Quadratically Regularised Principal Component Analysis over Multi-Relational Databases

Gabriel-Robert Ineluş

Keble College

University of Oxford

Supervisor: Prof. Dan Olteanu

4^{th} Year Project Report

Final Honour School of Computer Science - Part C

Trinity 2019

Abstract

This thesis puts forward two novel approaches to learning generalised low rank models over multi-relational databases. It focuses on learning Quadratically Regularised Principal Component Analysis models over the non-materialised data matrix, which is the result of a natural join query over a multi-relational database. This is achieved by providing variants of two existing fundamental algorithms: alternating minimisation and eigen-decomposition. These variants use factorised learning, a recent computational paradigm that decomposes the learning task into batches of aggregates which are pushed past the join query. We call these variants Factorised Alternating Minimisation - FAM, and Factorised Eigen-Decomposition - FED. We analyse FAM and FED from a computational complexity point of view and also benchmark our implementations regarding their complexity and numerical precision. This thesis is the first to look into factorised learning of Quadratically Regularised Principal Component Analysis models over multi-relational databases. When executed over Housing and Retailer data sets, FAM and FED achieve orders of magnitude speed-up compared to state-of-art approaches which materialise the data matrix. Computing generalised low rank models orders of magnitude faster means that more models can be computed and can be maintained up to date in the presence of changes to the underlying data. Hence, one could use one measure of performance (speed), to improve a seemingly orthogonal measure of performance (accuracy).

Acknowledgements

I would like to thank my supervisor Prof. Dan Olteanu for believing in me and for organising weekly meetings where we discussed the mathematics behind the project. I also want to thank Prof. Olteanu's DPhil students Djordje Zivanovic and Maximilian Schleich for helping me understand how to setup and execute LMFAO query engine in connection with the programs I developed for this thesis.

I would also like to thank my family for their continuous support and encouragement and especially my wife Isobel, for proofreading my thesis.

Contents

Conventions and Notations

$\mathbb{N}$	The set of natural numbers $0, 1, 2, \ldots$.
$\mathbb{N}^*$	$\mathbb{N} \setminus \{0\}$, i.e. $1, 2, 3, \ldots$.
$S^{m \times n}$	The set of matrices with m rows and n columns whose elements are members of the set S.
$\mathbf{v}$	We use **bold** lower case letters for vectors. Furthermore, if $\mathbf{v} \in \mathbb{R}^m$ then $\mathbf{v} \in \mathbb{R}^{m \times 1}$ i.e. we interpret vectors as columns.
$\mathbf{v}^T$	The transpose of $\mathbf{v}$.
$\|\mathbf{v}\|_2$	The Euclidean norm of a vector $\|\mathbf{v}\|_2 = \sqrt{\mathbf{v}^T\mathbf{v}}$.
x	We use lower case letters for scalars.
A	We use upper-case capitals in italics for matrices.
$A_{:,j}$	Denotes the j^{th} column of A.
$A_{i,:}$	Denotes the i^{th} row of A.
$A_{i,j}$	Denotes the element in row i and column j of A.
$\|A\|_F$	The Frobenius norm of $A \in \mathbb{R}^{m \times n}$: $\sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} A_{i,j}^2}$.
$Tr(A)$	The trace of the square matrix $A \in \mathbb{R}^{n \times n}$ is $\sum_{i=1}^{n} A_{i,i}$.
I_n	For a given $n \in \mathbb{N}^*$, $I_n \in \mathbb{R}^{n \times n}$ denotes the identity square matrix defined by the Kronecker delta function: $\delta_{i,j} = \begin{cases} 1 & i = j, \\ 0 & i \neq j. \end{cases}$
orthonormal matrix	We call a matrix "orthonormal" and denote it by Q if its columns $\mathbf{q}_1 \ldots \mathbf{q}_n$ for some $n \in \mathbb{N}^*$ are orthogonal unit vectors i.e. $\mathbf{q}_i^T\mathbf{q}_j = \begin{cases} 1 & i = j, \\ 0 & i \neq j. \end{cases}$ Immediately we get $Q^TQ = I_n$.
orthogonal matrix	We call an orthonormal matrix "orthogonal" if it is square i.e. $Q \in \mathbb{R}^{n \times n}$. Hence we get $QQ^T = I_n$ too.

1 Introduction

1.1 Motivation

This thesis investigates the problem of learning Quadratically Regularised Principal Component Analysis (L2-PCA) models over multi-relational databases. The primary motivation for considering L2-PCA is Data Science where Principal Component Analysis (PCA) is a key technique used to analyse data. PCA is a templatised method for **dimensionality reduction** which produces a compact representation of data whilst minimising information loss. PCA can be also used to [4]: discover patterns in data, produce less noisy and more informative representations of the data, perform matrix completion, and prepare data before further analysis such as clustering or finding the k-nearest neighbours. Thus, PCA is an important class of machine learning (ML) models. L2-PCA is considered over classic PCA for the same reason as Ridge Regression is considered over classic Linear Regression: the (L2) regularisers prevent the model from over-fitting [5].

Furthermore, multi-relational databases are considered because over 60% of the data that enterprises rely on is represented in **relational form** [24], [25]: 86% in Retail, 83% in Insurance, 82% in Marketing, 77% in Finance.

Let L2-PCA's input data matrix be described as the **non-materialised** result[1]: $A = Q(\mathcal{D}) \in \mathbb{R}^{m \times n}$ of an arbitrary fixed query Q over a multi-relational database $\mathcal{D}$ which in materialised listing form has m tuples and each tuple has n attributes. A's rank-k L2-PCA can be formulated as:

$$\text{minimise: } \|A - XY\|_F^2 + \gamma\|X\|_F^2 + \gamma\|Y\|_F^2 \tag{1.1}$$

$$\text{subject to: } X \in \mathbb{R}^{m \times k} \text{ and } Y \in \mathbb{R}^{k \times n}, \tag{1.2}$$

where k is a user-defined parameter representing the dimensionality reduction rank and $\gamma \geq 0$ is a tunable hyper-parameter that penalises PCA to avoid over-fitting.

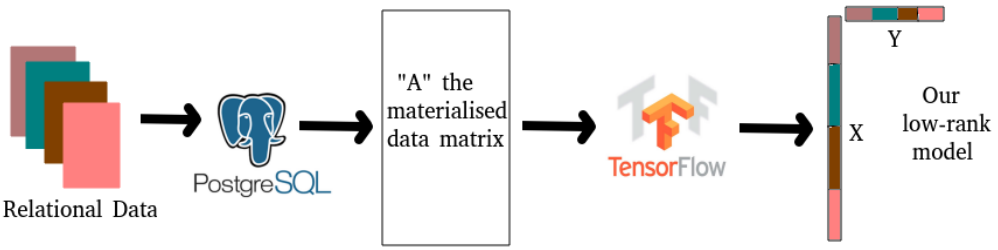

Figure 1.1: Today's workflow for computing L2-PCA over multi-relational databases.

The workflow (Fig. 1.1) used today by data scientists to train L2-PCA model is **structure-agnostic**. We see that the internal structure of the multi-relational database is completely **thrown away** and replaced by the *materialised* data matrix $A \in \mathbb{R}^{m \times n}$ which is computed by some Database Management System (DBMS) such as PostgreSQL. This computation is highly costly both from a memory and a time complexity point of view as **the number of rows that the resulting matrix A has can be exponential in the number of relations joined by Q**. Matrix A is then exported in some generic representation (possibly CSV) then is loaded into a company's favourite ML toolkit (e.g. TensorFlow - Google, PyTorch - Facebook). This data pipelining is also very expensive as it is likely to move GiBs of data. Then, after A is loaded into the ML toolkit, state-of-art algorithms are

[1]In case the data matrix $A \in \mathbb{R}^{m \times n}$ contains categorical features (columns), they are expanded using one-hot encoding. In this situations, n will actually become $\mathcal{O}(nd)$ where d is the size of the largest active domain among the categorical features. This worst-case number of columns is actually over estimating the complexity results as LMFAO uses a more succinct encoding based on FAQ-width [9] (see equation 2.19 for more details).

used to produce the desired low rank model of A. There is potentially exponential redundancy in this workflow coming from the fact that the matrix A is materialised. This causes cross products between relations to be expanded while the relational structure (potentially exponentially more succinct) is thrown away.

The relational structure comes with a fresh opportunity: **aggregates over the columns** of A, such as the product between two columns of A, can be efficiently computed by batch aggregate query engines like LMFAO [12], F [17] and AC/DC [18], directly over the input database, without materialising A. LMFAO is able to evaluate aggregates over the columns of A as batches of operations which it **pushes past the joins** of the query Q and thus evaluates the aggregates **exponentially faster** than naive approaches which use materialised A. This performance is achieved by factorisation of the join query Q, which essentially exploits the distributivity of $\times$ over $\cup$ (relational algebra -RA- terminology). This enables LMFAO to compute aggregates over A's columns without computing the actual joins. This computation paradigm is called **factorised computation**. More details about factorised computation can be found in Section 2.4.

1.2 Contributions

This thesis introduces two novel algorithms for computing L2-PCA models over the data matrix A which is the **non-materialised** result of a join query Q over a multi-relational database $\mathcal{D}$. The first algorithm, Factorised Alternating Minimisation (FAM), is a version of Alternating Minimisation, an **iterative approximation** which at convergence produces A's L2-PCA. The second algorithm, Factorised Eigen-Decomposition (FED), is a **closed form solution** based on A's Singular Value Decomposition (SVD) and spectral theorems of linear algebra ([1] Chapter 6.4 page 318 and Chapter 6.7 page 352). Its core component is the eigen-decomposition of $A^T A$. Both algorithms use **factorised computation paradigm** to efficiently calculate the necessary aggregates over A's columns in order to perform the learning tasks.

