Efficient Construction of Nonlinear Models over Normalized Data

Zhaoyue Cheng
University of Toronto
cheng.zhaoyue@cs.toronto.edu

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

Zhe Zhang
York University
zhezhang@yorku.ca

Xiaohui Yu
York University
xhyu@yorku.ca

***Abstract*—Machine Learning (ML) applications are proliferating in the enterprise. Relational data which are prevalent in enterprise applications are typically normalized; as a result, data has to be denormalized via primary/foreign-key joins to be provided as input to ML algorithms. In this paper, we study the implementation of popular nonlinear ML models, Gaussian Mixture Models (GMM) and Neural Networks (NN), over normalized data addressing both cases of binary and multi-way joins over normalized relations.**

For the case of GMM, we show how it is possible to decompose computation in a systematic way both for binary joins and for multi-way joins to construct mixture models. We demonstrate that by factoring the computation, one can conduct the training of the models much faster compared to other applicable approaches, without any loss in accuracy.

For the case of NN, we propose algorithms to train the network taking normalized data as the input. Similarly, we present algorithms that can conduct the training of the network in a factorized way and offer performance advantages. The redundancy introduced by denormalization can be exploited for certain types of activation functions. However, we demonstrate that attempting to explore this redundancy is helpful up to a certain point; exploring redundancy at higher layers of the network will always result in increased costs and is not recommended.

We present the results of a thorough experimental evaluation, varying several parameters of the input relations involved and demonstrate that our proposals for the training of GMM and NN yield drastic performance improvements typically starting at 100%, which become increasingly higher as parameters of the underlying data vary, without any loss in accuracy.

I. Introduction

Machine learning (ML) applications in enterprise settings are increasingly becoming mission-critical. As a result, numerous projects both in industry and academia integrate ML techniques in RDBMS, Spark and a variable of other production systems [27]. ML algorithms however have been developed on the assumption that data is readily available on a single input source, with the right formats and encodings. Such effective integration of ML technology in critical systems faces a data representation mismatch. Relational data are typically normalized [4] and as a result, data has to be denormalized via primary/foreign-key joins and materialized as a single (temporary) relation to be provided as an input to the ML algorithms. Therefore, the learning process commences after joins have been performed on the relations involved.

Consider the example of an analyst modeling customer shopping trends in a store. The analyst builds a model utilizing order details: Orders(OrderID, CustomerID, ItemID, Time, Amount) where ItemID is a foreign key referring to a new table that stores the items sold by the store: Items(ItemID, Price, Size, Colour, Category). A join is required between the two tables because some of the information in the Items table, e.g. Price, Size and Colour are essential features of a predictive model on buying patterns. After the join, the tables are materialized to be a temporary table that has to be provided as an input to the various ML algorithms utilized for predictive analysis. Examples like this abound in data analytics: in a video streaming company building recommendation models one has to join user viewing history with video information; at a banking application, building fraud detection models or conducting soft customer segmentation requires a join of customer purchasing/spending records with merchant data; in review sites, one has to associate via a join the reviewer meta-data with their reviews for user modeling applications, etc.

Numerous problems arise when building ML models after a primary/foreign-key join. Firstly, normalization removes redundancy in the data, so in the temporary materialized table, redundancy is reintroduced into the table [4]. As a result, this imposes increased storage requirements in the temporary materialized table. Moreover, it increases computation costs since redundancy introduces additional computation during the training processes. Essentially computations have to be repeated for each redundant set of attribute values. Furthermore, maintaining the materialized table when the base tables evolve for continuous learning applications introduces unnecessary overheads as well. Lastly, ML is commonly used for data analytics, which is often an exploratory process utilizing different slices of the data resulting from joining various base tables. Conducting such joins for every exploration is time-consuming and should be avoided [27].

Kumar et al. [22] recognized these issues and proposed specific algorithms to build generalized linear models and execute various linear algebra operations by pushing ML computations through joins to base tables. That way to the extent possible for the specific class of ML models considered, they demonstrated performance advantages in certain scenarios. Yang et al. [33] investigated the case of learning support vector machines with Gaussian kernels over normalized data. We are not aware of any work considering the general case of Gaussian Mixture Models (GMM) [26], various forms of Neural Networks (NN) and Deep Neural Networks (DNN) [15] executed over normalized input. GMM abound in various modeling tasks; it is an established method to model complex data spaces with diverse multidimensional characteristics. In addition, GMM is prevalent in financial analysis, quantitative finance, astronomy [23] as well as banking applications especially dealing with the returns of asset classes [26].

For these reasons, we focus on GMM and NN over normalized data sources and investigate the extent to which we can push the computation through the joins to offer performance advantages. To address the problems raised above, firstly, it is essential to reduce the I/O cost by removing the step of materializing the join result due to its large size. Secondly, we have to investigate how to utilize factorization to eliminate redundant calculations to save computation time. From a technical standpoint, the important issue is how to execute these popular ML algorithms while removing redundancy and saving costs without reducing the quality of the outcomes or the scalability of the algorithms.

In this paper, we compare three different applicable algorithms to construct such models. One alternative approach to materializing the join results is to compute the joins in batches on the fly. Our proposed algorithm factorizes the model and pushes the computation through join without redundancy. It not only reduces I/O cost avoiding unnecessary storage but also saves time by eliminating the repeated calculations. Various trade-offs among three algorithms are analyzed. Furthermore, the proposal is derived for the case of binary join, suitably generalizing to multi-way joins as well. All the algorithms proposed can deliver the correct models with the same quality as the original models thanks to the exact decomposition introduced.

In particular, for the most general case of GMM, we propose a factorized algorithm named *F-GMM* over data from normalized relations. We use the EM algorithm to train the parameters during E-step and M-step utilizing the reused calculation effectively. For the case of NN, the proposed algorithm *F-NN* demonstrates the BP algorithm can be decomposed to take normalized data into account. Large benefits can be achieved between the input layer and the first hidden layer during forward and backward propagation. We however do not continue pursuing this at higher layers of the network, because depending on the activation function we can no longer guarantee the exactness of the decomposition.

To our knowledge, this is the first work that deals with the most general form of GMM and NN in the context of normalized data. In summary, our work makes the following contributions:

- For GMM, we present three algorithms (*M-GMM*, *S-GMM* and *F-GMM*) and analyze their performance for binary and multi-way joins. In particular, *F-GMM* is a novel technique utilizing the normalized data to explore the reuse of computation.
- As for NN, three algorithms (*M-NN*, *S-NN* and *F-NN*) are proposed in a similar way. We explore the opportunity for algorithm *F-NN* to use normalized data directly during the forward and backward propagation training phases at the first layer to improve performance. Similarly, we propose solutions for both the binary and multi-way join case.
- In addition, for the case of NN, we analyze the impact of different activation functions on sharing computations during the training phase and investigate the computation cost at higher layers of the network.
- We present the results of a thorough experimental evaluation testing the impact of dataset parameters on the performance of the proposed algorithms and quantify the benefits. We also utilize publicly available datasets demonstrating impressive performance improvement in real scenarios.

This paper is organized as follows. Section II reviews related work. In Section III, we present background material for GMM and NN as well as the notation required for what follows. Section IV, formally defines the problems we focus on in this paper. In Section V, we present our solution for GMM followed by section VI in which we present our proposed solutions for training NN. In Section VII, we present the results of a thorough experimental evaluation of the proposed approaches. Finally, Section VIII concludes the paper and discusses avenues for future work in this area.

II. Related Work

Recently Cheng and Koudas [10] presented an algorithm to factorize construction over normalized data focusing on the restricted case of Independent Gaussian Mixture Models(IGMM). The work herein presents a significant generalization to the case of general GMM making no statistical assumptions on the properties of the underlying Gaussians and presents a comprehensive treatment of NN models.

Factorized databases [3] were proposed with the basic idea to represent relations with join dependencies using algebraically equivalent forms that store less data physically.

Extending this work, [21], [31], [32] apply factorized database ideas to ML computation, utilizing primary/foreign-key joins over relations scaling the work to apply in cases where data do not fit in memory. They focus on a general class of linear models (such as logistic regression) which are applicable in certain ML scenarios. Similarly, [29] present a factorized framework and demonstrate its applicability to regressions. More specifically, given a feature vector (x) in the table generated from primary/foreign-key joins of two relations (S and R), using logistic regression as an example, we need to calculate $w^T x$ where w is the parameter for the model. Since the table after joins introduces redundancy, factorized learning can reduce the computations in most cases by calculating $w_S^T x_S + w_R^T x_R$ on the relations before executing the join, where x_S and x_R are the features from S and R respectively. This offers performance benefits over approaches that materialize the join results. Shah et al. [30] present experimental evidence showing that in some cases avoiding joins has little impact in classification accuracy. Their study however includes datasets in which joins are required for improved accuracy and our work has applicability in all such cases.

In the same general thread, there exist recent works on scalable ML and data mining algorithms [12], [7], [2], [6], [25], [16], [8], [18], [17]. The main emphasis of such works is on the efficient implementation of scalable ML algorithms on a data management platform or their effective execution over programming paradigms. We focus on the specific implementation of nonlinear operations used factorization ideas. Another related research thread includes the implementation of linear algebra systems on data management systems [1], [24]. There is increasing interest in building systems with the aim to achieve closer integration of ML with data management [1], [6], [9], [14], [20], [19], [13]. In the context of nonlinear models, [28] is concerned with Bayesian Markov Chain Monte Carlo as applied to Factorization Machine (FM) models and [33] present the algorithms for Support Vector Machine (SVM). In our case, we focus on factorizing the training processes of GMM and NN over normalized data.

III. Background and Preliminaries

In this section, we will present the material and notation necessary for the remainder of the paper.

