interactions across two matrices. Now we are ready to present the first key proposition for representing the self interaction of a matrix in terms of its column-wise splitting. Assume that there is a matrix M which is split column-wisely by $M = [M_l, M_r]$. Then the self-interaction within M involves two columns, which may both come from M_l (thus falls in $\mathring{M}_l$), or M_r (thus falls in $\mathring{M}_r$), or M_l and M_r separately (thus falls in $M_l \otimes M_r$). Nevertheless, the ordering of the columns within $\mathring{M}$ can be different from $\mathring{M}_l, \mathring{M}_r, M_l \otimes M_r$ and therefore a permutation of the columns is needed. Skipping the proof due to space constraint, we summary the resulting proposition as follows.

Proposition 3.1. *Consider a matrix $M = [M_l, M_r]$ where $M_l \in R^{n \times d_l}$ and $M_r \in R^{n \times d_r}$. Then we have*

$$\mathring{M} = \left[\mathring{M}_l, \left[\mathring{M}_r, M_l \otimes M_r\right] P\right],$$

where $P \in R^{\left(\binom{d_r+1}{2}+d_ld_r\right) \times \left(\binom{d_r+1}{2}+d_ld_r\right)}$ is a permutation matrix such that $P[i, j] = 1$ iff there exist integers $a \in [d_l + 1, d_l + d_r]$ and $b \in [1, a]$, s.t. the following conditions hold:

$$i = \begin{cases} \binom{d_r+1}{2} + (b-2)d_l + a, & \text{if } b \leq d_l \\ \binom{a-d_l}{2} + b - d_l, & \text{otherwise} \end{cases}$$

$$j = \binom{a}{2} - \binom{d_l + 1}{2} + b.$$

Now we consider feature interactions over normalized data. We are given the normalized matrix (S, K_1, R_1) s.t. $T = [S, K_1R_1]$. Let $X_1 \triangleq K_1R_1$ denote the intermediate denormalized part of T. As per Proposition 3.1, we have $\mathring{T} = \left[\mathring{S}, \left[\mathring{X}_1, S \otimes X_1\right] P_T\right]$, where $P_T \in R^{\left(\binom{d_1+1}{2}+d_sd_1\right) \times \left(\binom{d_1+1}{2}+d_sd_1\right)}$ is a a permutation matrix such that $P_T[i, j] = 1$ iff there exist integers $a \in [d_s + 1, d_s + d_1]$ and $b \in [1, a]$, s.t the same conditions on i and j as in Proposition 3.1 hold, except with d_l replaced by d_s. Thus, we have the following relationship:

$$inter(T) = [T, \mathring{T}] = \left[S, X_1, \left[\mathring{S}, \left[\mathring{X}_1, S \otimes X_1\right] P_T\right]\right]. \quad (1)$$

Convention on foreign key features. An important difference in our setting compared to factorized LA is the way we treat foreign key features (RID_i/K_i). Such features are known to be beneficial for ML accuracy in some cases [17]. But for linear models, such categorical features need be represented as "one-hot" encoded vectors (large sparse 0/1 vectors). Since these can be very large (e.g., tens of thousands), even quadratic interactions can be untenable for such features. Thus, we exclude them from interactions in the rest of this paper.

3.4 Baseline: Interactions in Morpheus

A basic question is: *Is it possible to realize quadratic interactions directly over the normalized matrix?* Strictly speaking, the answer is no since the permutation matrix P_T disables us from creating a fact table feature matrix as well as a dimension feature matrix. If we view matrices as equivalent up to a permutation (i.e., ignoring permutation and reordering the columns), the answer becomes yes, but the major downside of this approach is likely to be inefficient, since it *requires materializing cross-interactions*. Thus, substantial computational redundancy will still remain. More precisely, to compute T with the regular normalized matrix in the Morpheus tool from [6], we need to replace (S, K_1, R_1) with a new normalized matrix $(\bar{S}, K_1, \bar{R}_1)$, defined as follows using our new non-linear operators (note $X_1 \triangleq K_1R_1$).

$$\bar{S} = [inter(S), S \otimes X_1] \text{ and } \bar{R}_1 = [RID_1, inter(R_1)]$$

In the above, $\bar{S}$ has a shape $n_S \times (2d_S + \binom{d_S}{2} + d_S d_1)$, where d_1 is the number of features in R_1, while $\bar{R}_1$ has a shape $n_1 \times (1 + 2d_1 + \binom{d_1}{2})$. Note that unlike the case of linear features where $T = [S, K_1R_1]$, for quadratic interaction, $\mathring{T} \neq [\bar{S}, K_1\bar{R}]$, but the equation holds up to a permutation, i.e., there exists some permutation matrix $P_{\mathring{T}}$, such that $\mathring{T}P_{\mathring{T}} = [\bar{S}, K_1\bar{R}]$.

We now explain why this approach is inefficient. Prior work showed that speedups possible with factorized LA for a given normalized matrix is a function of the so-called *tuple ratio* and *feature ratio* [16]. The former is n_S/n_1; the latter is d_1/d_S. As the tuple ratio goes to infinity, the speedup possible for most LA operators gets capped at 1+ *feature ratio*. In our setting, while tuple ratio is unaffected, feature ratio changes from d_1/d_S to $\left(2d_1 + \binom{d_1}{2}\right) / \left(2d_S + \binom{d_S}{2} + d_S d_1\right)$ in the above formulation. As d_1 goes to infinity, the latter term becomes $d_1/(2d_S)$ and thus the *feature ratio becomes around 50% smaller*. Indeed, such inefficiency is introduced by the cross-interaction term $S \otimes X_1$, which has a special algebraic structure (embedded in $X_1 = K_1R$) and hence provides room for further optimization.

Motivation for MorpheusFI. Given the above analysis of the baseline approach, we now ask: *Is it possible to further factorize quadratic interactions to reduce computational redundancy further?* In short, the answer is yes, but it requires fundamentally reworking the rewrites rules of factorized LA. In the next section, we present such a new framework that carefully delineates the interplay of redundancy caused denormalization with quadratic interactions.

4 FACTORIZED QUADRATIC INTERACTIONS

We first explain our new abstraction for capturing quadratic interactions over joins. We then present our framework of algebraic rewrite rules using our abstraction. The proofs of correctness for the rewrite rules are deferred to the appendix.

4.1 Interacted Normalized Matrix: A New Data Abstraction

We present a new abstraction, *interacted normalized matrix*, layered on the top of existing matrix data types in LA. It is similar to the normalized matrix of factorized LA, but has first-class support for quadratic interactions. For simplicity's sake, we first focus on a 2-table join as explained before, with the given normalized matrix being (S, K_1, R_1). We create a new hexatruple $(\hat{S}, K_1, \hat{R}_1, S, X_1, \hat{P})$ called *Interacted Normalized Matrix* with the following relationships (K_1 is retained):

$$\hat{S} = [S, \mathring{S}] \text{ and } \hat{R}_1 = [R_1, \mathring{R}_1] \text{ and } \hat{P} \text{ is a permutation matrix}$$
$$\text{s.t. } inter(\mathring{T}) = \left[\hat{S}, K_1\hat{R}_1, S \otimes (K_1R_1)\right] \hat{P}$$

Note that compared to the normalized matrix representation (in Section 3.4), the cross-interaction between S and X_1 is not explicitly maintained. Instead, we only store S, K_1, R_1, and push the computations on $S \otimes X_1$ down through S, K_1, R_1 whenever needed (explained later). This is central to how our abstraction avoids double redundancy. For simplicity purpose, let $\hat{R}'_1 \triangleq S \otimes (K_1R_1)$. Compared to Equation (1), permutation matrix $\hat{P}$ is simply used to reorder $inter(T)$'s columns. $\hat{P}$ is not physically constructed but lazily evaluated during the LA operation rewrite.

4.2 Element-wise Scaling Operators

Element-wise scaling (/ and ×) operators are common in ML. Typically they appear when a regularizer or a step size is applied. In Morpheus, the rewrite rules are trivial and preserve the normalized matrix structure.

In our setting, we observe that the scaling operators can also be *lazily evaluated* until other operators are in need. Thus, we maintain a scaling value denoted by α (initially = 1) in our implementation until any other operators are needed. If a new scaling factor β comes, we update α by

α/β or $\alpha \times \beta$ depending on the scaling operator. When other operator is called, we apply the following update $\hat{S} = \alpha\hat{S}$, $\hat{R}_1 = \alpha\hat{R}_1$ and $R_1 = \alpha R_1$. Note that we do not update $\hat{P}$ at all.

4.3 Left Matrix Multiplication (LMM)

LMM is also common in ML, arising is all GLMs solved with batch gradient methods. For clarity of exposition, we present the Morpheus rewrite rule before explaining our new rewrite rule. Note that W is a $d \times d_W$ parameter matrix.

$$TW \rightarrow SW[:d_S,] + K_1(R_1W[d_S+1:d,])$$

With Interactions. In our setting, we first permute W to mitigate the effect of $\hat{P}$, then multiply parts of the results with the self-interactions and cross-interaction parts seperately, and finally add them. Formally, recall that $inter(\hat{T}) = \left[\hat{S}, K_1\hat{R}_1, \hat{R}'\right]\hat{P}$. We first compute $\hat{W} = \hat{P}W$ (where P is simply a permutation matrix and thus this is only reordering the rows in W). Split $\hat{W}$ column-wisely into 3 parts,$\hat{W}_S = \hat{W}[1:d_S + \binom{d_S+1}{2},]$, $\hat{W}_R = \hat{W}[1+d_S+\binom{d_S+1}{2} : d_1 + \binom{d_1+1}{2} + d_S + \binom{d_S+1}{2},]$, and $\hat{W}_{SR} = \hat{W}[1+d_1+\binom{d_1+1}{2}+d_S+\binom{d_S+1}{2} : d,]$. Then we have

$$inter(T)W = \hat{S}\hat{W}_S + K_1\hat{R}_1\hat{W}_R + \hat{R}'_1\hat{W}_{SR}$$

Since $\hat{S}$ has no join-related redundancy, we leave the first term as it is. For the second term, similar to the case of Morpheus, we mutiply $\hat{R}_1\hat{W}_R$ and then compute the result times K_1.