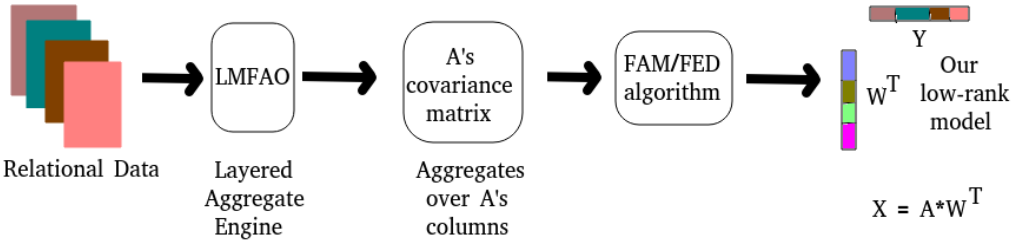

Figure 1.2: The workflow proposed by this thesis.

Fig. 1.2 displays the proposed workflow for training the L2-PCA model of the non-materialised data matrix $A \in \mathbb{R}^{m \times n}$. The core idea is to use the recent LMFAO aggregate query engine to efficiently compute (without depending on m) batches of aggregates over A's columns as required by FED and FAM algorithms such that the full listing representation of A is never materialised while $W \in \mathbb{R}^{k \times n}$ (such that $X = AW^T$) and $Y \in \mathbb{R}^{k \times n}$, are computed. Observe that the computed matrices W and Y do not depend on m. Furthermore, even though X can be materialised, in order to make a prediction one actually needs a single row of X say $X_{i,:} = A_{i,:}W^T$ where $A_{i,:}$ is the join of one tuple from each of the input relations:

$$A_{i,j} \approx X_{i,:}Y_{:,j} = (A_{i,:}W^T)Y_{:,j}. \tag{1.3}$$

This can be calculated and cached in $\mathcal{O}(nk)$. Note that in the classic algorithms, when X and Y are both materialised, predicting $A_{i,j}$ is $\mathcal{O}(k)$. This is the trade-off we have to pay in order to train the L2-PCA model and make predictions without depending on the number of rows m of the data matrix A.

A **direct application** of L2-PCA is low rank matrix completion. The data matrix A can be the result of non equi-joins (such as left outer joins), in which case *NULL* values will appear amongst A's entries. These *NULL* values can be replaced by default values (e.g. 0 in continuous case). Hence, L2-PCA can be naturally used to impute these missing values. One can recover a unknown matrix $A \in \mathbb{R}^{m \times n}$ of rank k from as little as $\Omega(nk \log n)$ known entries [7].

Asymptotic time complexity results:
When computing the L2-PCA of the data set matrix $A = Q(\mathcal{D}) \in \mathbb{R}^{m \times n}$, the two complexities below do not depend on m, but rather on $|\mathcal{D}|^{fhtw(Q)}$ where $|\mathcal{D}| = \sum_{R \in \mathcal{D}} |R|$ and $|R|$ is the number of tuples in relation R. Furthermore, $fhtw(Q)$ is the fractional hyper-tree width [23] of the query Q.

- FAM can iteratively compute a rank-k approximation of $A \in \mathbb{R}^{m \times n}$'s L2-PCA in $\mathcal{O}(n^2 k \times iterations + n^2 |\mathcal{D}|^{fhtw(Q)})$ time complexity, where $n^2|\mathcal{D}|^{fhtw(Q)}$ is the time LMFAO takes to compute the covariance matrix $(A^T A)$ required by FAM and $iterations$ is the number of steps until convergence to global minimum (proof in Appendix A.1.2 of [7]) or a hard limit set by the user (full details in Chapter 3).

- FED can compute an arbitrary-rank closed-form solution to L2-PCA in $\mathcal{O}(n^3 + n^2|\mathcal{D}|^{fhtw(Q)})$ time complexity where $n^2|\mathcal{D}|^{fhtw(Q)}$ is the time LMFAO takes to compute the covariance matrix $(A^T A)$ required by FED (full details in Chapter 4).

In case the query result A contains categorical features (columns), these need to be expanded using one-hot encoding. Thus, the number of columns of A becomes at most nd where d is the size of the largest active domain among the categorical features and n is the original number of columns. Thus, $A \in \mathbb{R}^{m \times nd}$. However, we will redefine $n := nd$ in categorical case because of simplicity and consistency between the complexity analysis for numerical and categorical data matrix A. **In the categorical case, LMFAO [12] actually uses a more succinct encoding than the assumed one-hot using FAQ-width [9], thus it achieves a factor of d speed-up where d is the size of the largest categorical active domain.**

FAM and FED can be **exponentially faster** than any approach that depends on the full listing (materialised) representation of the data matrix $A \in \mathbb{R}^{m \times n}$, which can have size $m = |\mathcal{D}|^{\rho^*(Q)} \gg |\mathcal{D}|^{fhtw(Q)}$, as the difference between $\rho^*(Q)$ (fractional edge cover [22]) and $fhtw(Q)$ (fractional hype-tree width [23]) can be as large as

the number of relations in Q (section 3.1 of [8]). See more details about query size measures in Section 2.4.

Implementation:

FAM and FED have been successfully implemented in *C++* programming language and connected to LMFAO. Details are presented in Chapter 5.

Experimental speed-up results:

Extensive performance experiments measuring the speed of calculating the L2-PCA of the data matrix A which is the result of a join query over the multi-relational databases $\mathcal{D}_{\text{housing}}$ and $\mathcal{D}_{\text{retailer}}$ have been conducted in Chapter 6. These experiments empirically showed that both FAM and FED implementations **outperform by several orders of magnitude** the current state-of-art PCA libraries (Scikit-Learn, TensorFlow, PyTorch) on top of PostgreSQL both from a time and memory complexity point of view even when the state-of-art algorithms are calculating a simpler model: A's PCA.

1.3 Outline

The structure of the remaining chapters is as follows:

Chapter 2 introduces the mathematics which underpins PCA and L2-PCA alongside the topic of factorised joins over multi-relational databases.

Chapter 3 presents *Factorised Alternating Minimisation* algorithm for computing L2-PCA, along with its asymptotic complexity analysis and numerical stability discussion.

Chapter 4 describes *Factorised Eigen-Decomposition* algorithm for computing L2-PCA, along with its asymptotic complexity analysis and numerical stability discussion.

Chapter 5 gives detail on the *C++* implementation of FAM and FED algorithms. The emphasis is on the key system design and algorithmic decisions.

Chapter 6 depicts the experimental setup and data sets. The performance of FAM and FED is contrasted against state-of-art PCA packages from TensorFlow, PyTorch and SkLearn which rely on PostgreSQL to compute the materialised join result.

Chapter 7 outlines related work on Factorised Learning and Linear Algebra over databases.

Chapter 8 summarises the outcomes of this thesis and considers ideas for future research in factorised learning generalised low rank models.

2
Preliminaries

Principal Component Analysis (PCA) uses the core of linear algebra i.e. *matrices*, *vector spaces* and various operations between them. These basic concepts alongside more advanced topics are explained in detail in Gilbert Strang's *Introduction to Linear Algebra* [1]. Precise references to this textbook will be given whenever linear algebra results are invoked without proof.

2.1 Principal Component Analysis Formulations

PCA is a templatised method for **dimensionality reduction** which produces a compact representation of data whilst minimising information loss. Let $A \in \mathbb{R}^{m \times n}$ be a data matrix. The following is a possible formulation for A's PCA which seeks the best rank-k approximation of A in the least squares sense: Assume $Z \in \mathbb{R}^{m \times n}$,

$$minimise: \quad \|A - Z\|_F^2 \quad \text{subject to: } Rank(Z) \leq k. \tag{2.1}$$

Note that we assumed $k \leq Rank(A)$ and perhaps that $\min(n, m) \gg k$.
Here is another formulation for the rank constraint:

$$Z \quad = \quad XY \text{ where } X \in \mathbb{R}^{m \times k} \text{ and } Y \in \mathbb{R}^{k \times n}, \tag{2.2}$$

which leads to:

$$minimise: \quad \|A - XY\|_F^2 \quad \text{subject to: } X \in \mathbb{R}^{m \times k} \text{ and } Y \in \mathbb{R}^{k \times n}. \tag{2.3}$$

In this formulation, the k columns of X and the k rows of Y represent a rank-k approximation model (using top k principal components) for A. This again assuming $k \leq Rank(A)$ and perhaps $\min(n, m) \gg k$.

Finally, we formulate A's Quadratically Regularised PCA (L2-PCA) - in the Ridge Regression sense - by adding L2 regularisers:

$$minimise: \quad \|A - XY\|_F^2 + \gamma\|X\|_F^2 + \gamma\|Y\|_F^2 \tag{2.4}$$

$$\text{subject to:} \quad X, \in \mathbb{R}^{m \times k} \text{ and } Y \in \mathbb{R}^{k \times n}. \tag{2.5}$$

Observe that L2-PCA is a generalised form of PCA (for instance, setting $\gamma = 0$ produces the least squares version).

2.2 Principal Component Analysis Solution

2.2.1 Formulation

Recall the matrix multiplication formulation for the rank constraint:

$$rank(Z) \leq k \iff Z = XY \text{ where } X \in \mathbb{R}^{m \times k} \text{ and } Y \in \mathbb{R}^{k \times n}, \tag{2.6}$$

which led to (equation 2.3 in the introduction chapter):

$$minimise: \quad \|A - XY\|_F^2 \quad \text{subject to: } X \in \mathbb{R}^{m \times k} \text{ and } Y \in \mathbb{R}^{k \times n}. \tag{2.7}$$

2.2.2 Closed form solution

We factorise A using SVD decomposition (see Strang's textbook chapter 6.7 p352 [1]):

$$A = U\Sigma V^T, \tag{2.8}$$

where $r = Rank(A)$ and:

- $U = \begin{bmatrix} \mathbf{u}_1 & \dots & \mathbf{u}_r \end{bmatrix} \in \mathbb{R}^{m \times r}$, $U^T U = I_r$ where $\mathbf{u}_i$ are the left singular vectors.

- $\Sigma = diag(\sigma_1, \ldots, \sigma_r) \in \mathbb{R}^{r \times r}$ has the singular values $\sigma_1 \geq \cdots \geq \sigma_r > 0$.
- $V = \begin{bmatrix} \mathbf{v}_1 & \ldots & \mathbf{v}_r \end{bmatrix} \in \mathbb{R}^{n \times r}$, $V^T V = I_r$ where $\mathbf{v}_i$ are the right singular vectors.

