FACTORIZED CONSTRUCTION OF MACHINE LEARNING METHODS OVER NORMALIZED DATA

ZHE ZHANG

A THESIS SUBMITTED TO
THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF ARTS

GRADUATE PROGRAM IN INFORMATION SYSTEMS AND TECHNOLOGY
YORK UNIVERSITY
TORONTO, ONTARIO
JULY 2021

Abstract

Enterprises are adopting machine learning to gain knowledge from the vast amount of data, which are normalized and stored in relational databases. All the features required in different relations must be combined through join operations and fed to machine learning processes. As a result, redundancy avoided by normalization is reintroduced, which incurs additional costs. This thesis proposes the factorized algorithms (F-GMM, F-NN and F-PPCA) for three widely used scenarios (GMM, NN and PPCA) in machine learning to eliminate the redundancy introduced by the joins. The training process can be conducted much faster without any loss in accuracy for the exact decomposition. The efficiency improvement depends on the relative redundancy of the original relations. Finally, we design extensive experiments on both synthetic and real datasets to evaluate the performance of the proposed algorithms by varying parameters of interest. The factorized method yields significant efficiency improvements, which increases with redundancy growth.

Acknowledgements

First and foremost, I would like to express my greatest appreciation to my respectable supervisor Prof. Xiaohui Yu for supporting my whole Master's study and research. Prof. Xiaohui Yu has supervised me during the past two years with his patient guidance and constant encouragement. With his help, I establish my confidence and enthusiasm for academic research and understand how to become a qualified researcher.

I also would like to extend my gratitude to Prof. Nick Koudas for his suggestion of this meaningful topic and the answers to the technical difficulties during the completion of the thesis. His rich research experience and enthusiasm give me inspiration and motivation in the academic field.

Besides, I would like to thank the supervisory committee and examination committee members for providing meaningful comments on this thesis. Prof. Manar Jammal kindly reviewed my proposal carefully at the beginning of this work and gave me advice on the design of the algorithms. Prof. Augustine gave me detailed

guidance on the language of my thesis and provided lots of valuable suggestions. Their rigorous academic attitude will have a profound impact on my future studies.

Last but not least, I would like to give my thanks to my beloved husband and heart-warming daughter for their loving considerations and great confidence in me during these two years. Many thanks also go to our upcoming new baby for the motivation he gives me, and I sincerely wish him healthy and safe.

Table of Contents

Abstract **ii**

Acknowledgements **iii**

Table of Contents **v**

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

1.1 Motivation 1

1.2 Methodology 5

1.3 Contributions and Outline 7

2 Literature Review **10**

2.1 Machine Learning on Big Data 10

2.2 Machine Learning and Data Management 11
2.3 Machine Learning Optimization 13
2.4 Factorized Database 14
2.5 Machine Learning over Normalized Data 15

3 Preliminaries **17**
3.1 Gaussian Mixture Models (GMM) 17
3.2 Neural Network (NN) 20
3.3 Probabilistic Principal Component Analysis (PPCA) 23

4 Problem Description and Algorithm Frameworks **27**
4.1 Problem Description 27
4.2 Three Algorithm Frameworks 29
4.2.1 Framework for M-algorithm 30
4.2.2 Framework for S-algorithm 32
4.2.3 Framework for F-algorithm 34

5 Algorithms for GMM **37**
5.1 Baseline Algorithms: M-GMM and S-GMM 37
5.2 Algorithm F-GMM and Decomposition Process 39
5.2.1 E-step 39
5.2.2 M-step 41

5.3 F-GMM for Multi-way Joins 45
5.3.1 E-step for Multi-way Joins 45
5.3.2 M-step for Multi-way Joins 47

6 Algorithms for NN **49**

6.1 Baseline Algorithms: M-NN and S-NN 49
6.2 Algorithm F-NN and Decomposition Process 50
6.2.1 Forward Propagation 51
6.2.2 Backward Propagation 55

7 Algorithms for PPCA **59**

7.1 Baseline Algorithms: M-PPCA and S-PPCA 59
7.2 Algorithm F-PPCA and Decomposition Process 60
7.2.1 Calculating W_{new} 60
7.2.2 Calculating σ^2_{new} 63

8 Experiments **67**

8.1 Settings 67
8.1.1 Environment 67
8.1.2 Parameters 67
8.1.3 Datasets 69
8.2 Results 71

8.2.1 Results on Synthetic Datasets 71

8.2.2 Results on Real Datasets 77

9 Conclusions and Future Work **79**

9.1 Conclusions 79

9.2 Future Work 80

Bibliography **82**

List of Tables

3.1 Notations used in GMM 18

3.2 Notations used in NN 20

3.3 Notations used in PPCA 24

4.1 Notations used in the algorithms 29

8.1 Synthetic datasets with varied parameters for GMM 69

8.2 Synthetic datasets with varied parameters for NN 69

8.3 Synthetic datasets with varied parameters for PPCA 70

8.4 Data dimensions of real datasets 70

8.5 Data dimensions of augmented real datasets 71

List of Figures

1.1 An example of redundant data 3

4.1 M-algorithm framework 31

4.2 S-algorithm framework 33

4.3 F-algorithm framework 35

6.1 Forward Propagation 52

6.2 Backward Propagation 57

8.1 Results for GMM algorithms varying parameters of interest 73

8.2 Results for NN algorithms varying parameters of interest 75

8.3 Results for PPCA algorithms varying parameters of interest 76

8.4 Results on real datasets 78

1 Introduction

1.1 Motivation

With the rapid development of data analysis and storage technologies, machine learning (ML) and data management are integrated closely during the enterprise's business process, such as discovering the value of data to support decision-making. Most enterprises adopt traditional relational databases that use tables to store data in rows and columns, and represent their relationships among data [1]. In relational databases, the normalized data is stored in separate minimal relational tables designed following strict normal forms, such as 2NF and 3NF, to avoid repetition, improve data consistency and retrieve information efficiently. Each table stores data of a particular subject, and there is no transitive dependency for them. They are linked to other tables with primary/foreign-key relationships. The primary key uniquely identifies a row in a table and the value of a foreign key corresponds to the values of the primary key in another table [1].

When data analysts take advantage of ML for enterprise data analysis, they

often select features scattered in various tables. However, most existing ML algorithms are assumed to take a single table as the input. To integrate all the required features, the traditional approach performs join operation for the normalized tables via primary/foreign key to get a new materialized table on the disk to feed the ML algorithms. Some adverse effects arise after such an operation. First of all, the new table reintroduces the redundancy removed by the normalization. As a result, it increases the repetitive computation costs on the redundant data during the training process. Besides, due to the enormous amount of data, materializing this new table imposes more storage space and the enterprise requires overheads to maintain it. Furthermore, the learning process is continuous as the data are updated frequently; thus, conducting joins for every exploration is time-consuming. Therefore, unnecessary costs from learning after joins lead to low efficiencies for the interaction application between database and ML.

Consider the example shown in Figure 1.1. The analysts intend to model customers' shopping trends in a store. Some features are the order details stored in the table **Orders** (*OrderID, CustomerID, ItemID, Time, Amount*), where *OrderID* is the primary key, and the other features are the items information stored in the table **Items** (*ItemID, Price, Size, Colour, Category*), where *ItemID* is the primary key. *ItemID* is a foreign key from **Orders** referencing **Items**. The original tables have four rows and five columns, two rows and five columns, respectively. After

the join operation, two small tables become one big table with four rows and nine columns. In the new table after joins, we can see that one row where $ItemID = 002$ in **Items** repeatedly appears two times for the matched tuples in **Orders**, and so does the row where $ItemID = 001$.

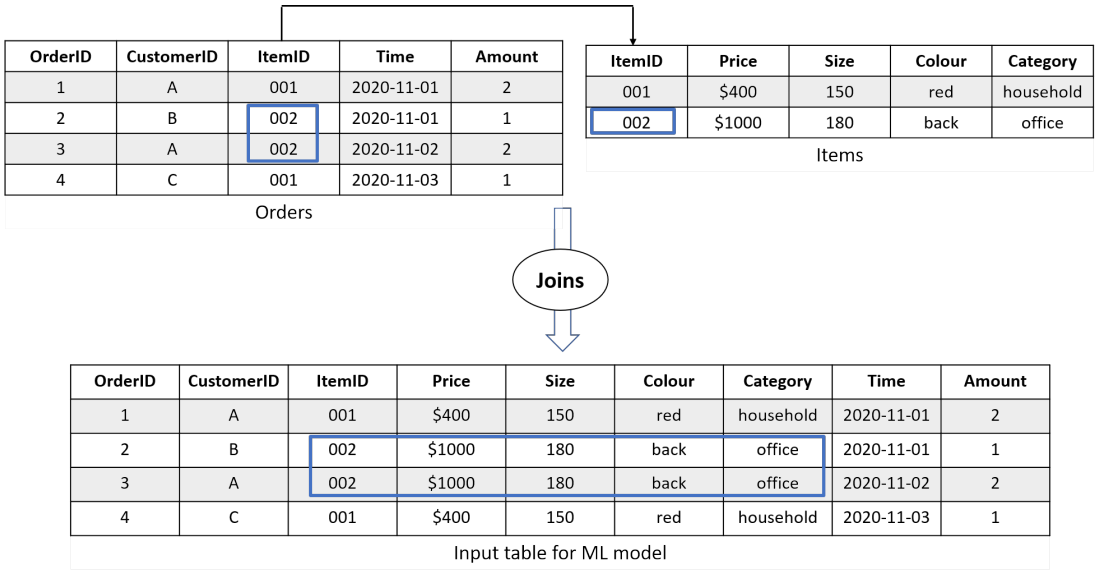


Figure 1.1: An example of redundant data

It is essential to find an effective way to avoid the additional costs of joining during the input stage. Recently, some studies have recognized this issue and utilize factorized ways to reduce redundancy in specific scenarios. They were initially aimed at for the linear models [2] and executed various linear algebra operations [3]. The follow-up articles explore the same issue for different models, such as Independent Gaussian Mixture Models (IGMM) [4] and Support Vector Machines

(SVM) with Gaussian Kernels [5]. However, the algorithms of ML are diverse and applicable to different scenarios. We pay more attention to whether the factorized method applies to other application scenarios.

In this thesis, we focus on two nonlinear ML models and a dimensionality reduction method to analyze whether they can be performed over normalized data. Our proposal is the factorized algorithm based on the original datasets instead of materializing the results after joins to push the computation. As far as we know, there is not any work considering the general case of Gaussian Mixture Models (GMM), various forms of Neural Networks (NN) and Probabilistic Principal Component Analysis (PPCA) executed over normalized input. Specifically, GMM is a training method used for clustering, while NN is one classical model to solve classification or regression problems. GMM is prevalent in financial analysis, quantitative finance, astronomy and banking applications, especially dealing with the returns of asset classes. NN is one widely used model to solve complex problems like pattern recognition or facial or handwriting recognition, weather prediction, and signal processing. Besides, we choose an iterative dimensionality reduction algorithm named PPCA to explore the application of data prepossessing. For the enterprise datasets we mentioned before, the features are in different relations. Most of the time, we cannot recognize which information is useful or not by experience. After the joins, all the features from different tables will be considered in reducing dimensionality

before getting the final low-dimensional data for the training process. We can directly utilize the data from the normalized relations to reduce the high-dimensional data and improve efficiency by removing the redundant part from the joins. One typical characteristic of these application scenarios is the training process needs complicated calculations. Therefore, there is a potential to improve efficiency when removing the joining step to reduce the redundant computations.

The challenge is how to decompose the computation precisely without reducing the quality of the outcomes or the scalability of the algorithms. Any approximation will affect the training quality and produce a model different from the original ones. Thus, the focus of this work is to provide the detailed decomposition steps for the training processes of specific algorithms to achieve exact decomposition.

1.2 Methodology

Based on the previous paper[6], we design three alternative methods to construct the above three ML application scenarios and analyze the performance improvement from both theoretical and experimental aspects.

There are two baseline approaches presented for comparison. The materialized method (M-algorithm) is the standard way to deal with this issue by materializing the join results on the disk. For GMM, NN and PPCA, the corresponding algorithms are M-GMM, M-NN and M-PPCA. The streaming method (S-algorithm) is

computing the joins in batches on the fly without storing the join results and we also provide the algorithms (S-GMM, S-NN and S-PPCA) for each ML case.

Our proposal is one factorized method (F-algorithm) that using the normalized data directly to calculate the equations after decomposition. In the algorithms of F-GMM, F-NN and F-PPCA, we factorize the training process step by step to separate the input data from different normalized tables and indicate which parts can bring savings. The target is to explore whether the factorized method is more effective than the other two baseline methods. Specifically, for F-GMM, 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 F-NN algorithm demonstrates the training process can be decomposed to take normalized data into account. Significant benefits can be achieved between the input layer and the first hidden layer during forward and backward propagation. However, we do not continue pursuing this at higher layers of the network because we can no longer guarantee the exactness of the decomposition for different activation functions. As to PPCA, the best choice for training is also the EM algorithm. The probabilistic latent variables in PPCA are continuous, so the decomposition process in F-PPCA is entirely different from that in F-GMM but reusing calculation still brings considerable efficiency improvement.

In general, the factorized approach's cost savings come from two sources: I/O

savings and computation savings. Firstly, F-algorithm removes the step of materializing the join results on the disk and instead executes the probing process on the fly in batches, bringing significant I/O savings. Secondly, we eliminate redundant computation by factorizing the calculation for the parameter updates. In F-algorithm, the partial parameter update equation that only involves the features in the dimension table can be calculated just once for all matching feature vectors from the fact table by primary/foreign-key relationship. In addition, various trade-offs between F-algorithms and baseline algorithms are compared. Thus, we can determine the efficiency improvement by qualitatively analyzing these two aspects before and after the decomposition. Besides, we also investigate how the differences of characteristics in the underlying relations involved affect the benefits of our proposal. The general conclusion is the more redundancy brings the more benefits.

To explain the process more clearly, all the algorithms are derived for binary joins in detail. Then, to prove its applicability, it is generalized to multi-way joins as well.

1.3 Contributions and Outline

In summary, we make the following contributions:

- Motivated by the linear models, we formally define the problem of three popular applications of ML over normalized data. To the best of our knowledge, this is the first work to address the general form of GMM, NN and PPCA.

- For each application of GMM, NN and PPCA, we present three algorithms to analyze their performance for binary joins. In particular, F-GMM, F-NN and F-PPCA are the factorized algorithms utilizing normalized data to explore the reuse of computation.

- We analyze the reason for performance improvement qualitatively depending on the characteristics in the relations, including the I/O cost savings and the calculation savings.

- We generalize the solution of binary joins to multi-way joins by taking GMM as an example.

- For the case of NN, we analyze the impact of different activation functions on sharing computations during the training phase. We investigate the computation cost at higher layers of the network and indicate the limitation of F-NN.

- We experimentally quantify the effectiveness of our proposal and present the results of a thorough evaluation testing the impact of different dataset param-

eters using synthetic datasets. We also utilize publicly available real datasets demonstrating impressive performance improvement.

The rest of the thesis is organized as follows. Chapter 2 reviews related work. In Chapter 3, we present background material for GMM, NN and PPCA as well as the notations required in the training process. Chapter 4 formally defines the problems and notations we focus on in this thesis and introduce the algorithm frameworks for our proposed method as well as the baseline approaches. In Chapter 5, we present three algorithms for GMM and the detailed decomposition process in algorithm F-GMM for both binary and multi-way joins. In Chapter 6, we introduce the three algorithms for training NN and the limitations existing in F-NN. Chapter 7 starts with two baseline algorithms for PPCA and then provides F-PPCA with the decomposition process. Chapter 8 illustrates the results of a thorough experimental evaluation of the proposed algorithms and the baseline algorithms across a variety of cases. Finally, Chapter 9 summarizes the conclusion in this thesis and discusses avenues for this research area in the future.

2 Literature Review

In this section, we organize the existing studies related to the problem into five general threads.

2.1 Machine Learning on Big Data

ML is the core of artificial intelligence and data science, which are fast-growing technical fields. Meanwhile, the rapid development of big data has also attracted widespread attention and applications from industry, academia, government, and organizations. Zhou et al. [7] introduced an extensive data ML framework to analyze its opportunities and challenges. It is centered on ML following the pre-processing, learning, evaluation stages and composes of big data, users, domain systems. The availability of high-quality intensive data will improve the result of ML. Elgohary et al. [8] combined compression techniques and sparse matrix representations to be compressed linear algebra to study many popular ML algorithms in memory. Khamis et al. [9] indicated in-database learning is more efficient than out-

of-database for costly data export loop and introduced the algorithms for training some models in relational databases using sparse tensors.

However, in many cases, a large part of the information in the data has not been exploited. Al-Jarrah et al. [10] proposed a sustainable data modeling to understand the large amount of data related to their field by discovering patterns and correlations effectively and efficiently. With its development, there is a list of the critical data processing issues caused by the ML pipelines deployed in production from understanding, validating, cleaning, and enriching data. To increase the quality of the models, the most common way is adding new features via joins [11]. Besides, Jan et al. [12] pointed out that the combination of deep learning technology and high-speed data processing mechanism will bring some limitations because it is not general and cannot learn features in big data. However, proposing a method to compare various deep learning techniques to process big data proves that using supervised and unsupervised training techniques is an excellent method to build deep learning.