A. Gaussian Mixture Models (GMM) and EM Algorithm

GMM [26] is a model comprising a fixed number of Gaussian distributions used for data clustering. Assume we are given N training data points $\mathbf{x}^{(n)}, 1 \leq n \leq N$ of dimension d. The distribution of a mixture of K Gaussian components is $p(\mathbf{x}^n) = \sum_{k=1}^{K} \pi_k N(\mathbf{x}^{(n)}|\mu_k, \Sigma_k)$, where π_k are the mixing coefficients s.t. $\sum_{k=1}^{K} \pi_k = 1$. The probability density function of the k^{th} Gaussian component of the mixture model is:

$$N(\mathbf{x}^{(n)}|\mu_k, \Sigma_k) = \frac{1}{\sqrt{(2\pi)^d |\Sigma_k|}} e^{-\frac{1}{2}(\mathbf{x}^{(n)}-\mu_k)^T \Sigma_k^{-1} (\mathbf{x}^{(n)}-\mu_k)} \tag{1}$$

There are a number of algorithms that can be applied for iteratively training GMM. However, the Expectation-Maximization (EM) algorithm [11] is the most widely used. EM is an iterative method to identify the maximum likelihood when the model contains an unobserved latent variable. The algorithm iteratively converges to a local minimum. The EM algorithm starts with some initial estimate of the parameters and then iteratively updates the parameters until convergence. Each iteration consists of one E-step and one M-step.

In the E-step, the posterior distribution of the latent variable $z^{(n)}$ for each observation is updated using the current parameters μ_k, Σ_k and π_k.

$$\gamma_k^{(n)} = p(z^{(n)} = k|\mathbf{x}) = \frac{\pi_k N(\mathbf{x}^{(n)}|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j N(\mathbf{x}^{(n)}|\mu_j, \Sigma_j)} \tag{2}$$

where $z \sim Categorical(\pi)$. Essentially $\gamma_k^{(n)}$ are our "soft" guesses for the values of $p(z^{(n)} = k|\mathbf{x})$ at this step.

In the M-step, we re-estimate the parameters using Maximum Likelihood Estimation (MLE) given the current $\gamma_k^{(n)}$, for all values of k and n:

$$\mu_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_k^{(n)} \mathbf{x}^{(n)} \tag{3}$$

$$\Sigma_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_k^{(n)} (\mathbf{x}^{(n)} - \mu_k)(\mathbf{x}^{(n)} - \mu_k)^T \tag{4}$$

$$\pi_k = \frac{N_k}{N} \quad \text{with} \quad N_k = \sum_{n=1}^{N} \gamma_k^{(n)} \tag{5}$$

GMM is iteratively updated by E-step and M-step until convergence criteria are met. One of the commonly used criteria for checking convergence is the difference in the following log-likelihood between two iterations is less than a certain threshold.

$$\ln p(\mathbf{x}|\pi, \mu, \Sigma) = \sum_{n=1}^{N} ln(\sum_{k=1}^{K} \pi_k N(\mathbf{x}^{(n)}|\mu_k, \Sigma_k)) \tag{6}$$

B. Neural Networks (NN) and BP Algorithm

NN [5] is a supervised model which recently gained popularity with the advent of deep learning [15]. The BP algorithm is the standard method for training NN. It utilizes the current parameters (weights and biases) to compute the error and uses the gradient descent method to propagate the error back to update the parameters. Due to space limitations, we only introduce basic notation and refer the reader to [5] for a comprehensive treatment.

A basic NN can be described as a sequence of linear transformations. Assuming x_i where $i \in \{1 \dots d\}$ is input feature of one instance, we construct linear combinations of the input as: $a_j = \sum_{i=1}^{d} w_{ji}^{(1)} x_i + b_j^{(1)}$ where $j \in \{1 \dots n_h\}$. The superscript (1) denotes the corresponding parameters at the first hidden layer of the network and n_h is the number of the hidden units at this layer. Parameter $w_{ji}^{(1)}$ where $i \in \{1 \dots d\}, j \in \{1 \dots n_h\}$ is the weight between the input feature x_i and the hidden unit h_j. $b_j^{(1)}$ is the bias for hidden unit h_j in the first hidden layer. The value of a_j is transformed via a differentiable activation function f to get $h_j = f(a_j)$ as the output of one hidden unit. Examples of f include the Sigmoid function $\sigma(a) = \frac{1}{1+exp(-a)}$ and the Rectified Linear Unit (ReLU) function $ReLU(a) = max(0, a)$. The outputs of the hidden units at the first layer are combined to synthesize the second hidden layer: $z_k = \sum_{j=1}^{n_h} w_{kj}^{(2)} h_j + b_k^{(2)}$ where $k \in \{1 \dots n_l\}$.

IV. Problem description

We now introduce formally the problems we focus in this paper. Following the style presented in [22], there are two relations **S** ($\underline{SID}$, X_S, FK) and **R** ($\underline{RID}$, X_R) with a primary/foreign-key relationship (**S**.FK refers to **R**.RID), where X_S (X_R) is the name of feature matrix $\mathbf{x}_S$ ($\mathbf{x}_R$). We assume that relation **S** has n_S tuples and relation **R** has n_R tuples satisfying $n_S > n_R$. There are d_S features in n-th feature vector $\mathbf{x}_S^{(n)}$ and $d_R = d - d_S$ features in $\mathbf{x}_R^{(n)}$. When learning a GMM over the result of the projected equi-join

TABLE I: Notations used in the paper

Symbol	Meaning
$\mathbf{R}$	Relation
$\mathbf{S}$	Relation
$\mathbf{T}$	Join result table
Y	Target
n_R	Number of tuples in $\mathbf{R}$
n_S	Number of tuples in $\mathbf{S}$
N	Number of tuples in $\mathbf{T}$ ($N = n_S$)
d_R	Number of features in $\mathbf{R}$
d_S	Number of features in $\mathbf{S}$
d	Number of features in $\mathbf{T}$ ($d = d_R + d_S$)
$\mathbf{x}_R$	Feature matrix from $\mathbf{R}$
$\mathbf{x}_S$	Feature matrix from $\mathbf{S}$
$\mathbf{x}^{(n)}$	n-th feature vector in $\mathbf{T}$
$x_i^{(n)}$	i-th feature in $\mathbf{x}^{(n)}$

$\mathbf{T}(\underline{SID}, [X_S\ X_R]) \leftarrow \pi_{SID,X_S,X_R}$ ($\mathbf{R} \bowtie_{RID=FK}$ $\mathbf{S}$), the feature vector of a tuple in $\mathbf{T}$ is the concatenation of the feature vectors from the joining tuples of $\mathbf{S}$ and $\mathbf{R}$. For the case of NN, relation $\mathbf{S}$ ($\underline{SID}$, Y, X_S, FK) has an additional attribute Y which is the target for learning purposes (the projected schema becomes $\mathbf{T}$ ($\underline{SID}$, Y, $[X_S\ X_R]$)). Table I summarizes the notations used in this paper.

The multi-way join case follows similarly. We are provided q attribute tables $\mathbf{R}_i(\underline{RID_i}, X_{R_i})$, $i \in \{1 \dots q\}$ and a table $\mathbf{S}(\underline{SID}, X_S, FK_1, \dots, FK_q)$ with q foreign keys. We are interested to learn a GMM over the projected equi-join T ($\underline{SID}$, $[X_S\ X_{R_1}\ \dots\ X_{R_q}]$) $\leftarrow \pi_{\underline{SID},X_S,X_{R_1}\dots X_{R_q}}$ ($\mathbf{R}_1 \bowtie_{RID_1=FK_1}$ $\dots$ $\mathbf{R}_q \bowtie_{RID_q=FK_q}$ $\mathbf{S}$). Similarly, target Y will be added to relation $\mathbf{S}$ and $\mathbf{T}$ for the training of NN.

Table $\mathbf{T}$ introduces redundancy back to the data representation which normalization removed in the first place. Depending on the redundancy introduced trade-offs may exist which we will explore in our ensuing discussion.

For purposes of exposition, we assume that the required joins execute in a block nested loops fashion. Our proposals are equally applicable when other types of joins are adopted such as partitioned hash joins.

V. General Gaussian Mixture Model Algorithm

We assume the most general case for GMM with arbitrary covariance matrices. This section starts by introducing the baseline materializing method and an improved algorithm, then proposes the decomposition of the various computations involved that constitutes the basis of our proposal, algorithm *F-GMM*.

A. Baseline Approaches

Algorithm *M-GMM* is the baseline solution widely used by analysts currently; it computes the join of the relations involved, materializes the results on the disk and subsequently reads them executing the EM algorithm to complete training. The algorithm is depicted as Algorithm 1.