Taking just the first k singular vectors and k singular values, we obtain $A_k = U_k \Sigma_k V_k^T$ which by Eckart-Young theorem [3] is the best rank-k approximation of A.

Thus, we derive

$$X = U_k \Sigma_k^{\frac{1}{2}}, \tag{2.9}$$

$$Y = \Sigma_k^{\frac{1}{2}} V_k^T, \tag{2.10}$$

as a possible closed form solution to Equation 2.7 minimisation problem. Note that if XY is a solution, so is $(XZ)(Z^{-1}Y)$ thus the solutions are not unique.

2.3 Quadratically Regularised PCA Solutions

Recall that we obtain L2-PCA formulation by adding regularisers:

$$minimise: \quad \|A - XY\|_F^2 + \gamma \|X\|_F^2 + \gamma \|Y\|_F^2 \tag{2.11}$$

$$\text{subject to:} \quad X, \in \mathbb{R}^{m \times k} \; and \; Y \in \mathbb{R}^{k \times n}. \tag{2.12}$$

where $X \in \mathbb{R}^{m \times k}$ and $Y \in \mathbb{R}^{k \times n}$.

2.3.1 Closed form solution

As before, a solution will be given by the rank-k SVD of A, $(A = U_k \Sigma_k V_k^T)$ where the regulariser γ takes a key role in the diagonal matrix of singular values, Σ'_k:

$$X = U_k \Sigma_k'^{\frac{1}{2}}, \tag{2.13}$$

$$Y = \Sigma_k'^{\frac{1}{2}} V_k^T, \tag{2.14}$$

$$\Sigma'_k = \mathbf{diag}((\sigma_1 - \gamma)_+, \ldots (\sigma_k - \gamma)_+), \tag{2.15}$$

where $(x)_+ = max(x, 0)$. A proof of correctness can be found in Appendix A.1.1 of Generalised Low Rank Models paper by M. Udell [7]. Observe that setting

$\gamma = 0$ produces the solution to classical PCA. Choosing a good γ is done using fundamental techniques such as cross-validation. In this thesis, γ is treated as a parameter which is given to the algorithm.

2.3.2 Iterative Solution using Alternating Minimisation

Another way to minimise the objective 2.11 is to use an iterative approximation approach: alternatively fix the value of X and optimise for Y and then fix the value of Y and optimise for X until convergence to the global minimum of the objective function 2.11. Of course that due to floating point precision, the iteration stops when the change for both X and Y is less than τ per iteration where τ is a small value representing the floating point error tolerance.
The iterative algorithm is given by the following:

$$X_{t+1} = AY_t^T(Y_tY_t^T + \gamma I_k)^{-1}, \tag{2.16}$$

$$Y_{t+1} = (X_{t+1}^TX_{t+1} + \gamma I_k)^{-1}X_{t+1}^TA. \tag{2.17}$$

The full derivation of the algorithm can be seen in the Appendix A.2, using matrix calculus formulae explained in Appendix A.1. Observe that setting $\gamma = 0$ produces the Alternating Minimisation solution for classical PCA. Furthermore, observe that the iteration for Y looks precisely as n simultaneous Ridge Regressions (one for each of A's columns).

2.4 Factorised Joins Over Multi-Relational Databases

This thesis uses factorised computation solely for the purpose of computing the covariance matrix of the data set on which L2-PCA is applied. This computation is done completely using previous work and this thesis simply builds applications on top of it.

A database is defined as efficient, reliable, convenient, and safe multi-user storage

of and access to **massive amounts of persistent data** [27]. A multi-relational database is a database which is **stored using at least two relations**.

Structured Query Language (SQL) is a programming language for relational queries over relational databases. A query Q is a request for information which can be compiled and evaluated over a multi-relational database $\mathcal{D}$. The result of a query Q is a relation.

This thesis considers specifically **SUM aggregate joins**, such as:

$$\begin{aligned}&\text{SELECT SUM}(a_i * a_j)\\&\text{FROM } \texttt{relation}_1 \text{ NATURAL JOIN } \texttt{relation}_2 \ \ldots\end{aligned}$$

which expresses $(A^T A)_{i,j}$ - the inner product between the columns i and j of the data matrix A which is the result of a join query Q over a multi-relational database $\mathcal{D}$. If the data matrix A is the result of a non equi-join, then it contains *NULL* values. Fortunately, **SUM aggregates are not affected** as *NULL* values can be replaced with default values (e.g. 0 in the continuous case) which do not affect the sum expressed by the aggregate.

Factorised Databases (FDB) are introduced in detail in [8] using Figure 2.1 as support for the core definitions.

Definition 1 *A Factorised Database (FDB) is defined using Figure 2.1 by its components: (note that a complete formal definition can be found in [8])*

- *a Multi-Relational Database consisting of three relations: Sales, Branch and Competition alongside their Natural Join - sub-figure (a),*
- *hypergraph of the natural join of the relations - sub-figure (b),*
- *variable order Δ defining a possible nesting structure of the factorised join - sub-figure (c)*
- *the factorised query $\Delta(\mathcal{D})$, grounding the variable order Δ over the database $\mathcal{D}$ - sub-figure (d).*

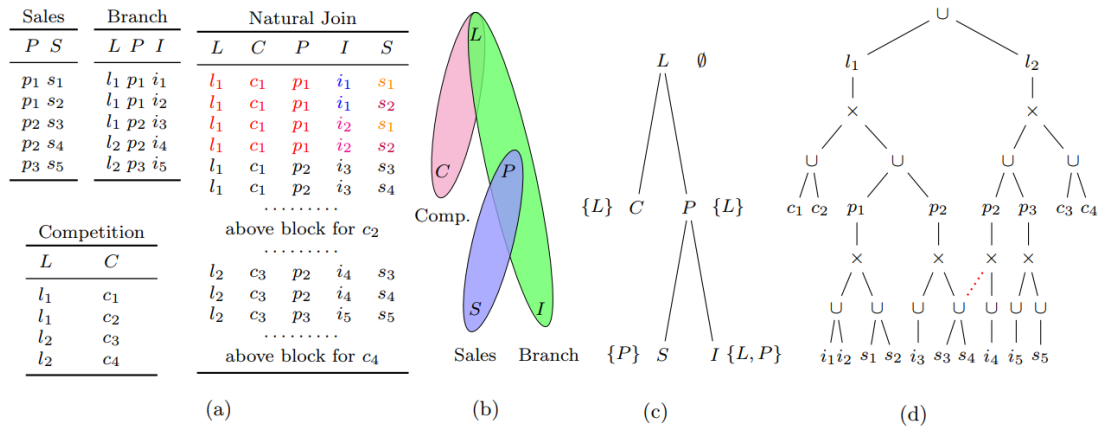

Figure 1: (a) Database with relations Branch(Location, Product, Inventory), Competition(Location, Competitor), Sales(Product, Sale), where the attribute names are abbreviated; (b) Hypergraph of the natural join of the relations; (c) Variable order Δ defining one possible nesting structure of the factorized join result given in (d). The union $s_3 \cup s_4$ is cached under the first occurrence of p_2 and referenced (via a dotted edge) from the second occurrence of p_2.

Figure 2.1: Factorised definitions by example, as presented in *Factorised Databases* [8] Section 2: A Factorization Example

Definition 2 *Factorised computation is defined by taking an algorithm and decomposing it into batches of aggregates which can be pushed past the joins of a query by using the distributivity of* $\times$ *over* $\cup$ *(see sub-figure (d) of Figure 2.1).*

Examples of factorised computation of aggregates:

- The query:

```
SELECT SUM(1)
FROM sales NATURAL JOIN branch
           NATURAL JOIN competition
```

  which aggregates over the multi-relational database presented in Figure 2.1 represents the cardinality of the join result. In this case, one simply turns $\cup$ in $+$ and $\times$ in $*$ in sub-figure (d) of Figure 2.1. Each data value is interpreted as a unit 1 and then the arithmetic expression is evaluated bottom up, yielding $|\Delta(\mathcal{D})|$ without materialising the join.

- The query:

```
SELECT SUM(C*P)
FROM sales NATURAL JOIN branch
           NATURAL JOIN competition
```

which aggregates over the columns C and P of the multi-relational database presented in Figure 2.1 represents the inner product between columns C and P of the natural join in sub-figure (a). This aggregate can be pushed past the join query as well. One turns all values except for C and P into 1, then just as before turns $\cup$ into $+$ and $\times$ into $*$. A bottom up evaluation of the arithmetic expression yields SUM(C*P) without materialising $\Delta(\mathcal{D})$.

As presented in Theorem 3.4 of [8], there are several complexity measures which this thesis will use when discussing factorised computation:

Theorem 1 *([22], [23], [31]) For an arbitrary join query Q and an arbitrary database $\mathcal{D}$, the query result $A = Q(\mathcal{D})$ can have:*

- *a flat representation of size $\mathcal{O}(|\mathcal{D}|^{\rho^*(Q)})$ [22]*
- *a factorised representation (say E) over d-trees of size $\mathcal{O}(|\mathcal{D}|^{fhtw(Q)})$ [23]*

Theorem 3.8 of [8] gives the desired aggregate evaluation complexity:

Theorem 2 *(Generalisation of [30] [17]) Given a variable order Δ and a factorised representation E over Δ, any SQL aggregate of the form SUM(X), MIN(X) or MAX(X) where X is an expression in the semi-ring $(\mathbb{N}[\Delta], +, *, 0, 1)$ can be computed in one pass over E.*

Furthermore, the measures $\rho^*(Q)$ and $fhtw(Q)$ are fractional edge cover number [22] and the fractional hyper-tree width [23] of the query Q. The following inequation holds:

$$1 \leq fhtw(Q) \leq \rho^*(Q) \leq |Q| \tag{2.18}$$

where $|Q|$ is the number of relation literals present in the join query Q.