2.2 Machine Learning and Data Management

It is a critical issue to deal with the extensive amount of data using appropriate data management techniques. The integration of ML technology with data management has attracted many researchers in academia. Miao et al. [13] illustrated

some significant data management challenges during the process of learning and managing deep learning models and presented the ModelHub system, including command-line version management tool and domain-specific language to address them. Cai et al. [14] applied SQL-based specification, simulation, and querying of database to Bayesian ML. Kraska et al. [15] designed MLbase following the idea of the database system to make ML applicable to datasets of various sizes. Mlog [16] is a declarative language that integrates ML into data management systems to deal with the data processing related to ML by operating over tensors with linear algebra. Kara et al. [17] explored the stochastic coordinate-descent method for the integration of ML into a column-store database to train generalized linear models.

More and more specific open-source platforms are designed to support ML algorithms and make them more scalable based on RDBMS or distributed platforms. Many academic papers provided the design and description for the projects recognized and applied in the industry. Gao et al. [18] proposed a concise declarative language named BUDS to implement ML algorithms on distributed computing platforms. MADlib [19] is a library used to provide SQL-based algorithms for ML that run at scale within a database engine. SystemML [20] provided an end-to-end description of declarative ML to run ML algorithms transparently on Spark. Apache Mahout [21] is a distributed linear algebra framework primarily used for creating scalable ML algorithms. Spark MLlib [22] is a scalable library to make practical

ML easy.

2.3 Machine Learning Optimization

The latest advances in ML are supported by new algorithms, availability of big explosive data, and continuous improvements in computing efficiency [23]. With the widespread application of ML, more and more people pay attention to its efficiency. Through reviewing and commenting on numerical optimization algorithms in ML applications, Leon, Frank and Jorge [24] explained how optimization problems arise for the ML area and their challenges. Dunjko et al. [25] from the perspective of quantum, proposed a method to improve the efficiency of ML. Park et al. [26] designed a specialized computing stack for supporting the scale-out acceleration of many ML algorithms and allowing programmers to break away from hardware design. Snoek, Larochelle and Adams [27] described their automatic method to optimize the performance of any given learning algorithm on the problem through the framework of Bayesian optimization and showed dramatic improvement on some widely used ML models. Mahajan et al. [28] integrated DAnA with PostgreSQL to realize an efficient hardware accelerator, which can map in-database advanced analytics queries for various ML algorithms. To deal with the repeatedly changing workflows of ML and shorten the time to attain the desired results, Xin et al. [29] designed a system that can accelerate the iteration and feedback intelligently.

2.4 Factorized Database

Factorized database is one exact representation for relational data utilizing the law of relational algebra to share the redundant data and reduce the computation in the query result, such as the Cartesian product over union [30]. Bakibayev, Olteanu and Zavodny [31] used compact factorized representations at the physical layer to reduce data redundancy and apply the queries for the in-memory datasets. Bakibayev et al. [32] expanded the factorized database to support a broader range of queries with aggregates and ordering by proposing new optimization and evaluation techniques. Rendle [33] proposed a method for linear regression and factorization machine models to encode the repeating patterns in the feature vectors to the predictor variables generated from relational data to get a considerable speed-up in the computation, but it only supports the in-memory datasets. Olteanu and Zavodny [34] applied two methods of factorized representations: namely f-representations and d-representations, for results of equi-join queries and compared their succinctness. Olteanu and Schleich [35] illustrated a system named F for building regression models over the computation and representation of materialized views to avoid redundancy by factorizing data and computation in memory. The factorized representation provides the basic idea to represent relations with join dependencies using algebraically equivalent forms. Here we will generalize this idea to applying

the ML algorithms for the data stored in the database.

2.5 Machine Learning over Normalized Data

As to learning the models over normalized datasets, scholars have made some contributions. Most of these works focus on specific ML algorithms. Kumar, Naughton and Patel [2] recognized the problems existing in the results after joins and developed specific algorithms to build generalized linear models using the factorized computation. Although their application scenarios are limited, it provides a standard direction for subsequent research. Schleich, Olteanu and Ciucanu [36] exploited a new structure to rewrite the objective function for the linear regression to decouple the co-factors computation of model parameters. Their factorized approach can be applied to complex user-defined aggregate functions instead of joins and simple aggregates. Chen et al. [3] utilized linear algebra operators to develop a new framework for generalizing the benefits of factorized ML over normalized data in a unified way and yielded significant speed-ups on several ML algorithms such as Linear Regression and K-Means clustering. In the following article, Li, Chen and Kumar [37] extended nonlinear operators for optimizing quadratic feature interactions within the factorized linear algebra framework for the ML over normalized data. Khamis et al. [38] implemented a gradient descent solver to optimize problems by iteratively improving the solution using normalized data. Cheng and Koudas [4] presented an

algorithm to factorize construction over normalized data focusing on the restricted case of Independent Gaussian Mixture Models (IGMM). Yang et al. [5] investigated the case of learning Support Vector Machines (SVM) via factorizing Gaussian kernel over normalized data from the view of linear algebra operations. However, their research did not involve the nonlinear models we have chosen or any dimensionality reduction method. This thesis intends to provide a more enlightening analysis system to expand its application scope further. It is the extension of our work [6] adding the new ML application scenario (PPCA) with more detailed theoretical analysis and thorough experimental reports.

At the same time, some articles discussed the feasibility and applicability of ML on normalized data. Kumar et al. [39] theoretically identified avoiding features brought in by joins may cause a decrease in accuracy for some cases and provided rules to measure the effects through simulations. They also conducted experiments to validate their rules for determining when it is safe to avoid joins. Kumar et al. [40] demonstrated an integrated toolkit for databases and models named Santoku. It is used to decide whether to denormalized the ML datasets based on factorized learning.

3 Preliminaries

This section will present materials about three ML application scenarios (GMM, NN and PPCA) discussed in the following thesis and introduce the notations involved. These are three standard ML models with different applicable scopes. We take the most common training method for each of them suitable for the factorized idea. This thesis only focuses on the efficiency improvement before and after using factorization for a specific application scenario. We do not consider comparing efficiency among different training methods in the same application scenario or different ML application scenarios.

3.1 Gaussian Mixture Models (GMM)

GMM is an unsupervised data clustering method for comprising a fixed number of Gaussian distributions [41]. The purpose of model learning is to estimate the parameters of the individual normal distribution components. Different from the previous paper [4], in this thesis, we consider the most general case for GMM with

arbitrary covariance matrices. Table 3.1 gives the notations used in the GMM.

Table 3.1: Notations used in GMM

Symbol	**Meaning**
K	Number of clusters
μ_k	Mean of component k
Σ_k	Covariance of component k
π_k	Mixing coefficients of component k
z	Latent variable
γ_k	Probability that x is generated by component k

Assume we are given N training data points $\mathbf{x}^{(n)}, 1 \leq n \leq N$ with dimension d. The probability density function of the k-th Gaussian component in the mixture model is:

$$\mathcal{N}(\mathbf{x}^{(n)}|\mu_k, \Sigma_k) = \frac{1}{\sqrt{(2\pi)^d|\Sigma_k|}} e^{-\frac{1}{2}(\mathbf{x}^{(n)}-\mu_k)^T\Sigma_k^{-1}(\mathbf{x}^{(n)}-\mu_k)} \tag{3.1}$$

The distribution of a mixture of K Gaussian components is:

$$\mathcal{P}(\mathbf{x}^n) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}^{(n)}|\mu_k, \Sigma_k) \quad s.t. \sum_{k=1}^{K} \pi_k = 1 \tag{3.2}$$

Expectation-Maximization (EM) algorithm is one of the most widely used methods to calculate the parameters [42]. It applies an iterative way to identify the maximum likelihood when the model contains an unobserved latent variable. The process includes two steps: estimation step (E-step) and maximization step (M-step). E-step is to estimate the expected value for each latent variable and M-step

is to optimize the parameters in the distribution using maximum likelihood. The EM algorithm starts with initial values of parameters and iteratively updates these values through repeating E-steps and M-steps until the convergence criteria are met.

For GMM, $\mathbf{z}$ is a latent variable, which means the probability of a given data point $\mathbf{x}^{(n)}$ from component k. In the E-step, the posterior distribution of $\mathbf{z}^{(n)}$ is updated using the current parameters μ_k, Σ_k and π_k:

$$\gamma_k^{(n)} = \mathcal{P}(\mathbf{z}^{(n)} = k|\mathbf{x}^{(n)}) = \frac{\pi_k \mathcal{N}(\mathbf{x}^{(n)}|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}^{(n)}|\mu_j, \Sigma_j)} \tag{3.3}$$

In M-step, μ_k, Σ_k and π_k are re-estimated using Maximum Likelihood Estimation (MLE) given the current γ_k using the following equations:

$$\mu_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_k^{(n)} \mathbf{x}^{(n)} \tag{3.4}$$

$$\Sigma_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_k^{(n)} (\mathbf{x}^{(n)} - \mu_k)(\mathbf{x}^{(n)} - \mu_k)^T \tag{3.5}$$

$$\pi_k = \frac{N_k}{N} \quad \text{with} \quad N_k = \sum_{n=1}^{N} \gamma_k^{(n)} \tag{3.6}$$

One general criterion to judge convergence is the difference of the log-likelihood between two consecutive iterations is less than a threshold set in advance. The log-likelihood function can be represented as:

$$\ln \mathcal{P}(\mathbf{x}|\pi, \mu, \Sigma) = \sum_{n=1}^{N} ln(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}^{(n)}|\mu_k, \Sigma_k)) \tag{3.7}$$

3.2 Neural Network (NN)

A neural network is composed of simple adaptable units for simulating the reaction of the biological nervous system to real-world objects [43]. Nowadays, this framework is used in the ML area, applying a series of algorithms to identify the relationships between large amounts of data. NN is one supervised regression model, which has gained popularity recently in data analysis with the development of deep learning [44]. Table 3.2 gives the notations used in the NN.

Table 3.2: Notations used in NN

Symbol	**Meaning**
h_j	Hidden unit in first layer
l_k	Hidden unit in second layer
p_j	Input value of h_j
q_k	Input value of l_k
w_{ji}	Weight between x_i and h_j
w_{kj}	Weight between h_j and l_k
b	Bias
n_h	Number of units in first layer
n_l	Number of units in second layer
n_o	Number of units in output layer
O	Output of the network
Y	Learning target
α	Learning rate

The input value of j-th hidden unit in first layer, where $j \in \{1 \dots n_h\}$, can be

described as a sequence of linear transformations shown in Equation 3.8. For one input data point $\mathbf{x}^{(n)}$ with dimension d, ${x_i}^{(n)}$ is one input feature, where $i \in \{1 \dots d\}$. The superscript [1] denotes the corresponding parameters at the first hidden layer of the network.

$$p_j^{(n)} = \sum_{i=1}^{d} w_{ji}^{[1]} {x_i}^{(n)} + b_j^{[1]} \quad \text{where} \quad j \in \{1 \dots n_h\} \tag{3.8}$$

After receiving the input value, the first hidden unit generates output value through a differentiable activation function f:

$$h_j^{(n)} = f(p_j^{(n)}) \tag{3.9}$$

The most widely used activation functions include Sigmoid, $\sigma(a) = \frac{1}{1+exp(-a)}$, ReLU, $ReLU(a) = max(0, a)$, and Tanh, $tanh(a) = 2\sigma(2a) - 1$. All the outputs of the first hidden layer are combined through similar linear transformations to be the input value of the hidden unit in the second hidden layer:

$$q_k^{(n)} = \sum_{j=1}^{n_h} w_{kj}^{[2]} {h_j}^{(n)} + b_k^{[2]} \quad \text{where} \quad k \in \{1 \dots n_l\} \tag{3.10}$$

Then, we can get the outputs of the second layer by applying the activation function. In this way, a multi-layer network will be calculated and the output value $O^{(n)}$ of the last layer is the final output of the whole neural network. The training process determines the weights and the bias between two adjacent layers according to the training data.

The Backward Propagation (BP) algorithm is an excellent method to minimize the error function for updating the parameters. It is based on Gradient Descent(GD) method, which includes Batch Gradient Descent, Mini-batch Gradient Descent and Stochastic Gradient Descent. There is no potential to explore the redundancy among different data points for Stochastic Gradient Descent because the parameters are changing when training every data point. BP is also an iterative learning process, which includes two phases. The first phase, also called the forward propagation phase, provides data to the input layer and forwards the signal layer by layer utilizing the current weights w and bias b until getting the output O to calculate the error E. One of the most commonly used ways to calculate E is to get the cumulative error of N data points, which can be represented as the following error function:

$$E = \frac{1}{2N} \sum_{n=1}^{N} \sum_{m=1}^{n_o} (O_m^{(n)} - Y_m^{(n)})^2 \tag{3.11}$$

The second phase, also called the backward propagation phase, is to propagate the error back to the hidden layer units to calculate the gradients of the units and update all the parameters w and b between each two adjacent layers of the network:

$$w = w + \Delta w \quad \text{where} \quad \Delta w = -\alpha \frac{\partial E}{\partial w} \tag{3.12}$$

$$b = b + \Delta b \quad \text{where} \quad \Delta b = -\alpha \frac{\partial E}{\partial b} \tag{3.13}$$

$\frac{\partial E}{\partial w}$ is the derivative of error with respect to weights evaluated by applying the

chain rule [44]. For example, suppose there is only one hidden layer, to update w_{kj}:

$$w_{kj} = w_{kj} - \alpha \frac{\partial E}{\partial w_{kj}}$$
$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial O_m} \quad \frac{\partial O_m}{\partial q_k} \quad \frac{\partial q_k}{\partial w_{kj}} \tag{3.14}$$

The further expansions of the gradients depend on the choice of the activation function. When the error reaches the expected small value, the iteration will stop.

3.3 Probabilistic Principal Component Analysis (PPCA)

Dimensionality reduction is a critical part of the data preparation in ML, which refers to the technologies that map the data points in the original high-dimensional space to the low-dimensional space. The significance lies in the extraction of useful information and the facilitation of data visualization. Principal component analysis (PCA) is one of the most prevalent dimensionality reduction techniques [45]. The target is to calculate the orthogonal projection of the original data onto a lower principal subspace of lower dimensionality by looking for the maximum variance of the projected data or the minimum average projection cost.

One effective way to solve PCA is to regard it as the maximum likelihood solution of a probabilistic latent variable model. Thus, the Probabilistic PCA (PPCA) technique is proposed [44]. PPCA is the optimized algorithm for regular PCA to estimate the principal axes without the intermediate steps to calculate the complex data covariance matrix [46] and provide the probabilistic distribution of

the data. It is more flexible and efficient than regular PCA when the dataset is huge or exists missing data. Table 3.3 provides the notations used in PPCA.

Table 3.3: Notations used in PPCA

Symbol	**Meaning**
p	Dimension after reduction
$\mathbf{z}$	Gaussian lalent variable ($p \times 1$)
W	Principal axes ($d \times p$)
μ	Mean
ϵ	Zero-mean noise variable
σ	Residual variance

Specifically, for a data point $\mathbf{x}^{(n)}$ with dimension d, we aim to find a latent variable $\mathbf{z}^{(n)}$ with lower dimension p. Thus, $\mathbf{x}^{(n)}$ is defined by the linear transformation of $\mathbf{z}^{(n)}$ plus d-dimensional Gaussian noise:

$$\mathbf{x}^{(n)} = W\mathbf{z}^{(n)} + \mu + \epsilon \quad \text{where} \quad \epsilon \sim \mathcal{N}(0, \sigma^2 I) \tag{3.15}$$

The prior distribution over $\mathbf{z}^{(n)}$ is defined as following, which is a zero-mean unit-covariance Gaussian:

$$\mathcal{P}(\mathbf{z}^{(n)}) = \mathcal{N}(0, I) \tag{3.16}$$

The conditional distribution for $\mathbf{x}^{(n)}$ can be presented as:

$$\mathcal{P}(\mathbf{x}^{(n)}|\mathbf{z}^{(n)}) = \mathcal{N}(W\mathbf{z}^{(n)} + \mu, \sigma^2 I) \tag{3.17}$$

Thus, it is a typical example of the linear Gaussian framework, because all of the marginal and conditional distributions are Gaussian. The purpose for training is

to determine the parameters W, μ and σ^2 in the model. The exact closed-form solution of μ is derived by the log likelihood [44]:

$$\mu = \overline{\mathbf{x}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}^{(n)} \tag{3.18}$$

However, getting the exact closed-form solutions for W and σ^2 is computationally intensive. As each data point $\mathbf{x}^{(n)}$ corresponds to a latent variable, we can utilize the EM algorithm to calculate parameters through maximum likelihood estimation. According to Equation 3.16 and 3.17, the log-likelihood function can be get:

$$ln\mathcal{P}(\mathbf{x},\mathbf{z}) = \sum_{n=1}^{N}\left\{\mathcal{P}(\mathbf{x}^{(n)}|\mathbf{z}^{(n)}) + \mathcal{P}(\mathbf{z}^{(n)})\right\} \tag{3.19}$$

For PPCA, in E-step of EM algorithm, we use the previous parameters W and σ^2 to get the expectation with respect to the posterior distribution of $\mathbf{z}^{(n)}$ and $\mathbf{z}^{(n)}{\mathbf{z}^{(n)}}^{\mathrm{T}}$:

$$\mathbb{E}[\mathbf{z}^{(n)}] = M^{-1}W^{\mathrm{T}}(\mathbf{x}^{(n)} - \overline{\mathbf{x}}) \tag{3.20}$$

$$\mathbb{E}[\mathbf{z}^{(n)}{\mathbf{z}^{(n)}}^{\mathrm{T}}] = \sigma^2 M^{-1} + \mathbb{E}[\mathbf{z}^{(n)}]\mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}} \tag{3.21}$$

$$\text{where} \quad M = W^{\mathrm{T}}W + \sigma^2 I$$

In M-step, we maximum the likelihood function with respect to W and σ^2 to get the equations of new value:

$$W_{new} = \left[\sum_{n=1}^{N}(\mathbf{x}^{(n)} - \overline{\mathbf{x}})\mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}}\right]\left[\sum_{n=1}^{N}\mathbb{E}[\mathbf{z}^{(n)}{\mathbf{z}^{(n)}}^{\mathrm{T}}]\right]^{-1} \tag{3.22}$$

$$\sigma^2_{new} = \frac{1}{ND}\sum_{n=1}^{N}\Big\{\left\|\mathbf{x}^{(n)} - \overline{\mathbf{x}}\right\|^2 - 2\mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}} {W_{new}}^{\mathrm{T}}(\mathbf{x}^{(n)} - \overline{\mathbf{x}}) + tr(\mathbb{E}[\mathbf{z}^{(n)}{\mathbf{z}^{(n)}}^{\mathrm{T}}]{W_{new}}^{\mathrm{T}} W_{new})\Big\} \tag{3.23}$$

Similarly, the E-step and M-step will be repeated until meeting a convergence criteria.