The third term, i.e., $\hat{R}'_1\hat{W}_{SR}$, is the most challenging and novel part. The rewrite rule for this part is illustrated in Figure 2. First, cut $\hat{W}_{SR}$ into d_S segments *row-wise*, each with the shape $d_1 \times d_W$ and the i-th chunk is denoted by W^i_{S1}. Then, multiply R_1 with each W^m_{S1} and stitch the resultant

$$R'_1 = [R_1W^1_{S1}, R_1W^2_{S1}, ..., R_1W^{d_S}_{S1}]$$

Let M denote $K_1R'_1$; its shape is $n_S \times d_W d_S$. If $d_W = 1$, do a Hadamard product ($\odot$) between M and S and take a *rowSum* of the result. Thus, we have this rewrite rule:

$$\hat{R}'_1\hat{W}_{SR} \rightarrow rowSum(M \odot S)$$

If $d_W > 1$, chunk M into d_W segments by columns, denoted $M^{(j)}$. Apply the Hadamard product and *rowSum* to each column and stitch all resultant matrices column-wise:

$$\hat{R}'_1\hat{W}_{SR} \rightarrow [rowSum(M^{(1)} \odot S), \cdots, rowSum(M^{(d_W)} \odot S)]$$

Overall, the full rewrite rule for LMM $inter(T)W$ with $\hat{P}W \triangleq [\hat{W}_S, \hat{W}_R, \hat{W}_{SR}]$ is as follows:

$$inter(T)W \rightarrow (\hat{S})\hat{W}_S + \hat{K}_1(\hat{R}_1\hat{W}_R) + \left[rowSum\left(M^{(1)} \odot S\right), \cdots, rowSum\left(M^{(d_W)} \odot S\right)\right]$$

where $\left[M^{(1)}, \cdots, M^{(d_W)}\right] \triangleq K_1\left(\underbrace{[R_1, R_1, \cdots, R_1]}_{d_S} W_{SR}\right)$.

4.4 Right Matrix Multiplication (RMM)

RMM is another common operator in ML (used in all GLMs as well). Given a parameter matrix W of shape $n_W \times n_S$, the Morpheus rewrite rule was as follows:

$$WT \rightarrow [WS, (WK_1)R_1]$$

With Interactions. In our setting, we need to handle self-interactions and cross-interactions. Note that

$$Winter(T) = [W\hat{S}, WK_1\hat{R}_1, W\hat{R}'_1]$$

Again, since $\hat{S}$ is not factorizable, so the first term is retained. For the second term, similar to the rewrite in Morpheus, we compute $W\mathring{K}_1$ first and then multiply the result by $\hat{R}$. $W\hat{R}'_1 = W(S \otimes (K_1R_1))$ is the most novel and challenging part. Its rewrite rule contains two main steps. First, we compute $\left[M^{(1)}, M^{(2)}, \cdots, M^{(d_W)}\right] \triangleq K_1^T\left(W^T \otimes S\right)$, where each $M^{(i)}$ has d_S columns. Next, we compute $colSum(M^{(i)} \otimes R_1)$ for each i and stitch the results row-wisely. This gives (proved in the appendix)

$$W\hat{R}'_1 = \left[colSum\left(M^{(1)} \otimes R_1\right); \cdots; colSum\left(M^{(d_W)} \otimes R_1\right)\right]$$

The overall rewrite rule is as follows.

$$Winter(T) \rightarrow \left[W\hat{S}, (WK_1)\hat{R}_1, \left[colSum\left(M^{(1)} \otimes R_1\right); \cdots; colSum\left(M^{(d_W)} \otimes R_1\right)\right]\right]$$

where $\left[M^{(1)}, M^{(2)}, \cdots, M^{(d_W)}\right] \triangleq K_1^T\left(W^T \otimes S\right)$ and each $M^{(i)}$ has d_S columns.

4.5 Matrix Aggregation

Matrix aggregations such as *rowSum*, *colSum*, and *sum* help compute loss functions and gradients in ML. Due to space constraints, we skip the Morpheus rewrite rules here.

With Interactions. Once again, the novelty is in handling self-interactions and cross-interactions. Interestingly, we can reduce redundancy further by pushing down aggregation through the interaction as well–the first known instance of such a push down, to the best of our knowledge. The multi-part rewrite rule is as follows:

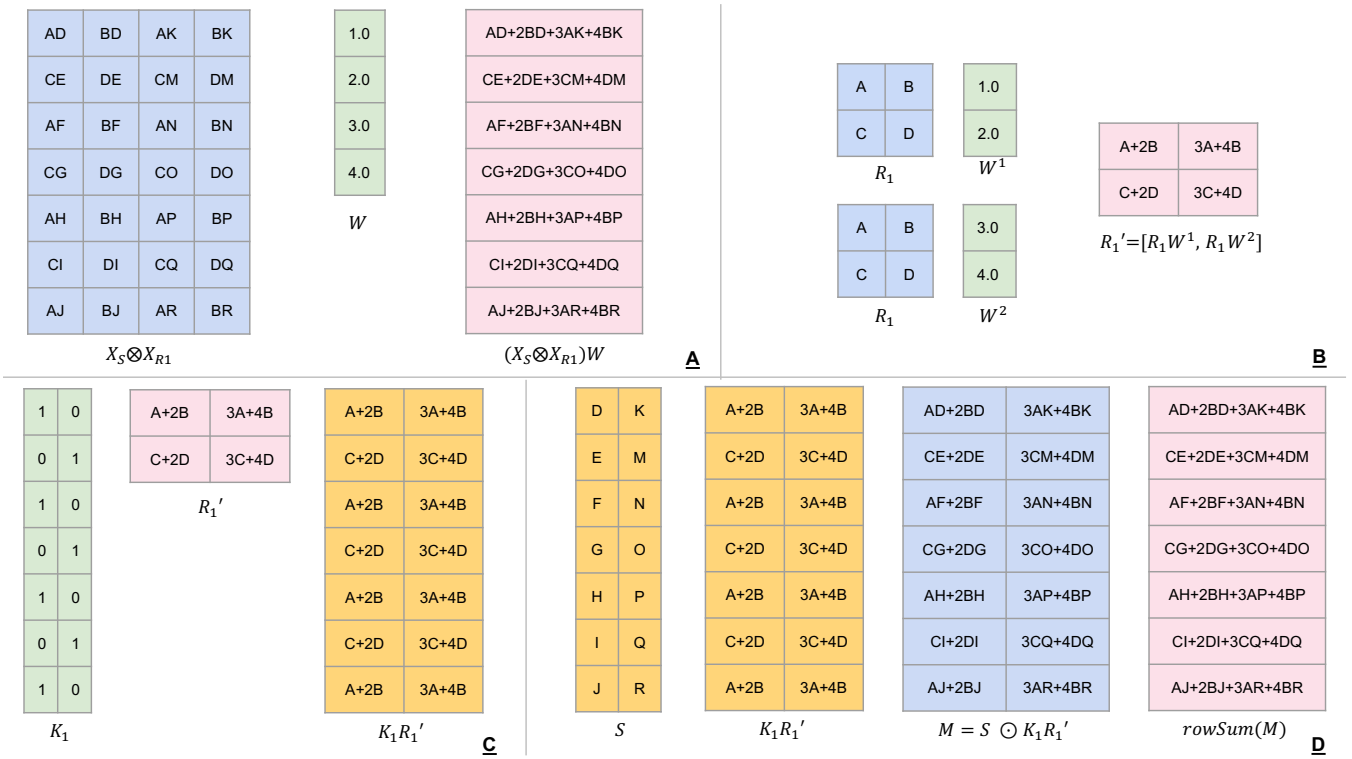

Figure 2: Illustration of LMM for $X_S \otimes X_1$ (X_1 is denoted X_{R1} for more clarity). (A) Materialized LMM. The individual entries of the dataset matrices are denoted with variables A, B, etc. instead of specific numbers to show the propagation of values. (B) Decompose W and multiply segments with R_1 to get R_1'. (C) Expand R_1' by multiplying it with K_1. (D) Hadamard product of S and K_1R_1', whose result is passed to $rowSum$.

Part 1: $\mathring{S}$: While this is not factorizable, we can push down aggregation through interaction as follows:

$$rowSum(\mathring{S}) \rightarrow \left((rowSum\,(S))^2 + rowSum\,(S \odot S)\right)/2$$

$$sum(\mathring{S}) \rightarrow sum\left(rowSum\left(\mathring{S}\right)\right)$$

Part 2: $\mathring{X}_1$: This part is factorizable; rewrite it like X_1:

$$rowSum(\mathring{X}_1) \rightarrow K_1\left((rowSum(R_1))^2 + rowSum(R_1 \odot R_1)\right)/2$$

$$colSum(\mathring{X}_1) \rightarrow colSum(K_1)\mathring{R}_1$$

$$sum(\mathring{X}_1) \rightarrow colSum\,(K_1)\,rowSum(\mathring{R}_1)$$

Part 3: $S \otimes X_1$: This part is also factorizable, in addition to pushing down aggregation through joins. Note how the aggregation of the cross-interaction becomes a Hadamard product over partial aggregations.

$$rowSum(S \otimes X_1) \rightarrow rowSum(S) \odot (K_1 rowSum(R_1))$$

$$colSum(S \otimes X_1) \rightarrow colSum\left(\left(K_1^T S\right) \otimes R_1\right)$$

$$sum(S \otimes X_1) \rightarrow colSum\left(rowSum(K_1^T S) \odot (rowSum(R_1))\right)$$

Using the above, the full rewrite rules can be easily derived and skipped here due to space limit.

4.6 Crossprod

Crossprod of T, which is $T^T T$ (aka Gramian), arises in least squares linear regression and other ML techniques. Its runtime is expensive–$O(nd^2)$, which becomes $O(nd^4)$ with quadratic interactions. Due to space constraints, we skip its tedious Morpheus rewrite rule.