Proposition 1 *Using the above two theorems, the recent LMFAO query engine 12 can efficiently make use of the factorised join representation and compute aggregates such as SUM($a_i * a_j$) in $\mathcal{O}(|\mathcal{D}|^{fhtw(Q)})$. One application of this is the computation of $S = A^T A$ for which $S_{i,j}$ is precisely the SUM($a_i * a_j$) aggregate. Thus, S can be calculated in $\mathcal{O}(n^2|\mathcal{D}|^{fhtw(Q)})$.*

A key observation to mention here concerns the case when A has categorical features. A categorical feature means a column whose entries are not in $\mathbb{R}$ but rather members of a discrete set, called the domain of the category. This thesis over-estimates the true complexity for computing S. The complexity, as presented in this thesis, will be assuming one-hot encoding. Thus, computing S will take $\mathcal{O}(n^2d^2|\mathcal{D}|^{fhtw(Q)})$ where d is the size of the largest active domain among categorical features (in the worst case, each column is replaced by d new columns). However, LMFAO is using a sparse encoding which is more succinct than one-hot, depending on the FAQ-width (presented in detail in the Appendix of [9]). This efficient encoding reduces the complexity by a factor of d, obtaining the following complexity for calculating S:

$$\mathcal{O}(n^2d|\mathcal{D}|^{fhtw(Q)}) \tag{2.19}$$

3

FAM: Factorised Alternating Minimisation

Chapter 2 introduced the preliminaries. This chapter presents in detail Factorised Alternating Minimisation (FAM) algorithm for computing the Quadratically Regularised Principal Component Analysis (L2-PCA) of some input data matrix described as the **non-materialised** result: $A = Q(\mathcal{D}) \in \mathbb{R}^{m \times n}$ of a query Q over a multi-relational database $\mathcal{D}$ which in materialised listing form has m tuples and each tuple has n attributes. Furthermore, we assume $m \gg n$.

Recall the Alternating Minimisation solution for A's L2-PCA (from Chapter 2):

$$X_{t+1} = AY_t^T(Y_tY_t^T + \gamma I_k)^{-1}, \tag{3.1}$$

$$Y_{t+1} = (X_{t+1}^T X_{t+1} + \gamma I_k)^{-1} X_{t+1}^T A. \tag{3.2}$$

3.1 Inlining of the Iterative Solution

In order to use factorised computation, we notice that when the equation for X_{t+1} is inlined in the equation for Y_{t+1} and the matrices are conveniently bracketed using associativity, one can rewrite the classic Alternating Minimisation to perform operations that only depend on A^TA which, as explained in Section 2.4, *does not require A to be materialised* in order to be computed:

$$
\begin{aligned}
Y_{t+1} &= \left(\left(AY_t^T(Y_tY_t^T+\gamma I_k)^{-1}\right)^T\left(AY_t^T(Y_tY_t^T+\gamma I_k)^{-1}\right)+\gamma I_k\right)^{-1} && (3.3)\\
&\quad \left(AY_t^T(Y_tY_t^T+\gamma I_k)^{-1}\right)^T A, && (3.4)\\
&= \left(\left((Y_tY_t^T+\gamma I_k)^{-1}Y_t\right)(A^TA)\left(Y_t^T(Y_tY_t^T+\gamma I_k)^{-1}\right)+\gamma I_k\right)^{-1} && (3.5)\\
&\quad \left((Y_tY_t^T+\gamma I_k)^{-1}Y_t\right)(A^TA), && (3.6)
\end{aligned}
$$

Let us use the following notation:

$$
\begin{aligned}
S &:= A^TA \in \mathbb{R}^{n\times n}, && (3.7)\\
Z_t &:= (Y_tY_t^T+\gamma I_k)^{-1} \in \mathbb{R}^{k\times k}, && (3.8)\\
W_t &:= Z_tY_t \in \mathbb{R}^{k\times n}. && (3.9)
\end{aligned}
$$

Then:

$$
\begin{aligned}
Y_{t+1} &= \left(\left((Y_tY_t^T+\gamma I_k)^{-1}Y_t\right)(A^TA)\left(Y_t^T(Y_tY_t^T+\gamma I_k)^{-1}\right)+\gamma I_k\right)^{-1} && (3.10)\\
&\quad \left((Y_tY_t^T+\gamma I_k)^{-1}Y_t\right)(A^TA), && (3.11)\\
&= \left((Z_tY_t)S(Y_t^TZ_t^T)+\gamma I_k\right)^{-1}(Z_tY_t)S, && (3.12)\\
&= (W_tSW_t^T+\gamma I_k)^{-1}W_tS, && (3.13)\\
X_{t+1} &= AY_t^T(Y_tY_t^T+\gamma I_k)^{-1}, && (3.14)\\
&= AW_t^T. && (3.15)
\end{aligned}
$$

The factorised query engine (LMFAO) efficiently calculates $S = A^TA$ in $\mathcal{O}(n^2|\mathcal{D}|^{fhtw(Q)})$ where $fhtw(Q)$ is the fractional hypertree width of the query Q. Thus, we do not depend on the number of rows (m) of the (**non-materialised**) matrix A when computing S. Furthermore, we only work with matrices whose sizes depend solely on n and k thus each iteration's complexity is independent of the number of rows (m) of the data matrix A.

3.2 Convergence Analysis

The alternating minimisation algorithm converges to the global optimum (proof in Appendix [A.3]). The convergence checks are:

$$\|Y_{t+1} - Y_t\|_F^2 \leq \tau, \tag{3.16}$$

$$\|X_{t+1} - X_t\|_F^2 = \|AW_t^T - AW_{t-1}^T\|_F^2 \leq \|A\|_F^2 \|W_t^T - W_{t-1}^T\|_F^2 \leq \tau, \tag{3.17}$$

where $\|A\|_F^2$ can be precomputed as $Tr(S)$ and we already know W_t and W_{t-1} as they are required to compute Y_t and Y_{t-1}. We used the submultiplicativity of Frobenius norm in inequation 3.17 which is a simple consequence of the remarkable Cauchy-Buniakovsky-Schwartz inequality (proof in Appendix [B.2.1]).

3.3 FAM Pseudo-code

```
1   /* Note that the possibility of setting a hard limit for
2      iterations has been removed for readability reasons. */
3
4   FUNCTION FAM(S, k, &Y, &W): void // output parameters Y and W
5     gamma := 10^-4
6     tau := 10^-6
7
8     Y_t := [I_k|0] ∈ R^{k×n} // the initial guess of correct shape
9     W_t := 0 ∈ R^{k×n} // the correct shape
10
11    done := false
12    while done ≠ true do
13      Z_t := (Y_t Y_t^T + I_k γ)^-1
14
15      W_old := W_t
16      W_t := Z_t Y_t
17
18      B_t := (W_t S W_t^T + I_k γ)^-1
19
20      Y_old := Y_t
21      Y_t := B_t W_t S
22
23      if Tr(S)‖W_old − W_t‖_F^2 ≤ τ and ‖Y_old − Y_t‖_F^2 ≤ τ
24        done := true
25
26    Y := Y_t
27    W := W_t
28  ■
```

Given $S = A^T A$ and k the desired low rank, it produces Y and W such that $X = AW^T$ and XY is an approximation of the best rank k embedding of A. Due to $Tr(S)$ being large relative to the machine precision, we will only require $\|W_{old} - W_t\|_F^2 \leq \tau$.

3.4 Complexity Analysis

Theorem 3 *Let the data matrix $A = Q(\mathcal{D}) \in \mathbb{R}^{m \times n}$ be the result of an arbitrary fixed join query Q over an arbitrary fixed multi-relational database $\mathcal{D}$. Let fhtw(Q) [23] be the fractional hypertree width of Q. Factorised Alternating Minimisation computes rank-k L2-PCA of A given by matrices $Y \in \mathbb{R}^{k \times n}$ and $W \in \mathbb{R}^{k \times n}$, such that $X = AW^T$ and $A \approx XY$ is the best rank-k approximation of A, in $\mathcal{O}(n^2k \times iterations + n^2|\mathcal{D}|^{fhtw(Q)})$ time.*

Proof 1 *We analyse the complexity of FAM's internal operations over $A \in \mathbb{R}^{m \times n}$ and rank k.*

- *$S = A^TA$ takes $\mathcal{O}(n^2|\mathcal{D}|^{fhtw(Q)})$ time complexity (see Proposition 1) using LMFAO engine (computed once and cached). If A has categorical features, then this is actually **over-estimating** the total complexity as n is replaced by nd (as we assume one-hot encoding) where d is the size of the largest active domain among the categorical features. In practice, LMFAO unleashes the full potential of factorised computation by using a more succinct representation based on FAQ-width [9] and achieves an extra $\mathcal{O}(d)$ time complexity speed-up.*
- *$\|A\|_F^2 = Tr(S)$ takes $\mathcal{O}(n)$ trivially (computed once and cached).*
- *$Y_tY_t^T + \gamma I_k$ takes $\mathcal{O}(k^2n)$ as $Y_t \in \mathbb{R}^{k \times n}$ and $I_k \in \mathbb{R}^{k \times k}$.*
- *$Z_t = (Y_tY_t^T + \gamma I_k)^{-1}$ takes $\mathcal{O}(k^3)$ using Cholesky decomposition.*
- *$W_t = Z_tY_t$ takes $\mathcal{O}(k^2n)$ as $Z_t \in \mathbb{R}^{k \times k}$ and $Y_t \in \mathbb{R}^{k \times n}$.*
- *$W_tSW_t^T + \gamma I_k$ takes $\mathcal{O}(n^2k + k^2n) = \mathcal{O}(n^2k)$ as k is significantly smaller than n and $W_t \in \mathbb{R}^{k \times n}$, $S \in \mathbb{R}^{n \times n}$, $W_t^T \in \mathbb{R}^{n \times k}$ and $I_k \in \mathbb{R}^{k \times k}$.*
- *$B_t = (W_tSW_t^T + \gamma I_k)^{-1}$ takes $\mathcal{O}(k^3)$ using Cholesky decomposition.*
- *$Y_{t+1} = B_tW_tS$ takes $\mathcal{O}(n^2k + k^2n) = \mathcal{O}(n^2k)$ as k is significantly smaller than n and $B_t \in \mathbb{R}^{k \times k}$, $W_t \in \mathbb{R}^{k \times n}$ and $S \in \mathbb{R}^{n \times n}$.*

Looking at the pseudo-code, the time complexity of an iteration step is: $\mathcal{O}(n^2k) + \mathcal{O}(nk^2) + \mathcal{O}(k^3) = \mathcal{O}(n^2k)$ *as* k *is significantly smaller than* n*. The precomputation of* $S = A^TA$ *which takes* $\mathcal{O}(n^2|\mathcal{D}|^{fhtw(Q)})$ *therefore the total complexity is:*

$$\mathcal{O}(n^2k \times iterations + n^2|\mathcal{D}|^{fhtw(Q)}) \blacksquare \tag{3.18}$$

The complexity of the classic Alternating Minimisation algorithm can be reduced to $\mathcal{O}(n^2k \times iterations + n^2m)$*. Our complexity in Equation 3.18 can be* ***exponentially faster*** *than any alternating minimisation approach that depends on the full listing (materialised) representation of* $A = Q(\mathcal{D})$*, which can have size* $m = |\mathcal{D}|^{\rho^*(Q)} \gg |\mathcal{D}|^{fhtw(Q)}$*, as the difference between* $\rho^*(Q)$ *(fractional edge cover) and* $fhtw(Q)$ *(fractional hypertree width) can be as large as the number of relation literals in* Q *(see inequality 2.18).*

3.5 Numerical Stability Optimisation

It can be shown that floating point standard IEEE754 is biased towards small values. The domains for 64-bit double and 80-bit long double can represent values in immense ranges (for example the range for long double is $\approx [-1.18 \times 10^{4932}, -3.65 \times 10^{-4951}] \cup [3.65 \times 10^{-4951}, 1.18 \times 10^{4932}]$). Out of all these values, 50% of the floating point numbers which can be represented are in the interval $(-1, 1)$. See more details in Appendix C.1.