4 Problem Description and Algorithm Frameworks

In this chapter, we first formally define the problem we focus on in this thesis and then introduce three algorithms involved in our analysis.

4.1 Problem Description

In order to make the problem more clear, we mainly consider scenario of binary join over normalized data. Following the style presented in [2], there are two relations $\mathbf{S}(\underline{SID}, \mathbf{X}_S, FK)$ and $\mathbf{R}(\underline{RID}, \mathbf{X}_R)$ with a primary/foreign-key relationship ($\mathbf{S}.FK$ refers to $\mathbf{R}.RID$), where $\mathbf{X}_S$ and $\mathbf{X}_R$ are the feature matrices in $\mathbf{S}$ and $\mathbf{R}$ respectively. We assume that relation $\mathbf{S}$ has n_S tuples and relation $\mathbf{R}$ has n_R tuples satisfying $n_S > n_R$. There are d_S features in n-th feature vector ${\mathbf{x}_S}^{(n)}$ of $\mathbf{X}_S$ and d_R features in ${\mathbf{x}_R}^{(n)}$ of $\mathbf{X}_R$. The feature matrix $\mathbf{X}$ in join result is the concatenation of the features from the joining tuples of $\mathbf{S}$ and $\mathbf{R}$. There are N feature vectors in $\mathbf{X}$ where $N = n_S$ and each vector $\mathbf{x}^{(n)}$ has d features satisfying $d = d_S + d_R$. When learning a GMM or using PPCA to reduce dimensionality, the result of the

projected equi-join table $\mathbf{T}$ can be expressed as:

$$\mathbf{T}(\underline{SID}, RID, \mathbf{X}) = \mathbf{T}(\underline{SID}, RID, [\mathbf{X}_S \mathbf{X}_R]) \leftarrow \Pi_{SID, RID, \mathbf{X}_S, \mathbf{X}_R}(\mathbf{R} \bowtie_{RID=FK} \mathbf{S}) \tag{4.1}$$

For the case of NN, relation $\mathbf{S}(\underline{SID}, Y, \mathbf{X}_S, FK)$ has an additional attribute Y, which is the target for learning purpose, so the projected schema becomes $\mathbf{T}$ $(\underline{SID}, RID, Y, [\mathbf{X}_S \mathbf{X}_R])$. Table 4.1 summarizes the notations used in this paper.

We generalize the binary case to multi-way case and take GMM as an example to analyze the factorized method in Section 5.3. There are q attribute tables $\mathbf{R}_i$ where $i \in \{1 \dots q\}$ to join with $\mathbf{S}$, which has q foreign keys. To ease notation, we denote $\mathbf{S}$ as $\mathbf{R}_0$ and d_S as d_{R_0}. Therefore, we perform the factorization over a join sequence for $\mathbf{R}_i$ where $i \in \{0 \dots q\}$, which can be represented as:

$$\begin{aligned} &\mathbf{T}(\underline{SID}, RID, [\mathbf{X}_{R_0} \mathbf{X}_{R_1} \dots \mathbf{X}_{R_q}]) \\ &\quad \leftarrow \Pi_{\underline{SID}, RID, \mathbf{X}_{R_0}, \mathbf{X}_{R_1} \dots \mathbf{X}_{R_q}} \mathbf{R}_1 \bowtie_{RID_1=FK_1} \dots \mathbf{R}_q \bowtie_{RID_q=FK_q} \mathbf{R}_0 \end{aligned} \tag{4.2}$$

The data in table $\mathbf{S}$ and $\mathbf{R}$ is represented in a normalized way, but table $\mathbf{T}$ reintroduces the redundancy back due to the join via the relationship between $\mathbf{S}.FK$ and $\mathbf{R}.RID$. In the following discussion, we will explore the effective way to eliminate the negative impacts. All the proposed algorithms are equally applicable to all types of joins, such as block nested-loop join and hash join.

Table 4.1: Notations used in the algorithms

Symbol	Meaning
$\mathbf{R}$	Relation
$\mathbf{S}$	Relation
R_i	i-th bach of $\mathbf{R}$
S_i	i-th bach of $\mathbf{S}$
$\mathbf{T}$	Join result table
$\|R\|$	Number of pages in $\mathbf{R}$
$\|S\|$	Number of pages in $\mathbf{S}$
$\|T\|$	Number of pages in $\mathbf{T}$
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 $\mathbf{T}$ ($d = d_R + d_S$)
$\mathbf{X}$	Feature matrix in $\mathbf{T}$
$\mathbf{X}_R$	Feature matrix in $\mathbf{R}$
$\mathbf{X}_S$	Feature matrix in $\mathbf{S}$
$\mathbf{x}^{(n)}$	n-th feature vector in $\mathbf{T}$
$\mathbf{x}_R^{(n)}$	The part from $\mathbf{R}$ in $\mathbf{x}^{(n)}$
$\mathbf{x}_S^{(n)}$	The part from $\mathbf{S}$ in $\mathbf{x}^{(n)}$
$x_i^{(n)}$	i-th feature in $\mathbf{x}^{(n)}$
$Iter$	Number of iterations in training algorithm
m	Number of times reading data in one iteration

4.2 Three Algorithm Frameworks

In this part, we introduce the frameworks for the baseline approaches (M-algorithm and S-algorithm) and our proposal method (F-algorithm) from a general perspec-

tive. For GMM, NN and PPCA, the corresponding algorithms will be concreted in Chapter 5, 6 and 7, such as M-GMM, S-GMM and F-GMM. We also analyze the costs to show the trade-offs may exist and the efficiency improvement for the factorized way.

4.2.1 Framework for M-algorithm

M-algorithm is the baseline algorithm to materialize the join results on the disk, which most data analysts widely use, as we introduced before. Its process is depicted in Figure 4.1. Essentially, we need to get a new table **T** after the join of the normalized tables **S** and **R** involved, materializing all the results in **T** on the disk, and reading them subsequently to complete the training process of the target model.

For the cost of the M-algorithm, we analyze it from two aspects: I/O cost and CPU cost. More specifically, I/O cost includes the time to fetch data from table **S** and **R** for joins, write it back to the database and read data from table **T** for training. CPU cost includes the time of a series operation for **S** and **R** during the join process and the computation time for updating the parameters during the training process.

For I/O cost, during the phase of data preparation, it needs to get table **S** and **R** from the database by reading $|R|+|R||S|$ pages for nested loop join. As to block nested loops join and hash join, that will be $|R|+\frac{|R|}{BlockSize}|S|$ pages and $3(|R|+|S|)$

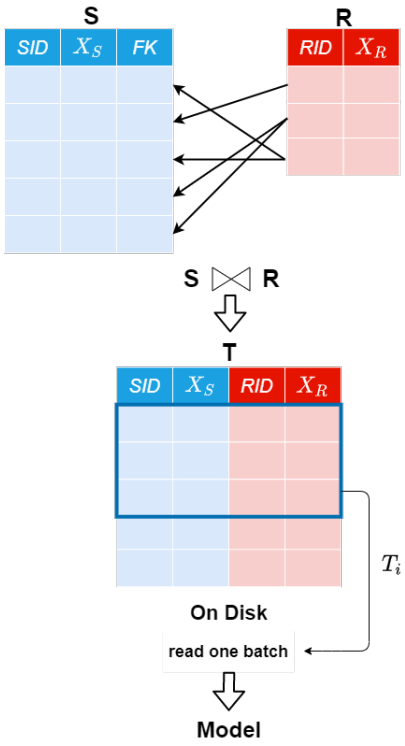


Figure 4.1: M-algorithm framework

pages, respectively. We refer to this time as $joinIOcost$. After completing the join operation, $|T|$ pages should be written back to the database. When training the model, we need to read all the tuples from $\mathbf{T}$ into memory m times during one iteration, which is determined by the training algorithm. For example, in GMM, all the data points will be read three times in one iteration because there are three parameters in the model and the update of each parameter requires all tuples to participate. In summary, the total I/O cost of M-algorithm is $joinIOcost + |T| + iter \times m \times |T|$.

As to CPU cost, different join methods consume different computation time in

the join process, which refer to $joinCPUcost$. Taking hash join as an example, $joinCPUcost$ includes the time of partitioning tables, constructing hash and comparing keys. When updating the parameters using ML method, $N \times d$ fields of $\mathbf{T}$ involved in the calculation because there are N tuples in $\mathbf{T}$ and d features in each tuple. Denote the time for one field to do the calculation in ML is $mlCPUcost$. Thus, the total CPU cost of the M-algorithm is $joinCPUcost + (N \times d) \times mlCPUcost$.

4.2.2 Framework for S-algorithm

S-algorithm is another baseline approach computing the joins in batches on the fly, adopting the idea of stream. Its aim is to eliminate the impact of materialization in the M-algorithm, but it still needs to feed non-normalized data into the training model. Specifically, we perform the joins of $\mathbf{S}$ and $\mathbf{R}$ on the fly by reading one batch of $\mathbf{R}$, retrieve tuples from $\mathbf{S}$ and form one sub-table of final join results in the memory to compute the model parameters immediately. In this way, we read all batches of $\mathbf{R}$ sequentially to execute the joins operation with $\mathbf{S}$ and compute the ML model. The whole process does not rewrite the join results on the disk so that there is no table $\mathbf{T}$. Figure 4.2 illustrates how the data be fed to the model, where S_i and R_i are the i-th batches of $\mathbf{S}$ and $\mathbf{R}$ respectively.

Since the join results are not stored, we need to fetch data from the original tables to execute the join operation for m times in one iteration. The intermediate

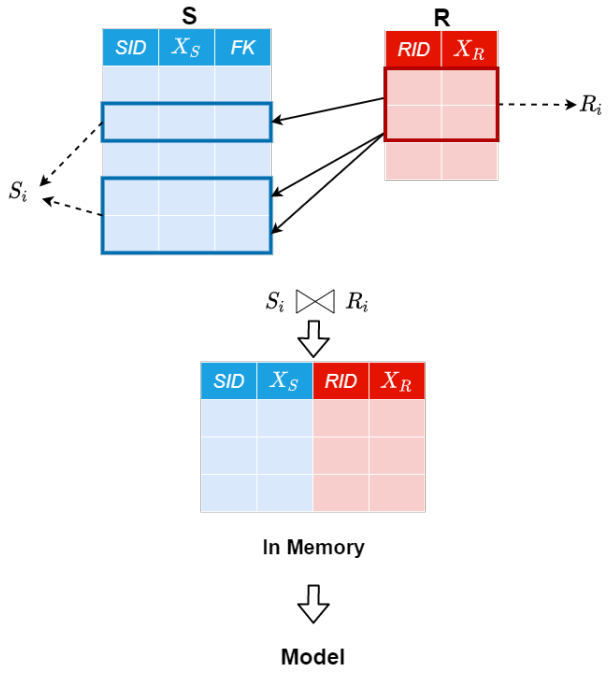


Figure 4.2: S-algorithm framework

results of the data will be fed to the model intermediately until the end of the training involving them. So, the total I/O cost for S-algorithm is $iter \times m \times joinIOcost$. Compared with M-algorithm's I/O cost, when $joinIOcost < \frac{iter \times m+1}{iter \times m-1} \times |T|$, S-algorithm has less I/O cost.

Meanwhile, the $joinCPUcost$ during the join process will increase $iter \times m$ times. We can find that the difference between S-algorithm and M-algorithm only exists in the data preparation phase. All the N tuples with d features after the join will participate in calculating the model no matter whether the tuples are stored in the disk or not. That is to say, S-algorithm has the same computation cost

for updating the ML model with M-algorithm. Therefore, the total CPU cost for S-algorithm is $iter \times m \times joinCPUcost + (N \times d) \times mlCPUcost$, which will be no less than that of the M-algorithm.

4.2.3 Framework for F-algorithm

To improve the efficiency of the ML applications by avoiding the redundancy bring by joins, we propose our solution named F-algorithm. Compared with the data preparation phase of the M-algorithm or S-algorithm, the actual computation and storage of the join results do not take place. This approach turns the previous task into finding the matching tuples from normalized tables according to the matching primary-key and foreign-key. In the process of training, instead of feeding every entire joined tuple to the ML algorithm, we factorize the initial equations for calculating required parameters into independent parts involving the feature vectors from the corresponding normalized tables, respectively.

The general idea of the F-algorithm is described in Figure 4.3. We read one batch R_i from table $\mathbf{R}$ in the database, which is only used to probe table $\mathbf{S}$ and identify the matching tuples (S_i) in it. Then, take the feature matrices $\mathbf{X}_{S_i}$ and $\mathbf{X}_{R_i}$ from S_i and R_i as the input to the training process. Parameters are independently computed for the parts of the factorized equations involving $\mathbf{X}_{S_i}$ or $\mathbf{X}_{R_i}$. The final result is obtained by recombining the two parts according to the matched foreign

key. Thus, in this way, there is no need to get every entire tuple after joins for calculation.

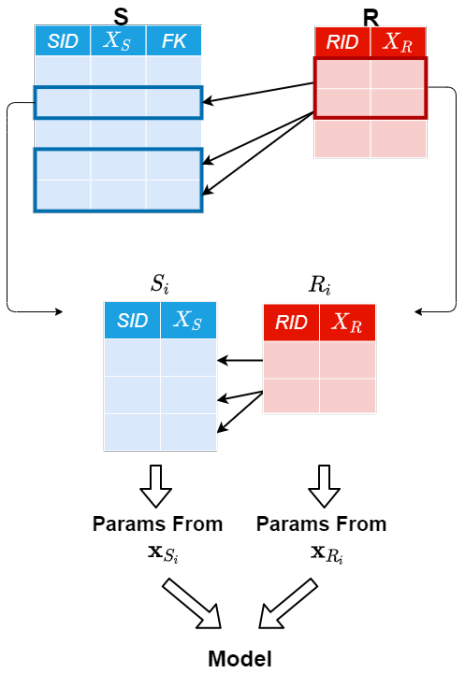


Figure 4.3: F-algorithm framework

For the costs of the F-algorithm, we also analyze them from two aspects. During the phase of data preparation, it reads the data from **S** and **R** to do the matching operation $iter \times m$ times due to the results not be saved. Thus, the total I/O cost is $iter \times m \times joinIOcost$, the same as S-algorithm. Although it simplifies the calculation of join operation that to find tuples matching the tuples in the **S** referring the target primary-keys in **R**, the actual time of CPU for this probing process is almost $joinCPUcost$. However, obvious efficiency improvements exist

in the calculation process of the model. Since only the data in two normalized tables **S** and **R** are involved in the calculation for ML, the total number of fields is $n_S \times d_S + n_R \times d_R$, where $n_S = N$ and $n_R < N$. Thus, the total CPU cost for F-algorithm is $iter \times m \times joinCPUcost + (n_S \times d_S + n_R \times d_R) \times mlCPUcost$.

To guarantee the correctness of this approach, the critical issue is to make sure all the steps in the training process needing feature matrices are exactly decomposed and have no approximate calculation. Besides, other factors, such as the input data, the estimation of parameters, and the number of iterations required for training, cannot be changed. Therefore, we display the decomposition process in detail for all the ML application scenarios considered in this thesis to prove its feasibility and accuracy.

5 Algorithms for GMM

In this chapter, we apply the frameworks of three approaches in Section 4.2 for the general case of GMM with arbitrary covariance matrices. We formally introduce the baseline algorithms (M-GMM and S-GMM) and then give the detailed decomposition and analysis for the computation in F-GMM.