Algorithm *S-GMM* is another baseline approach that computes the join of the relations involved on the fly without materializing the join result on the disk and executes the EM algorithm directly. This algorithm is essentially the same as Algorithm 1; however, it does not perform Line 1 which is

Algorithm 1 Algorithm *M-GMM*

```
1:  Apply join S and R and materialize the table T after join in the database
2:  repeat
3:      E-step:
4:      for $i \leq$ number of batches do
5:          Read batch $i$ of T into memory
6:          Update responsibility of data points in this batch
7:          $\gamma_k^{(n)} \leftarrow \frac{\pi_k N(\mathbf{x}^{(n)}|\mu_k,\Sigma_k)}{\sum_{j=1}^{K} \pi_j N(\mathbf{x}^{(n)}|\mu_j,\Sigma_j)}$    $\forall n \in$ batch $i$
8:      end for
9:      M-step:
10:     for $i \leq$ number of batches do
11:         Read batch $i$ of T into memory
12:         Add to the sum results from this batch
13:         $Sum_{\mu_k} += \sum_{n=1}^{|batch\ size\ i|} \gamma_k^{(n)} \mathbf{x}^{(n)}$
14:     end for
15:     Update $\mu_k \leftarrow \frac{1}{N_k} Sum_{\mu_k}$
16:     for $i \leq$ number of batches do
17:         Read batch $i$ of T into memory
18:         Add to the sum results from this batch
19:         $Sum_{\Sigma_k} += \sum_{n=1}^{|batch\ size\ i|} \gamma_k^{(n)} (\mathbf{x}^{(n)} - \mu_k)(\mathbf{x}^{(n)} - \mu_k)^T$
20:     end for
21:     Update $\Sigma_k \leftarrow \frac{1}{N_k} Sum_{\Sigma_k}$
22:     Update $\pi_k \leftarrow \frac{N_k}{N}$ with $N_k \leftarrow \sum_{n=1}^{N} \gamma_k^{(n)}$
23: until Convergence
```

materializing the table $\mathbf{T}$ in the database. In addition, in Lines 5, 11, 17, instead of reading i-th batch from table $\mathbf{T}$, it is accomplished by reading the i-th batch of $\mathbf{R}$ and retrieving tuples from $\mathbf{S}$ using the primary/foreign-key relationship to get the i-th sub-table of the join in the memory as the input to compute the GMM on the fly. Figure 1(a) and (b) present the overview of the two baseline approaches. *S-GMM* naturally generalizes to multi-way joins.

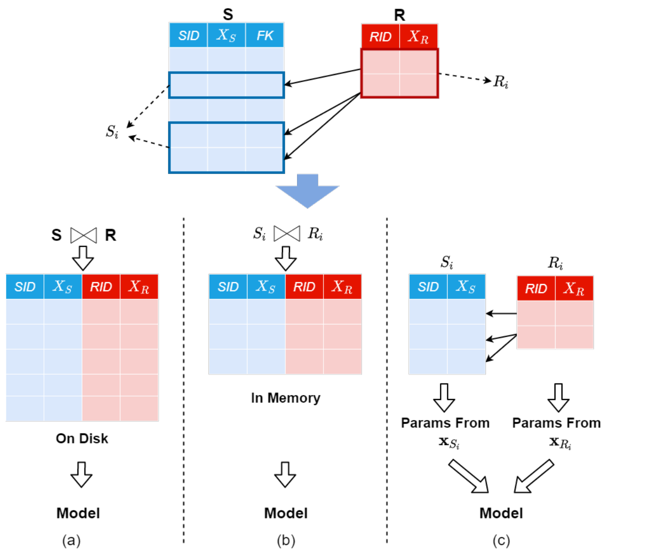


Fig. 1: (a) *M-GMM* (b) *S-GMM* (c) *F-GMM*

Here we analyze the cost for the baseline approaches according to Algorithm 1. We use $|S|$, $|R|$, $|T|$ to represent the number of pages in each table. For the case of *M-GMM*, the total I/O cost includes computing block nested loops join of **S** and **R** ($|R| + \frac{|R|}{BlockSize}|S|$), materializing the table **T** ($|T|$) and executing EM iteratively for $iter$ times where each iteration will read **T** 3 times ($3 \times iter \times |T|$). For the case of *S-GMM*, reading **T** is replaced by executing joins on the fly. Thus, the total I/O cost is $3 \times iter \times (|R| + \frac{|R|}{BlockSize}|S|)$. Which of these two approaches has less I/O cost depends on the size of tables and $BlockSize$. When $BlockSize > \frac{(3\times iter-1)|R||S|}{(3\times iter+1)|T|-(3\times iter-1)|R|}$, *S-GMM* has less I/O cost. On the other hand, for each iteration of EM, all the tuples generated from joins must participate in the calculation of each parameter whether they are materialized or not. In other words, there is no difference in terms of computation cost between them.

B. Algorithm F-GMM for Binary Joins

Algorithm *F-GMM* computes the parameters of GMM in a factorized way. The algorithm is derived from *M-GMM* and *S-GMM* as follows. As in *S-GMM*, materializing the table (Line 1) is not required; for Lines 5, 11, 17, relation **R** is processed in batches, probing **S** for matching tuples using the primary/foreign key. The difference from *S-GMM* is that, instead of feeding every joined tuple to the network for parameter updates in Lines 7, 13, 19, we factorize the equations into two parts involving $\mathbf{x}_S$ and $\mathbf{x}_R$ respectively. As illustrated by Figure 1(c), the i-th batch of **R** (R_i) is only used in probing **S** to identify S_i; the actual computation and storage of the join result does not take place. The updated values are then independently computed for those parameters involving $\mathbf{x}_{S_i}$ and those involving $\mathbf{x}_{R_i}$. More specifically, we factorize the computation of $\gamma_k^{(n)}$, μ_k and Σ_k in the E-step as well as the M-step. Some bookkeeping during the factorization is required and provided below.

1) E-step

In the E-step, $\gamma_k^{(n)}$ needs to be calculated involving Equations (1) and (2). In Equation (1), feature vectors are not directly involved in the calculation of $\frac{1}{\sqrt{(2\pi)^d|\Sigma_k|}}$. In contrast, for the part of $e^{-\frac{1}{2}(\mathbf{x}^{(n)}-\mu_k)^T\Sigma_k^{-1}(\mathbf{x}^{(n)}-\mu_k)}$, feature vectors from both table **S** and **R** are required when computing $(\mathbf{x}^{(n)} - \mu_k)^T\Sigma_k^{-1}(\mathbf{x}^{(n)} - \mu_k)$.

To ease notation, we denote Σ_k^{-1} as I_k and ignore the subscript k in the expanded equations for all the parameters belonging to the k^{th} Gaussian component.

$$\underset{1\times 1}{(\mathbf{x}^{(n)} - \mu_k)^T I_k(\mathbf{x}^{(n)} - \mu_k)} =$$
$$\underset{1\times d}{\begin{bmatrix} x_1^{(n)} - \mu_1 & x_2^{(n)} - \mu_2 & \cdots & x_d^{(n)} - \mu_d \end{bmatrix}} \times$$
$$\underset{d\times d}{\begin{bmatrix} I_{1,1} & I_{1,2} & \cdots & I_{1,d} \\ I_{2,1} & I_{2,2} & \cdots & I_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ I_{d,1} & I_{d,2} & \cdots & I_{d,d} \end{bmatrix}} \times \underset{d\times 1}{\begin{bmatrix} x_1^{(n)} - \mu_1 \\ x_2^{(n)} - \mu_2 \\ \vdots \\ x_d^{(n)} - \mu_d \end{bmatrix}} \tag{7}$$

Denote the first d_S dimensions of vector $\mathbf{x}^{(n)} - \mu_k$ as PD_S and the remaining d_R dimensions as PD_R:

$$\underset{d_S\times 1}{PD_S} = \begin{bmatrix} x_1^{(n)} - \mu_1 \\ x_2^{(n)} - \mu_2 \\ \vdots \\ x_{d_S}^{(n)} - \mu_{d_S} \end{bmatrix} \quad \underset{d_R\times 1}{PD_R} = \begin{bmatrix} x_{d_S+1}^{(n)} - \mu_{d_S+1} \\ x_{d_S+2}^{(n)} - \mu_{d_S+2} \\ \vdots \\ x_d^{(n)} - \mu_d \end{bmatrix} \tag{8}$$

Then $(\mathbf{x}^{(n)} - \mu_k)^T I_k(\mathbf{x}^{(n)} - \mu_k)$ = **UL** (upper left matrix) + **UR** (upper right matrix) + **LL** (lower left matrix) + **LR** (lower right matrix), where

$$\underset{1\times 1}{\mathbf{UL}} = \underset{1\times d_S}{PD_S^T} \underset{d_S\times d_S}{\begin{bmatrix} I_{1,1} & \cdots & I_{1,d_S} \\ \vdots & \ddots & \vdots \\ I_{d_S,1} & \cdots & I_{d_S,d_S} \end{bmatrix}} \underset{d_S\times 1}{PD_S} \tag{9}$$

$$\underset{1\times 1}{\mathbf{UR}} = \underset{1\times d_S}{PD_S^T} \underset{d_S\times d_R}{\begin{bmatrix} I_{1,d_S+1} & \cdots & I_{1,d} \\ \vdots & \ddots & \vdots \\ I_{d_S,d_S+1} & \cdots & I_{d_S,d} \end{bmatrix}} \underset{d_R\times 1}{PD_R} \tag{10}$$

$$\underset{1\times 1}{\mathbf{LL}} = \underset{1\times d_R}{PD_R^T} \underset{d_R\times d_S}{\begin{bmatrix} I_{d_S+1,1} & \cdots & I_{d_S+1,d_S} \\ \vdots & \ddots & \vdots \\ I_{d,1} & \cdots & I_{d,d_S} \end{bmatrix}} \underset{d_S\times 1}{PD_S} \tag{11}$$

$$\underset{1\times 1}{\mathbf{LR}} = \underset{1\times d_R}{PD_R^T} \underset{d_R\times d_R}{\begin{bmatrix} I_{d_S+1,d_S+1} & \cdots & I_{d_S+1,d} \\ \vdots & \ddots & \vdots \\ I_{d,d_S+1} & \cdots & I_{d,d} \end{bmatrix}} \underset{d_R\times 1}{PD_R} \tag{12}$$

We are decomposing a multiplication of three matrices in Equation (7) into a sum of four values (Equations (9), (10), (11), (12)) each of which is the result of a multiplication of three smaller matrices. They are the intermediate products that do not require every entire tuple $\mathbf{x}^{(n)}$ to be calculated. Instead, all of them are calculated using $\mathbf{x}_S^{(n)}$ and $\mathbf{x}_R^{(n)}$ from **S** and **R** directly. In particular, PD_R in Equation (10) and (11) as well as **LR** in Equation (12) only involve $\mathbf{x}_R^{(n)}$. Their values can be reused because each tuple in **R** can match several tuples in **S** through a primary/foreign-key probing. Therefore, there is a potential for large savings in this way of decomposition by removing the repeated calculations.