With Interactions. The goal is to compute $crossprod(inter(T)) = inter(T)^T inter(T)$. We need to carefully handle the double redundancy in self-interactions and cross-interactions. We create the following novel multi-part rewrite rule, wherein cp stands for $crossprod$.

$$P_1 = \hat{R}_1^T(K_1^T \hat{S})\ ;\ P_2 = \hat{R}_1'^T \hat{S} = (\hat{S}^T \hat{R}_1')^T$$

$$P_3 = \hat{R}_1'^T(K_1 \hat{R}_1) = (\hat{R}_1^T(K_1^T(S \otimes X_1)))^T = \left(\hat{R}_1^T\left(\left(K_1^T S\right) \otimes R_1\right)\right)^T$$

$$Q = (diag(colSums(K)))^{\frac{1}{2}} \hat{R}_1\ ;$$

$$cp(\hat{R}_1') = cp(S \otimes (K_1 R_1)) = reshape\left((R_1 \otimes R_1)^T\left(K_1^T\,(S \otimes S)\right)\right)$$

where "reshape" is an operator used to change the shape of a matrix as well as reorder the elements within it. We leave it to the appendix due to space limit.

Overall, the rewritten expression for $cp(inter(T))$ is:

$$\begin{bmatrix} cp(\hat{S}) & P_1^T & P_2^T \\ P_1 & cp(Q) & P_3^T \\ P_2 & P_3 & cp(\hat{R}_1') \end{bmatrix}$$

P_2 can be factorized using the rewrite rules for RMM as presented earlier. Here the most challenging part is to compute $cp(\hat{R}_1')$ without materializing $\hat{R}_1'$. The key insight is that we can pre-aggregate the rows in S based on K_1 to avoid $O(d^2)$ many innner-product of vectors with length n_S. Instead, the rewrite rule only asks for $O(d_S^2)$ many such inner-product.

4.7 Matrix Pseudo-Inverse

The rewrite rule pseudo-inverse in our framework is identical to that of Morpheus, since the structure of this operation is not affected by feature interactions.

5 ANALYSIS AND EXTENSIONS

We now formally analyze the runtime complexity of our rewritten LA operations. We then extend our framework to star schema multi-table joins. Finally, we discuss the runtime trade-offs of our rewrite rules and explain why crossovers with materialized execution can arise. Due to space constraints, the formal proofs of correctness for LMM and RMM rewrite rules are given in the Appendix.

5.1 Runtime Complexity Analysis

Table 2 summarizes the runtime complexity comparison. Since we want to know how much computational redundancy has been avoided in terms of the number of FLOPS (for floating point adds and multiplies), we skip the big O notation. Instead, we express the proportional dependency of the arithmetic computation cost (FLOPS), on the data size parameters. Recall that S has shape $n_S \times d_S$, while R_1 has shape $n_1 \times d_1$. Also, $d = d_S + d_1$, while after quadratic interaction, the number of features goes up to $2d + \binom{d}{2}$.

5.2 Extension to Star Schema

Star schemas are common. For example, in a recommendation system like Netflix, one often joins the fact table with ratings with at least 2 dimension tables: user and movie details. Having more than one dimension tables complicates feature interactions, especially across said tables. We explain how our rewrite rules generalize to a 3-table star schema with 1 fact table **S** and 2 dimension tables $\mathbf{R}_1$ and $\mathbf{R}_2$. It is straightforward to extend to more multiple dimension tables, and we have implemented them in MorpheusFI. But we skip showing the most general forms for exposition sake, since they are tedious. Similarly, we omit the permutation matrix for exposition sake. Formally, **T** has the schema $\mathbf{T}(Y, [X_S, X_1, X_2])$. The base table feature matrices

Operator	Materialized	Our Framework
Scaling Op	$n_S f(d)$	1 (lazy)
LMM	$d_W n_S(f(d_s)+ f(d_1) + d_S d_1)$	$d_W(n_S f(d_S) + n_1 f(d_1) + n_1 d_1 d_S + n_S d_S)$
RMM	$n_W n_S(f(d_S)+ f(d_1) + d_S d_R)$	$n_W(n_S f(d_S) + n_1 f(d_1) + 2n_S d_1 + n_1 d_1 d_S)$
sum	$n_S f(d)$	$4(n_S d_S + n_1 d_1)$
crossprod	$\frac{1}{2} n_S f(d_S + d_1)^2$	$n_S f(d_s)^2 + 2n_1 f(d_1)^2 + n_1 f(d_S) f(d_1) + n_1 (d_S d_1)^2$

Table 2: Arithmetic computation costs (time complexity). We denote $f(x) = 2x + \binom{x}{2}$.

are S, R_1, and R_2, while the foreign key indicator matrices are K_1 and K_2. Thus, modulo a column permutation, we have $inter(T) = [S, \mathring{S}, S \otimes X_1, S \otimes X_2, X_1, \mathring{X_1}, X_2, \mathring{X_2}, X_1 \otimes X_2]$, Note that $X_1 = K_1 R_1$ and $X_2 = K_2 R_2$.

We see a new component: $X_1 \otimes X_2$, cross-interaction across dimension tables. *The technical novelty of this extension is in carefully avoiding double redundancy for this component.* The other components area handled as explained in Section 4. We skip scalar operations for brevity sake, since their behavior is similar to the 2-table join case. Other LA operators, however, require novel rewrite rules to factorize $X_1 \otimes X_2$.

5.2.1 LMM. Denote the chunk of W that multiplies with $X_1 \otimes X_2$ as W_{12}; its shape is $d_1 d_2 \times d_W$. To handle $(X_1 \otimes X_2) W_{12}$, we first compute $X_1 = K_1 R_1$, and then use the similar rewrite in LMM for 2-table joins. More precisely,

$$(X_1 \otimes X_2)\, W_{12} \rightarrow$$
$$\left[rowSum\left(M^{(1)} \odot X_1\right), \cdots, rowSum\left(M^{(d_W)} \odot X_1\right) \right]$$

where $\left[M^{(1)}, \cdots, M^{(d_W)}\right] \triangleq K_2 \left(\underbrace{[R_2, R_2, \cdots, R_2]}_{d_1} W_{12} \right)$.

Note that changing the ordering of X_1, X_2 gives a different rewrite rules (by exchaning 1 and 2). Therefore, it becomes an interesting question of how to order the attribute tables. We leave a detailed discussion to Section 5.3. Given the above rewrite rule, the full rewrite rule for LMM can be easily obtained and skipped here due to space limit.

5.2.2 RMM. Similar to LMM, the key novelty is in handling $W(X_1 \otimes X_2)$. We first fix X_1 and then we have:

$$W(X_1 \otimes X_2) \rightarrow$$
$$\left[colSum\left(M^{(1)} \otimes R_2\right); \cdots; colSum\left(M^{(d_W)} \otimes R_2\right) \right]$$

where $\left[M^{(1)}, M^{(2)}, \cdots, M^{(d_W)}\right] \triangleq K_2^T \left(W^T \otimes (K_1 R_1)\right)$ and each $M^{(i)}$ has d_1 columns.

Note that similar to LMM, if we change the order of X_1, X_2, we will have another rewrite rule.

5.2.3 Aggregation. As in Section 4.5, the $X_1 \otimes X_2$ part is not just factorizable but also amenable to push down of aggregation through interactions. Indeed, aggregation of cross-interaction becomes a Hadamard product of partial aggregations, avoiding computational redundancy further.

$$rowSum(X_1 \otimes X_2) \rightarrow K_1 rowSum(R_1) \odot K_2 rowSum(R_2)$$
$$colSum(X_1 \otimes X_2) \rightarrow colSum(K_1)R_1 \otimes colSum(K_2)R_2$$
$$sum(X_1 \otimes X_2) \rightarrow sum(K_1 rowSum(R_1) \odot K_2 rowSum(R_2))$$

Due to space constraints, we present the overall rewrite rules for aggregations of $inter(T)$, as well as the rewrite rule for *crossprd* in the Appendix.

5.3 Performance Trade-offs and Crossovers

Prior work on factorized LA showed that the rewritten factorized approach will not always be faster than materialized execution. As in relational query optimization, there are performance trade-offs based on the data sizes, leading to crossovers in runtime trends between alternate plans. In particular, [16] showed that two quantities are critical to quantify these trade-offs: *tuple ratio* and *feature ratio*. For a 2-table join, the tuple ratio is n_S/n_1; feature ratio is d_1/d_S. If the feature ratio is high (say, > 1) and tuple ratio is also high (say, > 5), the efficiency gains of factorized LA will be significant. Otherwise, materialized execution is comparable or even slightly faster due to the overheads of extra LA operations introduced by the rewrite rules [6].

The above trade-offs matter in our setting too, but an additional factor also matters: *sparsity* (fraction of non-zero entries). Sparse matrices are common in ML applications, since categorical features are usually "one-hot" encoded to get long 0/1 vectors, especially for linear models. In our customer churn example, country is a categorical feature (with say, 200 unique values) that will be recoded to a 200-D vector with exactly one feature being 1 (rest are 0). Sparse matrix representation avoids storing zeros by storing only triples of row-column-value for non-zeros.

Sparse Feature Interaction Trade-offs. Given the above, crucial observation is this: *quadratic interactions amplify sparsity quadratically*. For instance, if we have a matrix M with sparsity e, the sparsity of materialized $inter(M)$ is roughly e^2. But in our framework's rewrite rules, we create many *dense intermediate tables*. For instance, in the rewrite rule for LMM in Section 4.3, rewriting $(S \otimes X_1)W$ creates dense intermediates R_1' and M. Thus, if the base tables' matrices are too sparse, factorized interactions might be more expensive than materialized execution. More precisely, consider only the cross-interaction between S and R_1 in the LMM example of Figure 2. Suppose both S and R_1 have a sparsity of e. Materialized execution (A in the figure) costs $n_S d_S d_1 e^2$, while our rewrite rule costs $n_1 d_1 d_S e + n_S d_S e$. The speedup is $n_S d_S d_1 e^2/(n_1 d_1 d_S e + n_S d_S e)$, which can be < 1, if e is very small ($< 5\%$). But if e is large or just equal to 100% (i.e., the base table matrices are dense), the speedups could be reasonable or even substantial.

Heuristic Decision Rule. To deal with the sparsity-related trade-offs in addition to the feature and tuple ratios, we propose a simple *conservative* heuristic decision rule to predict if our rewritten approach is likely to be significantly faster. This could help users decide for their dataset instance whether to use our tool; it can also be integrated into an "automated optimizer" for a higher-level ML system. Our decision rule is derived from the above cost ratio calculation for LMM. Assume all base table matrices have sparsity e. Ignore $n_S d_S e$ in the denominator. The ratio gets simplified to roughly $\frac{n_S}{n_1} e$. Our decision rule then is as follows: If $\frac{n_S}{n_1} e > 1$, use the factorized interaction rewrite rule. We can also extend this decision rule to multi-table joins. In general, the more dimension tables we have that satisfy the above rule, the more likely it is that the factorized interaction approach will be faster. Note that even if one base table matrix is dense (or almost dense), all cross-interactions with it will likely become dense. With this intuition, we extend our decision rule as follows. Let p be the number of base tables. Let q be the number of dimension tables that are "sparse" (sparsity $< 5\%$). Let e_i denote the sparsity of R_i. Our decision rule is as follows (empirically validated in Section 6).