5.1 Baseline Algorithms: M-GMM and S-GMM

As we introduced in Section 4.2.1, M-GMM is the baseline algorithm adopted nowadays to training the model. Algorithm 1 describes how to prepare the data and how to apply the EM algorithm in GMM. Before calculating the parameters, we execute joins in the database for table **S** and table **R**. The results are materialized in a large table **T**, which is also stored in the database. Due to memory limitations, it reads **T** subsequently in batches with the appropriate size to the memory as input data points. Then, follow the steps in Algorithm 1 to update the parameters until it meets the condition of termination. According to the training process, in Lines 8, 14 and 20, feature vectors $\mathbf{x}^{(n)}$ participate in the calculation. Parameters μ_k and

σ_k are updated through the sum for all N data points in the equations. Therefore, we need to read table **T** three times in each iteration, respectively, in Lines 6, 12 and 18.

Algorithm 1 Algorithm M-GMM

1: **Data preparation:** Apply join of table **S** and table **R** in the database and materialize the results (table **T**) on the disk.
2: **Initialize** μ_k**,** Σ_k **and** π_k **:**
3: **repeat**
4: **E-step:**
5: **for** i in [*1 : number of batches*] **do**
6: Read batch i of **T** into memory
7: Update the posterior distribution of **z**:
8: $\gamma_k^{(n)} = \frac{\pi_k \mathcal{N}(\mathbf{x}^{(n)}|\mu_k,\Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}^{(n)}|\mu_j,\Sigma_j)} \qquad \forall n \in$ batch i
9: **end for**
10: **M-step:**
11: **for** i in [*1 : number of batches*] **do**
12: Read batch i of **T** into memory
13: Update the sum results of μ_k:
14: $Sum_{\mu_k} += \sum_{n=1}^{batchsize} \gamma_k^{(n)} \mathbf{x}^{(n)}$
15: **end for**
16: Update $\mu_k = \frac{1}{N_k} Sum_{\mu_k}$
17: **for** i in [*1 : number of batches*] **do**
18: Read batch i of **T** into memory
19: Update the sum results of σ_k:
20: $Sum_{\Sigma_k} += \sum_{n=1}^{batchsize} \gamma_k^{(n)} (\mathbf{x}^{(n)} - \mu_k)(\mathbf{x}^{(n)} - \mu_k)^T$
21: **end for**
22: Update $\Sigma_k = \frac{1}{N_k} Sum_{\Sigma_k}$
23: Update $\pi_k = \frac{N_k}{N}$ with $N_k = \sum_{n=1}^{N} \gamma_k^{(n)}$
24: **until** Convergence

The only difference between algorithm S-GMM and M-GMM is S-GMM exe-

cutes joins on the fly. So the calculation process is essentially the same as Algorithm 1, but the way to obtain data has changed. It does not perform Line 1, which is materializing the table $\mathbf{T}$ in the database. In addition, in Lines 6, 12 and 18, instead of reading i-th batch from table $\mathbf{T}$, it computes joins for S_i and R_i (as described in Figure 4.2) using the primary/foreign-key relationship. It gets the sub-table in the memory as the input to update the parameters in Lines 8, 14 and 20.

5.2 Algorithm F-GMM and Decomposition Process

In this section, we will provide the improved algorithm F-GMM to analyze how the Algorithm 1 to be factorized for the computation. The difference from S-GMM is that, instead of feeding every joined tuple to the model for updates of $\gamma_k^{(n)}$, μ_k and Σ_k in Lines 8, 14 and 20, we factorize the equations into two parts involving the features from $\mathbf{S}$ and $\mathbf{R}$ respectively. Such factorization is required during the E-step as well as the M-step. We will give a detailed analysis for them.

5.2.1 E-step

In Line 8, $\gamma_k^{(n)}$ involves the feature vectors $\mathbf{x}^{(n)}$ for calculating $\mathcal{N}(\mathbf{x}^{(n)}|\mu_k, \Sigma_k)$, which is given in Equation 3.11. They are not required in the calculation of $\frac{1}{\sqrt{(2\pi)^d|\Sigma_k|}}$, but in the part of $(\mathbf{x}^{(n)} - \mu_k)^T\Sigma_k^{-1}(\mathbf{x}^{(n)} - \mu_k)$. To make the equation more intuitive and clear, we ignore the subscript k in the expanded equations for all the parameters belonging to the k^{th} Gaussian component and denote Σ^{-1} as I. The decomposition

process is as follows :

$$(\mathbf{x}^{(n)} - \mu_k)^T \underset{1\times 1}{I_k} (\mathbf{x}^{(n)} - \mu_k) = \underset{1\times d}{\begin{bmatrix} x_1^{(n)} - \mu_1 & x_2^{(n)} - \mu_2 & \cdots & x_d^{(n)} - \mu_d \end{bmatrix}} \times \underset{d\times d}{\begin{bmatrix} I_{1,1} & I_{1,2} & \cdots & I_{1,d} \\ I_{2,1} & I_{2,2} & \cdots & I_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ I_{d,1} & I_{d,2} & \cdots & I_{d,d} \end{bmatrix}} \times \underset{d\times 1}{\begin{bmatrix} x_1^{(n)} - \mu_1 \\ x_2^{(n)} - \mu_2 \\ \vdots \\ x_d^{(n)} - \mu_d \end{bmatrix}} \tag{5.1}$$

To ease notation, we denote the first d_S dimensions of vector $\mathbf{x}^{(n)} - \mu_k$ as PF_S, in which features come from table $\mathbf{S}$, and the remaining d_R dimensions as PF_R. Therefore, $\mathbf{x}^{(n)} - \mu_k$ can be split into two parts due to $d = d_S + d_R$:

$$\underset{d_S\times 1}{PF_S} = \begin{bmatrix} x_1^{(n)} - \mu_1 \\ x_2^{(n)} - \mu_2 \\ \vdots \\ x_{d_S}^{(n)} - \mu_{d_S} \end{bmatrix} \qquad \underset{d_R\times 1}{PF_R} = \begin{bmatrix} x_{d_S+1}^{(n)} - \mu_{d_S+1} \\ x_{d_S+2}^{(n)} - \mu_{d_S+2} \\ \vdots \\ x_d^{(n)} - \mu_d \end{bmatrix} \tag{5.2}$$

Similarly, I_k can be divided into four parts:

$$\underset{d_S\times d_S}{I_{SS}} = \begin{bmatrix} I_{1,1} & \cdots & I_{1,d_S} \\ I_{2,1} & \cdots & I_{2,d_S} \\ \vdots & \ddots & \vdots \\ I_{d_S,1} & \cdots & I_{d_S,d_S} \end{bmatrix} \qquad \underset{d_S\times d_R}{I_{SR}} = \begin{bmatrix} I_{1,d_S+1} & \cdots & I_{1,d} \\ I_{2,d_S+1} & \cdots & I_{2,d} \\ \vdots & \ddots & \vdots \\ I_{d_S,d_S+1} & \cdots & I_{d_S,d} \end{bmatrix}$$

$$\underset{d_R\times d_S}{I_{RS}} = \begin{bmatrix} I_{d_S+1,1} & \cdots & I_{d_S+1,d_S} \\ I_{d_S+2,1} & \cdots & I_{d_S+2,d_S} \\ \vdots & \ddots & \vdots \\ I_{d,1} & \cdots & I_{d,d_S} \end{bmatrix} \qquad \underset{d_R\times d_R}{I_{RR}} = \begin{bmatrix} I_{d_S+1,d_S+1} & \cdots & I_{d_S+1,d} \\ I_{d_S+2,d_S+1} & \cdots & I_{d_S+2,d} \\ \vdots & \ddots & \vdots \\ I_{d,d_S+1} & \cdots & I_{d,d} \end{bmatrix} \tag{5.3}$$

We denote:

$$\mathbf{SS} = {PF_S}^T \times I_{SS} \times PF_S \tag{5.4}$$

$$\mathbf{SR} = {PF_S}^T \times I_{SR} \times PF_R \tag{5.5}$$

$$\mathbf{RS} = {PF_R}^T \times I_{RS} \times PF_S \tag{5.6}$$

$$\mathbf{RR} = {PF_R}^T \times I_{RR} \times PF_R \tag{5.7}$$

Thus, the expansion in Equation 5.1 can be written as the sum of four parts:

$$(\mathbf{x}^{(n)} - \mu_k)^T I_k (\mathbf{x}^{(n)} - \mu_k) = \mathbf{SS} + \mathbf{SR} + \mathbf{RS} + \mathbf{RR} \tag{5.8}$$

Equations 5.4-5.7 are the multiplication of three small matrices consisted by PF_S or PF_R. Thus, in Equation 5.8, we successfully decompose the multiplication of three big matrices in Equation 5.1 into the parts that can be calculated use the feature vectors $\mathbf{x}_S^{(n)}$ and $\mathbf{x}_R^{(n)}$ from table **S** or table **R** instead of the entire tuple $\mathbf{x}^{(n)}$ in table **T**. As we know, each tuple in table **R** can match several tuples in table **S** through the primary/foreign-key relationship. Thus, in Equations 5.5 and 5.6, the results of PF_R can be reused for the matching PF_S to calculate **SR** and **RS**. In Equation 5.7, as **RR** only involves PF_R not PF_S, the results of **RR** can be calculated only once and reused for the matching **SS**, **SR** and **RS**. Finally, We can obtain the same calculation result of Equation 5.1 by computing PF_R and **RR** for n_R tuples in table **R** and combining all the matching results involved PF_S. After decomposition, there is no need to use the join results to calculate the parameters and it brings the potential for significant savings by removing the repeated calculations.

5.2.2 M-step

In M-step, there are three parameters that need to be updated: μ_k, Σ_k and π_k. From Equation 3.6 we can see that π_k does not involve the feature vectors of data

points and can be calculated using the current γ_k. For μ_k, Equation 3.4 can be decomposed into two parts containing features vectors from **S** or **R**:

$$\begin{aligned}\underset{d\times 1}{\mu_k} &= \frac{1}{N_k}\sum_{n=1}^{N}\gamma_k^{(n)}\mathbf{x}^{(n)} \\ &= \frac{1}{N_k}\times\sum_{n=1}^{N}\times\begin{bmatrix}\underset{d_S\times 1}{\gamma_k^{(n)}\mathbf{x}_S^{(n)}} \\ \\ \underset{d_R\times 1}{\gamma_k^{(n)}\mathbf{x}_R^{(n)}}\end{bmatrix}\end{aligned} \tag{5.9}$$

However, $\gamma_k^{(n)}$ changes with n and there is no opportunity to reuse the calculation results of $\gamma_k^{(n)}\mathbf{x}_R^{(n)}$ for all the matching $\mathbf{x}_R^{(n)}$ in **S**. But Equation 5.9 can be updated directly using the feature vectors from **S** and **R** based on the decomposition.

Then we focus on how to update Σ_k using Equation 3.5. We aim to calculate $(\mathbf{x}^{(n)}-\mu_k)(\mathbf{x}^{(n)}-\mu_k)^T$ in a factorized way before multiplying $\gamma_k^{(n)}$ and summing over all data points. Similarly, subscript k is ignored in the following expansions for simplifying notation:

$$\underset{d\times d}{(\mathbf{x}^{(n)}-\mu_k)(\mathbf{x}^{(n)}-\mu_k)^T} = \underset{d\times 1}{\begin{bmatrix}x_1^{(n)}-\mu_1 \\ x_2^{(n)}-\mu_2 \\ \vdots \\ x_d^{(n)}-\mu_d\end{bmatrix}}\times\underset{1\times d}{\begin{bmatrix}x_1^{(n)}-\mu_1 & x_2^{(n)}-\mu_2 & \cdots & x_d^{(n)}-\mu_d\end{bmatrix}} \tag{5.10}$$

$$= \begin{bmatrix}\underset{d_S\times d_S}{\mathbf{UL}} & \underset{d_S\times d_R}{\mathbf{UR}} \\ \underset{d_R\times d_S}{\mathbf{LL}} & \underset{d_R\times d_R}{\mathbf{LR}}\end{bmatrix} \tag{5.11}$$

where:

$$\mathbf{UL} = PF_S\times {PF_S}^T \tag{5.12}$$

$$\mathbf{UR} = PF_S\times {PF_R}^T \tag{5.13}$$

$$\mathbf{LL} = PF_R\times {PF_S}^T \tag{5.14}$$

$$\mathbf{LR} = PF_R\times {PF_R}^T \tag{5.15}$$

In Equation 5.10, the big matrix with dimension $d \times d$ can be factorized according to the source of feature vectors into four matrices with smaller dimensions in Equation 5.11: **UL**(upper left matrix), **UR**(upper right matrix), **LL**(lower left matrix) and **LR**(lower right matrix). Similarly, each of them can be represented as the simple multiplications of the intermediate results in Equation 5.2 calculated by $\mathbf{x}_S^{(n)}$ or $\mathbf{x}_R^{(n)}$ from **S** or **R** directly. For each unique $\mathbf{x}_R^{(n)}$, calculating the parts involved it and combining the parts involved $\mathbf{x}_S^{(n)}$ for all the matching cases. In particular, PF_R and **LR** are the parts only involved $\mathbf{x}_R^{(n)}$. Thus, they can be computed only once and then reused for any matching results, although they are not materialized on the disk or in the memory.

To measure the gains more clearly, we take $(\mathbf{x}^{(n)} - \mu_k)(\mathbf{x}^{(n)} - \mu_k)^T$ as an example to analyze the computational savings due to factorization and how redundancy influences its performance. Firstly, when we use the data from table **T** to calculate Equation 5.10, one tuple in **T** requires d subtractions and d^2 multiplications. τ_s and τ_m are used to represent the time required for one subtraction and one multiplication operation respectively. Thus, for all N tuples in **T**, the computation time τ is:

$$\tau = Nd(\tau_s + d\tau_m) \tag{5.16}$$

Next, we consider the changes after decomposition by F-GMM in Equation 5.11. PF_R is only calculated for n_R tuples in **R** and PF_S is calculated for n_S tuples in

S. We execute $n_S d_S + n_R d_R$ subtractions for PF_S and PF_R. In Equation 5.12-5.15, there are $n_S {d_S}^2$, $n_S d_S d_R$, $n_S d_R d_S$ and $n_R {d_R}^2$ multiplications respectively. Thus the total computation time τ' can be represented as:

$$\tau' = (n_S d_S + n_R d_R)\tau_s + (n_S {d_S}^2 + 2 n_S d_S d_R + n_R {d_R}^2)\tau_m \tag{5.17}$$

As $N = n_S$ and $d = d_S + d_R$, the time saving is:

$$\Delta\tau = \tau - \tau' = (n_S - n_R) d_R (\tau_s + d_R \tau_m) \tag{5.18}$$

and the saving rate is:

$$r = \frac{\Delta\tau}{\tau} = \frac{n_S - n_R}{n_S} \quad \frac{d_R}{d_S + d_R} \quad \frac{\tau_s + d_R \tau_m}{\tau_s + (d_S + d_R)\tau_m} \tag{5.19}$$

Equation 5.19 explains the underlying factors that influence the saving rate for this factorized method. On the one hand, assuming d_S, d_R and n_R are fixed, with the increase of n_S, r grows. Thus, the factorized approach enjoys more computational cost savings over the baseline algorithms. Correspondingly, if the n_S is fixed, the decrease of n_R will bring the same trend. On the other hand, when n_S, n_R and d_S are fixed, the saving rate will rise with the increase of d_R. Similarly, if d_R is fixed and decreases d_S, the result will be consistent. As we introduced before, for two tables **S** and **R** that need to be joined, if **R** has fewer tuples than that of **S** (i.e. $n_R < n_S$) and more features than that of **S** (i.e. $d_R > d_S$), more redundancy exists in the join result. Thus, the efficiency improvement for our proposal depends on the redundancy after joins and the more redundancy, the more benefits.

Notice that the correctness of all the calculations can be guaranteed because they are exactly decomposed to covert the large matrix operation into several small parts, and no approximation is involved. Although M-GMM reads the i-th batch from $\mathbf{T}$ and S-GMM/F-GMM reads the i-th batch from $\mathbf{R}$ for probing and then training, the values of parameters updated in each iteration are the same. All N tuples are involved in calculating the parameters in Lines 8, 14, and 20, regardless of the number of matching tuples in each batch. Thus, the parameters are the same after training and the accuracy of the models does not change for the algorithms M-GMM, S-GMM and F-GMM.

5.3 F-GMM for Multi-way Joins

For complex business, the required features for ML are scattered in more than two relations of the enterprise database. Thus, it is essential to generalize multi-way joins based on binary joins. We take F-GMM as an example to give the detailed steps. Similar decomposition of other ML cases for multi-way joins can be carried out following its pattern. We have introduced the problem setting and notations for the multi-way joins in section 4.1.