2) M-step

In the M-step, we can update π_k using Equation (5) directly which does not involve the data from table **S** or **R**. When updating μ_k, Equation (3) can be decomposed into two parts:

$$\underset{d\times 1}{\mu_k} = \frac{1}{N_k}\sum_{n=1}^{N}\gamma_k^{(n)}\mathbf{x}^{(n)} = \begin{bmatrix} \underset{d_S\times 1}{\frac{1}{N_k}\sum_{n=1}^{N}\gamma_k^{(n)}\mathbf{x}_S^{(n)}} \\ \underset{d_R\times 1}{\frac{1}{N_k}\sum_{n=1}^{N}\gamma_k^{(n)}\mathbf{x}_R^{(n)}} \end{bmatrix} \tag{13}$$

We next outline how to update Σ_k using Equation (4). We only need to focus on the computation of $(\mathbf{x}^{(n)} - \mu_k)(\mathbf{x}^{(n)} - \mu_k)^T$ since multiplying by a constant $\gamma_k^{(n)}$ and summing over all data points can be easily accomplished as long as we determine how to compute it in a factorized way. Subscript k is also ignored for simplifying notation.

$$\underset{d\times d}{(\mathbf{x}^{(n)}-\mu_k)(\mathbf{x}^{(n)}-\mu_k)^T} =$$

$$\underset{d\times 1}{\begin{bmatrix} x_1^{(n)}-\mu_1 \\ x_2^{(n)}-\mu_2 \\ \vdots \\ x_d^{(n)}-\mu_d \end{bmatrix}} \times \underset{1\times d}{\begin{bmatrix} x_1^{(n)}-\mu_1 & x_2^{(n)}-\mu_2 & \cdots & x_d^{(n)}-\mu_d \end{bmatrix}}$$

$$= \begin{bmatrix} \underset{d_S\times d_S}{\mathbf{UL}} & \underset{d_S\times d_R}{\mathbf{UR}} \\ \underset{d_R\times d_S}{\mathbf{LL}} & \underset{d_R\times d_R}{\mathbf{LR}} \end{bmatrix} \tag{14}$$

where

$$\underset{d_S\times d_S}{\mathbf{UL}} = \underset{d_S\times 1}{PD_S} \quad \underset{1\times d_S}{PD_S^T} \tag{15}$$

$$\underset{d_S\times d_R}{\mathbf{UR}} = \underset{d_S\times 1}{PD_S} \quad \underset{1\times d_R}{PD_R^T} \tag{16}$$

$$\underset{d_R\times d_S}{\mathbf{LL}} = \underset{d_R\times 1}{PD_R} \quad \underset{1\times d_S}{PD_S^T} \tag{17}$$

$$\underset{d_R\times d_R}{\mathbf{LR}} = \underset{d_R\times 1}{PD_R} \quad \underset{1\times d_R}{PD_R^T} \tag{18}$$

In this case, similar to the calculation of $\gamma_k^{(n)}$, all the sub-matrices in Equations (13) and (14) can be updated directly from the feature vectors of **S** and **R**. Equations (15), (16), (17), (18) are four matrices with smaller dimensions where PD_R and **LR** can be computed only once and then reused for any tuple in the join result with same $\mathbf{x}_R^{(n)}$. The potential for great savings which will be verified in Section VII depends on the degree of redundancy introduced.

We take Equation (14) as an example to analyze the computational savings due to factorization and how redundancy influences its performance. Before decomposition, computing one tuple in **T** requires d subtractions and d^2 multiplications. For N tuples, the original computation time is represented as $\tau = Nd(\tau_s + d\tau_m)$ where τ_s and τ_m represent the time required for one subtraction and multiplication operation respectively. When PD_R and **LR** are reused after decomposition by *F-GMM*, it requires $n_S d_S + n_R d_R$ subtractions and $n_S(d_S^2 + 2d_S d_R) + n_R d_R^2$ multiplications. As $N = n_S$ and $d = d_S + d_R$, the time saving is $\Delta\tau = (n_S - n_R)d_R(\tau_s + d_R\tau_m)$ and the saving rate is $\frac{\Delta\tau}{\tau} = \frac{(\frac{n_S}{n_R}-1)(\tau_s + d_R\tau_m)}{\frac{n_S}{n_R}(\frac{d_S}{d_R}+1)(\tau_s + d\tau_m)}$. That is, when d_S is fixed, with the increase of d_R or $\frac{n_S}{n_R}$, the factorized algorithm enjoys more computational cost savings over the baseline algorithms. Moreover, *F-GMM* has the same I/O cost as *S-GMM*, which we have analyzed in V-A.

Notice that the correctness of the calculation can be guaranteed for the reason that it is an exact decomposition to covert the large matrix operation into several small parts and no approximation is involved. All the other factors, such as the input data, the calculation of parameters, and the number of iterations required for training, remain unchanged. Although *M-GMM* reads the i-th batch from **T** and *S-GMM* / *F-GMM* reads the i-th batch from **R** for probing and then training, the values of parameters updated in each iteration are the same. This is because all N tuples are involved in calculating the parameters in Lines 7, 13, and 19, regardless of the number of matching tuples in each batch. Thus, the parameters are the same after finishing training and the accuracy of the models will not change for the algorithms *M-GMM*, *S-GMM* and *F-GMM*.

C. Algorithm F-GMM for Multi-way Joins

For large warehouses, multi-way joins are the norm and such a generalization is imperative. We now present the generalization of *F-GMM* to multi-way joins. In this section, we perform the factorization over a join sequence $\mathbf{R}_1 \bowtie_{RID_1=FK_1} \ldots \mathbf{R}_q \bowtie_{RID_q=FK_q} \mathbf{S}$. To ease notation in what follows, we denote **S** as $\mathbf{R}_0$ and d_S as d_{R_0}.

1) E-step

Similarly to the binary case, the E-step involves Equations (1) and (2). In contrast, in the calculation of $e^{-\frac{1}{2}(\mathbf{x}^{(n)}-\mu_k)^T\Sigma_k^{-1}(\mathbf{x}^{(n)}-\mu_k)}$, data from tables **S** ($\mathbf{R}_0$) to $\mathbf{R}_q$ are required when computing $(\mathbf{x}^{(n)}-\mu_k)^T\Sigma_k^{-1}(\mathbf{x}^{(n)}-\mu_k)$ in Equation (7).

Denote the first d_{R_0} dimensions of vector $\mathbf{x}^{(n)}$ - μ_k as PD_{R_0} and the remaining dimensions can be decomposed into q parts corresponding to relations $\mathbf{R}_1$ to $\mathbf{R}_q$. They are denoted as PD_{R_i} with dimension d_{R_i} for $i = 0\ldots q$. Then

$$\underset{1\times 1}{(\mathbf{x}^{(n)}-\mu_k)^T I_k(\mathbf{x}^{(n)}-\mu_k)} = \sum_{i=0}^{q}\sum_{j=0}^{q} PD_{R_i}^T I_{ij} PD_{R_j} \tag{19}$$

where

$$\underset{d_{R_m}\times 1}{PD_{R_m}} = \begin{bmatrix} x_{d_{m-1}+1}^{(n)} - \mu_{d_{m-1}+1} \\ x_{d_{m-1}+2}^{(n)} - \mu_{d_{m-1}+2} \\ \vdots \\ x_{d_m}^{(n)} - \mu_{d_m} \end{bmatrix} \tag{20}$$

$$\underset{d_{R_m}\times d_{R_n}}{I_{mn}} = \begin{bmatrix} I_{d_{m-1}+1,d_{n-1}+1} & \cdots & I_{d_{m-1}+1,d_n} \\ \vdots & \ddots & \vdots \\ I_{d_m,d_{n-1}+1} & \cdots & I_{d_m,d_n} \end{bmatrix} \tag{21}$$

I_{mn} is the partial I_k corresponding to relations R_m and R_n, where $m \in \{1\ldots q\}, n \in \{1\ldots q\}$ and $d_m = \sum_{i=0}^{m} d_{R_i}$.

In Equation (19), $(\mathbf{x}^{(n)}-\mu_k)^T I_k(\mathbf{x}^{(n)}-\mu_k)$ is decomposed into a sum of $(q+1)\times(q+1)$ smaller matrices. Similar to the analysis for the binary join case, when $i = j$ and $i \neq 0$, $PD_{R_i}^T I_{ij} PD_{R_j}$ can also be reused due to the redundancy. Furthermore, $\forall m \in \{1\ldots q\}$, for each feature vector in $\mathbf{R}_m$, Equation (20) is only calculated once to removing the repeated calculation in *S-GMM*.