Rule. *Use our factorized interaction framework if and only if one of the following two conditions hold:*
$q < \lfloor \frac{p}{2} \rfloor$, *or* $q \geq \lfloor \frac{p}{2} \rfloor$ *and* $\forall i \in 1$ *to* q, $e_i \frac{n_S}{n_i} > 1$.

Ordering of Dimension Tables. As mentioned in Section 5.2.1, the ordering among dimension tables matters for performance in our framework. Recall that the LMM rewrite rule multiples R_1 with d_2 chunks of W, yielding R_1' of shape $n_1 \times d_2 d_W$. Had we swapped R_1 for R_2 in this ordering, the shape of the intermediate matrix will be $n_2 \times d_1 d_W$ instead. Clearly, the relative sizes of R_i matter for ordering. We now formally analyze what ordering is likely to be most beneficial. We use LMM as our prototypical LA operator to understand this trade-off more deeply. Let the sparsity of R_1 and R_2 be e_1 and e_2, respectively. The cost ratio relative to materialized execution of our factorized rewrite rule for the $X_1 \otimes X_2$ component is as follows:

$$\frac{n_S d_1 d_2 e_1 e_2}{e_1 n_1 d_1 d_2 + n_S d_2 e_2} = \frac{n_S e_1 e_2}{e_1 n_1 + \frac{n_S}{d_R} e_2} = \frac{1}{\frac{1}{e_2 \frac{n_S}{n_1}} + \frac{1}{d_1 e_1}}$$

Thus, the smaller the values of $\frac{1}{e_2 \frac{n_S}{n_1}}$ and $\frac{1}{d_1 e_1}$, the larger the speedup for this cross-interaction across dimension tables.

This observation suggests a simple *heuristic ordering rule* for dimension tables. Given a pair of dimension tables R_i and R_j, R_i should go before R_j for the rewrite rules if the following holds. If there are ties in the global ordering based on conflicts between such local pairwise orderings, we can break the ties randomly.

$$\frac{1}{e_j \frac{n_S}{n_{R_i}}} + \frac{1}{d_{R_i} e_i} < \frac{1}{e_i \frac{n_S}{n_{R_j}}} + \frac{1}{d_{R_j} e_j}$$

6 EXPERIMENTAL EVALUATION

We prototype our framework in PyTorch to create a tool we call MorpheusFI. We briefly explain its implementation and then present an extensive empirical evaluation of its performance on various synthetic and real-world datasets. We seek to answer three questions. (1) How do our rewrites affect runtimes of various LA operators and LA-based ML algorithms? (2) Is our characterization of the runtime tradeoffs accurate? (3) How do the runtimes and accuracy of the ML algorithms on real data in MorpheusFI compare with the alternatives?

Implementation of MorpheusFI. We implement our interacted normalized matrix as a Python class using NumPy's *ndarray* and SciPy's *COO sparse matrix*. We integrate it with the popular ML package PyTorch to exploit pre-existing gradient methods. Our class supports a star schema. Every LA operator is overloaded for our class. As in Morpheus, transposed LA operations are handled using the regular rewrite rules with a double transpose. A flag in our class records if it is transposed. Scalar operators are cached for lazy evaluation. Another flag lets the user indicate if foreign key features should be used. As mentioned in Section 4.1, our class has a copy of S an R_i along with the self-interactions $\mathring{S}$ and $\mathring{R}_1$, but it does not physically store the denormalized versions or cross-interactions (to avoid redundancy). A user can load a star schema dataset as CSV files using a simple file reading API. Given an LA-based ML algorithm, MorpheusFI invokes our rewrite rules for each LA operation in that algorithm automatically under the covers. Due to space constraints, we defer examples of such automatically factorized interactions to the appendix: logistic regression and linear SVM with gradient descent and ordinary least squares linear regression.

Synthetic Data. We generate data of various shapes in NumPy to analyze runtimes in depth. We focus primarily on a 2-table join. We set $n_1 = 10^5$ and $d_S = 20$ and vary the tuple ratio $\frac{n_S}{n_1}$ and feature ratio $\frac{d_1}{d_S}$ to set n_S and d_1. All matrices are dense except for the sparsity experiment.

Real-world Datasets. We use the 7 real-world star schema datasets from [17]. Each dataset has at least 2 dimension tables. Some of the datasets have no features in the fact table, in which case S is empty. We pre-processed all datasets as per standard ML practice: drop primary keys in fact tables, "whiten" all numeric features (subtract mean and divide by standard deviation), and convert all categorical features with one-hot encoding. For binary classification with logistic regression and linear SVM, we binarized all targets to 0/1 values for the former and 1/-1 for the latter. Overall, many tables' matrices become very sparse. Overall, the shapes and sparsity of the tables' matrices are listed in Table 3.

Dataset	(n_S, d_S)	S Sparsity	(n_i, d_i)	R_i Sparsity
Walmart	(421570, 81)	0.0123	(2340, 9) (45, 4)	0.9951 0.5000
Yelp	(215879,0)	NA	(11537, 111) (43873, 7)	0.2883 0.8571
Movie	(1000209,0)	NA	(6040, 3463) (3706, 120)	0.0012 0.1750
Expedia	(942142, 6)	0.8771	(37021, 3205) (11939, 43)	0.0044 0.1860
LastFM	(343747,0)	NA	(4999, 12) (50000, 229)	0.5833 0.0175
Books	(253120,0)	NA	(27876, 140) (49972, 3662)	0.0143 0.0011
Flights	(66548, 20)	1.0	(540, 178) (3182, 3301) (3182, 3301)	0.0281 0.0018 0.0018

Table 3: Shapes and sparsity of the tables in the real-world datasets. "NA" means S is empty in that dataset.

Experimental Setup and Protocol. All experiments were run on CloudLab [7]. The machine had 2 Intel E5-2660 v2 10-core CPUs, 256 GB RAM and 2 TB disk. The OS was Ubuntu 16.04 LTS. We used Python 2.7, NumPy 1.13, SciPy 1.1, PyTorch 0.4.0, and gcc 5.4.0 20160609. For the baseline comparison with Morpheus, we use *MorpheusPy* [19], which implements factorized LA in Python NumPy. The *Materialized* execution plan uses the single-table matrix T, which is pre-materialized by joining the base tables in Python. We *exclude* this materialization in our results (note this favors *Materialized*). We exclude all data pre-processing times for the real datasets, as well as all data loading times for all compared tools to let us focus on LA/ML computations times. All runtimes reported are averages of three runs.

6.1 Results on Synthetic Data

6.1.1 ***LA Operators.*** We first compare the relative runtimes of MorpheusFI against *Materialized* for 3 time-intensive LA operators: LMM, RMM, and *crossprod*. Figure 3 presents the results. We see that MorpheusFI achieves higher speedups for higher feature and tuple ratios on all operators, which

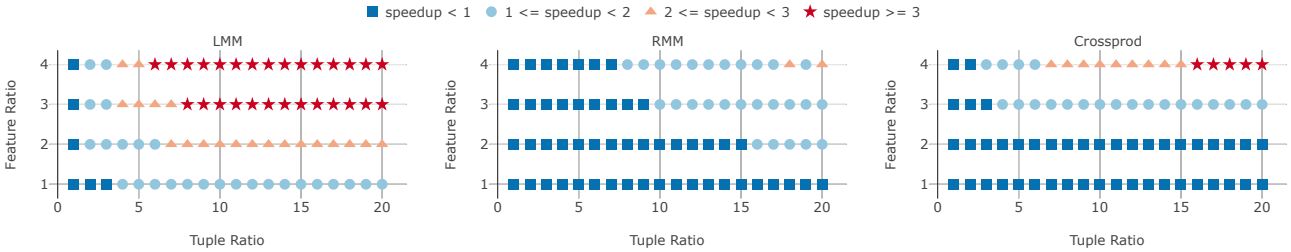

Figure 3: Relative runtimes of MorpheusFI against Materialized for major LA operators on synthetic data.

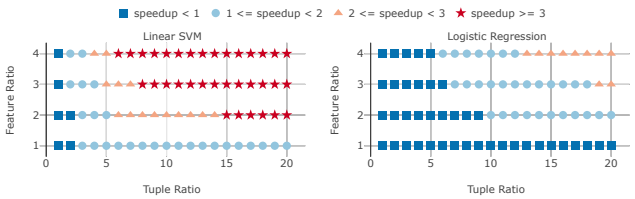

Figure 4: Relative runtimes of MorpheusFI against Materialized for LA-based ML algorithms on synthetic data.

is consistent with past results for regular factorized LA. For LMM, the speedups grow with the feature ratio even for a tuple ratio of just 10. But for *crossprod*, MorpheusFI is faster only at much higher tuple and feature ratios. This is because the fraction of the time spent on the factorized portion gets reduced with feature interactions–recall that $\mathring{S}$ has no redundancy. The trends for RMM are in between because its overheads are more substantial than LMM.

6.1.2 ***LA-based ML Algorithms.*** We now compare the relative runtimes for end-to-end training of LA-based ML algorithms. We study 2 popular classifiers for which feature interactions are widely used: linear SVM and logistic regression. The automatically factorized interaction versions of both algorithms are shown in the appendix. Figure 4 presents the results. We see that the trends for linear SVM resemble that of LMM. This is expected because LMM is the dominant LA operator in that algorithm. In contrast, the trends for logistic regression look like a hybrid of LMM and RMM; this is because both of these operators arise in this algorithm. Overall, MorpheusFI yields speedups for both LA-based ML algorithms commensurate with the amount of redundancy in the data, as captured by the tuple and feature ratios.