5.3.1 E-step for Multi-way Joins

Similar to the binary joins, the E-step involves data from tables $\mathbf{S}$ ($\mathbf{R}_0$) to $\mathbf{R}_q$ when calculating Equation 5.1. Denote the first d_{R_0} dimensions in $\mathbf{x}^{(n)} - \mu_k$ as PF_{R_0}

and the remaining dimensions can be decomposed into q parts corresponding to relations $\mathbf{R}_1$ to $\mathbf{R}_q$ as PF_{R_g}, where $g \in \{1 \dots q\}$ and $d_g = \sum_{i=0}^{g} d_{R_i}$. Thus, the vector $\mathbf{x}^{(n)} - \mu_k$ can be divided into $q+1$ parts.

$$\underset{d_{R_0}\times 1}{PF_{R_0}} = \begin{bmatrix} x_1^{(n)} - \mu_1 \\ x_2^{(n)} - \mu_2 \\ \vdots \\ x_{d_0}^{(n)} - \mu_{d_0} \end{bmatrix} \qquad \underset{d_{R_g}\times 1}{PF_{R_g}} = \begin{bmatrix} x_{d_{g-1}+1}^{(n)} - \mu_{d_{g-1}+1} \\ x_{d_{g-1}+2}^{(n)} - \mu_{d_{g-1}+2} \\ \vdots \\ x_{d_g}^{(n)} - \mu_{d_g} \end{bmatrix} \tag{5.20}$$

Similarly, I_k can be divided into $(q+1)^2$ parts, where $g \in \{1 \dots q\}$ and $h \in \{1 \dots q\}$:

$$\underset{d_{R_0}\times d_{R_0}}{I_{00}} = \begin{bmatrix} I_{1,1} & \cdots & I_{1,d_0} \\ I_{2,1} & \cdots & I_{2,d_0} \\ \vdots & \ddots & \vdots \\ I_{d_0,1} & \cdots & I_{d_0,d_0} \end{bmatrix} \qquad \underset{d_{R_0}\times d_{R_h}}{I_{0h}} = \begin{bmatrix} I_{1,d_{h-1}+1} & \cdots & I_{1,d_h} \\ I_{2,d_{h-1}+1} & \cdots & I_{2,d_h} \\ \vdots & \ddots & \vdots \\ I_{d_0,d_{h-1}+1} & \cdots & I_{d_0,d_h} \end{bmatrix}$$

$$\underset{d_{R_g}\times d_{R_0}}{I_{g0}} = \begin{bmatrix} I_{d_{g-1}+1,1} & \cdots & I_{d_{g-1}+1,d_0} \\ I_{d_{g-1}+2,1} & \cdots & I_{d_{g-1}+2,d_0} \\ \vdots & \ddots & \vdots \\ I_{d_g,1} & \cdots & I_{d_g,d_0} \end{bmatrix} \qquad \underset{d_{R_g}\times d_{R_h}}{I_{gh}} = \begin{bmatrix} I_{d_{g-1}+1,d_{h-1}+1} & \cdots & I_{d_{g-1}+1,d_h} \\ I_{d_{g-1}+2,d_{h-1}+1} & \cdots & I_{d_{g-1}+2,d_h} \\ \vdots & \ddots & \vdots \\ I_{d_g,d_{h-1}+1} & \cdots & I_{d_g,d_h} \end{bmatrix} \tag{5.21}$$

For unified expression, they are denoted as PF_{R_i} with dimension d_{R_i} and I_{ij} with dimension $d_{R_i} \times d_{R_j}$ where $i \in \{0 \dots q\}$ and $j \in \{0 \dots q\}$. Then Equation 5.8 can be represented as:

$$(\mathbf{x}^{(n)} - \mu_k)^T I_k (\mathbf{x}^{(n)} - \mu_k) = \sum_{i=0}^{q} \sum_{j=0}^{q} PF_{R_i}^T I_{ij} PF_{R_j} \tag{5.22}$$

In Equation 5.22, $(\mathbf{x}^{(n)} - \mu_k)^T I_k (\mathbf{x}^{(n)} - \mu_k)$ is decomposed into a sum of $(q+1)^2$ smaller matrices. The opportunity to eliminate redundancy exists in $PF_{R_i}^T I_{ij} PF_{R_j}$, when $i = j \neq 0$. Furthermore, $\forall g \in \{1 \dots q\}$, for each feature vector in $\mathbf{R}_g$, PF_{R_g}

only needs to be calculated once and reused for the matching cases. As the number of tables that need to be joined increases, there is more space to improve efficiency by eliminating redundancy.

5.3.2 M-step for Multi-way Joins

In the M-step, because $\mathbf{x}^{(n)}$ is combined by the features vectors from table $\mathbf{S}$ ($\mathbf{R}_0$) and tables $\mathbf{R}_1$ to $\mathbf{R}_q$, we can decompose μ_k in Equation (3.4) as follows:

$$\mu_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_k^{(n)} \mathbf{x}^{(n)} \tag{5.23}$$

$$= \frac{1}{N_k} \times \sum_{n=1}^{N} \times \begin{bmatrix} \gamma_k^{(n)} \mathbf{x}_{R_0}^{(n)} \\ \gamma_k^{(n)} \mathbf{x}_{R_1}^{(n)} \\ \vdots \\ \gamma_k^{(n)} \mathbf{x}_{R_q}^{(n)} \end{bmatrix} \tag{5.24}$$

As to Σ_k, $(\mathbf{x}^{(n)} - \mu_k)(\mathbf{x}^{(n)} - \mu_k)^T$ for multi-way joins can be written as:

$$(\mathbf{x}^{(n)} - \mu_k)(\mathbf{x}^{(n)} - \mu_k)^T = \begin{bmatrix} M_{00} & M_{01} & \cdots & M_{0q} \\ M_{10} & M_{11} & \cdots & M_{1q} \\ \vdots & \vdots & \ddots & \vdots \\ M_{q0} & M_{q1} & \cdots & M_{qq} \end{bmatrix} \tag{5.25}$$

where:

$$\underset{d_{R_i} \times d_{R_j}}{M_{ij}} = \underset{d_{R_i} \times 1}{PF_{R_i}} \quad \underset{1 \times d_{R_j}}{PF_{R_j}{}^T} \tag{5.26}$$

In Equation 5.25, we decompose the big matrix with dimension $d \times d$ into $(q+1)^2$ blocks of much smaller matrices. It is evident that there are large savings if we

reuse the computation of PF_{R_i} $(i \neq 0)$ and M_{ij} $(i = j \neq 0)$ by getting the features from tables $\mathbf{S}$ and $\mathbf{R}_1$ to $\mathbf{R}_q$ directly for the factorized equation.

6 Algorithms for NN

In this chapter, three algorithms, M-NN, S-NN and F-NN, are provided. In F-NN, we decompose the training process during forward and backward propagation to explore the space for efficiency improvement. Assuming this is a fully connected network and we adopt mini-batch gradient descent in the BP algorithm. The parameters will be updated according to the accumulated error calculated by the data points in one batch.

6.1 Baseline Algorithms: M-NN and S-NN

We also start by introducing the algorithm M-NN and the detailed steps are depicted in Algorithm 2. Lines 7 and 12 involve the feature vectors $\mathbf{x}^{(n)}$, which are read from the join results table $\mathbf{T}$ on the disk in batches. Because we choose the mini-batch gradient descent in the BP algorithm, all the tuples need to be read once in Line 5 in each iteration. All the parameters (w and b) will be updated many times in one iteration, which is determined by the number of batches. For algorithm S-NN, the differences exist in Lines 1 and 5; It reads one batch from

table **R** and gets the matching tuples in **S** to do joins on the fly without storing the join results on the disk.

Algorithm 2 Algorithm M-NN

1: **Data preparation:** Apply join of table **S** and table **R** in the database and materialize the results (table **T**) on the disk.
2: **Initialize all the** w **and** b **:**
3: **repeat**
4: **for** t in [*1 : number of batches*] **do**
5: Read batch t of **T** into memory
6: Calculate the output value for each node j in input layer:
7: $h_j^{(n)} = f(\sum_{i=1}^{d} w_{ji} x_i^{(n)} + b_j) \qquad \forall n \in$ batch t,
8: Propagate forward to each hidden layer until get the output $O^{(n)}$ of the network
9: Calculate the accumulated error:
10: $E = \frac{1}{2batchsize} \sum_{n=1}^{batchsize} \sum_{m=1}^{n_o} (O_m^{(n)} - Y_m^{(n)})^2$
11: Propagate the error backward using chain rule to calculate the gradient for all the w and b:
12: $\Delta w = -\alpha \frac{\partial E}{\partial w} \qquad \Delta b = -\alpha \frac{\partial E}{\partial b}$
13: Update all the w and b:
14: $w = w + \Delta w \qquad b = b + \Delta b$
15: **end for**
16: **until** Convergence

6.2 Algorithm F-NN and Decomposition Process

The factorized algorithm derived from Algorithm 2 for NN is called F-NN. As in S-GMM, materializing table **T** in Line 1 is not required but reading the data from table **R** and **S** in batches to calculate the factorized equations in Lines 7 and 10. Next, we will discuss the detailed decomposition for the training process from forward propagation and backward propagation.

6.2.1 Forward Propagation

We start with the first hidden layer, which contains n_h hidden units and receives d features from input layer. As shown in Equation 3.8, before applying the activation function f, $p_j^{(n)}$ is the input value for a single hidden unit calculated by the n-th feature vector $\mathbf{x}^{(n)}$ in table $\mathbf{T}$, where $j \in \{1 \dots n_h\}$. In order to make it clearer, we use the form of matrix for d features and n_h units to re-represent Equation 3.8, where $\mathbf{W}$ is the weight matrix and $\mathbf{b}$ is the bias vector. Thus, vector $\mathbf{p}^{(n)}$ is $\mathbf{W}\mathbf{x}^{(n)} + \mathbf{b}$ and it can be decomposed as:

$$
\begin{aligned}
\underset{n_h \times 1}{\mathbf{p}^{(n)}} &= \mathbf{W}\mathbf{x}^{(n)} + \mathbf{b} \\
&= \underset{n_h \times d}{\begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,d} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_h,1} & w_{n_h,2} & \cdots & w_{n_h,d} \end{bmatrix}} \times \underset{d \times 1}{\begin{bmatrix} x_1^{(n)} \\ x_2^{(n)} \\ \vdots \\ x_d^{(n)} \end{bmatrix}} + \underset{n_h \times 1}{\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n_h} \end{bmatrix}} \\
&= \underset{n_h \times d_S}{\begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,d_S} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,d_S} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_h,1} & w_{n_h,2} & \cdots & w_{n_h,d_S} \end{bmatrix}} \times \underset{d_S \times 1}{\begin{bmatrix} x_1^{(n)} \\ x_2^{(n)} \\ \vdots \\ x_{d_S}^{(n)} \end{bmatrix}} \\
&+ \underset{n_h \times d_R}{\begin{bmatrix} w_{1,d_S+1} & w_{1,d_S+2} & \cdots & w_{1,d} \\ w_{2,d_S+1} & w_{2,d_S+2,} & \cdots & w_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_h,d_S+1} & w_{n_h,d_S+2} & \cdots & w_{n_h,d} \end{bmatrix}} \times \underset{d_R \times 1}{\begin{bmatrix} x_{d_S+1}^{(n)} \\ x_{d_S+2}^{(n)} \\ \vdots \\ x_d^{(n)} \end{bmatrix}} + \underset{n_h \times 1}{\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n_h} \end{bmatrix}} \\
&= \mathbf{W}_S \mathbf{x}_S^{(n)} + \mathbf{W}_R \mathbf{x}_R^{(n)} + \mathbf{b}
\end{aligned} \tag{6.1}
$$

In Equation 6.1, all the features can be divided into two vectors: $\mathbf{x}_S^{(n)}$ and $\mathbf{x}_R^{(n)}$ respectively. For each iteration, the values of the weights and biases are constant and each tuple in $\mathbf{R}$ may match several tuples in $\mathbf{S}$. The result of partial inner products $\mathbf{W}_R\mathbf{x}_R^{(n)} + \mathbf{b}$ only needs to be calculated once, which can bring calculation savings in F-NN and we will quantify the benefits in the experimental section. After applying activation function on the result, we obtain the output matrix $\mathbf{h}^{(n)} = f(\mathbf{p}^{(n)})$ for the hidden units in the first layer. As depicted in Figure 6.1, we read tuples directly in batches from $\mathbf{R}$ and probe $\mathbf{S}$ for matching tuples so that the two partial computations of $\mathbf{W}$ from the feature vector of $\mathbf{S}$ and $\mathbf{R}$ ($\mathbf{W}_S$ and $\mathbf{W}_R$) will be combined and pushed to the activation function.

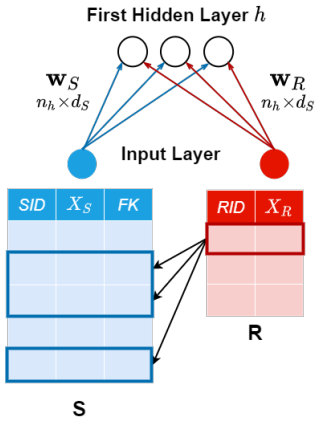


Figure 6.1: Forward Propagation

Afterwards, we discuss the calculation of the next layer with n_l hidden units. The input value for every unit is shown in Equation 3.10. Due to $h_j^{(n)} = f(p_j^{(n)})$, we replace $p_j^{(n)}$ with Equation 3.8 to get:

$$q_k^{(n)} = \sum_{j=1}^{n_h} w_{kj}^{[2]} f(\sum_{i=1}^{d} w_{ji}^{[1]} x_i^{(n)} + b_j^{[1]}) + b_k^{[2]} \tag{6.2}$$

If we use the idea of decomposition to separate Equation 6.2, f must be an additive function, which means a solution to the Cauchy functional form: $f(x+y) = f(x) + f(y)$. In this case, we can factorize $q_k^{(n)}$ as:

$$q_k^{(n)} = \sum_{j=1}^{n_h} (w_{kj}^{(2)} f(\sum_{i=1}^{d_S} w_{ji}^{(1)} x_i^{(n)}) + w_{kj}^{(2)} f(\sum_{i=d_S+1}^{d} w_{ji}^{(1)} x_i^{(n)} + b_j^{(1)})) + b_k^{(2)} \tag{6.3}$$

$$= \sum_{j=1}^{n_h} w_{kj}^{(2)} f(T_1) + \sum_{j=1}^{n_h} w_{kj}^{(2)} f(T_2) + b_k^{(2)} \tag{6.4}$$

$$= \sum_{j=1}^{n_h} w_{kj}^{(2)} f(T_1) + T_3 \tag{6.5}$$

where:

$$T_1 = \sum_{i=1}^{d_S} w_{ji}^{(1)} x_i^{(n)} \tag{6.6}$$

$$T_2 = \sum_{i=(d_S+1)}^{d} w_{ji}^{(1)} x_i^{(n)} + b_j^{(1)} \tag{6.7}$$

$$T_3 = \sum_{j=1}^{n_h} w_{kj}^{(2)} f(T_2) + b_k^{(2)} \tag{6.8}$$

Here T_1 and T_2 are the parts that have been computed in the first hidden layer and easily to save and reuse. T_3 is the value only involved the features from table **R** multiplying the weights in the second layer. Similarly, according to the principle of the factorized algorithm, T_3 is computed when one tuple in table **R** appears for the first time and reused for all the matching tuples in table **S** subsequently.

However, $q_k^{(n)}$ can be decomposed in this way based on the assumption that the activation function is the additive function. Popular activation functions are

empirically restricted to certain choices such as *sigmoid*, *tanh* and *Relu*, which have been highly successful in ML and deep learning areas primarily due to its simplicity and low overhead during optimization [47]. It is easy to prove that *sigmoid* and *tanh* are not additive functions. If the NN adopts them, there are no opportunities to factorize the calculation in the second layer and share the repeated part. The *Relu* function is a piece-wise linear function so that if T_1 and T_2 have the same sign, it is an additive function. Thus, this requirement of the function is the limitation for the usage in the second hidden layer.

Furthermore, suppose we choose one additive function as activation function, let us consider the total operations after decomposition in Equation 6.3. T_1 and T_2 have been calculated in the first layer. In Equation 6.5, it requires n_h multiplications and n_h additions to sum up the results of multiplication $(w_{kj}^{(2)} f(T_1))$ and add T_3. For T_3 in Equation 6.8, it requires another n_h multiplications and n_h additions for for n_R feature vectors in $\mathbf{R}$ before they can be reused. In other words, the total cost after decomposition is $2n_h$ multiplications and $2n_h$ additions. Compared with the operations required before decomposition, we only need n_h multiplications and n_h additions once we received the value of $h_j^{(n)}$ to calculate the input values of second layer. Therefore, if we attempt to reuse the result computed from $\mathbf{S}$ and $\mathbf{R}$ respectively at the second hidden layer, the computation cost is higher than before. Following the analysis in this way, this cost is going to increase even further at higher

layers, which eventually outweighs the savings brought by the decomposition in the first layer.

In conclusion, during the process of forward propagation, F-NN can make sense beyond the first layer only when we choose additive activation functions. Even though the equations in the higher layers can be decomposed, the additional computation incurred by it may render any attempt to share the reused parts at higher layers unattractive, even will bring more costs.

6.2.2 Backward Propagation

Before utilizing gradient descent algorithm to calculate w and b, we should get the result of error function (E) in Equation 3.11. Backward propagation starts from the output layer and proceeds to the lower layers. In the entire process, feature vectors are involved in the computation only once when computing the gradient of the weights between the input layer and the first hidden layer. Therefore, we only consider the decomposition in this step.