2) M-step

In the M-step, when combining the features in table **S** ($\mathbf{R}_0$) and tables $\mathbf{R}_1$ to $\mathbf{R}_q$, we can decompose μ_k in Equation (3) as follows:

$$\underset{d\times 1}{\mu_k} = \frac{1}{N_k}\sum_{n=1}^{N}\gamma_k^{(n)}\mathbf{x}^{(n)} = \begin{bmatrix} \underset{d_{R_0}\times 1}{\frac{1}{N_k}\sum_{n=1}^{N}\gamma_k^{(n)}\mathbf{x}_{R_0}^{(n)}} \\ \underset{d_{R_1}\times 1}{\frac{1}{N_k}\sum_{n=1}^{N}\gamma_k^{(n)}\mathbf{x}_{R_1}^{(n)}} \\ \vdots \\ \underset{d_{R_q}\times 1}{\frac{1}{N_k}\sum_{n=1}^{N}\gamma_k^{(n)}\mathbf{x}_{R_q}^{(n)}} \end{bmatrix} \tag{22}$$

Next, for updating Σ_k, Equation(14) for multi-way joins can be written as:

$$\begin{bmatrix} \underset{d_{R_0}\times d_{R_0}}{M_{00}} & \underset{d_{R_0}\times d_{R_1}}{M_{01}} & \cdots & \underset{d_{R_0}\times d_{R_q}}{M_{0q}} \\ \vdots & \vdots & \ddots & \vdots \\ \underset{d_{R_q}\times d_{R_0}}{M_{q0}} & \underset{d_{R_q}\times d_{R_1}}{M_{q1}} & \cdots & \underset{d_{R_q}\times d_{R_q}}{M_{qq}} \end{bmatrix} \tag{23}$$

where

$$\underset{d_{R_i}\times d_{R_j}}{M_{ij}} = \underset{d_{R_i}\times 1}{PD_{R_i}} \; \underset{1\times d_{R_j}}{PD_{R_j}^T} \tag{24}$$

In this step, we decompose the result of $(\mathbf{x}^{(n)} - \mu_k)(\mathbf{x}^{(n)} - \mu_k)^T$ with dimension $d \times d$ into $(q+1) \times (q+1)$ blocks of much smaller matrices. It is evident that there are large savings if we reuse the computation of PD_{R_i} $(i \neq 0)$ and M_{ij} $(i = j$ and $i \neq 0)$ by getting the features from tables **S** and $\mathbf{R}_1$ to $\mathbf{R}_q$ directly for the factorized Equation (23).

VI. Neural Networks (NN)

The baseline approach to train a NN, with input from table **T** on the disk, is referred to as *M-NN*. When joining **R** and **S** on the fly and feeding the sub-tables to the model without materializing the result, it yields the other baseline algorithm *S-NN*. *F-NN* is the factorized approach we proposed. Training the NN can proceed according to standard algorithms namely batch, mini-batch or stochastic gradient descent (SGD) [5]. Note that for the case of SGD training, the join of **S** and **R** has to be permuted per epoch. Similarly, to perform SGD training in algorithm *S-NN* and *F-NN*, we can permute the keys of **R** for each training epoch, accessing the keys in a different order per epoch while probing relation **S**. Thus, the entire discussion that follows applies equally to mini-batch, batch and SGD training. The detailed decomposition for the computation process in the training process of NN using *F-NN* is discussed below.

A. *Algorithm F-NN for Binary Joins*

1) *Forward Propagation in the First Layer*

Assume there is one NN that receives d features in the input layer and have n_h hidden units in the first hidden layer h. When receiving the n-th feature vector $\mathbf{x}^{(n)}$, the value for a single hidden unit in layer h before applying the activation function f is: $a_j^{(n)} = \sum_{i=1}^{d} w_{ji}^{(1)} x_i^{(n)} + b_j^{(1)}$ where $j \in \{1 \dots n_h\}$. In order to make it clearer, we use the form of matrix to present the decomposition process. Thus, the weights matrix between the input and hidden layer is w and the bias vector is b. To ease notation, we eliminate the superscript (1) denoting the layer index. The vector $a^{(n)}$ for all $a_j^{(n)}$ can be written as:

$$\underset{n_h\times 1}{a^{(n)}} = \underset{n_h\times d}{w} \times \underset{d\times 1}{\mathbf{x}^{(n)}} + \underset{n_h\times 1}{b}$$

$$= \underset{n_h\times d}{\begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,d} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_h,1} & w_{n_h,2} & \cdots & w_{n_h,d} \end{bmatrix}} \times \underset{d\times 1}{\begin{bmatrix} x_1^{(n)} \\ x_2^{(n)} \\ \vdots \\ x_d^{(n)} \end{bmatrix}} + \underset{n_h\times 1}{\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n_h} \end{bmatrix}}$$

$$= \underset{n_h\times d_S}{\begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,d_S} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,d_S} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_h,1} & w_{n_h,2} & \cdots & w_{n_h,d_S} \end{bmatrix}} \times \underset{d_S\times 1}{\begin{bmatrix} x_1^{(n)} \\ x_2^{(n)} \\ \vdots \\ x_{d_S}^{(n)} \end{bmatrix}}$$

$$+ \underset{n_h\times d_R}{\begin{bmatrix} w_{1,d_S+1} & w_{1,d_S+2} & \cdots & w_{1,d} \\ w_{2,d_S+1} & w_{2,d_S+2}, & \cdots & w_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_h,d_S+1} & w_{n_h,d_S+2} & \cdots & w_{n_h,d} \end{bmatrix}} \times \underset{d_R\times 1}{\begin{bmatrix} x_{d_S+1}^{(n)} \\ x_{d_S+2}^{(n)} \\ \vdots \\ x_d^{(n)} \end{bmatrix}} + \underset{n_h\times 1}{\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n_h} \end{bmatrix}}$$

In the computation above, the features can be divided into two vectors from relation **S** and **R** respectively. After adding the bias vector to apply activation function on the result, we obtain the values of the hidden units $h^{(n)} = f(a^{(n)})$ in the first layer. As depicted in Figure 2 (a), we can read tuples directly by batches from **R** and probe **S** for matching tuples so that parts of the computation of the vector can be pushed to the activation function. For each iteration, the values of the weights and biases are constant and each tuple in **R** may match several tuples in **S**. The result of partial inner products involved the features from **R** adding the bias only needs to be calculated once. In other words, $\sum_{i=d_S+1}^{d} w_{ji}^{(1)} x_i^{(n)} + b_j^{(1)}$ where $j \in \{1 \dots n_h\}$ is the reused calculation which can bring calculation savings in *F-NN* and we will quantify the benefits in Section VII.

2) *Forward Propagation in the Second Layer and Beyond*

Let us now observe the computation between the first layer h and the second layer l with n_l hidden units. For a single unit l_k in the second layer, where $k \in \{1 \dots n_l\}$, the output after the activation function f is:

$$l_k^{(n)} = f(\sum_{j=1}^{n_h} w_{kj}^{(2)} h_j^{(n)} + b_k^{(2)}) \tag{25}$$

$$= f(\sum_{j=1}^{n_h} w_{kj}^{(2)} f(\sum_{i=1}^{d} w_{ji}^{(1)} x_i^{(n)} + b_j^{(1)}) + b_k^{(2)}) \tag{26}$$

If f is an additive function[1], $l_k^{(n)}$ can be factored into:

$$f(\sum_{j=1}^{n_h}(w_{kj}^{(2)} f(\sum_{i=1}^{d_S} w_{ji}^{(1)} x_i^{(n)}) + w_{kj}^{(2)} f(\sum_{i=d_S+1}^{d} w_{ji}^{(1)} x_i^{(n)} + b_j^{(1)})) + b_k^{(2)})$$

$$= f(\sum_{j=1}^{n_h} w_{kj}^{(2)} f(T_1) + \sum_{j=1}^{n_h} w_{kj}^{(2)} f(T_2) + b_k^{(2)}) \tag{27}$$

where $T_1 = \sum_{i=1}^{d_S} w_{ji}^{(1)} x_i^{(n)}$ and $T_2 = \sum_{i=(d_S+1)}^{d} w_{ji}^{(1)} x_i^{(n)} + b_j^{(1)}$, which can be stored after being computed in the first layer and reused for the second layer to save additional operations. Moreover, let $T_3 = \sum_{j=1}^{n_h} w_{kj}^{(2)} f(T_2) + b_k^{(2)}$, which is the sum

[1]A solution to the Cauchy functional form $f(x + y) = f(x) + f(y)$.

of partial output involved the features from **R** in the first layer multiplying the weights in the second layer. Similarly, T_3 can be computed when one tuple in **R** appears for the first time and reused for the remaining matching tuples in **S** separately.

However in NN literature, popular activation functions f are empirically restricted to certain choices such as $sigmoid$, $tanh$ and recently $Relu$ [15], which has been highly successful in deep learning applications primarily due to its simplicity and low overhead during optimization [15]. It is fairly easy to confirm that both $sigmoid$ and $tanh$ are not additive functions; thus when networks utilize them, there are no opportunities to decompose and share the computation beyond the first layer. The $Relu$ function, on the other hand, is a piece-wise linear function. We observe that when the two terms T_1 and T_2 have the same sign, the $Relu$ function is additive. Thus, only additive functions could satisfy the requirements for exact decomposition, which limits the usage of factorized algorithms in the second layer.

Even when additive activation functions are used, it follows from Equation (25) that computing the value before applying the activation function of a single unit requires n_h multiplications and n_h additions. After decomposition, computing Equation (27) requires summing up the results of multiplication $(w_{kj}^{(2)} f(T_1))$ and an addition (adding T_3). That is, it requires n_h multiplications and n_h additions. Furthermore, it requires another n_h multiplications and n_h additions to compute T_3 for each feature vector in **R** before it can be reused. Thus, the total number of operations required to compute a single value at the second layer is higher if we attempt to reuse the result computed from **S** and **R** respectively across layers. At higher layers, this cost is going to increase even further with a similar reasoning, which may eventually outweigh the cost savings brought by the decomposition. To summarize, reusing the computation beyond the first layer of an NN is only possible for additive activation functions, and even when this is the case, the overhead incurred by the decomposition may render any attempt to share computation at higher layers unattractive.