Analysing equation[3.13]:

$$Y_t = (W_tSW_t^T + \gamma I_k)^{-1}W_tS \tag{3.19}$$

$$\iff \quad Y_t = \left(W_t\frac{S}{\alpha}W_t^T + \frac{\gamma}{\alpha}I_k\right)^{-1}W_t\frac{S}{\alpha} \tag{3.20}$$

$$\text{where} \quad \alpha = \frac{Tr(S)}{n} \tag{3.21}$$

We used the fact that the constant α comes out of the inverse as $\frac{1}{\alpha}$ and then used commutativity of scalar with matrices. This observation implies that we can factor away a constant α from S without affecting the result. Hence, the values of S

can be kept small. There is empirical evidence that this observation improves the numerical stability of FAM by orders of magnitude for the Housing data set (see the penultimate experiment in Chapter 6).

3.6 Chubby $\neq$ Bad

It is often the case that it is costly (or impossible) to obtain many data points. In fields such as medical research it is more likely to have a small number of data points which through complicated analysis produced large numbers of features (each sample can represent a patient's medical record). Principal Component Analysis (with its various applications) is nevertheless vital in this domain as well.

Just as before, we assume that the data matrix on which we apply quadratically regularised PCA on is the result of a query Q over a factorised database $\mathcal{D}$: $A = Q(\mathcal{D}) \in \mathbb{R}^{m \times n}$. The key difference is that we now assume $n \gg m$.

3.6.1 Solution using Alternating Minimisation

Recall that the iterative algorithm is given by the following recurrences:

$$X_{t+1} = AY_{t+1}^T(Y_{t+1}Y_{t+1}^T + \gamma I_k)^{-1}, \tag{3.22}$$

$$Y_{t+1} = (X_t^T X_t + \gamma I_k)^{-1} X_t^T A, \tag{3.23}$$

obtained from the objective function:

$$\mathcal{L}(X, Y) = \|A - XY\|_F^2 + \gamma\|X\|_F^2 + \gamma\|Y\|_F^2, \tag{3.24}$$

by fixing X and minimising for Y alternating with fixing Y and minimising for X.

3.6.2 Inlining of the Iterative Solution Revisited

This time the tables have turned: we inline the equation for Y_{t+1} in the equation for X_{t+1} then we can group the matrices such that we perform operations that do not depend on the number of columns (n) of the factorised database data matrix A:

$$
\begin{aligned}
X_{t+1} &= A\Big(A^T X_t (X_t^T X_t + \gamma I_k)^{-1}\Big) && (3.25)\\
&\quad \Big((X_t^T X_t + \gamma I_k)^{-1} X_t^T A (A^T X_t (X_t^T X_t + \gamma I_k)^{-1}) + \gamma I_k\Big)^{-1} && (3.26)\\
&= (AA^T)\Big(X_t (X_t^T X_t + \gamma I_k)^{-1}\Big) && (3.27)\\
&\quad \Big(\big((X_t^T X_t + \gamma I_k)^{-1} X_t^T\big)(AA^T)\big(X_t (X_t^T X_t + \gamma I_k)^{-1})\big) + \gamma I_k\Big)^{-1} && (3.28)
\end{aligned}
$$

Let us use the following notation:

$$
\begin{aligned}
G &:= AA^T \in \mathbb{R}^{m\times m}, && (3.29)\\
K_t &:= (X_t^T X_t + \gamma I_k)^{-1} \in \mathbb{R}^{k\times k}, && (3.30)\\
L_t &:= X_t K_t \in \mathbb{R}^{m\times k}. && (3.31)
\end{aligned}
$$

Then:

$$
\begin{aligned}
X_{t+1} &= (AA^T)\Big(X_t (X_t^T X_t + \gamma I_k)^{-1}\Big) && (3.32)\\
&\quad \Big(\big((X_t^T X_t + \gamma I_k)^{-1} X_t^T\big)(AA^T)\big(X_t (X_t^T X_t + \gamma I_k)^{-1})\big) + \gamma I_k\Big)^{-1} && (3.33)\\
&= G(X_t K_t)\Big((K_t^T X_t^T) G (X_t K_t) + \gamma I_k\Big)^{-1}, && (3.34)\\
&= G L_t (L_t^T G L_t + \gamma I_k)^{-1}, && (3.35)\\
Y_{t+1} &= (X_t^T X_t + \gamma I_k)^{-1} X_t^T A, && (3.36)\\
&= L_t^T A. && (3.37)
\end{aligned}
$$

During iterations, we do not depend on the number of columns (i.e. n) in the time complexity for computing the solution to the quadratically regularised PCA, as we work with matrices whose sizes depend only on m and k.

We will now focus on proving that equations 3.35 and 3.37 can be recovered equivalently assuming FAM runs on $G = AA^T$ instead of $S = A^T A$:

- clearly S became G.

- assuming we rename Y_t to X_t^T, we obtain:

$$Z_t = (Y_t Y_t^T + \gamma I_k)^{-1} = (X_t^T X_t + \gamma I_k)^{-1} = K_t^T$$
$$W_t = Z_t Y_t = K_t^T X_t^T = (X_t K_t)^T = L_t^T$$

- furthermore, we conclude the proof by obtaining:

$$X_{t+1} = A^T W_t^T = A^T L_t = (L_t^T A)^T = Y_{t+1}^T \blacksquare$$

Thus, we obtain in W the value L^T and in Y the value X^T which means that one implementation suffices for both scenarios. $\blacksquare$

4

FED: Factorised Eigen-Decomposition

The previous chapter presented Factorised Alternating Minimisation (FAM) algorithm for performing Quadratically Regularised Principal Component Analysis (L2-PCA). This chapter focuses on the details of Factorised Eigen-Decomposition (FED) algorithm for computing the L2-PCA of some input data set described as the **non-materialised** data matrix: $A = Q(\mathcal{D}) \in \mathbb{R}^{m \times n}$ of a query Q over a multi-relational database $\mathcal{D}$ which in materialised listing form has m tuples and each tuple has n attributes. Furthermore, we assume $m \gg n$.

4.1 Eigenvalue Eigenvector Decomposition

Computing the SVD of A can be very expensive. However, we will rewrite the algorithm which computes A's SVD using the eigen-decomposition of $A^T A$, taking advantage of the fact that we can use factorised computation to efficiently compute $S = A^T A$ using the recent LMFAO [12] query engine:

$$S = A^T A \xrightarrow{\text{eigenvalue-eigenvector decomposition}} L \Lambda L^{-1}. \tag{4.1}$$

Hence, by the spectral theorems[B.3] we have:

$$L \Lambda L^{-1} = L \Lambda L^T. \tag{4.2}$$

On the other hand:

$$A = U\Sigma V^T \tag{4.3}$$

$$\iff \quad S = A^T A = V\Sigma U^T U\Sigma V^T \tag{4.4}$$

$$\iff \quad S = V\Sigma^2 V^T \text{ because } U \text{ is orthonormal} \tag{4.5}$$

Therefore, we know that $S = V\Sigma^2 V^T$ for some orthonormal matrix V and some diagonal matrix Σ^2. But we already have means to obtain these kinds of matrices using equation 4.2. Therefore, we can retrieve:

$$V = L \tag{4.6}$$

$$\Sigma = \Lambda^{\frac{1}{2}}. \tag{4.7}$$

For the curious reader, **it is always the case that in equation 4.7 we have non-negative eigenvalues for** S, proof in appendix[B.3.1].

We can recover Y and W produced by alternating minimisation, except in closed form solution, as presented in section[2.3.1]:

$$Y = \Sigma_k'^{\frac{1}{2}} V_k^T, \tag{4.8}$$

$$W = \Sigma_k'^{-\frac{1}{2}} V_k^T \tag{4.9}$$

The equation 4.9 comes from the equation 3.15 ($X = AW^T$) corroborated with the equation 2.13 ($X = U_k\Sigma_k'^{\frac{1}{2}}$), when $A = U_k\Sigma_k V_k^T$. Also, **the** k **chosen (eigenvector, eigenvalue) pairs will be the top k pairs with the greatest eigenvalues**.