For the reason that we adopt the mini-batch gradient descent, the new value of the w is determined by the average of Δw after calculating all the gradient of the weights ($\frac{\partial E}{\partial w}$) for one batch tuples. Let $\mathbf{X}_B$ to denote the feature matrix of one batch tuples after joins consisted of B feature vectors. In this batch, $\mathbf{X}_{BS}$ and $\mathbf{X}_{BR}$ are the feature matrices from $\mathbf{S}$ and $\mathbf{R}$ respectively. $\frac{\partial E}{\partial \mathbf{p}}$ is the gradient of the error

concerning the value before the activation function between the input and hidden layer, which is obtained via the chain rule. Thus, the calculation for the $\frac{\partial E}{\partial w}$ can be presented in the matrix form:

$$\underset{n_h \times d}{\frac{\partial E}{\partial w}} = \frac{\partial E}{\partial \mathbf{p}} \quad {\mathbf{X}_B}^{\mathrm{T}} \tag{6.9}$$

$$= \underset{n_h \times B}{\begin{bmatrix} \frac{\partial E}{\partial p_1^{(1)}} & \frac{\partial E}{\partial p_1^{(2)}} & \cdots & \frac{\partial E}{\partial p_1^{(B)}} \\ \frac{\partial E}{\partial p_2^{(1)}} & \frac{\partial E}{\partial p_2^{(2)}} & \cdots & \frac{\partial E}{\partial p_2^{(B)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial p_{n_h}^{(1)}} & \frac{\partial E}{\partial p_{n_h}^{(2)}} & \cdots & \frac{\partial E}{\partial p_{n_h}^{(B)}} \end{bmatrix}} \times \underset{B \times d_S \qquad\qquad B \times d_R}{\begin{bmatrix} x_1^{(1)} & \cdots & x_{d_S}^{(1)} & x_{d_S+1}^{(1)} & \cdots & x_d^{(1)} \\ x_1^{(2)} & \cdots & x_{d_S}^{(2)} & x_{d_S+1}^{(2)} & \cdots & x_d^{(2)} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ x_1^{(B)} & \cdots & x_{d_S}^{(B)} & x_{d_S+1}^{(B)} & \cdots & x_d^{(B)} \end{bmatrix}} \tag{6.10}$$

$$= \begin{bmatrix} PG_S & PG_R \\ n_h \times d_S & n_h \times d_R \end{bmatrix} \tag{6.11}$$

where

$$PG_S = \underset{n_h \times B}{\begin{bmatrix} \frac{\partial E}{\partial p_1^{(1)}} & \frac{\partial E}{\partial p_1^{(2)}} & \cdots & \frac{\partial E}{\partial p_1^{(B)}} \\ \frac{\partial E}{\partial p_2^{(1)}} & \frac{\partial E}{\partial p_2^{(2)}} & \cdots & \frac{\partial E}{\partial p_2^{(B)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial p_{n_h}^{(1)}} & \frac{\partial E}{\partial p_{n_h}^{(2)}} & \cdots & \frac{\partial E}{\partial p_{n_h}^{(B)}} \end{bmatrix}} \times \underset{B \times d_S}{\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_{d_S}^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_{d_S}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(B)} & x_2^{(B)} & \cdots & x_{d_S}^{(B)} \end{bmatrix}} = \frac{\partial E}{\partial \mathbf{p}} \quad {\mathbf{X}_{BS}}^{\mathrm{T}} \tag{6.12}$$

$$PG_R = \underset{n_h \times B}{\begin{bmatrix} \frac{\partial E}{\partial p_1^{(1)}} & \frac{\partial E}{\partial p_1^{(2)}} & \cdots & \frac{\partial E}{\partial p_1^{(B)}} \\ \frac{\partial E}{\partial p_2^{(1)}} & \frac{\partial E}{\partial p_2^{(2)}} & \cdots & \frac{\partial E}{\partial p_2^{(B)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial p_{n_h}^{(1)}} & \frac{\partial E}{\partial p_{n_h}^{(2)}} & \cdots & \frac{\partial E}{\partial p_{n_h}^{(B)}} \end{bmatrix}} \times \underset{B \times d_R}{\begin{bmatrix} x_{d_S+1}^{(1)} & x_{d_S+2}^{(1)} & \cdots & x_d^{(1)} \\ x_{d_S+1}^{(2)} & x_{d_S+2}^{(2)} & \cdots & x_d^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d_S+1}^{(B)} & x_{d_S+2}^{(B)} & \cdots & x_d^{(B)} \end{bmatrix}} = \frac{\partial E}{\partial \mathbf{p}} \quad {\mathbf{X}_{BR}}^{\mathrm{T}} \tag{6.13}$$

In Equation 6.11, even though we can decompose the computation into PG_S and PG_R, there are no opportunities to explore redundancy in the computation. For PG_R, each row in $\frac{\partial E}{\partial \mathbf{p}}$ multiplies the corresponding column of $\mathbf{X}_{BR}^{\mathrm{T}}$ not $\mathbf{X}_{BR}$.

As we known, the redundancy exists among different columns of $\mathbf{X}_B$, such as $[x_1^{(a)} x_2^{(a)} \cdots x_d^{(a)}]^{\mathrm{T}}$ and $[x_1^{(b)} x_2^{(b)} \cdots x_d^{(b)}]^{\mathrm{T}}$, due to sharing the same part of $\mathbf{X}_{BR}$. In other words, there is no redundancy among the columns of $\mathbf{X}_B^{\mathrm{T}}$, such as $[x_i^{(1)} x_i^{(2)} \cdots x_i^{(B)}]^{\mathrm{T}}$ and $[x_j^{(1)} x_j^{(2)} \cdots x_j^{(B)}]^{\mathrm{T}}$. Thus, the decomposition in this part does not bring any efficiency improvement due to the reused calculation.

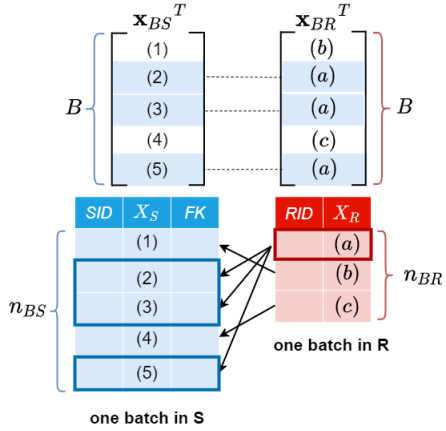


Figure 6.2: Backward Propagation

However, decomposing the matrix into two parts in the backward propagation will bring I/O cost savings. In Equation 6.13, $\mathbf{X}_R^{\mathrm{T}}$ is a matrix with B rows and some of them are repeated. Actually, after decomposition, there is no need to read all the B rows but obtaining features vectors in batches directly from table $\mathbf{R}$. Then, we can use the primary/foreign-key relationship to retrieve corresponding features from $\mathbf{S}$ and populate $\mathbf{X}_{BR}^{\mathrm{T}}$. As shown in Figure 6.2, for a feature vector (a) in $\mathbf{R}$, the matching feature vectors (2), (3) and (5) in $\mathbf{S}$ populate $\mathbf{X}_S^{\mathrm{T}}$. Then the feature

vector (a) is inserted in the positions corresponding to $\mathbf{X}_S^{\mathrm{T}}$ in $\mathbf{X}_R^{\mathrm{T}}$. Specifically, both $\mathbf{X}_{BS}^{\mathrm{T}}$ and $\mathbf{X}_{BR}^{\mathrm{T}}$ have B rows, where B is the cardinality of $\mathbf{X}_{BS}$. Instead of retrieving $B \times (d_S + d_R)$ fields in one batch of $\mathbf{T}$, we only need $n_{BS} \times d_S + n_{BR} \times d_R$ fields where $n_{BS} = B$ and $n_{BR} < B$. If taken all tuples required in training into consideration, $n_S \times d_S + n_R \times d_R$ fields will be much less than $N \times (d_S + d_R)$ fields. Therefore, even through there are no reused calculations, I/O cost can be saved due to getting data directly from the normalized tables $\mathbf{S}$ and $\mathbf{R}$. That is the another one benefit from the factorized approach.

In summary, although algorithm F-NN has many limitations, considerable savings can be achieved between the input layer and the first hidden layer during forward and backward propagation. However, we do not continue pursuing it at higher layers of the network because we no longer guarantee the exactness of the decomposition for different types of activation functions. We will experimentally evaluate the efficiency improvement for the F-NN with one hidden layer in Chapter 8.

7 Algorithms for PPCA

In this chapter, we give the description and analysis for the baseline algorithms and the factorized way about PPCA.

7.1 Baseline Algorithms: M-PPCA and S-PPCA

Algorithm 3 shows the training process of M-PPCA. We carry out the joins for table **S** and table **R** to get table **T** stored in the database. In the EM algorithm of PPCA, the calculations for W_{new} and σ^2_{new} both require to be calculated using all N tuples in table **T**. We only consider one-batch tuples for each loop and divide the polynomials of Equation 3.22 and 3.23 into several partial sum results. Specially, in Line 15, W_{new} can be written as the concatenation of W_{Part_1} and W_{Part_2}, which is represented in Lines 11 and 13, while in Line 26, σ^2_{new} is the combination of $\sigma^2_{Part_1}$, $\sigma^2_{Part_2}$ and $\sigma^2_{Part_3}$ as calculated in Lines 20, 22 and 24. In this process, the calculation of σ^2 will use the new value of W so that we should read all N data points twice in each iteration. In addition, the intermediate results $\mathbb{E}[\mathbf{z}^{(n)}]$ and $\mathbb{E}[\mathbf{z}^{(n)}\mathbf{z}^{(n)^{\mathrm{T}}}]$ will be both used in calculation of W_{new} and σ^2_{new} but they cannot be

stored due to the limitation of memory under our setting. Therefore, in Line 18, we should take $\mathbf{x}^{(n)}$ as the input to calculate $\mathbb{E}[\mathbf{z}^{(n)}]$ again.

The construction method for S-PPCA is in the same way as S-GMM and S-NN; reading the data from normalized tables directly in Line 1, getting the sub-tables of the join results on the fly in the memory and using them as the input in Lines 7, 11, 18, 20 and 22 sequentially for calculating parameters.

7.2 Algorithm F-PPCA and Decomposition Process

In this section, all the detailed decomposition steps will be provided for algorithm F-PPCA. It is derived from Algorithm 3; Line 1 for materializing the table is not required and relation $\mathbf{R}$ is processed in batches, probing $\mathbf{S}$ for matching tuples using the primary/foreign key. However, instead of storing and feeding every joined tuple for updating the parameters in Lines 7, 11, 20 and 22, we factorize the equations into two parts involving $\mathbf{X}_S$ and $\mathbf{X}_R$ respectively.

7.2.1 Calculating W_{new}

In Line 7, we calculate the expectations of $\mathbf{z}^{(n)}$ using Equation 3.20. Because W and σ^2 are initialized parameters, $M^{-1}W^{\mathrm{T}}$ is an invariant matrix with dimension $p \times d$, which does not change for different n. To ease notation, we denote $M^{-1}W^{\mathrm{T}}$ as G, so $\mathbb{E}[\mathbf{z}^{(n)}]$ can be represented into the form of a matrix:

Algorithm 3 *M-PPCA*

1: **Data preparation:** Apply join of table **S** and table **R** in the database and materialize the results (table **T**) on the disk.
2: **Initialize** W **and** σ^2 **:**
3: **repeat**
4: **for** i in [*1 : number of batches*] **do**
5: Read batch i of **T** into memory
6: Update the expectations of $\mathbf{z}^{(n)}$:
7: $\mathbb{E}[\mathbf{z}^{(n)}] = M^{-1}W^{\mathrm{T}}(\mathbf{x}^{(n)} - \overline{\mathbf{x}})$ with $M = W^{\mathrm{T}}W + \sigma^2 I \qquad \forall n \in$ batch i
8: Update the expectations of $\mathbf{z}^{(n)}\mathbf{z}^{(n)^{\mathrm{T}}}$:
9: $\mathbb{E}[\mathbf{z}^{(n)}\mathbf{z}^{(n)^{\mathrm{T}}}] = \sigma^2 M^{-1} + \mathbb{E}[\mathbf{z}^{(n)}]\mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}} \qquad \forall n \in$ batch i
10: Update the first partial sum results of W_{new}:
11: $W_{Part_1} += \sum_{n=1}^{batchsize}(\mathbf{x}^{(n)} - \overline{\mathbf{x}})\mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}}$
12: Update the second partial sum results of W_{new}:
13: $W_{Part_2} += \sum_{n=1}^{batchsize}\mathbb{E}[\mathbf{z}^{(n)}\mathbf{z}^{(n)^{\mathrm{T}}}]$
14: **end for**
15: Update $W_{new} = W_{Part_1}{W_{Part_2}}^{-1}$
16: **for** i in [*1 : number of batches*] **do**
17: Read batch i of **T** into memory
18: Update $\mathbb{E}[\mathbf{z}^{(n)}]$ and $\mathbb{E}[\mathbf{z}^{(n)}\mathbf{z}^{(n)^{\mathrm{T}}}]$ in Lines 7 and 9 respectively.
19: Update the first partial sum results of σ^2_{new}:
20: $\sigma^2_{Part_1} += \sum_{n=1}^{batchsize}\|\mathbf{x}^{(n)} - \overline{\mathbf{x}}\|^2$
21: Update the second partial sum results of σ^2_{new}:
22: $\sigma^2_{Part_2} += \sum_{n=1}^{batchsize} 2\mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}}{W_{new}}^{\mathrm{T}}(\mathbf{x}^{(n)} - \overline{\mathbf{x}})$
23: Update the third partial sum results of σ^2_{new}:
24: $\sigma^2_{Part_2} += \sum_{n=1}^{batchsize} tr(\mathbb{E}[\mathbf{z}^{(n)}\mathbf{z}^{(n)^{\mathrm{T}}}]{W_{new}}^{\mathrm{T}}W_{new})$
25: **end for**
26: Update $\sigma^2_{new} = \frac{1}{ND}(\sigma^2_{Part_1} - \sigma^2_{Part_2} + \sigma^2_{Part_3})$
27: $W = W_{new}$
28: $\sigma^2 = \sigma^2_{new}$
29: **until** Convergence

$$\underset{p\times 1}{\mathbb{E}[\mathbf{z}^{(n)}]} = G(\mathbf{x}^{(n)} - \overline{\mathbf{x}}) \tag{7.1}$$

$$= \underbrace{\begin{bmatrix} G_{1,1} & G_{1,2} & \cdots & G_{1,d} \\ G_{2,1} & G_{2,2} & \cdots & G_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ G_{p,1} & G_{p,2} & \cdots & G_{p,d} \end{bmatrix}}_{p\times d} \times \underbrace{\begin{bmatrix} x_1^{(n)} - \overline{x}_1 \\ x_2^{(n)} - \overline{x}_2 \\ \vdots \\ x_d^{(n)} - \overline{x}_d \end{bmatrix}}_{d\times 1} \tag{7.2}$$

In Equation 7.2, we divide G into two small matrices G_S and G_R with dimension $p \times d_S$ and $p \times d_R$ respectively, which can be represented as:

$$\underset{p\times d_S}{G_S} = \begin{bmatrix} G_{1,1} & G_{1,2} & \cdots & G_{1,d_S} \\ G_{2,1} & G_{2,2} & \cdots & G_{2,d_S} \\ \vdots & \vdots & \ddots & \vdots \\ G_{p,1} & G_{p,2} & \cdots & G_{p,d_S} \end{bmatrix} \qquad \underset{p\times d_R}{G_R} = \begin{bmatrix} G_{1,d_S+1} & G_{1,d_S+2} & \cdots & G_{1,d} \\ G_{2,d_S+1} & G_{2,d_S+2} & \cdots & G_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ G_{p,d_S+1} & G_{p,d_S+2} & \cdots & G_{p,d} \end{bmatrix} \tag{7.3}$$

Then, we denote the first d_S dimensions of $\mathbf{x}^{(n)} - \overline{\mathbf{x}}$ as $PD_S^{(n)}$ and the remaining d_R dimensions as $PD_R^{(n)}$:

$$\underset{d_S\times 1}{PD_S^{(n)}} = \begin{bmatrix} x_1^{(n)} - \overline{x}_1 \\ x_2^{(n)} - \overline{x}_2 \\ \vdots \\ x_{d_S}^{(n)} - \overline{x}_{d_S} \end{bmatrix} \qquad \underset{d_R\times 1}{PD_R^{(n)}} = \begin{bmatrix} x_{d_S+1}^{(n)} - \overline{x}_{d_S+1} \\ x_{d_S+2}^{(n)} - \overline{x}_{d_S+2} \\ \vdots \\ x_d^{(n)} - \overline{x}_d \end{bmatrix} \tag{7.4}$$

Therefore,

$$\mathbb{E}[\mathbf{z}^{(n)}] = \begin{bmatrix} G_S & G_R \end{bmatrix} \times \begin{bmatrix} PD_S^{(n)} \\ \\ PD_R^{(n)} \end{bmatrix} \tag{7.5}$$

$$= G_S \times PD_S^{(n)} + G_R \times PD_R^{(n)} \tag{7.6}$$

In Equation 7.6, $\mathbb{E}[\mathbf{z}^{(n)}]$ is factorized perfectly as the sum of two parts. $G_R \times PD_R^{(n)}$ is the part that only involves $\mathbf{x}_R^{(n)}$ and can be reused. For all the matching tuples in table $\mathbf{S}$ that have the same foreign-key with the primary-key in table $\mathbf{R}$, they

just need to add the same value of $G_R \times PD_R^{(n)}$. Thus, we can reduce the cost of calculation for $\mathbb{E}[\mathbf{z}^{(n)}]$.