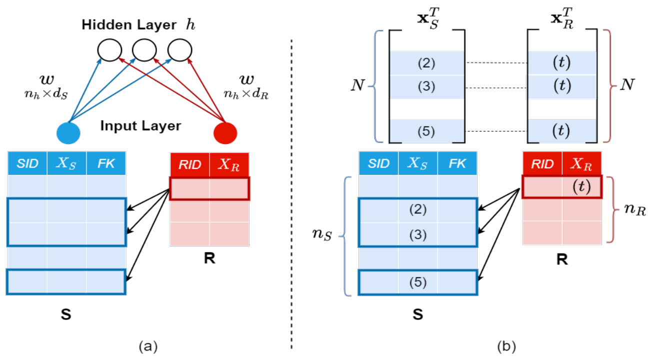


Fig. 2: (a) Forward Propagation (b) Backward Propagation

3) Computation in Backward Propagation

In the backward propagation phase, the error function is $E = \frac{1}{2N}\sum_{n=1}^{N}(o^{(n)} - Y^{(n)})^2$, where o is the output from the neural network and Y is the target. Backward propagation starts from the output layer and proceeds to the lower layers. In the entire process, all the feature vectors are involved in the computation for only one time when computing the gradient of the weights between the input layer and the first hidden layer.

$\frac{\partial E}{\partial a}$ is the gradient of error with respect to the value before the activation function between the input and hidden layer. It is obtained via BP algorithm applying the chain rule.

$$\underset{n_h \times d}{\frac{\partial E}{\partial w}} = \underset{n_h \times N}{\frac{\partial E}{\partial a}} \times \underset{N \times d}{\mathbf{x}^{\mathrm{T}}} \tag{28}$$

$$= \underset{n_h \times N}{\frac{\partial E}{\partial a}} \times \begin{bmatrix} \underset{N \times d_S}{\mathbf{x}_S^{\mathrm{T}}} & \underset{N \times d_R}{\mathbf{x}_R^{\mathrm{T}}} \end{bmatrix} = \begin{bmatrix} \underset{n_h \times d_S}{PG_S} & \underset{n_h \times d_R}{PG_R} \end{bmatrix} \tag{29}$$

where $PG_S = \frac{\partial E}{\partial a} \times \mathbf{x}_S^{\mathrm{T}}$ and $PG_R = \frac{\partial E}{\partial a} \times \mathbf{x}_R^{\mathrm{T}}$.

From Equation (28), as the result of the multiplication of the feature matrix $\mathbf{x}$, even though we can decompose the computation into two parts as in Equation (29), there are no opportunities to explore redundancy in the computation. The reason is that the redundancy exists among the columns of $\mathbf{x}$ due to their sharing the same part of $\mathbf{x}_R$. For PG_R, each row in $\frac{\partial E}{\partial a}$ multiplies the corresponding column of $\mathbf{x}_R^{\mathrm{T}}$ not $\mathbf{x}_R$, i.e., $[x_i^{(1)} x_i^{(2)} \cdots x_i^{(N)}]^{\mathrm{T}}$ not $[x_1^{(n)} x_2^{(n)} \cdots x_d^{(n)}]^{\mathrm{T}}$.

One benefit of decomposing the matrix into two parts in the backward propagation is I/O cost savings. **R** contains n_R tuples, while in Equation (29), the matrix $\mathbf{x}_R^{\mathrm{T}}$ has N tuples. We can obtain features directly from **R** and use the primary/foreign-key relationship to retrieve corresponding features from **S**. This way, $\mathbf{x}_S^{\mathrm{T}}$ can be populated along with the matrix $\mathbf{x}_R^{\mathrm{T}}$ required. In particular, as shown in Figure 2(b), for a feature vector (t) in **R**, the matching feature vectors (2), (3) and (5) in **S** populate $\mathbf{x}_S^{\mathrm{T}}$. Then the feature vector (t) is inserted in positions corresponding to $\mathbf{x}_S^{\mathrm{T}}$ in $\mathbf{x}_R^{\mathrm{T}}$. Note that both $\mathbf{x}_S^{\mathrm{T}}$ and $\mathbf{x}_R^{\mathrm{T}}$ have N rows, where N is the cardinality of **S**. Thus, instead of retrieving $N \times (d_S + d_R)$ fields in **T**, we only need $n_S \times d_S + n_R \times d_R$ fields where $n_S = N$ and $n_R < N$. Thus, *F-NN* can bring savings by reducing I/O cost during backward propagation even when there are no reused calculations.

To summarize, our final proposal is *F-NN*, which follows from algorithms *M-NN* and *S-NN* and applies the optimizations during the forward and backward propagation. *F-NN* reduces computational cost by removing repeated calculations through decomposition as outlined in Section VI-A1 and reduces I/O cost by avoiding reading the redundant fields in **T** as discussed in Section VI-A3. We will experimentally evaluate such savings in Section VII.

One may notice that the proposed optimizations are compatible with popular techniques in DNN such as batch normalization [15] (as it affects all input and applied before data enters the network) as well as Dropout [15]. Dropout can be applied either after activation at a layer or at the network input. In both cases, the linearity between the weights and the input, allows Dropout to be applied effectively with suitable bookkeeping.

B. Algorithm F-NN for Multi-way Joins

The algorithm generalizes naturally to multi-way joins. We assume the same setting involving a join of $q+1$ relations as in Section V-C.

During forward propagation, we read features directly from tables $\mathbf{S}$, $\mathbf{R}_1 \ldots \mathbf{R}_q$. To unify notation, we denote $\mathbf{S}$ as $\mathbf{R}_0$ in the sequel. Let us focus on the computation of a single hidden unit $h_j^{(n)}$ in the first layer:

$$h_j^{(n)} = f(\sum_{i=1}^{d} w_{ji}^{(1)} x_i^{(n)} + b_j^{(1)}) \tag{30}$$

$$= f(\sum_{i=1}^{d_{R_0}} w_{ji}^{(1)} x_i^{(n)} + \sum_{m=1}^{q} \sum_{i=d_{m-1}+1}^{d_m} w_{ji}^{(1)} x_i^{(n)} + b_j^{(1)}) \tag{31}$$

We compute the weights multiplied with features from table $\mathbf{R}_0$ ($\mathbf{S}$) and then compute the weights multiplied with features from tables $\mathbf{R}_1$ to $\mathbf{R}_q$. After sum them along with the bias, we feed the result to the activation function to get the value of the neural network at this hidden unit.

During backward propagation, the gradient of error with respect to the weights between the input and the first hidden layer is obtained by the BP algorithm applying the chain rule:

$$\underset{n_h \times d}{\frac{\partial E}{\partial w}} = \underset{n_h \times N}{\frac{\partial E}{\partial a}} \times \underset{N \times d}{\mathbf{x}^{\mathrm{T}}} = \underset{n_h \times N}{\frac{\partial E}{\partial a}} \times \begin{bmatrix} \underset{N \times d_S}{\mathbf{x}_{R_0}^{\mathrm{T}}} & \cdots & \underset{N \times d_{R_q}}{\mathbf{x}_{R_q}^{\mathrm{T}}} \end{bmatrix}$$

$$= \begin{bmatrix} \underset{n_h \times d_S}{PG_{R_0}} & \cdots & \underset{n_h \times d_{R_q}}{PG_{R_q}} \end{bmatrix} \tag{32}$$

where $PG_{R_m} = \frac{\partial E}{\partial a} \times \mathbf{x}_{R_m}^{\mathrm{T}}, m \in \{0, ..., q\}$. The same optimization as in Section VI-A3 is applied to decompose the matrix multiplication into $q+1$ parts in Equation (32). Feature vectors can be divided into $q+1$ parts and obtained directly from $\mathbf{R}_0$ to $\mathbf{R}_q$ to populate $\mathbf{x}_{R_m}^{\mathrm{T}}$ in the corresponding position. This is a way to reduce the I/O cost compared to retrieving the entire feature vectors in $\mathbf{T}$.

VII. Experiments

In this section, we present a detailed experimental evaluation of all the algorithms presented comparing their performance. We compare the runtime performance of *M-GMM*, *S-GMM* and *F-GMM* for the case of GMM as well as *M-NN*, *S-NN* and *F-NN* for the NN. There are two main parameters of interest in the underlying relations that essentially quantify the impact of normalization in terms of eliminating redundancy: the number of features in R (d_R) and the tuple ratio of $\mathbf{S}$ and $\mathbf{R}$ ($rr = n_S/n_R$). We vary these two parameters controlling the amount of redundancy that the join introduces. In addition, we vary the number of clusters (K) for GMM and the number of hidden units (n_h) for NN . We utilize both synthetic and real datasets in our evaluation as outlined below.

A. Datasets

In order to be able to vary the parameters of interest in a controlled way to observe trends, we utilize synthetic datasets. We generate synthetic datasets for primary/foreign-key joins with a wide range of attributes in the relations involved. The parameters varied are shown in Table II, III for GMM and NN experiments respectively. We generate synthetic data sampling from multiple Gaussian distributions and add random noise in accordance with previous work [22].