6.1.3 ***Effects of Sparsity.*** As explained in Section 5.3, sparsity is a key factor for the runtime trade-offs of MorpheusFI. To understand these trade-offs quantitatively, we synthesize data for a 3-table join. All R_i have the same shape. We set $n_1 = n_2 = 10^5$, $d_S = 20$, both tuple ratios to 20, and both feature ratios to 4. We study various sparsity regimes: only R_1 is sparse, both R_1 and R_2 are sparse, etc., with the same sparsity used for all sparse tables. We plot the runtimes

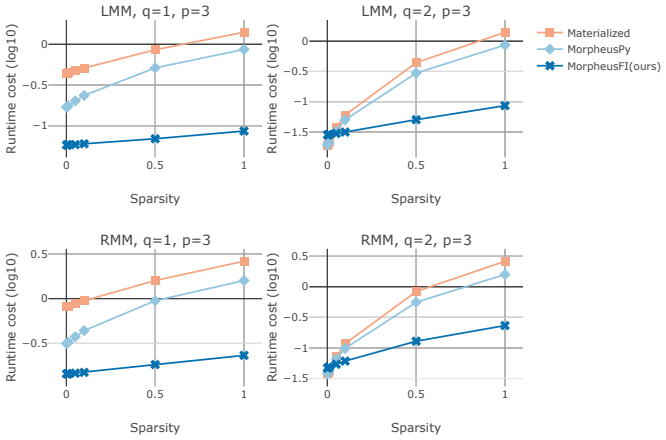

Figure 5: Effect of sparsity on LMM and RMM. p is the number of joined tables and q is the number of dimension tables that are sparse; their sparsity factor is varied on the x axis.

of LMM and RMM in regime for varying sparsity. Figure 5 shows the runtimes for 2 regimes comparing MorpheusFI, Materialized, and MorpheusPy. Other regimes yielded similar insights; we defer them to the appendix due to space constraints.

Overall, we see that factorized interactions in MorpheusFI can be *even an order of magnitude faster* than both Materialized and MorpheusPy as the sparsity factor goes to 1 (i.e., the dimension tables become dense). As expected, the gaps are larger when only 1 dimension table is sparse ($q = 1$) compared to both being sparse, which validates our explanation of the trade-offs in Section 5.3. Interestingly, when $q = 1$, MorpheusFI is *always* faster regardless of the sparsity factor, i.e., there are no crossovers. But for $q = 2$, we see crossovers below a sparsity factor of 0.05, wherein both Materialized and MorpheusPy become (slightly) faster. These results justify our heuristic decision rule that also considers sparsity for predicting when using MorpheusFI might actually be beneficial.

6.2 Results on Real-world Datasets

We now present the runtime and accuracy results for logistic regression and linear SVM with feature interactions on

Dataset	# Batches Seen		Validation Accuracy		Time to Convergence (sec)				Speedup of MFI over:		
	Adam	LBFGS	Adam	LBFGS	Adam	LBFGS (MFI)	LBFGS (Mor)	LBFGS (Mat)	Adam	LBFGS (Mor)	LBFGS (Mat)
Walmart	6727	87	0.9287	0.9336	13.72	9.2	12.2	29.2	1.5	1.3	3.2
Yelp	1080	50	0.7595	0.7601	23.98	16.1	26.4	75.7	1.5	1.6	4.7
Movie	5040	116	0.6830	0.6840	398.39	108.4	137.9	314.3	3.7	1.3	2.9
Expedia	7467	80	0.7685	0.7629	671.15	118.3	176.4	267.5	5.7	1.5	2.3
LastFM	5244	39	0.6845	0.6786	31.53	8.4	5.6	10.7	3.7	0.7	1.3
Books	525	48	0.5999	0.6005	50.94	32.8	20.0	19.4	1.6	0.6	0.6
Flights	3751	167	0.8360	0.8560	1099.35	441.6	272.2	271.3	2.5	0.6	0.6

Table 4: End-to-end training results of logistic regression with feature interactions. "MFI" is MorpheusFI. "Mor" is MorpheusPy. "Mat" is Materialized. NB: MorpheusFI, MorpheusPy, and Materialized have the same LBFGS accuracy.

Dataset	# Batches Seen		Validation Accuracy		Time to Convergence (sec)				Speedup of MFI over:		
	Adam	LBFGS	Adam	LBFGS	Adam	LBFGS (MFI)	LBFGS (Mor)	LBFGS (Mat)	Adam	LBFGS (Mor)	LBFGS (Mat)
Walmart	48447	166	0.9257	0.9284	102.5	16.7	23.0	55.1	6.1	1.4	3.3
Yelp	3352	112	0.7571	0.7490	76.0	34.4	57.0	168.3	2.2	1.7	4.9
Movie	10220	196	0.6774	0.6313	989.7	193.6	216.4	673.0	4.7	1.1	3.5
Expedia	16726	36	0.7557	0.7338	1760.0	48.8	84.8	129.2	36.1	1.7	2.6
LastFM	18612	84	0.6634	0.6783	113.6	17.2	10.5	21.3	6.6	0.6	1.2
Books	2484	84	0.5940	0.5905	295.8	58.9	38.8	38.4	5.0	0.7	0.7
Flights	2285	311	0.8395	0.8617	651.2	705.9	557.0	563.4	0.9	0.8	0.8

Table 5: End-to-end training results of linear SVM with feature interactions. "MFI" is MorpheusFI. "Mor" is MorpheusPy. "Mat" is Materialized. NB: MorpheusFI, MorpheusPy, and Materialized have the same LBFGS accuracy.

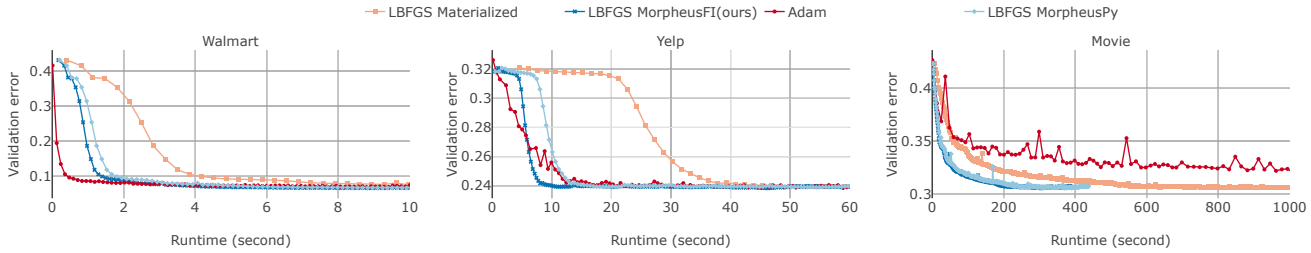

Figure 6: Convergence behavior of logistic regression validation error over wall-clock runtime of training for 3 real-world datasets. For LBFGS, errors are computed after every epoch. For Adam, errors are computed after every mini-batch. Note that the time to compute the errors is exclude from the runtime.

the real data. We use the LBFGS optimization procedure, which has a similar data access pattern as BGD. Since PyTorch already implements LBFGS, we just overloaded the *Autograd* function of PyTorch to compute "forward" (loss) and "backward" (gradient) passes. We use PyTorch's native *SparseTensor* for LA operators. We compare 4 approaches: *LBFGS+MorpheusFI*, *LBFGS+MorpheusPy*, *LBFGS+Materialized*, and *Adam*, a popular SGD procedure [14]. Since SGD is not expressible with bulk LA operators (it needs mini-batch sampling), *Adam* uses the pre-materialized T. We prepare all mini-batches beforehand and store them in memory; this pre-processing time is excluded from the results (this could favor *Adam*).

ML Methodology. We follow standard ML training methodology [10]. The datasets are pre-split into train-validation-test sets. We use L2 regularization and tune two hyper-parameters: *regularizer* (λ) and initial *stepsize* (α) using a standard grid search: $\alpha \in \{1, 0.5, 0.25, 0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001, 0.00001\}$ and $\lambda \in \{1, 0.5, 0.1, 0.05, 0.01\}$. Other hyper-parameters used default PyTorch values (batch size 50 for *Adam* and 20 cached gradients for LBFGS). All models are trained till *convergence*, defined as follows per standard practice. Compute the moving average of the validation error for the last 5 iterations. If it drops by $< 0.01\%$, stop. An "iteration" for LFBGS is a full pass over the data; for *Adam*, it is a mini-batch pass. So, Adam can converge even "within"

an epoch. Note that the moving average helps smooth over the noisy convergence behavior of SGD/Adam. We *exclude* the time to compute validation errors for all approaches.

Heuristic Decision Rule Predictions. We first check if we can even expect MorpheusFI to be faster than Materialized using our heuristic decision rule from Section 5.3. Note that this check only applies to the LBFGS-based approaches and relates only to per-epoch runtimes. It is not possible to predict time to convergence. Based on the values of the ratios and sparsity listed in Table 3, the predictions of our decision rule are as follows: "Yes" for 5 of 7 datasets: Walmart, Yelp, Movies, Expedia, and LastFM, and "No" for the other 2 datasets: Books and Flights. In other words, we can expect MorpheusFI to be (slightly) slower than Materialized for Books and Flights in terms of per-epoch runtimes. For completeness sake, we still run our runtime-accuracy experiment for all 7 datasets to verify which predictions hold true.

6.2.1 ***Results: Runtimes.*** Table 4 and Table 5 present the runtimes till convergence and validation accuracy at convergence for the chosen hyper-parameters. We first focus on the runtimes and see three main takeaways:

First, MorpheusFI with LBFGS is significantly faster than the other approaches for many datasets on both ML models: up to 36.1× faster than Adam, up to 4.9× faster than Materialized, and up to 1.7× faster than MorpheusPy. While the speedup against MorpheusPy is relatively lower, this is an artifact of all the real datasets being very sparse. Recall from Figure 5 that for denser data, MorpheusFI can be even faster.

Second, while MorpheusFI is substantially faster than Adam for almost all datasets across both models, Materialized was interestingly slower than Adam in some cases (e.g., logistic regression on Walmart). This means MorpheusFI can actually swap the relative performance trend between SGD and batch gradient methods for multi-table data by avoiding computational redundancy for LA operations.

Third, our heuristic decision rule correctly predicted that MorpheusFI is likely to be slower than Materialized on Books and Flights. This slowdown, albeit minor ($< 2\times$) is seen on both models. Thus, practitioners can easily apply our decision rule to check beforehand if MorpheusFI is likely to benefit them compared to Materialized or MorpheusPy. Interestingly, except for linear SVM on Flights, MorpheusFI is still faster than Adam in all these cases.