Observe the fact that since we obtain the full matrix Σ, we can pick the rank k of the decomposition by simply looking at the sum representation and choosing k with a satisfying reconstruction accuracy:

$$\begin{aligned} accuracy = 1 - relativeError &= 1 - \frac{\|A' - A_k\|_F^2}{\|A'\|_F^2} = 1 - \frac{\|U_n\Sigma_n' V_n^T - U_k\Sigma_k' V_k^T\|_F^2}{\|U_n\Sigma_n' V_n^T\|_F^2} \\ &= 1 - \frac{\|\sum_{i=1}^{n}(\sigma_i - \gamma)_+ \mathbf{u}_i\mathbf{v}_i^T - \sum_{i=1}^{k}(\sigma_i - \gamma)_+ \mathbf{u}_i\mathbf{v}_i^T\|_F^2}{\|\sum_{i=1}^{n}(\sigma_i - \gamma)_+ \mathbf{u}_i\mathbf{v}_i^T\|_F^2} \\ &= 1 - \frac{\sum_{i=k+1}^{n}(\sigma_i - \gamma)_+^2}{\sum_{i=1}^{n}(\sigma_i - \gamma)_+^2} = \frac{\sum_{i=1}^{k}(\sigma_i - \gamma)_+^2}{\sum_{i=1}^{n}(\sigma_i - \gamma)_+^2}. \end{aligned}$$

Note that Factorised Alternating Minimisation (FAM) did not have such a rank revealing property, but rather assumed the rank is given as an argument. Hence, since FED takes the reconstruction accuracy as an argument, this method seems to be superior in this respect.

4.2 Pseudo-code - FED

```
1  FUNCTION FED(S, k, &Y, &W, accuracy): void // output parameters Y and W
2    L := S_eigenvectors // ordered in decreasing value of eigenvalues
3    D := diag(S_eigenvalues) // in decreasing order σ_1 ≥ ··· ≥ σ_n
4
5    γ := 10^-4
6    EPS := 10^-8
7
8    sum := 0
9    for i := 1 to n do
10     D_{i,i} := max(0, sqrt(D_{i,i}) − gamma)
11     sum := sum + D_{i,i}^2
12
13   current := 0
14   for i := 0 to n do
15     current := current + D_{i,i}^2
16     k = max(k, i + 1)
17     if current + EPS ≥ accuracy ∗ sum break
18
19   for i := 0 to k do
20     Σ'_{i,i} := D_{i,i}
21     V_{:,i} := L_{:,i}
22
23   Y := Σ'^{-1/2} V^T
24   W := Σ'^{-1/2} V^T
25 ■
```

Given $S = A^T A$ and the desired low rank k or the required reconstruction *accuracy*, we return Y and W such that $X = AW^T$ and $\frac{\|A'-XY\|_F^2}{\|A'\|_F^2} \geq accuracy$ where the chosen rank is the maximum between the given k and the smallest rank which satisfies the requested reconstruction *accuracy* (note that full rank gives 100% reconstruction accuracy) and $A' = U\Sigma' V^T$, with $\Sigma' = diag((\sigma_i - \gamma)_+)$, $i \in \{1, \ldots, n\}$ and $a_+ = max(0, a)$.

4.3 Complexity Analysis

Theorem 4 *Let $A = Q(\mathcal{D}) \in \mathbb{R}^{m \times n}$ be the result of an arbitrary fixed join query Q over an arbitrary fixed multi-relational database $\mathcal{D}$. Let fhtw(Q) [23] be the fractional*

hypertree width of Q. Factorised Eigen-Decomposition computes arbitrary-rank (k) L2-PCA of A given by matrices $Y \in \mathbb{R}^{k \times n}$ and $W \in \mathbb{R}^{k \times n}$, such that $X = AW^T$ and $A \approx XY$ is the best rank-k approximation of A, in $\mathcal{O}(n^3 + n^2|\mathcal{D}|^{fhtw(Q)})$ time.

Proof 2 *The time complexity for eigen-decomposition is $9n^3$ steps (as presented by the interface of the algorithm [21]) and A^TA is computable in $\mathcal{O}(n^2|\mathcal{D}|^{fhtw(Q)})$ by LMFAO [12] (see Proposition 1). Thus, in total we have $\mathcal{O}(n^3 + n^2|\mathcal{D}|^{fhtw(Q)})$.* ■.

Notice here that if A has categorical features, then this is actually ***over-estimating*** *the total complexity as n is replaced by nd (as we assume one-hot encoding) where d is the size of the largest active domain among the categorical features. In practice, LMFAO unleashes the full potential of factorised computation by using a more succinct representation based on FAQ-width [9] and achieves an extra $\mathcal{O}(d)$ time complexity speed-up.*

According to Golub & Van's textbook Matrix Computations [6], the best algorithm for computing SVD: $A = U\Sigma V^T \in \mathbb{R}^{m \times n}$ is $\mathcal{O}(n^3 + n^2m)$, hence our approach can be ***exponentially faster*** *than any SVD approach that depends on the full listing (materialised) representation of $A = Q(\mathcal{D})$, which can have size $m = |\mathcal{D}|^{\rho^*(Q)} \gg |\mathcal{D}|^{fhtw(Q)}$, as the difference between $\rho^*(Q)$ (fractional edge cover) and $fhtw(Q)$ (fractional hypertree width) can be as large as the number of relation literals in Q (see inequality 2.18).*

4.4 Numerical Stability

The numerical stability of FED falls back on the numerical stability of self-adjoint eigen-decomposition classified as *good* by Eigen's catalogue of decompositions [20].

4.5 FAM vs FED

In this case where A^TA is efficiently computed, it turns out that one prefers the closed form solution when: $iterations \times k \geq 9n$ where k is the low rank given as argument to FAM algorithm. Thus, if n is very small (hundreds) and k is very small (tens) then the closed form solution is preferred from a time complexity point of view.

4.6 Chubby $\neq$ Bad

Again, it can sometimes be costly (or impossible) to obtain many data points. Throughout this section, the assumption is that the data set we apply quadratically regularised PCA on is the result of a query Q over a factorised database $\mathcal{D}$. Furthermore, we assume that the result of the query is a **non-materialised** relation which has $m \in \mathbb{N}^*$ tuples (data points) with $n \in \mathbb{N}^*$ attributes with $n \gg m$. We represent the result of the query as the matrix $A \in \mathbb{R}^{m \times n}$.

4.6.1 Closed form solution via Eigen-Decomposition

We take advantage of the fact that we have computed $G = AA^T$ which is symmetric:

$$G \xrightarrow{\text{eigenvalue-eigenvector decomposition}} L\Lambda L^{-1}. \tag{4.10}$$

Hence, by the spectral theorems[B.3] we have:

$$L\Lambda L^{-1} = L\Lambda L^T. \tag{4.11}$$

On the other hand:

$$A = U\Sigma V^T \tag{4.12}$$

$$\iff \quad G = AA^T = U\Sigma V^T V \Sigma U^T \tag{4.13}$$

$$\iff \quad G = U\Sigma^2 U^T \text{ because } V \text{ is orthonormal} \tag{4.14}$$

Therefore, we know that $G = U\Sigma^2 U^T$ for some orthonormal matrix U and some diagonal matrix Σ^2. But we already have means to obtain these kinds of matrices using equation 4.11. Therefore, we can retrieve:

$$U = L \tag{4.15}$$

$$\Sigma = \Lambda^{\frac{1}{2}}. \tag{4.16}$$

For the curious reader, it is always the case that in equation 4.16 we have non-negative eigenvalues for G proof in appendix[B.3.1].

But now we can recover X_{final} and L_{final} produced by alternating minimisation, except in closed form solution, as presented in section[2.3.1]:

$$X_{final} = U_k \Sigma_k'^{\frac{1}{2}} \tag{4.17}$$

$$L_{final} = U_k \Sigma_k'^{-\frac{1}{2}} \tag{4.18}$$

The equation 4.18 comes from the equation 3.37: $Y = L^T A$ with the equation 2.14: $Y = \Sigma_k'^{\frac{1}{2}} V_k^T$, when $A = U_k \Sigma_k V_k^T$. Also, **the k chosen (eigenvector, eigenvalue) pairs will be the top k pairs with the greatest eigenvalues**.

Proof that classic FED algorithm produces the same result (transposed):

- We set $S = AA^T$ instead of $A^T A$. Thus, the eigen-decomposition actually produced U_k and Σ_k instead of V_k and Σ_k.
- Thus, FED sets:

$$Y = \Sigma_k'^{\frac{1}{2}} V_k^T = \Sigma_k'^{\frac{1}{2}} U_k^T = (U_k \Sigma_k'^{\frac{1}{2}})^T = X_{final}^T$$
$$W = \Sigma_k'^{-\frac{1}{2}} V_k^T = \Sigma_k'^{-\frac{1}{2}} U_k^T = (U_k \Sigma_k'^{-\frac{1}{2}})^T = L_{final}^T \blacksquare$$

Therefore we can use the same FED algorithm with AA^T as input and obtain the desired results. $\blacksquare$

5 Implementation

The previous two chapters introduced Factorised Alternating Minimisation (FAM) and Factorised Eigen-Decomposition (FED) algorithms for computing Quadratically Regularised Principal Component Analysis (L2-PCA) model. This chapter discusses the essential design decisions regarding programming FAM and FED. The focus will be on scalability, reliability and speed of the two implementations. The two algorithms are implemented in *C++* programming language, using *Eigen* template library for linear algebra. The programs were written on top of knowledge mainly gathered from the following courses: 1^{st}-Year Linear Algebra, 2^{nd}-Year OOP, 2^{nd}-Year Concurrent Programming, 2^{nd}-Year Compilers, 3^{rd}-Year Computer Architecture and 4^{th}-Year Concurrent Algorithms and Data Structures. The chapter begins by explaining the choice of programming language.

5.1 Why *C++* Programming Language

The main reason why *C++* programming language is chosen is because *C++* is the closest programming language to *Intel 64 Architecture* [26] instruction set, which is used by most of today's processors. Therefore, the code is (almost) translated straight to *Intel 64* instructions. Furthermore, there are many free *C++* compilers available. This thesis uses *g++* from the GNU Compiler Collection (GCC). The main

reason behind it is that it supports state-of-art compilation optimisation flags such as $-O3$ and $Ofast$. Using such a compilation flag will cause the compiler to: use heuristics regarding access patterns (for memory hierarchy), loop unrolling, heuristic predictions and Single Instruction Multiple Data (SIMD) operations, creating little to no redundant assembly instructions (note here that interpreted languages tend to have redundant instructions introduced by the byte code interpreters, redundancies which are non-trivial for a compiler to eliminate).

Among the positive aspects of *C++* programming we find:

- Its remarkable compilation and run-time speed (due to closeness to I-64 Architecture as previously mentioned). Note here that *C* and *Fortran* might achieve faster execution times, however the lack of template libraries tipped the balance towards *C++*.