TABLE II: Synthetic data dimensions for GMM

Experiment	n_S	n_R	d_S	d_R	K
Vary rr	Varied	1000	5	5 and 15	5
Vary d_R	10^6 and 5×10^6	1000	5	Varied	5
Vary K	10^6	1000	5	15	Varied

TABLE III: Synthetic data dimensions for NN

Experiment	n_S	n_R	d_S	d_R	n_h
Vary rr	Varied	1000	5	5 and 15	50
Vary d_R	10^6 and 5×10^6	1000	5	Varied	50
Vary n_h	10^6	1000	5	15	Varied

For the real datasets, we utilize the *Expedia*, *Walmart* and *Movies* datasets from the Hamlet Plus Project[2]. We derive *Expedia1* dataset by joining *R1_Hotels* (relation $\mathbf{R}$) with *S_Listings* (relation $\mathbf{S}$) and *Expedia2* dataset by joining *R2_Searches* (relation $\mathbf{R}$) with *S_Listings* (relation $\mathbf{S}$). For the *Walmart* dataset, we join *R1_Indicators* (relation $\mathbf{R}$) with *S_Sales* (relation $\mathbf{S}$) and for the *Movies* dataset, we join *R2_movies* (relation $\mathbf{R}$) with *S_ratings* (relation $\mathbf{S}$). The details of the datasets are available in Table IV. For GMM, we use the original representation of the data (labeled *Not Sparse*). For NN, we use the one-hot representation of the data (labeled *Sparse*). Unless stated otherwise, we run NN training for 10 epochs and utilize a single hidden layer in our experiments.

TABLE IV: Data dimensions of real datasets

Dataset	n_S	d_S	n_R	d_R
Expedia1(Not Sparse)	942142	7	11938	8
Expedia2(Not Sparse)	942142	7	37021	14
Walmart (Not Sparse)	421570	3	2340	9
Movies (Not Sparse)	1000209	1	3706	21
Walmart (Sparse)	421570	126	2340	175
Movies (Sparse)	1000209	1	3706	21

Since the dimension in the real datasets is limited, we construct datasets derived from the *Expedia1* dataset with larger dimensions. These are constructed by picking tuples with a high rr and increasing d_R by repeating the features (as well as adding random Gaussian noise). These are depicted in Table V as *Expedia3* to *Expedia5* along with their associated characteristics.

TABLE V: Data dimensions of augmented real datasets

Dataset	n_S	d_S	n_R	d_R
Expedia3	634133	7	2899	29
Expedia4	634133	7	2899	78
Expedia5	634133	7	2899	218

To generate data for multi-way joins, we utilize the *Movies* dataset (relations *S_ratings*, *R1_users* and *R2_movies*) and

[2]Available at https://adalabucsd.github.io/hamlet.html

inject synthetic data to relation *R1_users* ($\mathbf{R}_1$) keeping the size of relation *R2_movies* ($\mathbf{R}_2$) unchanged. In this way, we can vary the redundancy ratio between $\mathbf{R}_1$ and $\mathbf{R}_2$. Each time we generate a synthetic tuple for $\mathbf{R}_1$, we select a random tuple from $\mathbf{R}_2$, extract the key and insert a synthetic tuple on *S_ratings* ($\mathbf{S}$) enforcing any relational constraints in the process. We also vary the dimension for relation *R1_users* (d_{R_1}) during the experiments.

B. Experimental Setup

All experiments were run on a cluster of machines with 16 Intel Xeon E5630 2.53 GHz cores, 96 GB RAM and 338 GB disk with CentOS 6.2. Our code is implemented in Python 2.7.13 using NumPy for all the matrix calculations and psycopg2 to read and write data from PostgreSQL 9.6. The RDBMS is utilized primarily for storage of relations and all algorithm logic is implemented on top of the RDBMS. We also conducted experiments with TensorFlow in place of Numpy, as well as on GPUs instead of CPUs, and the trends are consistent with what we are presenting here; they are thus omitted due to space limitation.

C. Results on Synthetic Datasets

1) GMM

Figure 3 presents the results for the case of the GMM algorithms varying tuple ratio (rr) in Figure 3(a), varying the number of features in R (d_R) in Figure 3(b) and varying the number of clusters (K) in Figure 3(c). In all cases, *F-GMM* is fastest than the other two applicable approaches. In Figure 3(a), with the increase of rr, the benefits of the proposed *F-GMM* become increasingly larger. It can be seen that the trend will persist becoming increasingly larger as d_R increases. For $d_R = 5$, *F-GMM* is 2 times faster than *S-GMM*, which becomes 2.4 times faster when $d_R = 15$. In Figure 3(b), it is evident that as d_R increases, *F-GMM* becomes two to six and a half times faster than the other approaches for different values of rr. The benefit will keep on increasing as we increase d_R. Finally Figure 3(c) presents that *F-GMM* is two to three times faster when we vary K for fixed rr and d_R. This benefit will increase as we vary rr and/or d_R.

Figure 4 presents the corresponding experiments for multi-way joins. The results are overall consistent, however, the benefits of our proposal for increasing rr (ratio of synthetic tuples in $\mathbf{R}_1$ to the number of tuples in $\mathbf{R}_2$) vary from three to five times faster. As d_{R_1} increases, it is three to fourteen times faster than others. Moreover, when increasing K, the time will be saved three to five times than alternative approaches. Compared with the results of binary joins, the optimizations introduced by *F-GMM* pay off as the number of joins increases.

2) NN

F-NN gains obvious performance improvement for the case of NN. Figure 5(a) presents the results when increasing rr. For $d_R = 5$, *F-NN* becomes more than two times faster. These savings will keep an upward tendency as rr increases further. For $d_R = 15$, it progressively becomes three times faster, with higher benefits as rr increases. Figure 5(b) shows the results of a corresponding experiment varying d_R. With the growth of d_R, performance advantages vary increasingly from two to three times faster for $rr = 1000$ and from 2.2 to 3.5 times for $rr = 5000$. Finally, Figure 5(c) reveals the results for increasing n_h in the network. For fixed rr and d_R, as n_h increases, the performance benefits of *F-NN* vary from two to three times faster than the others. These advantages will increase as we vary d_R and/or rr.

It is evident that for very small values of rr, depending on the values of d_R, the proposed approach may not offer performance advantages. For example, for $d_R = 5$, we observe performance benefits for values of $rr > 200$. Similarly, when $d_R = 15$, benefits start appearing for values of $rr > 50$.

Figure 6 presents the results of the same experiments for multi-way joins. The results are consistent, however, the benefits of *F-NN* range from three to four times faster as we vary rr and from three times faster for small values of rr to six times faster for larger rr as we increase d_{R_1}. Similarly, the benefits persist as we increase n_h, up to four times in Figure 6(c). It reveals that the proposed approach offers large performance benefits especially as the number of joins increases.

D. Results on Real Datasets

The following tables present the results for the real datasets for the case of GMM and NN respectively (all times in the tables in seconds).

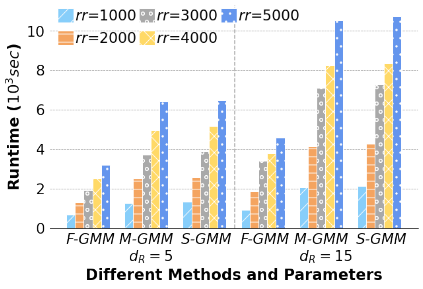
TABLE VI: Results on real datasets for GMM

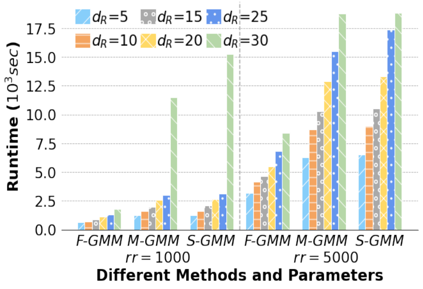
Dataset	*M-GMM*	*S-GMM*	*F-GMM*
Expedia1(Not Sparse)	2140.1	2244.3	1014.2
Expedia2(Not Sparse)	1221.1	1248.5	593.1
Walmart (Not Sparse)	595.9	602.9	212.1
Movies (Not Sparse)	1691.7	1755.8	514.6
Expedia3 (Augmented)	1673.5	1750.9	639.3
Expedia4 (Augmented)	6129.6	6311.4	1843.3
Expedia5 (Augmented)	23270.6	23375.1	9779.3
Movies-3way	2455.3	2883.1	715.1

From Tables VI, the performance benefits of *F-GMM* for different datasets are up to 3.4 times faster than *M-GMM* or *S-GMM*. For the same experiment in the case of *Expedia3* to *Expedia5* datasets, we can see that the benefits of *F-GMM* range from 2.4 to 3.4 times faster than others while as to *Expedia1*, they are up to 2.2 times. Extending to multi-way joins, we display the results for *Movies-3way*. In particular, *F-GMM* is 4.4 times (for *F-GMM*) faster compared to the materialized methods. It is evident that compared with the binary join case, the performance benefits of our approach are larger since there are more opportunities for exploiting the inherent redundancy after the join.