6.2.2 ***Results: Accuracy and Convergence Behavior.*** To understand the convergence behavior, Table 4 and Table 5 also report the number of batches/iterations seen till convergence. Note that a batch for LBFGS is the full dataset, but for Adam, it is a mini-batch. Figure 6 also shows the learning curves of validation errors (1− accuracy) over time for logistic regression on 3 datasets for more intuition (due to space constraints, the plots for the other datasets are in the Appendix). We see two main takeaways.

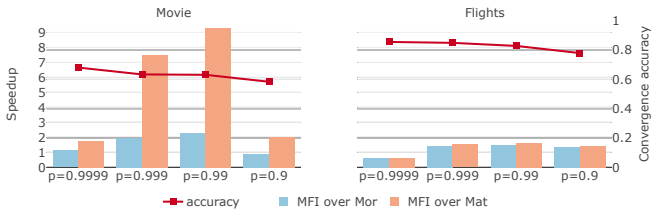

Figure 7: LR speedups and accuracies after feature selections on Movie and Flights

First, the accuracy of LBFGS and Adam are mostly similar for almost all datasets and both models. In a couple of cases, LBFGS has slightly more accuracy (both models on Flights), while there is one case where Adam has slightly more accuracy (linear SVM on Movie). Given that accuracy is similar, time to convergence (previous subsection) and convergence behavior are crucial. On that count, MorpheusFI with LBFGS is substantially faster than Adam, while still yielding similar accuracy. Second, the learning curves of LBFGS with the three tools–MorpheusFI, Materialized, and MorpheusPy– exhibits interesting differences, especially when contrasted with Adam (Figure 6). On Walmart, Adam reduces errors quickly initially but then slows down. MorpheusFI with LBFGS catches up and eventually converges earlier. On Movie, however, Adam exhibits rather noisy behavior, leading to all 3 LBFGS approaches converge earlier. But the most interesting case is Yelp. While Adam converges earlier than both Materialized and MorpheusPy, the faster per-epoch efficiency of MorpheusFI (due to our factorized interaction rewrite rules) inverts the trend for LBFGS and converges earlier than Adam.

6.2.3 ***Results: Runtimes and Sparsity.*** As mentioned before, the real datasets all have sparse features, resulting in relatively lower speedups for MorpheusFI over MorpheusPy. To understand if this gap increase if the features were densers, we include this experiment. We perform an unsupervised feature subset selection on real datasets to reduce sparsity; we filtered out features in dimension tables that do not meet a certain variance threshold; the threshold is $p(1-p)$ and p was varied. Table 6 in the appendix depicts the new reduced dimensions of all tables. Figure 7 compares MorpheusFI with Morpheus on this modified Movie and Flights for logistic regression with LBFGS; other datasets are shown in the appendix due to space constraints. We have two main takeaways:

First, MorpheusFI now reports higher speedups over Morpheus on these denser features, as predicted. In fact, the speedup over Materialized goes up even further. for instance, on Movie, MorpheusFI now reports 9× speedup over Materialized and 3× speedup over MorpheusPy. We also note

an interesting reversal of trend on Flights: MorpheusFI was slower than MorpheusPy and Materialized in Tables 4 and 5, but it is now 2× faster than both MorpheusPy and Materialized. Second, the accuracy is reduced slightly even as feature selection reduced runtimes across the board. This is because accuracy is tied to the richness of the features used, and our low-variance thresholding reduces this richness. This is a classical runtime-accuracy trade-off due to feature subset selection, and data scientists can pick an operating point based on their application-specific constraints.

6.3 Summary and Discussion

Our results show that MorpheusFI can offer substantial speedups for LA operators and LA-based linear ML algorithms over normalized data with non-linear feature interactions. In some corner cases, predictable with our decision rule, materialized execution is slightly faster. Our work generalizes factorized LA along a novel direction and studies hitherto unknown trade-off spaces for ML runtime efficiency. By avoiding materialization of both join and feature interactions, our work reduces memory and storage requirements. As ML analytics increasingly move to the cloud, such memory and runtimes savings can translate to monetary cost savings. While our prototype used PyTorch, our ideas are generic and applicable to other LA frameworks such as TensorFlow or R as well.

7 RELATED WORK

Factorized ML. Our work generalizes the recent line of work on optimizing ML over multi-table data [6, 15, 16, 21, 25, 27]. Most of these works focus on specific ML algorithms or are dependent on specific systems/platforms (e.g., RDBMS). For instance, [21, 27] focus only on linear regression, while [25] focus on a recommendation algorithm called "factorization machine," and [15] studied in-RDBMS factorized ML. Morpheus [6, 19] is the closest to our work in that it decouples LA operators from ML algorithms or system platforms and created a general factorized LA framework. The novelty of our work is that we support and optimize quadratic feature interactions within such a factorized LA setting. Our framework of rewrite rules are novel in tackling the unique double redundancy caused by the interplay of joins and feature interactions. So, our work expands the benefits of factorized ML/LA idea to a larger set of use cases.

LA Systems. There is much prior work on LA systems [3–5, 9, 22, 23, 25, 29, 30, 32]. Some systems build LA operations on top of RDBMSs or dataflow systems, including SystemML [4, 5, 32], RIOT [32], and SimSQL [20]. In particular, SystemML has extensive logical and physical optimizations for LA workloads [8, 9]. All these systems are *complementary* to our work, since none of them focus on multi-table data or address non-LA operations such as feature interactions. It is possible to integrate our framework with any of these scalable LA systems, which we leave for future work. Another lines of work aims to optimize physical layout and compression of matrices. For instance, TileDB [22] employs fragments to manage random rewrites for array data, while SciDB [29] also does blocked array management. These systems support only a limited set of LA operators and do not target ML workloads. A bevy of recent deep learning tools also support LA operators: TensorFlow [3], PyTorch [23], SINGA [31], etc. None of them focus on multi-table data or optimizing feature interactions. They are also *orthogonal* to our focus on logical rewrite rules; our framework can be integrated such tools, as our prototype with PyTorch shows.

Cross-Algebra Optimization. Some recent works [11, 13] combine relational algebra (RA) and linear algebra (LA) for cross-algebra optimization. For instance, TensorDB [13] pushes tensor decomposition through joins and unions. LaraDB [11] proposes a minimal logical abstraction, called associative table, to combine LA and RA. With three basic operators, it captures many functionalities of both RA and LA. Such cross-algebra approaches and their optimizations are *complementary* to our focus on non-linear feature interactions in a factorized LA setting. It is interesting future work to integrate our ideas with such approaches without losing the efficiency gains we see in an LA setting.

8 CONCLUSION AND FUTURE WORK

Factorized ML techniques accelerate ML over multi-table data but have hitherto been restricted to ML algorithms expressible with only linear data transformations. We fundamentally extend this paradigm to support a popular non-linear data transformation: quadratic feature interactions. With a new abstraction and an extensive framework of novel algebraic rewrite rules, our work exploits the peculiar double redundancy caused by the interplay of joins with feature interactions. Our tool, MorpheusFI, offers substantial efficiency gains over approaches that ignore such redundancy. As integration of ML with data management receives more attention, our work shows the benefits of looking beyond LA-only or RA-only algebraic formalisms to optimize end-to-end ML workflows more holistically. As for future work, we plan to support more complex LA operations and integrate our framework with distributed LA systems.

ACKNOWLEDGMENTS

This work was supported in part by a Hellman Fellowship, Google PhD Fellowship, and Faculty Research Awards from Google and Opera Solutions. We thank the members of UC San Diego's Database Lab for their feedback on this work.

REFERENCES

[1] Kaggle Survey: The State of Data Science and Machine Learning. https://www.kaggle.com/surveys/2017.

[2] Scikit-learn Preprocessing: Polynomial Features.

[3] Abadi, M., et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 265–283.

[4] Boehm, M., Dusenberry, M. W., Eriksson, D., Evfimievski, A. V., Manshadi, F. M., Pansare, N., Reinwald, B., Reiss, F. R., Sen, P., Surve, A. C., and Tatikonda, S. Systemml: Declarative machine learning on spark. *Proc. VLDB Endow. 9*, 13 (Sept. 2016), 1425–1436.

[5] Cai, Z., Vagena, Z., Perez, L., Arumugam, S., Haas, P. J., and Jermaine, C. Simulation of database-valued markov chains using simsql. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 637–648.

[6] Chen, L., Kumar, A., Naughton, J., and Patel, J. M. Towards linear algebra over normalized data. *Proc. VLDB Endow. 10*, 11 (Aug. 2017), 1214–1225.

[7] Eide, R. R. E., and Team, C. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *;login:the magazine of USENIX 39*, 6 (2014), 36–38.

[8] Elgamal, T., Luo, S., Boehm, M., Evfimievski, A. V., Tatikonda, S., Reinwald, B., and Sen, P. Spoof: Sum-product optimization and operator fusion for large-scale machine learning. In *CIDR* (2017).

[9] Elgohary, A., Boehm, M., Haas, P. J., Reiss, F. R., and Reinwald, B. Compressed linear algebra for large-scale machine learning. *Proc. VLDB Endow. 9*, 12 (Aug. 2016), 960–971.

[10] Hastie, T., et al. *The Elements of Statistical Learning: Data mining, Inference, and Prediction.* Springer-Verlag, 2001.

[11] Hutchison, D., Howe, B., and Suciu, D. Laradb: A minimalist kernel for linear and relational algebra computation. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond* (New York, NY, USA, 2017), BeyondMR'17, ACM, pp. 2:1–2:10.

[12] Jones, E., Oliphant, T., Peterson, P., et al. SciPy: Open source scientific tools for Python, 2001–.

[13] Kim, M., and Candan, K. S. Tensordb: In-database tensor manipulation with tensor-relational query plans. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management* (New York, NY, USA, 2014), CIKM '14, ACM, pp. 2039–2041.

[14] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *CoRR abs/1412.6980* (2014).

[15] Kumar, A., Jalal, M., Yan, B., Naughton, J., and Patel, J. M. Demonstration of santoku: Optimizing machine learning over normalized data. *Proc. VLDB Endow. 8*, 12 (Aug. 2015), 1864–1867.

[16] Kumar, A., Naughton, J., and Patel, J. M. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1969–1984.