- Memory management is done by the user. This implies that there are no graphs of references that are being constructed in the background, alongside reachability algorithms which try to detect unreachable memory regions. Thus, the program run-time is smooth and easy to benchmark, focusing again on performance. Note here that we are going to avoid using any kind of smart *C++* pointers.

- LMFAO [12] query engine is written in *C++*.

There are negative aspects of *C++* programming (otherwise all programs would be written in *C++*):

- Memory leaking is easily caused by the lack of a garbage collection system.

- Objects can be passed by reference or by value (as a copy), thus it is easy to write inefficient *C++* code.

- It is easy to create Segmentation Faults by dereferencing NULL or previously deleted (dangling) pointers.

- There are few to no good IDEs, thus command line debugging is common practice among *C++* programmers.

One might prefer to trade the easy difficulty of writing buggy/inefficient code for the chance of writing extremely efficient (and profilable) code.

5.2 Why Eigen for Linear Algebra

Eigen is a *C++* template library for linear algebra algorithms. The fact that it is a template library implies that it is easy to install as it will be inlined and compiled by the compiler at run-time (there are only headers, no *.cpp* files). Thus, it is easy to use regardless of Operating System (OS) or compiler version.

There are quite a few remarkable advantages of using Eigen for linear algebra:

- It supports **dense and sparse** matrix linear algebra,
- Since it is **templatised**, it supports the use of variable data types,
- It uses **Expression Templates** (ET) which allow **lazy evaluation**, and **removal of temporary containers**,
- It allows 64-bit and 80-bit **SIMD optimisations**,
- It automatically spawns **multiple threads** if algorithms support it,
- The algorithms used are carefully selected for **reliability, numerical stability** and memory-time complexity trade-offs are always taken into consideration,
- Google's TensorFlow is built on top of Eigen (this indicates the trust of a major company),
- And nevertheless, it is very **elegant** to code with and has plenty of detailed documentation available [19].

Since no perfect linear algebra library exists, there are a few drawbacks as well:

- It is sometimes difficult to understand whether one should use column major or row major matrices (this can lead to poor performance.)

- Sometimes there is incomplete documentation and undefined behaviour can thus easily leak in the program by incorrectly initialising objects.

5.2.1 Why Dense Matrices

It is the case that LMFAO query engine can compute the matrix $S = A^T A$ where $A = Q(\mathcal{D})$, the result of an arbitrary query Q over a multi-relational database $\mathcal{D}$ when A has categorical features. A compressed version of one-hot encoding will then be used. This means that a sparse representation of $A^T A$ might come in handy. However, we will be using dense matrices and the reason for this is because we often have to take inverses of matrices, and the inverse of a sparse matrix can be dense. See the proof in appendix [B.1.1].

5.3 Matrix Utility Library

Using a templatised class called *MatrixUtil* provides several features. The main feature is that the algorithms implemented as static members of this class can be used with different data types such as: int, float, double, long double, complex. Being tied to a class is a preference as all the relevant linear algebra algorithms alongside their documentations would follow OOP conventions. Furthermore, being a template class, it has the implementation as part of the header. Hence, just as Eigen library, it does not require any pre-compilation and can be used as is.

```
1  template <typename T>
2  class MatrixUtil
3  {
4    private:
5      typedef Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> MatrixT;
6      static constexpr long double EPS = 1e-16;
7      static constexpr long double tau = 1e-5;
8    public:
9  [...]
10 }
```

5.4 Factorised Alternating Minimisation Implementation

See below the interface:

```
template <typename T>
class MatrixUtil
{
[...]
  static void FactorisedAlternatingMinimisation(MatrixT  ATA,
                                                int      rankK,
                                                MatrixT &W,
                                                MatrixT &Y,
                                                int      iterations = -1);
    /**
     * Given the matrix A^T A (i.e. 'ATA'), and a 'rankK', perform Inlined
     * L2-PCA on matrix A. This will produce the eigen matrix 'W' such that
     * X = AW^T and the eigen matrix 'Y' such that XY is the best rank-k
     * approximation of A. If specified, the algorithm will perform 'iterations'
     * otherwise it will iterate until the iteration improvement is below 'tau':
     * if ((Wold - Wt).squaredNorm() < tau &&
     *     (Yold - Yt).squaredNorm() < tau)
     *   done = true;
     * Complexity per iteration: O(N^2 rankK) where N = 'ATA.size()'.
     * Speed: Very fast
     */
[...]
}
```

See below the implementation:

```
template<typename T>
void MatrixUtil<T>::FactorisedAlternatingMinimisation(MatrixT  ATA,
                                                      int      rankK,
                                                      MatrixT &W,
                                                      MatrixT &Y,
                                                      int      iterations)
{
  MatrixT Yt   = MatrixT::Zero(rankK, ATA.rows());
  MatrixT Yold = MatrixT::Zero(rankK, ATA.rows());
  MatrixT Wt   = MatrixT::Zero(rankK, ATA.rows());
  MatrixT Wold = MatrixT::Zero(rankK, ATA.rows());
  MatrixT S = ATA, Zt,Ztaux, Bt,Btaux;

  T traceS = S.trace();
  T avgTr = traceS/(1.0*ATA.rows());
  for (int i = 0; i < rankK; ++i)
    Yt(i,i) = 1;

  MatrixT Ik = MatrixT::Zero(rankK, rankK);
  for (int i = 0; i < rankK; ++i)
    Ik(i,i) = 1;

  MatrixT Ikgamma = MatrixT::Zero(rankK, rankK);
  MatrixT IkgammaPeTr = MatrixT::Zero(rankK, rankK);
  MatrixT SpeTr = S*(1.0/avgTr);

  T gamma = 0.0001;
  Ikgamma = Ik*gamma;
  IkgammaPeTr = Ik*(gamma/avgTr);
```

```
30
31    int currentIteration = 0;
32    bool done = false;
33
34    Eigen::LLT<MatrixT> choleskySolver;
35    while (!done) {
36      ++currentIteration;
37
38
39      choleskySolver.compute(Yt * Yt.transpose() + Ikgamma);
40      Zt = choleskySolver.solve(Ik);
41
42      Wold = Wt;
43      Wt = Zt*Yt;
44
45
46      choleskySolver.compute(Wt*SpeTr*Wt.transpose() + IkgammaPeTr);
47      Bt = choleskySolver.solve(Ik);
48
49      Yold = Yt;
50      Yt = Bt*Wt*SpeTr;
51
52      if ( (Wold - Wt).squaredNorm() < tau &&
53       (Yold - Yt).squaredNorm() < tau)
54        done = true;
55      if (iterations == currentIteration)
56        done = true;
57    }
58    W = Wt;
59    Y = Yt;
60  }
```

The algorithm uses Eigen library Cholesky LL^T matrix factorisation whenever an inverse was computed. The main features of this algorithm are: **very high speed** and **proved numerical stability**. The algorithm has as requirement that the matrix to be Cholesky decomposed is **positive-definite**. The matrices we want to perform Cholesky decomposition on (in order to invert them) are all **positive-definite** (proof in Appendix B.3.1). Furthermore, the complexity analysis matches with the one analysed on the pseudo-code: $\mathcal{O}(n^2k \times iterations + n^2|\mathcal{D}|^{fhtw(Q)})$.

Furthermore, the reason why we work with dense matrices (except the obvious reason of speed) is that the inverse of a sparse matrix can be dense. Thus, it is not worth the slow-down of using sparse matrices at all. Proof of this claim in Appendix B.1.1.

5.5 Factorised Eigen Decomposition Implementation

See below the interface:

```
template <typename T>
class MatrixUtil
{
[...]
  static void FactorisedEigenDecomposition(MatrixT  ATA,
                                           int      rankK,
                                           MatrixT &W,
                                           MatrixT &Y,
                                           double   accuracy = 0);
  /**
   * Given the matrix A^T A (i.e. 'ATA'), and a 'rankK', perform
   * Quadratically Regularised PCA (L2-PCA) on matrix A. This will produce
   * the matrix 'W' such that X = AW^T and matrix 'Y' such that XY is the
   * best rank-k approximation of A. The method will use eigen-decomposition.
   * If 'accuracy' is provided, then the algorithm will use the minimum rank
   * k >= rankK which achieves the requested 'accuracy'. For example
   * rankK = rank(A^T A) if 100% accuracy is required.
   * Complexity: O(N^3) actually about 9N^3 steps.
   * Numerical Stability: Good (evaluated by Eigen)
   */
 [...]
}
```

See below the implementation:

```
template<typename T>
void MatrixUtil<T>::FactorisedEigenDecomposition(MatrixT  ATA,
                                                 int      rankK,
                                                 MatrixT &W,
                                                 MatrixT &Y,
                                                 double   accuracy)
{
  MatrixT L    = MatrixT::Zero(ATA.rows(), ATA.rows());
  MatrixT Diag = MatrixT::Zero(ATA.rows(), ATA.rows());
  VectorT D    = VectorT::Zero(ATA.rows());

  Eigen::SelfAdjointEigenSolver<MatrixT> eigenDecomposer(ATA);

  L = eigenDecomposer.eigenvectors();
  D = eigenDecomposer.eigenvalues();

  T EPS = 1e-8;
  T gamma = 1e-4;

  T sum = 0;
  auto VSTimeStart = Clock::now();
  for (int i = 0; i < ATA.rows(); ++i) {
    Diag(i,i) = max((T)0, sqrt(D(ATA.rows() - i - 1)) - gamma);
    sum += Diag(i,i) * Diag(i,i);
  }

  /**
   *  We will choose the rank which meets the required accuracy.
   */

  T current = 0;
  for (int i = 0; i < ATA.rows(); ++i) {
    current += Diag(i,i)*Diag(i,i);
    rankK = max(rankK, i + 1);
    if (current + EPS >= accuracy * sum)
      break;
  }

  MatrixT V     = MatrixT::Zero(ATA.rows(), rankK);
```

```
40    MatrixT Sh    = MatrixT::Zero(rankK, rankK);
41    MatrixT Shinv = MatrixT::Zero(rankK, rankK);
42
43    for (int i = 0; i < rankK; ++i) {
44      Sh(i,i)    = sqrt(Diag(i,i));
45      Shinv(i,i) = 1.0 / Sh(i,i);
46      V.col(i)   = L.col(ATA.rows() - i - 1);
47    }
48
49    Y = Sh*V.transpose();
50    W = Shinv*V.transpose();
51  }
```

The algorithm uses Eigen library self-adjoint eigen-decomposition which performs the eigen-decomposition of a self-adjoint matrix as follows: it reduces the matrix to tridiagonal form and then the matrix is brought to diagonal form with implicit symmetric QR steps with Wilkinson shift. The main features of this algorithm are: **speed** and **numerical stability**. Details can be found in Section 8.3 of Golub & Van Loan, Matrix Computation [6]. The matrix on which we want to perform eigen-decomposition is self-adjoint (symmetric for real matrices). Furthermore, the overall complexity analysis matches with the one analysed on the pseudo-code: $\mathcal{O}(n^3 + n^2|\mathcal{D}|^{fhtw(Q)})$.