In Line 11, feature vectors are involved in the calculation of $(\mathbf{x}^{(n)} - \overline{\mathbf{x}})\mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}}$ before summation. Here, $\mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}}$ has been calculated in Line 7 and the decomposition equation is as follows:

$$(\underset{d\times p}{\mathbf{x}^{(n)} - \overline{\mathbf{x}}})\mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}} = \begin{bmatrix} \underset{d_S\times 1}{PD_S^{(n)}} \\ \\ \underset{d_R\times 1}{PD_R^{(n)}} \end{bmatrix} \times \underset{1\times p}{\mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}}} \tag{7.7}$$

$$= \begin{bmatrix} PD_S^{(n)} \times \mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}} \\ \\ PD_R^{(n)} \times \mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}} \end{bmatrix} \tag{7.8}$$

Similarly, for $\mathbf{x}^{(n)} - \overline{\mathbf{x}}$, we can divide it into $PD_S^{(n)}$ and $PD_R^{(n)}$. $PD_R^{(n)}$ can be easily reused as the intermediate results following the previous calculation to remove the redundancy for the matching tuples. However, in Equation 7.8, the values of $\mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}}$ are varied for different n. There are no possibility to reuse the multiplication results for $PD_R^{(n)} \times \mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}}$.

7.2.2 Calculating σ_{new}^2

From Equation 3.23, σ_{new}^2 is updated with $\mathbb{E}[\mathbf{z}^{(n)}]$, $\mathbb{E}[\mathbf{z}^{(n)}\mathbf{z}^{(n)\mathrm{T}}]$ and W_{new}, thus we cannot calculate it before getting the final value of W_{new}. Besides, due to the limited memory, the intermediate results of $\mathbb{E}[\mathbf{z}^{(n)}]$ and $\mathbb{E}[\mathbf{z}^{(n)}\mathbf{z}^{(n)\mathrm{T}}]$ in Lines 7 and 9 cannot be used for σ_{new}^2 and need to be calculated again. Calculating Line 7

will bring more potential to reduce computational cost based on our analysis for Equation 7.6.

For $\sigma^2_{part_1}$ in Line 20, the calculation of $\|\mathbf{x}^{(n)} - \overline{\mathbf{x}}\|^2$ can also be divided into two parts using our factorized method. $\|PD_R^{(n)}\|^2$ is the part that we expect to reuse for saving the calculations. The process for the decomposition is:

$$\begin{aligned}\|\mathbf{x}^{(n)} - \overline{\mathbf{x}}\|^2 &= (x_1^{(n)} - \overline{x}_1)^2 + \cdots + (x_d^{(n)} - \overline{x}_d)^2 \\ &= \sum_{i=1}^{d_s}(x_i^{(n)} - \overline{x}_i)^2 + \sum_{i=d_{s+1}}^{d}(x_i^{(n)} - \overline{x}_i)^2 \\ &= \|PD_S^{(n)}\|^2 + \|PD_R^{(n)}\|^2 \end{aligned} \tag{7.9}$$

The last part involved feature vectors is the equation for calculating $\sigma^2_{part_2}$ in Line 22. $\mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}}$ changes with different n, but W_{new} has been calculated in the previous step as a constant matrix for all the feature vectors. Thus, according to the associative law of matrix multiplication, we can regard ${W_{new}}^{\mathrm{T}}(\mathbf{x}^{(n)} - \overline{\mathbf{x}})$ as a whole equation to decompose firstly and then multiply $\mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}}$:

$$\mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}}{W_{new}}^{\mathrm{T}}(\mathbf{x}^{(n)} - \overline{\mathbf{x}}) = \mathbb{E}[\mathbf{z}^{(n)}]^{\mathrm{T}}({W_{new}}^{\mathrm{T}}(\mathbf{x}^{(n)} - \overline{\mathbf{x}})) \tag{7.10}$$

The process of ${W_{new}}^{\mathrm{T}}(\mathbf{x}^{(n)} - \overline{\mathbf{x}})$ is similar as Line 7 and its steps are as follows:

$$
\underset{p\times 1}{{W_{new}}^{\mathrm{T}}}(\mathbf{x}^{(n)}-\overline{\mathbf{x}}) = \underset{p\times d}{\begin{bmatrix} W_{new_{1,1}} & W_{new_{2,1}} & \cdots & W_{new_{d,1}} \\ W_{new_{1,2}} & W_{new_{2,2}} & \cdots & W_{new_{d,2}} \\ \vdots & \vdots & \ddots & \vdots \\ W_{new_{1,p}} & W_{new_{2,p}} & \cdots & W_{new_{d,p}} \end{bmatrix}} \times \underset{d\times 1}{\begin{bmatrix} x_1^{(n)}-\overline{x}_1 \\ x_2^{(n)}-\overline{x}_2 \\ \vdots \\ x_d^{(n)}-\overline{x}_d \end{bmatrix}} \tag{7.11}
$$

$$
\begin{aligned}
&= \begin{bmatrix} {W_{new_S}}^{\mathrm{T}} & {W_{new_R}}^{\mathrm{T}} \end{bmatrix} \times \begin{bmatrix} PD_S^{(n)} \\ \\ PD_R^{(n)} \end{bmatrix} \\
&= {W_{new_S}}^{\mathrm{T}} \times PD_S^{(n)} + {W_{new_R}}^{\mathrm{T}} \times PD_R^{(n)}
\end{aligned} \tag{7.12}
$$

where:

$$
\underset{p\times d_S}{{W_{new_S}}^{\mathrm{T}}} = \begin{bmatrix} W_{new_{1,1}} & \cdots & W_{new_{d_S,1}} \\ W_{new_{1,2}} & \cdots & W_{new_{d_S,2}} \\ \vdots & \ddots & \vdots \\ W_{new_{1,p}} & \cdots & W_{new_{d_S,p}} \end{bmatrix} \qquad \underset{p\times d_R}{{W_{new_R}}^{\mathrm{T}}} = \begin{bmatrix} W_{new_{d_{S+1},1}} & \cdots & W_{new_{d,1}} \\ W_{new_{d_{S+1},2}} & \cdots & W_{new_{d,2}} \\ \vdots & \ddots & \vdots \\ W_{new_{d_{S+1},p}} & \cdots & W_{new_{d,p}} \end{bmatrix} \tag{7.13}
$$

The original big matrix in 7.11 can be decomposed to the sum equations for two separate parts involved the feature vectors from **S** and **R** in Equation 7.12. Based on this decomposition, we can remove the redundancy for the calculation of ${W_{new_R}}^{\mathrm{T}} \times PD_R^{(n)}$ to get significant savings.

After decomposing GMM, NN and PPCA in Chapter 5, 6 and 7, we can conclude several common laws. Firstly, we should decompose all the equations that contain the feature vectors into two parts: one only involves the features from table **S** and the other only involves the features from table **R**. Secondly, if the equations contain both feature vectors and other elements, we should determine whether the parts after decomposition can be reused among different data points. The most suitable case for decomposition is that the rest of the equation is constant for all

data points, such as the parameter W_{new} in Equation 7.12. If some elements are different for each data point, we only can take the remaining part as the reused part, such as the decomposition in Equation 7.8. However, as long as the equations can be decomposed, we can benefit from I/O cost savings. Because instead of reading the results after the joins containing much redundancy, we fetch the data directly from the original tables. Thirdly, different types of models may require different processing methods. They may not be decomposed due to the complicated calculation or may bring extra costs due to decomposition. Taking NN as an example, we cannot pursue the benefits of decomposition at higher layers in forward propagation for both these two reasons.

8 Experiments

In this chapter, we present a thorough experimental evaluation for all the algorithms we have discussed before to compare their performance using synthetic and real datasets. Besides, we vary the parameters of interest to explore the underlying factors that impact our proposal.

8.1 Settings

8.1.1 Environment

All experiments are conducted on the machine with 16 Intel Xeon E5630 2.53 GHz cores, 96 GB RAM and 338 GB disk with CentOS 6.2. Our codes are implemented in Python 3.8. We use the NumPy library for all the matrix calculations and the Psycopg2 library for operating PostgreSQL. All the relations are stored in PostgreSQL and the algorithms are implemented on top of it.

8.1.2 Parameters

As we discussed in Section 5.2.2, the amount of redundancy for the table after joins impact the performance of our proposals when conducting ML algorithms

over the original normalized data. Thus, we choose two main parameters of interest that can essentially measure the redundancy introduced by joins: tuple ratio of $\mathbf{S}$ and $\mathbf{R}$, denoted as $tr = n_S/n_R$, and feature ratio of $\mathbf{R}$ and $\mathbf{S}$, denoted as $fr = d_R/d_S$. Besides, the values of hyper-parameters in each application scenario may also influence their efficiency, so we take them into consideration. We vary the number of clusters(K) for GMM, the number of hidden units(n_h) for NN and the dimension after dimensionality reduction(p). Therefore, we not only compare the performance of three algorithms designed for GMM, NN and PPCA, but also investigate the underlying relations that essentially quantify the impact of normalization in eliminating redundancy.

The performance measurement is the runtime of each algorithm on different datasets, which includes the time for acquiring the data and the time for training the parameters. As long as we guarantee the exact decomposition, factorized algorithm and the two baseline algorithms will reach the same training results. Because it takes a long time to get convergence, all the algorithms are trained for ten epochs as the iteration's terminal condition. For NN, based on the analysis in Section 6.2.1, we design the network with one single hidden layer.

8.1.3 Datasets

We generate a series of synthetic datasets and also verify them on real datasets. Although the issue investigated in this research is very common in enterprises, it is not easy to find the large open-source real datasets that satisfied the primary and foreign key relationship. We utilize synthetic datasets to vary the parameters of interest in a controlled way and explore their trends. All the synthetic datasets are normalized generated from multiple Gaussian distributions and added random noise. Table **S** and its corresponding table **R** satisfy the primary and foreign key relationship. The values of the primary key in **R** are evenly distributed in the column of the foreign key in **S**. According to the parameters we selected in Section 8.1.2, all the cases considered for GMM, NN and PPCA are listed in Table 8.1, Table 8.2 and Table 8.3, respectively.

Table 8.1: Synthetic datasets with varied parameters for GMM

Cases	n_S	n_R	$tr(n_S/n_R)$	d_S	d_R	$fr(d_R/d_S)$	K
vary tr	Varied	1000	Varied	5	5 and 15	1 and 3	5
vary fr	10^6 and 5×10^6	1000	1000 and 5000	5	Varied	Varied	5
vary K	10^6	1000	1000	5	15	3	Varied

Table 8.2: Synthetic datasets with varied parameters for NN

Cases	n_S	n_R	$tr(n_S/n_R)$	d_S	d_R	$fr(d_R/d_S)$	n_h
vary tr	Varied	1000	Varied	5	5 and 15	1 and 3	50
vary fr	10^6 and 5×10^6	1000	1000 and 5000	5	Varied	Varied	50
vary n_h	10^6	1000	1000	5	15	3	Varied

Table 8.3: Synthetic datasets with varied parameters for PPCA

Cases	n_S	n_R	$tr(n_S/n_R)$	d_S	d_R	$fr(d_R/d_S)$	p
vary tr	Varied	1000	Varied	2	2 and 6	1 and 3	3
vary fr	10^6 and 5×10^6	5000	200 and 1000	2	Varied	Varied	3
vary p	10^6	1000	1000	2	6	3	Varied

The real datasets are the *Expedia*, *Walmart* and *Movies* datasets from the Hamlet Plus Project[1]. We choose the suitable tables as **S** and **R** to be our experimental group. *Expedia1* dataset is by joining *R1_Hotels* (**R**) with *S_Listings* (**S**) and *Expedia2* dataset is by joining *R2_Searches* (**R**) with *S_Listings* (**S**). For the *Walmart* dataset, we join *R1_Indicators* (**R**) with *S_Sales* (**S**) and for the *Movies* dataset, we join *R2_movies* (**R**) with *S_ratings* (**S**). The dimensions of every real datasets are available in Table 8.4. For NN, we use the one-hot representation of the data, which can also be found in Hamlet Plus Project.

Table 8.4: Data dimensions of real datasets

Dataset	n_S	d_S	n_R	d_R
Expedia1	942142	7	11938	8
Expedia2	942142	7	37021	14
Walmart	421570	3	2340	9
Movies	1000209	1	3706	21

Since the dimension in the real datasets is limited, we construct datasets derived from the *Expedia1* dataset with larger parameters of interest. These are constructed by picking some tuples from *S_Listings* and *R1_Hotels* to increase tr, and repeating

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

features with random Gaussian noise to increase fr. The augmented datasets are depicted in Table 8.5 as *Expedia3* to *Expedia5* along with their associated characteristics.

Table 8.5: 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

8.2 Results

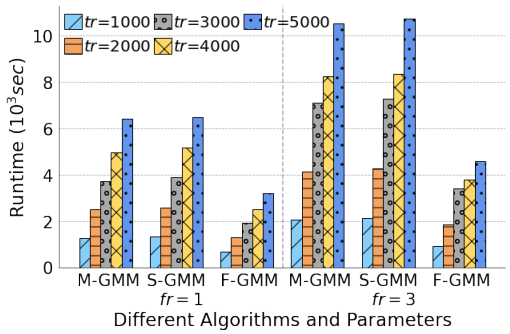
8.2.1 Results on Synthetic Datasets

We depict the results of all the cases listed for GMM, NN and PPCA in Table 8.1, Table 8.2 and Table 8.3 for three algorithms introduced in Chapter 5, 6 and 7. They are M-GMM, S-GMM, F-GMM for GMM, M-NN, S-NN, F-NN for NN, and M-PPCA, S-PPCA, F-PPCA for PPCA.

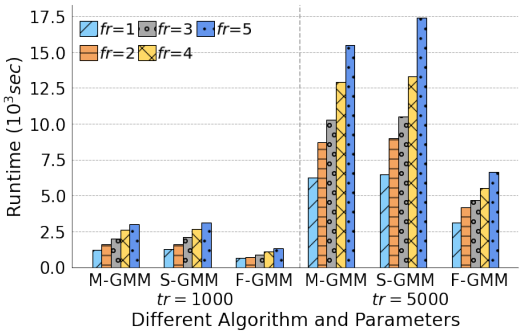
The results for GMM are shown in Figure 8.1. In all cases, F-GMM is the fastest among all the approaches and S-GMM is a bit slower than M-GMM. For the parameters of interest, in Figure 8.1(a), no matter $fr = 1$ or $fr = 3$, with the increase of tuple ratio (tr), the runtime of the F-GMM grows more slowly than the other two algorithms. In other words, the benefit of F-GMM will become more obvious as tr increases. Besides, we compare the difference of this trend between

two different fr. For $tr = 1000$, when $fr = 1$, F-GMM is 2 times faster than S-GMM, which becomes 2.4 times faster when $fr = 3$. As to Figure 8.1(b), it is easily found that when $tr = 1000$, the benefits of F-GMM rise from 2 times to 2.4 times and when $tr = 5000$, the range of benefits is from 2.1 times to 2.6 times. Thus, the efficiency will keep on increasing as we increase tr or fr. Finally, Figure 8.1(c) illustrates that the benefits of F-GMM will drop from 2.8 times to 1.9 times when varying K from 1 to 10 for fixed tr and fr. So the number of clusters will affect the amount of efficiency improvement. When K reaches a large value, the benefit may disappear.

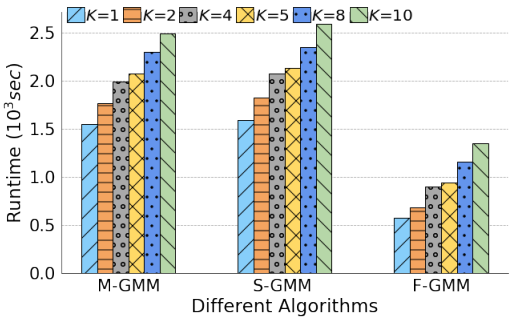
The performance improvement for the case of NN is shown in Figure 8.2. It can be seen that the savings of F-NN are more obvious for all the selection of parameters, but this is based on our setting that only one hidden layer is in our neural network. Figure 8.2(a) presents the results when increasing tr. For $fr = 1$, F-NN becomes more than 2 times faster than S-NN. These savings will keep a slight upward tendency as tr increases further. For $fr = 3$, F-NN is around 3 times faster than S-NN and gradually rises with the increase of tr. Figure 8.2(b) reveals the corresponding experiments varying fr. With the growth of fr, performance advantages vary increasingly from 2 times to 3 times faster for $tr = 1000$ and from 2.1 to 3.2 times for $tr = 5000$. Besides, Figure 8.2(c) presents the results for increasing n_h in the network. For fixed tr and fr, as n_h increases, F-NN still gains



(a) Varying tr



(b) Varying fr

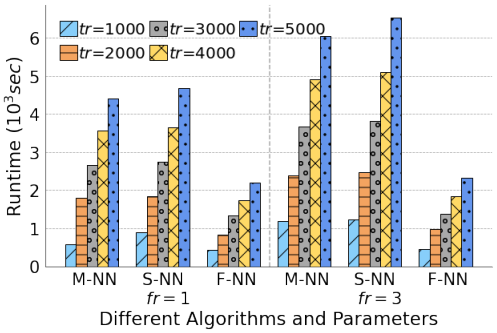


(c) Varying K

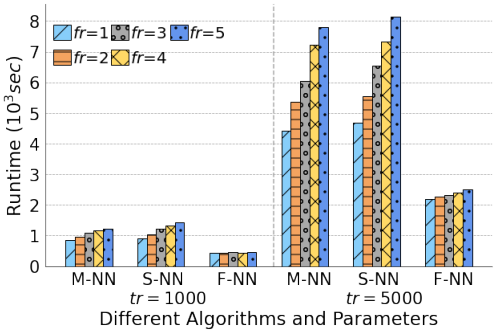
Figure 8.1: Results for GMM algorithms varying parameters of interest

obvious benefits. Even when $n_h = 90$, it is 2.7 times faster than S-NN. Besides, although n_h increases 9 times, the runtimes for them only have a small difference.