[17] Kumar, A., Naughton, J., Patel, J. M., and Zhu, X. To join or not to join?: Thinking twice about joins before feature selection. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, ACM, pp. 19–34.

[18] Langford, J. Vowpal Wabbit. https://github.com/JohnLangford/vowpal_wabbit/wiki.

[19] Li, S., and Kumar, A. Morpheuspy: Factorized machine learning with numpy. Tech. rep. https://adalabucsd.github.io/papers/TR_2018_MorpheusPy.pdf.

[20] Luo, S., Gao, Z. J., Gubanov, M., Perez, L. L., and Jermaine, C. Scalable linear algebra on a relational database system. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)* (April 2017).

[21] Olteanu, D., and Schleich, M. F: Regression models over factorized views. *Proc. VLDB Endow. 9*, 13 (Sept. 2016), 1573–1576.

[22] Papadopoulos, S., Datta, K., Madden, S., and Mattson, T. The tiledb array data storage manager. *Proc. VLDB Endow. 10*, 4 (Nov. 2016), 349–360.

[23] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NIPS-W* (2017).

[24] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2018.

[25] Rendle, S. Scaling factorization machines to relational data. In *Proceedings of the 39th international conference on Very Large Data Bases* (2013), PVLDB'13, VLDB Endowment, pp. 337–348.

[26] S. Walt, S. C. C., and Varoquaux, G. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering 13*, 2 (March 2011), 22–30.

[27] Schleich, M., Olteanu, D., and Ciucanu, R. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, ACM, pp. 3–18.

[28] Shalev-Shwartz, S., and Ben-David, S. *Understanding Machine Learning: From Theory to Algorithms.* Cambridge University Press, 2014.

[29] Stonebraker, M., Brown, P., Poliakov, A., and Raman, S. The architecture of scidb. In *Proceedings of the 23rd International Conference on Scientific and Statistical Database Management* (Berlin, Heidelberg, 2011), SSDBM'11, Springer-Verlag, pp. 1–16.

[30] Venkataraman, S., et al. Sparkr: Scaling r programs with spark. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, ACM, pp. 1099–1104.

[31] Wang, W., et al. Singa: Putting deep learning in the hands of multimedia users. In *Proceedings of the 23rd ACM International Conference on Multimedia* (2015), MM '15, ACM, pp. 25–34.

[32] Zhang, Y., Herodotou, H., and Yang, J. RIOT: i/o-efficient numerical computing without SQL. *CoRR abs/0909.1766* (2009).

A PROOFS OF REWRITE RULES

We now prove the correctness of our rewrite rules for the cross-interaction part $S \otimes X_1$ of LMM, RMM and Crossprod. For brevity sake, we focus on matrix-vector multiplication over a 2-table join, but our proofs can be easily generalized to matrix-matrix multiplication, and other operators, including for multi-table joins. We first introduce some notation used in the proofs. $X[i, j]$ is the entry in the i^{th} row and j^{th} column of matrix X. Similarly, $W[i]$ is i^{th} element of vector W.

A.1 Proof for LMM

Lemma A.1. *Given R_1 (shape $n_1 \times d_1$), S (shape $n_S \times d_S$), W (shape $d_S d_1 \times 1$), $X_1 \triangleq K_1 R_1$, and $R' \triangleq S \otimes (K_1 R_1)$, $R'W \triangleq rowSum(KR'_1 \odot S)$ where $R'_1 = [R_1 W^1, ..., R_1 W^{d_S}]$.*

Proof. To prove the lemma, we compare the first row in the results of $R'W$ (materialized) and $rowSum(M \odot S)$ (factorized) interaction executions, denoted as m and f respectively. Since W is a vector, both m and f are just scalars. We derive the expressions for both to show their equivalence. Since this line of reasoning applies directly to every row of the respective results, the proof follows directly from this result.

Materialized: $m = \sum_{i=1}^{d_S} \sum_{j=1}^{d_1} S[1,i]X_1[1,j]W[d_1(i-1)+j]$

Factorized: We write out the intermediate matrix R_1':

$$\begin{bmatrix} \sum_{j=1}^{d_1} R_1[1,j]W[j] & \dots & \sum_{j=1}^{d_1} R_1[1,j]W[d_1(d_S-1)+j] \\ \dots & \dots & \dots \\ \sum_{j=1}^{d_1} R_1[n_1,j]W[j] & \dots & \sum_{j=1}^{d_1} R_1[n_1,j]W[d_1(d_S-1)+j] \end{bmatrix}$$

Note that the rewrite then computes $rowSum((K_1R_1') \odot S)$. Since we only need the first entry of this resultant column vector, we only need the expression for the first row of K_1R_1' next. Consider the i^{th} entry of the first row (i goes from 1 to d_S):

$$\begin{aligned}(K_1R_1')[1,i] &= \sum_{l=1}^{n_1} K_1[1,l]R_1'[l,i] \\ &= \sum_{l=1}^{n_1} K_1[1,l] \sum_{j=1}^{d_1} R_1[l,j]W[d_1(i-1)+j]\end{aligned}$$

We then reorder the summation over the indices as follows (note that $X_1 \triangleq K_1R_1$):

$$\begin{aligned}(K_1R_1')[1,i] &= \sum_{j=1}^{d_1} W[d_1(i-1)+j] \sum_{l=1}^{n_1} K_1[1,l]R_1[l,j] \\ &= \sum_{j=1}^{d_1} W[d_1(i-1)+j]X_1[1,j]\end{aligned}$$

Given the above, the full expression for f, which is the first entry of $rowSum((K_1R_1') \odot S)$, is as follows:

$$\begin{aligned}f &= \sum_{i=1}^{d_S} (K_1R_1')[1,i]S[1,i] \\ &= \sum_{i=1}^{d_S} \sum_{j=1}^{d_1} W[d_1(i-1)+j]X_1[1,j]S[1,i] \\ &= m \quad \square\end{aligned}$$

We have proved the lemma. Now we prove that LMM works end-to-end.

$$\begin{aligned}&\hat{S}\hat{W}_S + K_1\hat{R}_1\hat{W}_R + rowSum(K_1\hat{R}_1' \odot \hat{S}) \\ &\quad \text{where} \quad R_1' = [\hat{R}_1W_{SR}^1, ..., \hat{R}_1W_{SR}^{d_S}] \\ =&\hat{S}\hat{W}_S + K_1\hat{R}_1\hat{W}_R + \hat{R}'\hat{W}_{SR} \\ =&(\hat{S} + K_1\hat{R}_1 + \hat{R}')\hat{W} \\ =&(\hat{S} + K_1\hat{R}_1 + \hat{S} \otimes K_1\hat{R}_1)\hat{P}W = inter(T)W\end{aligned}$$

□

A.2 Proof for RMM

Lemma A.2. *Given R_1 (shape $n_1 \times d_1$), S (shape $n_S \times d_S$), W (shape $1 \times n_S$), $X_1 \triangleq K_1R_1$, and $R' \triangleq S \otimes (K_1R_1)$, $WR' \triangleq colSum(K_1^T(W^T \otimes S) \otimes R_1)$.*

Proof. To prove the lemma, we compare the first column in the results of materialized and factorized interaction executions, denoted as m and f respectively. Since W is a vector, both m and f are just scalars. We derive the expressions for both to show their equivalence. As with LMM, this line of reasoning applies to every column and so, the proof follows.

Materialized: $m = \sum_{i=1}^{n_S} W[i]S[i,1]X_1[i,1]$

Factorized: We write out the intermediate matrix S':

$$S' = \begin{bmatrix} S[1,1]W[1] & \dots & S[1,d_S]W[1] \\ \dots & \dots & \dots \\ S[n_S,1]W[n_S] & \dots & S[n_S,d_S]W[n_S] \end{bmatrix}$$

Note the rewrite then computes $colSum((K_1^TS') \otimes R_1)$. Since we only need the first entry of this resultant row vector, we only need the expression for the first column of K_1^TS'. Consider the j^{th} entry of the first column ($j = 1$ to n_1):

$$\begin{aligned}(K_1^TS')[j,1] &= \sum_{i=1}^{n_S} K_1^T[j,i]S'[i,1] \\ &= \sum_{i=1}^{n_S} K_1^T[j,i]S[i,1]W[i]\end{aligned}$$

Given the above, the full expression for f, which is the first entry of $colSum(U \times R_1)$, is as follows:

$$\begin{aligned}f &= \sum_{j=1}^{n_1} (K_1^TS')[j,1]R_1[j,1] \\ &= \sum_{j=1}^{n_1} \sum_{i=1}^{n_S} K_1^T[j,i]S[i,1]W[i]R_1[j,1]\end{aligned}$$

Reordering the sum gives the following ($X_1 \triangleq K_1R_1$):

$$\begin{aligned}f &= \sum_{i=1}^{n_S} S[i,1]W[i] \sum_{j=1}^{n_1} K_1^T[j,i]R_1[j,1] \\ &= \sum_{i=1}^{n_S} S[i,1]W[i]X_1[i,1] \\ &= m \quad \square\end{aligned}$$

We have proved the lemma. Now we prove that RMM works end-to-end.

$$
\begin{aligned}
&[W\hat{S}, WK_1\hat{R}_1, colSum(K_1^T(W^T \otimes \hat{S}) \otimes \hat{R}_1)] \\
=&[W\hat{S}, WK_1\hat{R}_1, W(\hat{S} \otimes K_1\hat{R}_1)] \\
=&W[\hat{S}, K_1\hat{R}_1, \hat{S} \otimes K_1\hat{R}_1] = W\,inter(T)
\end{aligned}
$$

□

A.3 Proof for Crossprod

Lemma A.3. *Given R_1 (shape $n_1 \times d_1$), S (shape $n_S \times d_S$), W (shape $1 \times n_S$), $X_1 \triangleq K_1R_1$, and $R' \triangleq S \otimes (K_1R_1)$, $cp(S \otimes K_1R_1) \triangleq reshape((R_1 \otimes R_1)^T(K_1^T(S \otimes S))$*

Proof. To prove the lemma, we compare the set of entries (without considering the ordering of entries) in materialized and factorized executions, denoted as m and n respectively.
Materialized:

$$
\begin{aligned}
m = \{&\sum_{k=1}^{n_S} S[k,i]X_1[k,j]S[o,a]X_1[o,b] \;\mid\; 1 \le i \le d_S, \\
&1 \le j \le d_1, 1 \le a \le d_S, 1 \le b \le d_1\}
\end{aligned}
$$