TABLE VII: Results on real datasets for NN

Dataset	*M-NN*	*S-NN*	*F-NN*
Walmart(Sparse)	743.1	845.5	104.1
Movies (Sparse)	437.4	507.2	112.3
Movies-3way	890.1	1022.3	202.1


(a) Varying rr


(b) Varying d_R

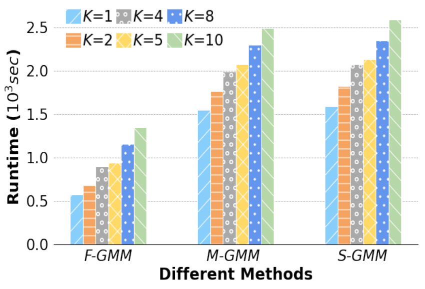
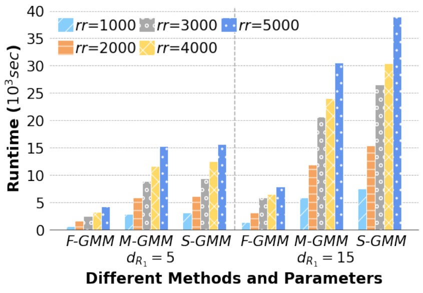
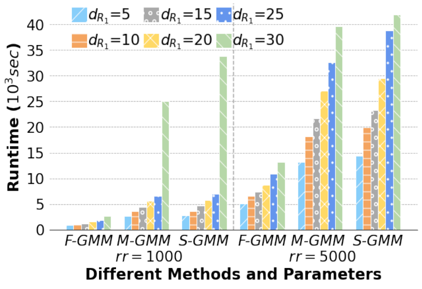

(c) Varying K

Fig. 3: Performance results for GMM algorithms varying parameters of interest


(a) Varying rr


(b) Varying d_{R_1}

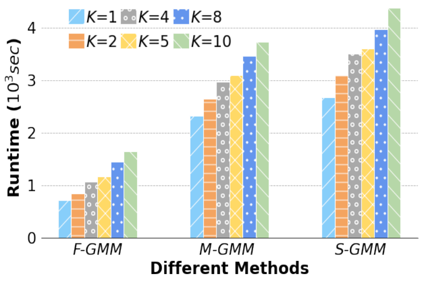
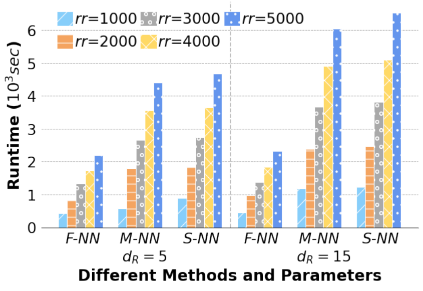
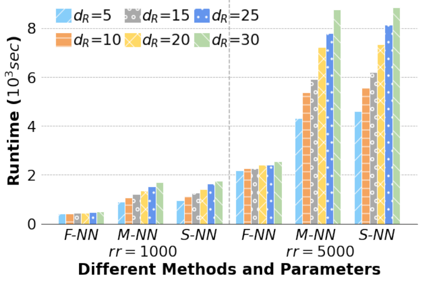

(c) Varying K

Fig. 4: Performance results for GMM algorithms on multi-way joins varying parameters of interest


(a) Varying rr


(b) Varying d_R

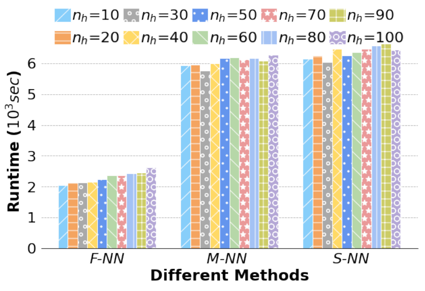
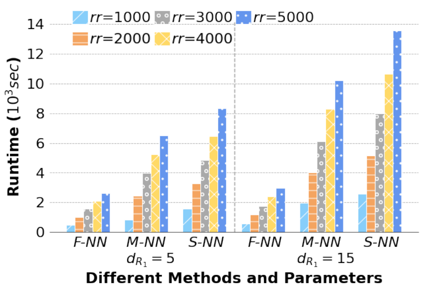
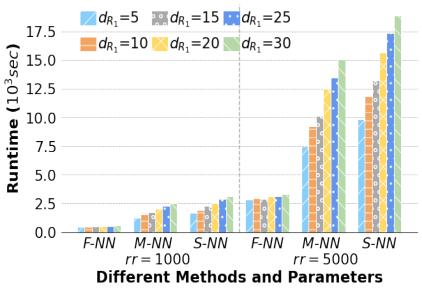

(c) Varying n_h

Fig. 5: Performance results for NN algorithms varying parameters of interest


(a) Varying rr


(b) Varying d_{R_1}

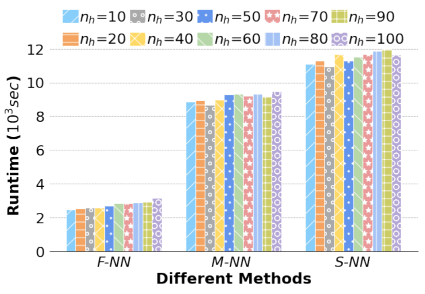

(c) Varying n_h

Fig. 6: Performance results for NN algorithms on multi-way joins varying parameters of interest

Finally, Table VII reveals the results for the case of NN. *F-NN* demonstrates 8.1 times faster execution for *Walmart* dataset and 4.5 times faster execution for *Movies* dataset, As we have demonstrated on the synthetic datasets, the performance benefits of our approach become larger as redundancy increases. In the case of the *Sparse* datasets, the redundancy ratio is high after being encoded. *F-NN* can take advantage of this redundancy during both forward and backward propagation phases and offer superior performance benefits. These results as demonstrated using real datasets attest to the significance of our proposals given the enormous recent interest in NN and associated learning technologies in academia and industry. For the multi-way join, the benefit of *Movies-3way* is 3.4 times for *F-NN*. In general, the benefits of the proposed approach depend on the amount of redundancy the join introducing.

VIII. Conclusions

We propose a set of algorithms to execute popular non-linear algorithms such as Gaussian Mixture Models and Neural Networks over normalized databases. We demonstrate that by carefully reworking the basic operations of these algorithms they can be executed over normalized relational inputs offering large performance benefits over alternative approaches. In addition to proposing and documenting our algorithms, we present the results of a detailed experimental evaluation utilizing both synthetic and real datasets demonstrating the performance advantages on our proposals. We experimentally establish that the proposed algorithms executed over normalized relations offer very significant performance advantages that become increasingly larger as the characteristics of the underlying datasets change. In the future, it is natural to focus on other types of popular deep network architectures, including various types of autoencoders, generative models as well as deep belief networks. Finally, the mode of delivery of ML algorithms exploring factorization ideas to end users (via a UDF library or native implementations) merits further investigation.

IX. Acknowledgements

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] Oracle R Enterprise.

[2] Apache Mahout: Scalable machine learning and data mining, 2008.

[3] N. Bakibayev, T. Kociský, D. Olteanu, and J. Zavodny. Aggregation and ordering in factorised databases. *PVLDB*, 6(14):1990–2001, 2013.

[4] C. Beeri, P. A. Bernstein, and N. Goodman. A sophisticate's introduction to database normalization theory. In *VLDB*, pages 113–124, 1978.

[5] C. M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Springer, 2007.

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

[7] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. R. Burdick, and S. Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in SystemML. *PVLDB*, 7(7):553–564, 2014.

[8] Z. Cai, Z. J. Gao, S. Luo, L. L. Perez, Z. Vagena, and C. Jermaine. A comparison of platforms for implementing and running very large scale machine learning algorithms. In *SIGMOD*, pages 1371–1382, 2014.

[9] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine. Simulation of database-valued markov chains using simsql. In *SIGMOD*, pages 637–648, 2013.

[10] Z. Cheng and N. Koudas. Non linear models over normalized data. In *Poster Paper, Proceedings of IEEE ICDE*, 2019.

[11] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38, 1977.

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

[13] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *PVLDB*, 5(11):1268–1279, 2012.

[14] Z. J. Gao, S. Luo, L. L. Perez, and C. Jermaine. The BUDS language for distributed bayesian machine learning. In *SIGMOD*, pages 961–976, 2017.

[15] I. J. Goodfellow, Y. Bengio, and A. C. Courville. *Deep Learning*. MIT Press, 2016.

[16] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library: Or MAD skills, the sql. *PVLDB*, 5(12):1700–1711, 2012.

[17] M. A. Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. AC/DC: in-database learning thunderstruck. In *DEEM@SIGMOD*, pages 8:1–8:10, 2018.

[18] M. A. Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. In-database learning with sparse tensors. In *PODS*, pages 325–340, 2018.

[19] M. L. Koc and R. Christopher. Incrementally maintaining classification using an RDBMS. *PVLDB*, 4(5):302–313, 2011.

[20] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. MLbase: A Distributed Machine-learning System. In *CIDR*, volume 1, pages 2–1, 2013.

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

[22] A. Kumar, J. F. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, pages 1969–1984, 2015.

[23] K. J. Lee, L. Guillemot, Y. L. Yue, M. Kramer, and D. J. Champion. Application of the Gaussian mixture model in pulsar astronomy - pulsar classification and candidates ranking for the Fermi 2FGL catalogue. *Monthly Notices of the Royal Astronomical Society*, 424(4):2832–2840, 2012.

[24] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. Scalable linear algebra on a relational database system. In *ICDE*, pages 523–534, 2017.

[25] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine learning in Apache Spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, 2016.

[26] K. P. Murphy. *Machine learning - a probabilistic perspective*. Adaptive computation and machine learning series. MIT Press, 2012.

[27] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich. Data management challenges in production machine learning. In *SIGMOD*, pages 1723–1726, 2017.

[28] S. Rendle. Scaling factorization machines to relational data. In *PVLDB*, volume 6, pages 337–348, 2013.

[29] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *SIGMOD*, pages 3–18, 2016.

[30] V. Shah, A. Kumar, and X. Zhu. Are key-foreign key joins safe to avoid when learning high-capacity classifiers? *PVLDB*, 11(3):366–379, 2017.

[31] A. K. Side Li, Lingjiao Chen. Enabling and optimizing non linear feature interactions in factorized linear algebra. In *SIGMOD*, pages 1571–1588, 2019.

[32] A. Thomas and A. Kumar. A comparative evaluation of systems for scalable linear algebra-based analytics. *PVLDB*, 11(13):2168–2182, 2018.

[33] K. Yang, Y. Gao, L. Liang, B. Yao, S. Wen, and G. Chen. Towards factorized svm with gaussian kernels over normalized data. In *ICDE*, pages 1453–1464, 2020.