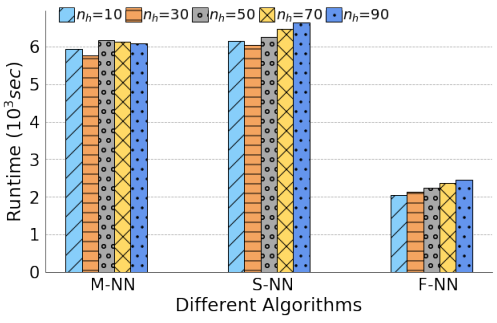
For the results of PPCA, Figure 8.3 describes the runtimes for all the cases. The overall trend is the same as the previous two models. As to varying tr in Figure 8.3 (a), with the rise of tr, all three algorithms will increase their runtimes, but F-PPCA always gains the fastest one than the other two and the benefits will be more and more obvious. When $fr = 1$, it changes from 1.5 times ($tr = 1000$) to 2.4 times ($tr = 50000$). When $fr = 3$, it becomes 1.9 times ($tr = 1000$) to 3 times ($tr = 50000$) and the benefit of it is more than the case of $fr = 1$. In Figure 8.3 (b), the differences among the results of F-PPCA are tiny when varying fr between 1 and 9. However, both M-PPCA and S-PPCA increase significantly with the growth of fr. Thus, compared with M-PPCA and S-PPCA, F-PPCA gains more time savings. Specifically, for $tr = 200$, it is 1.2 times to 3 times faster than others and for $tr = 1000$, its range is from 1.3 times to 3.3 times. Obviously, the larger tr brings more benefits. Figure 8.3(c) represents the performance for different choices of p. The changes are not significant for all the algorithms for the fixed tr and fr. Therefore, the choice of p has no great effect on our experimental results.



(a) Varying tr

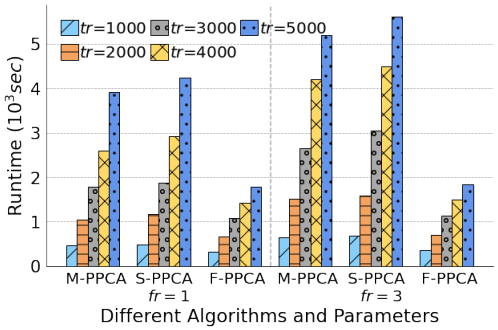


(b) Varying fr

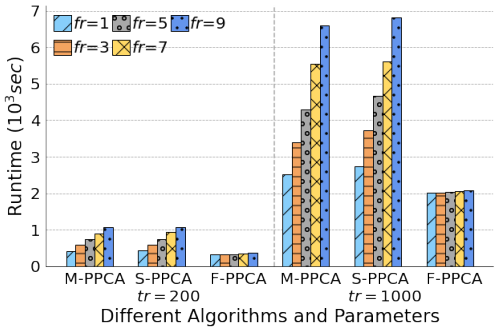


(c) Varying n_h

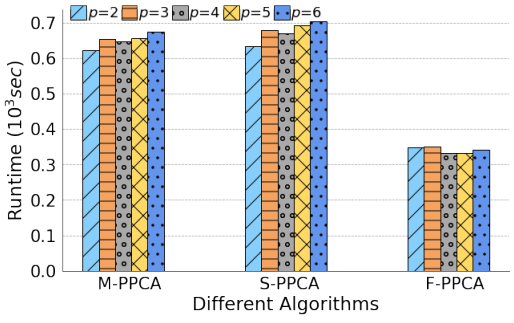
Figure 8.2: Results for NN algorithms varying parameters of interest



(a) Varying tr



(b) Varying fr



(c) Varying p

Figure 8.3: Results for PPCA algorithms varying parameters of interest

8.2.2 Results on Real Datasets

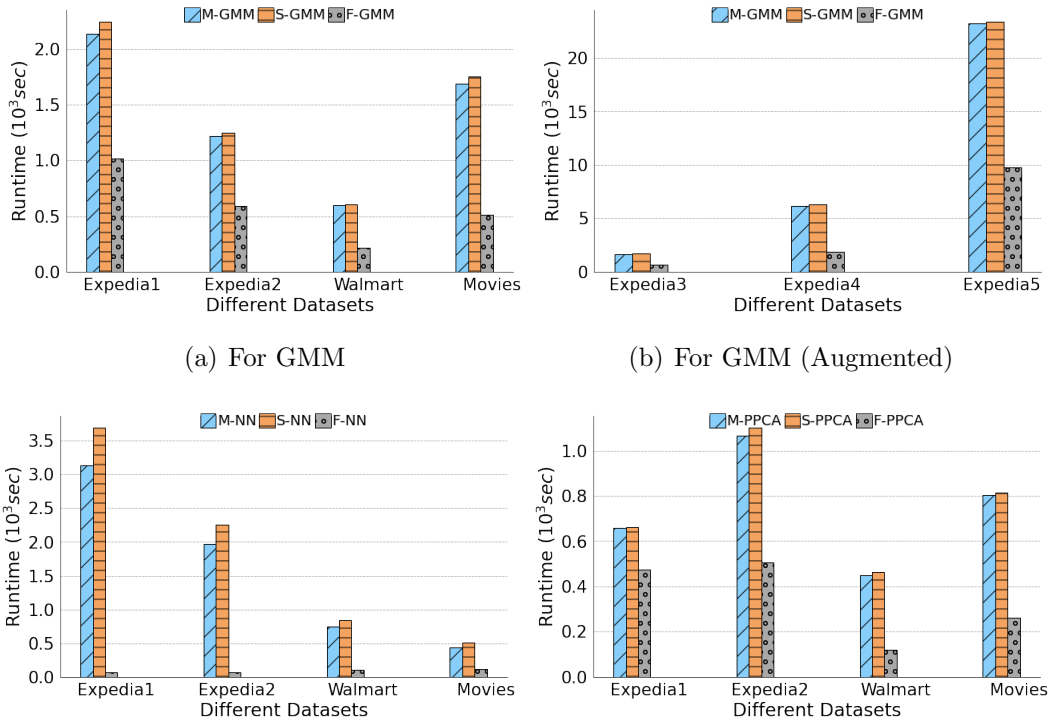
The following figures show the performance for the real datasets. F-GMM, F-NN and F-PPCA gain much more significant advantages than the other two baseline approaches.

The results of GMM are shown in Figure 8.4(a). The performance benefits of F-GMM for different datasets are up to 3.4 times faster than others. When we use the augmented datasets to repeat the same experiment for *Expedia3*, *Expedia4* and *Expedia5* datasets, the benefits of F-GMM range from 2.4 to 3.4 times faster as both tr and fr have been changed while as to *Expedia1*, it is 2.2 times.

For the cases of NN, we can see more benefits from Figure 8.4(b). Due to the reason that we use sparse datasets, the redundancy ratio is high after being encoded. As we have demonstrated on the synthetic datasets, the performance benefits of our approach become larger as redundancy increases. Specifically, F-NN demonstrates 52 times faster execution for the *Expedia* dataset, 8.1 times faster execution for *Walmart* dataset and 4.5 times faster execution for *Movies* dataset. As demonstrated using real datasets, these results attest to the significance of our proposals given the enormous recent interest in NN and associated learning technologies in academia and industry.

As to PPCA in 8.4(c), the benefit of performance for F-PPCA is most significant

for the *Walmart* dataset with up to 3.9 times faster than other algorithms. For the *Movies* dataset, it is 3.1 times, which is also a critical improvement.



(a) For GMM

(b) For GMM (Augmented)

(c) For NN

(d) For PPCA

Figure 8.4: Results on real datasets

As introduced before, for the results of the experiments, we only compare the speed improvement among our proposed factorized algorithm and baseline algorithms for one specific ML application scenario. Differences in efficiency among GMM, NN and PPCA (for example, why PPCA is faster than NN) are determined by the training algorithms themselves and not within our consideration.

9 Conclusions and Future Work

9.1 Conclusions

This thesis focuses on ML over normalized data without materializing the table after joins as the input.

Firstly, we have proposed a factorized approach suitable for three applications in ML area, based on previous studies about linear models. It utilizes the original normalized data before joins for training directly without storing the join results on the disk or in the memory to eliminate the redundancy existing in the input table for ML, which is usually obtained by joining normalized tables. In the meantime, we conclude that as the characteristics of the underlying datasets change, the performance of this method can gain more benefits.

Secondly, to compare the effectiveness of the factorized way, we also have designed two baseline approaches to analyze the efficiency improvement. One is materializing the join results on the disk and feeding the tuples after joins to the models. The other one is performing joins on the fly to directly provide the sub-results of joins in the memory to the models.

Thirdly, we have adopted Gaussian Mixture Models (GMM), Neural Networks (NN) and Probabilistic Principal Component Analysis (PPCA) as specific examples to prove the feasibility and correctness of our proposed method. We give the three alternative algorithms for each, including two baseline algorithms and one factorized algorithm (F-GMM for GMM, F-NN for NN and F-PPCA for PPCA). Meanwhile, we analyze how the factorized algorithms improve efficiency when obtaining the normalized data and calculate the training process compared with the other two.

Fourthly, to make the proposed algorithms more applicable, we use binary joins to give the decomposition process and then generalize the solutions for multi-way joins.

Lastly, we have designed and conducted a series of experiments on synthetic and real datasets to measure the efficiency quantitatively among all the cases introduced and explore the factors that influence the benefits of our proposed algorithms. The results show that the factorized algorithms significantly outperform the baseline methods in terms of efficiency improvement. The amount of redundancy existing in the two joined tables is critical in determining efficiency improvement.

9.2 Future Work

While ML application scenarios require different treatment, we expect some of the principles adopted in this work will apply to broader fields. Each of these

algorithms has its merits in specific application scenarios and this would be the case for algorithms relying on the factorization concepts proposed herein. It is an excellent direction to build a library of such implementations of ML scenarios. We can make the initial contributions but anticipate it to evolve into an open-source project with broad involvement from the community. For managing the model implementation in the community, we can adopt a UDF library approach, similar to MADLib, and deliver the algorithms with native implementations, similar to SparkML.

Bibliography

[1] A. Silberschatz, H. F. Korth, S. Sudarshan *et al.*, *Database system concepts*. Mcgraw-hill New York, 1997, vol. 4.

[2] A. Kumar, J. Naughton, and J. M. Patel, "Learning generalized linear models over normalized data," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 1969–1984.

[3] L. Chen, A. Kumar, J. Naughton, and J. M. Patel, "Towards linear algebra over normalized data," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, 2017.

[4] Z. Cheng and N. Koudas, "Nonlinear models over normalized data," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019, pp. 1574–1577.

[5] K. Yang, Y. Gao, L. Liang, B. Yao, S. Wen, and G. Chen, "Towards factorized svm with gaussian kernels over normalized data," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1453–1464.

[6] Z. Cheng, N. Koudas, Z. Zhang, and X. Yu, "Efficient construction of nonlinear models over normalized data," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1140–1151.

[7] L. Zhou, S. Pan, J. Wang, and A. V. Vasilakos, "Machine learning on big data: Opportunities and challenges," *Neurocomputing*, vol. 237, pp. 350–361, 2017.

[8] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, "Compressed linear algebra for large-scale machine learning," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 960–971, 2016.

[9] M. Abo Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich, "In-database learning with sparse tensors," in *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2018, pp. 325–340.

[10] O. Y. Al-Jarrah, P. D. Yoo, S. Muhaidat, G. K. Karagiannidis, and K. Taha, "Efficient machine learning for big data: A review," *Big Data Research*, vol. 2, no. 3, pp. 87–93, 2015.

[11] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, "Data management challenges in production machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1723–1726.

[12] B. Jan, H. Farman, M. Khan, M. Imran, I. U. Islam, A. Ahmad, S. Ali, and G. Jeon, "Deep learning in big data analytics: a comparative study," *Computers & Electrical Engineering*, vol. 75, pp. 275–287, 2019.

[13] H. Miao, A. Li, L. S. Davis, and A. Deshpande, "Towards unified data and lifecycle management for deep learning," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 2017, pp. 571–582.

[14] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine, "Simulation of database-valued markov chains using simsql," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 637–648.

[15] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan, "Mlbase: A distributed machine-learning system," in *Cidr*, vol. 1, 2013, pp. 2–1.

[16] X. Li, B. Cui, Y. Chen, W. Wu, and C. Zhang, "Mlog: Towards declarative in-database machine learning," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1933–1936, 2017.

[17] K. Kara, K. Eguro, C. Zhang, and G. Alonso, "Columnml: Column-store machine learning with on-the-fly data transformation," *Proceedings of the VLDB Endowment*, vol. 12, no. 4, pp. 348–361, 2018.

[18] Z. J. Gao, S. Luo, L. L. Perez, and C. Jermaine, "The buds language for distributed bayesian machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 961–976.

[19] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li *et al.*, "The madlib analytics library," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, 2012.

[20] 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 *et al.*, "Systemml: Declarative machine learning on spark," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1425–1436, 2016.

[21] R. Anil, G. Çapan, I. Drost-Fromm, T. Dunning, E. Friedman, T. Grant, S. Quinn, P. Ranjan, S. Schelter, and Ö. Yilmazel, "Apache mahout: Machine learning on distributed dataflow systems." *J. Mach. Learn. Res.*, vol. 21, pp. 127–1, 2020.

[22] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[23] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[24] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *Siam Review*, vol. 60, no. 2, pp. 223–311, 2018.

[25] V. Dunjko, J. M. Taylor, and H. J. Briegel, "Quantum-enhanced machine learning," *Physical review letters*, vol. 117, no. 13, p. 130501, 2016.

[26] J. Park, H. Sharma, D. Mahajan, J. K. Kim, P. Olds, and H. Esmaeilzadeh, "Scale-out acceleration for machine learning," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 367–381.

[27] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Proceedings of the 25th International Conference on Neural Information Processing Systems-Volume 2*, 2012, pp. 2951–2959.

[28] D. Mahajan, J. K. Kim, J. Sacks, A. Ardalan, A. Kumar, and H. Esmaeilzadeh, "In-rdbms hardware acceleration of advanced analytics," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, 2018.

[29] D. Xin, L. Ma, J. Liu, S. Macke, S. Song, and A. Parameswaran, "Accelerating human-in-the-loop machine learning: challenges and opportunities," in *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, 2018, pp. 1–4.

[30] D. Olteanu and M. Schleich, "Factorized databases," *ACM SIGMOD Record*, vol. 45, no. 2, pp. 5–16, 2016.

[31] N. Bakibayev, D. Olteanu, and J. Závodnỳ, "Fdb: A query engine for factorised relational databases," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, 2012.

[32] N. Bakibayev, T. Kociskỳ, D. Olteanu, and J. Závodnỳ, "Aggregation and ordering in factorised databases," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, 2013.

[33] S. Rendle, "Scaling factorization machines to relational data," *Proceedings of the VLDB Endowment*, vol. 6, no. 5, pp. 337–348, 2013.

[34] D. Olteanu and J. Závodnỳ, "Size bounds for factorised representations of query results," *ACM Transactions on Database Systems (TODS)*, vol. 40, no. 1, pp. 1–44, 2015.

[35] D. Olteanu and M. Schleich, "F: Regression models over factorized views," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, 2016.

[36] M. Schleich, D. Olteanu, and R. Ciucanu, "Learning linear regression models over factorized joins," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 3–18.

[37] S. Li, L. Chen, and A. Kumar, "Enabling and optimizing non-linear feature interactions in factorized linear algebra," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1571–1588.

[38] M. A. Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich, "Ac/dc: In-database learning thunderstruck," in *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, 2018, pp. 1–10.

[39] A. Kumar, J. Naughton, J. M. Patel, and X. Zhu, "To join or not to join? thinking twice about joins before feature selection," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 19–34.

[40] A. Kumar, M. Jalal, B. Yan, J. Naughton, and J. M. Patel, "Demonstration of santoku: Optimizing machine learning over normalized data," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1864–1867, 2015.

[41] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[42] D. A. Reynolds, "Gaussian mixture models." *Encyclopedia of biometrics*, vol. 741, 2009.

[43] T. Kohonen, "An introduction to neural computing," *Neural networks*, vol. 1, no. 1, pp. 3–16, 1988.

[44] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006.

[45] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[46] M. E. Tipping and C. M. Bishop, "Probabilistic principal component analysis," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 61, no. 3, pp. 611–622, 1999.

[47] I. J. Goodfellow, Y. Bengio, and A. C. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org/