Factorized: We write out the intermediate results S' by aggregating rows in $S \otimes S$ with K_1^T such that $S' = K_1^T(S \otimes S)$. As with RMM, the j-th row in S' is:

$$
\begin{aligned}
S'[j,:] &= K_1^T(S \otimes S) \\
&= [\sum_{j=1}^{n_1}\sum_{i=1}^{n_S} K_1^T[j,i]S[i,1]S[i,1], \\
&\quad \ldots, \\
&\quad \sum_{j=1}^{n_1}\sum_{i=1}^{n_S} K_1^T[j,i]S[i,d_S]S[i,d_S]]
\end{aligned}
$$

Performing a matrix multiplication with $R_1 \otimes R_1$ gives:

$$
f = \{\sum_{k=1}^{n_1} R_1[k,i]R_1[k,j]S'[k,m] \;\mid\; 1 \le i,j \le d_1, 1 \le m \le d_S^2\}
$$

Because K has preaggregated rows of $S \otimes S$, we can expand it in the expression as well expand R_1 to X_1 such that:

$$
\begin{aligned}
f &= \{\sum_{k=1}^{n_1} R_1[k,i]R_1[k,j]S'[k,m] \;\mid\; 1 \le i,j \le d_1, 1 \le m \le d_S^2\} \\
&= \{\sum_{k=1}^{n_S} S[k,i]S[o,a]X_1[k,j]X_1[o,b] \;\mid\; 1 \le i \le d_S, \\
&\quad 1 \le j \le d_1, 1 \le a \le d_S, 1 \le b \le d_1\} \\
&= m \quad \square
\end{aligned}
$$

Without considering the order of entries in the result matrices, both executions are equivalent. We apply *reshape* function to modify factorized executions to align to the correct ordering. We have therefore proved the lemma. As such, we can prove the whole crossprod works:

$$
crossprod(inter(T)) = \begin{bmatrix} cp(\hat{S}) & P_1^T & P_2^T \\ P_1 & cp(Q) & P_3^T \\ P_2 & P_3 & cp(R') \end{bmatrix}
$$

where

$$
P_1 = \hat{R}_1^T(K_1^T\hat{S})\;;\; P_2 = \hat{R}_1'^T\hat{S} = (\hat{S}^T\hat{R}_1')^T
$$

$$
P_3 = \hat{R}_1'^T(K_1\hat{R}_1) = (\hat{R}_1^T(K_1^T(S \otimes X_1)))^T = \left(\hat{R}_1^T\left(\left(K_1^TS\right) \otimes R_1\right)\right)^T
$$

$$
Q = (diag(colSums(K)))^{\frac{1}{2}}\hat{R}_1\;;
$$

$$
cp(\hat{R}_1') = cp(S \otimes (K_1R_1)) = reshape\left((R_1 \otimes R_1)^T\left(K_1^T(S \otimes S)\right)\right)
$$

□

B FACTORIZED ALGORITHMS

Algorithms 1 and 2 present logistic regression, Algorithms 3 and 4 present linear SVM, and Algorithms 5 and 6 present ordinary least squares linear regression. While we show the simple BGD method for the classifiers for exposition sake, LBFGS has the same data access pattern.

C MORE EXPERIMENTAL RESULTS

Figure 9 presents the convergence behavior learning curves for logistic regression on the remaining 4 real datasets. We skip the convergence behavior learning curves for linear SVM here due to space constraints; they showed similar trends. Table 6 present data dimensions after feature selections on all real-world datasets. Figure 8 shows visualized speedups and accuracies tradeoffs on Expedia and Yelp.

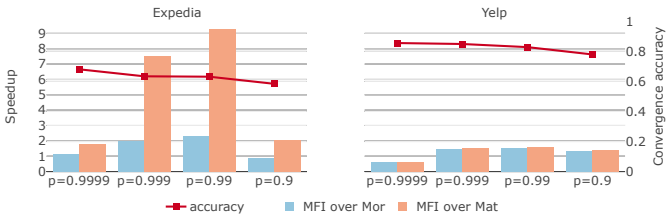

Figure 8: LR speedups and accuracies after feature selections on Expedia and Yelp

Algorithm 1: Standard Logistic Regression

Data: Regular Matrix T , Y , W
for *i in* 1 : *max_iter* **do**
  $W = W + \alpha * (T^T(Y/(1 + exp(TW))))$
end

	p=0.9999	p=0.999	p=0.99	p=0.9
Dataset	(n_i, d_i), R_i, sparisty nnz	(n_i, d_i), R_i, sparisty nnz	(n_i, d_i), R_i, sparisty nnz	(n_i, d_i), R_i, sparisty nnz
Yelp	(11537, 93) 0.3441 369166 (43873, 7) 0.8571 263238	(11537, 73) 0.4245 357531 (43873, 7) 0.8571 263238	(11537, 55) 0.5443 345398 (43873, 7) 0.8571 263238	(11537, 22) 0.7971 202307 (43873, 7) 0.8571 263238
Movie	(6040, 3463) 0.0012 24160 (3706, 120) 0.1750 77826	(6040, 109) 0.0288 18965 (3706, 112) 0.1875 77811	(6040, 23) 0.1303 18103 (3706, 59) 0.3521 76981	(6040, 6) 0.3926 14229 (3706, 11) 0.5455 22236
Expedia	(11939, 41) 0.1951 95510 (37021, 1308) 0.0106 515395	(11939, 34) 0.2352 95463 (37021, 292) 0.0467 504537	(11939, 14) 0.5664 94676 (37021, 85) 0.1536 483356	(11939, 9) 0.8629 92718 (37021, 27) 0.4202 420020
LastFM	(4999, 12) 0.5833 34993 (50000, 142) 0.0281 199816	(4999, 12) 0.5833 34993 (50000, 65) 0.0612 198770	(4999, 12) 0.5833 34993 (50000, 30) 0.1277 191589	(4999, 10) 0.6000 29994 (50000, 8) 0.3809 152375
Books	(27876, 57) 0.0350 55652 (49972, 922) 0.0042 195664	(27876, 25) 0.0795 55407 (49972, 216) 0.0172 185882	(27876, 9) 0.21551 54066 (49972, 41) 0.0782 160224	(27876, 2) 0.8396 46811 (49972, 2) 1.0000 99944
Flights	(540, 178) 0.0281 2700 (3182, 3301) 0.0018 19092 (3182, 3301) 0.0018 19092	(540, 178) 0.0280 2700 (3182, 123) 0.0402 15740 (3182, 123) 0.0402 15740	(540, 35) 0.1293 2443 (3182, 38) 0.1222 14779 (3182, 38) 0.1222 14779	(540, 7) 0.4219 1595 (3182, 12) 0.3397 12971 (3182, 12) 0.3397 12971

Table 6: Data dimensions after feature selection. Walmart is omitted, since its dimension tables have few features.

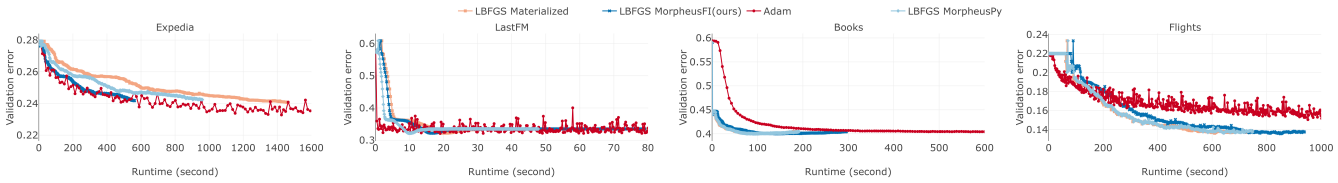

Figure 9: Logistic Regression convergence behaviors.

Algorithm 2: Factorized Interacted Logistic Regression

Data: Normalized matrix (S, K, R) , Y , W
for i *in* 1 : max_iter **do**
$W = [W_S, W_{SS}, W_1, W_{11}, W_{S1}]$;
$W_{S1} = [W_{S1}^1, W_{S1}^2, ..., W_{S1}^{d_S}]$;
$P = (Y/(1 + exp(SW_S + K_1(R_1W_1) + (\mathring{S})W_{SS} + K_1(\mathring{R}_1W_{11}) + rowSum(K_1[R_1W_{S1}^1, ..., R_1W_{S1}^{d_S}] \odot S))))^T$;
$W = W + \alpha[PS, P\mathring{S}, (PK_1)R_1, PK_1\mathring{R}_1, colSum(K_1^T(P^T \otimes S) \otimes R_1)]^T$;
end

Algorithm 3: Standard Linear SVM

Data: Regular Matrix T , Y , W
for i *in* 1 : max_iter **do**
$P = sign(max(0, 1 - Y \odot TW))$;
$W = W + (Y \odot P)^T T$;
end

Algorithm 4: Factorized Interacted Linear SVM

Data: Normalized matrix (S, K, R), Y , W
for i *in* 1 : max_iter **do**
$W = [W_S, W_{SS}, W_1, W_{11}, W_{S1}]$;
$W_{S1} = [W_{S1}^1, W_{S1}^2, ..., W_{S1}^{d_S}]$;
$Q = SW_S + K_1(R_1W_1) + (\mathring{S})W_{SS} + K_1(\mathring{R}_1W_{11}) + rowSum(K_1[R_1W_{S1}^1, ..., R_1W_{S1}^{d_S}] \odot S)$;
$P = sign(max(0, 1 - Y \odot Q))$;
$O = (Y \odot P)^T$;
$W = W + [OS, O\mathring{S}, (OK_1)R_1, OK_1\mathring{R}_1, colSum(K_1^T(O^T \otimes S) \otimes R_1)]$;
end

Algorithm 5: Standard Linear Regression

Data: Regular matrix (S, K, R) , Y
$w = ginv(crossprod(S, K, R))((S, K, R)^T Y)$

Algorithm 6: Factorized Interacted Linear Regression

Data: Normalized matrix (S, K, R) , Y
$P = ginv(Crossprod(S, R, K))$;
$W = P[Y^TS, Y^T\mathring{S}, (Y^TK_1)R_1, Y^TK_1\mathring{R}_1, colSum(K_1^T(Y \otimes S) \otimes R_1)]^T$;