5.6 Parallelism

Note that Eigen library can be set at compile time to use the available CPU Cores in order to run algorithms in parallel (in addition to SIMD operations). During local machine tests, up to 4 times speed-up was achieved for some matrix operations (block matrix multiplications) and 2 times speed-up for computing matrix inverses. More details about these test can be seen in the Test Driven Development (TDD) section 5.7.

5.7 Test Driven Development

The tests have been performed using Google Test platform:

```
1  #include "matrix_engine.h"
2
3  int main(int argc, char **argv) {
4    testing::InitGoogleTest(&argc, argv);
```

```
5   return RUN_ALL_TESTS();
6 }
```

See below part of the tests used while developing. Test Driven Development principles have been successfully used. Tests which failed and then progressively passed were key for checking implementation correctness. Furthermore, both classic and chubby FAM and FED algorithms have been implemented and the theoretical results have been checked. This implied both implementation correctness and empirical proof of the observations regarding chubby matrices.

```
1   TEST(Matrix, FactorisedAlternatingMinimisationTest1) {
2   cout <<setprecision(25) <<fixed;
3   Eigen::MatrixXd A(6,4), ATA, W, Y, X, approx;
4   A << 1, 2, 3, 2,
5        4, 5, 6, 3,
6        1, 2, 8, 1,
7        4, 4, 7, 4,
8        1, 1, 5, 1,
9        1, 2, 3, 4;
10  ATA = A.transpose() * A;
11  MatrixUtil<double>::FactorisedAlternatingMinimisation(ATA, 2, W, Y);
12  cout << "W:\n" << W << endl << endl;
13  cout << "Y:\n" << Y << endl << endl;
14
15  X = A*W.transpose();
16  approx = X * Y;
17  cout << "approx:\n" << approx << endl << endl;
18 }
19
20 TEST(Matrix, FactorisedEigenDecompositionTest1) {
21  cout <<setprecision(25) <<fixed;
22  Eigen::MatrixXd A(6,4), ATA, W, Y, X, approx;
23  A << 1, 2, 3, 2,
24       4, 5, 6, 3,
25       1, 2, 8, 1,
26       4, 4, 7, 4,
27       1, 1, 5, 1,
28       1, 2, 3, 4;
29  ATA = A.transpose() * A;
30  MatrixUtil<double>::FactorisedEigenDecomposition(ATA, 2, W, Y, 1.0);
31  cout << "W:\n" << W << endl << endl;
32  cout << "Y:\n" << Y << endl << endl;
33
34  X = A*W.transpose();
35  approx = X * Y;
36  cout << "approx:\n" << approx << endl << endl;
37  cout << "original:\n" << A << endl << endl;
38 }
39
40
41 TEST(Matrix, MultithreadingTest) {
42  cout << "Number of threads: " << Eigen::nbThreads() << endl;
43  int N = 500;
44  int M = 800;
45  Eigen::MatrixXd A = Eigen::MatrixXd::Zero(M, N);
46
47  Eigen::MatrixXd S = Eigen::MatrixXd::Zero(N, N);
48  Eigen::MatrixXd B = Eigen::MatrixXd::Zero(N, N);
49  Eigen::MatrixXd C = Eigen::MatrixXd::Zero(N, N);
```

```
50    Eigen::MatrixXd Cinv = Eigen::MatrixXd::Zero(N, N);
51
52    Eigen::MatrixXd Ik = Eigen::MatrixXd::Zero(N, N);
53    for (int i = 0; i < N; ++i)
54      Ik(i,i) = 1;
55    for (int i = 0; i < M; ++i)
56      for (int j = 0; j < N; ++j)
57        A(i,j) = 11*(i+1)*(j+1);
58    cout << "Computed A" << endl;
59    S = A.transpose() * A;
60    cout << "Computed covariance matrix" << endl;
61    S = S + Ik; // make it invertible
62    cout << "Computed invertible matrix" << endl;
63    B = S + Ik*5;
64    cout << "Computed second invertible matrix" << endl;
65    auto t1 = Clock::now();
66    C = S * B;  /// 3 times faster when using 4 threads
67    auto t2 = Clock::now();
68
69    std::cout << "Time taken to multiply: "
70          << std::chrono::duration_cast<std::chrono::nanoseconds>(t2 - t1).count()↩
                / 1e6
71          << " miliseconds\n";
72
73              /*************************************/
74              // Testing various inverting mechanisms //
75              /*************************************/
76
77              Eigen::LLT<Eigen::MatrixXd> choleskySolver;
78    Eigen::LDLT<Eigen::MatrixXd> choleskySolver2;
79    Eigen::PartialPivLU<Eigen::MatrixXd> LUSolver;
80    auto t3 = Clock::now();
81    LUSolver.compute(C);
82    Cinv = LUSolver.solve(Ik);
83
84    //Cinv = C.inverse();
85
86    //choleskySolver.compute(C);
87    //Cinv = choleskySolver.solve(Ik);
88
89    //choleskySolver2.compute(C);
90    //Cinv = choleskySolver.solve(Ik);
91
92    auto t4 = Clock::now();
93
94    std::cout << "Time taken to invert: "
95          << std::chrono::duration_cast<std::chrono::nanoseconds>(t4 - t3).count()↩
                / 1e6
96          << " miliseconds\n";
97    cout << "Checking that the inverse is correct" << endl;
98    assert(Cinv*C == Ik);
99    cout << "Done all OK" << endl;
100 }
101
102 TEST(Matrix, FactorisedAlternatingMinimisationTest2) {
103   freopen(CATEGORICAL_MATIN, "r", stdin);
104
105   int N, M;
106   scanf("%d", &N); M = N;
107   Eigen::MatrixXd ATA = Eigen::MatrixXd::Zero(N,M);
108   int i, j;
109   double val = 0;
110   //for (int i = 0; i < N; ++i) {
111   //for (int j = 0; j < M; ++j) {
112   while (scanf("%d%d%lf", &i, &j, &val) == 3)
113       ATA(i,j) = val;
114
115   Eigen::MatrixXd Y1, W1;
116
```

```
117   auto t1 = Clock::now();
118   MatrixUtil<double>::FactorisedAlternatingMinimisation(ATA, 7, W1, Y1);
119   auto t2 = Clock::now();
120   cout << setprecision(30) << fixed;
121   cout << "Matrix W \n";
122   cout << W1 << endl << endl;
123   cout << "Matrix Y \n";
124   cout << Y1 << endl << endl;
125   fclose(stdin);
126 }
127
128 TEST(Matrix, FactorisedEigenDecompositionTest2) {
129   freopen(CATEGORICAL_MATIN, "r", stdin);
130
131   int N, M;
132   scanf("%d", &N); M = N;
133   Eigen::MatrixXd ATA = Eigen::MatrixXd::Zero(N,M);
134   int i, j;
135   double val = 0;
136   while (scanf("%d%d%lf", &i, &j, &val) == 3)
137       ATA(i,j) = val;
138   Eigen::MatrixXd Y1, W1;
139
140   MatrixUtil<double>::FactorisedEigenDecomposition(ATA, 7, W1, Y1);
141
142   cout << setprecision(30) << fixed;
143   cout << "Matrix W \n";
144   cout << W1 << endl << endl;
145   cout << "Matrix Y \n";
146   cout << Y1 << endl << endl;
147   fclose(stdin);
148 }
```

The full testing environment can be found on the GitHub repository of this thesis [29] in the "*tests*" folder. The *MatrixUtil* library can be found in the "include" folder. The repository is ready to be tested and the setup is explained in the "readme.md" file alongside a quick "how to run" guide.

6 Experiments

Chapters 3 and 4 depicted the theory behind Factorised Alternating Minimisation (FAM) and Factorised Eigen-Decomposition (FED) algorithms. Chapter 5 focused on practice: the *C++* implementations of FAM and FED. The current chapter concentrates on benchmarking FAM and FED runtime performance on both real-life and in-house tailored data sets. Both algorithms will calculate the rank-k Quadratically Regularised Principal Component Analysis (L2-PCA) of a complete data matrix $A = Q(\mathcal{D}) \in \mathbb{R}^{m \times n}$ which is the result of a natural join query Q over a multi-relational database $\mathcal{D}$. This will be done in both cases by computing matrices $Y \in \mathbb{R}^{k \times n}$ and $W \in \mathbb{R}^{k \times n}$ such that $X = AW^T$ and $A \approx XY$.

Two types of extensive runtime performance experiments are conducted. The first type of experiment uses the Housing data set and varies the number of tuples m of the result matrix A. The second type of experiment uses the Retailer data set and varies the number of attributes n of the result matrix A. All experiments assume that: 64-bit precision is used, the chosen low-rank is $k = 7$ and regulariser $\gamma = 0.0001$.

6.1 Summary of Findings

- **FAM** and **FED** successfully computed the rank-7 L2-PCA models for all 20 Housing data sets and for all 4 Retailer data sets. The competitor algorithms

(SkLearn/TensorFlow/Torch on top of PostgreSQL) have only managed to compute the rank-7 PCA models for Housing data sets 1 to 6, running out of memory on Housing data sets 7-20 and on all Retailer data sets. FAM and FED were more than 4 orders of magnitude faster than the quickest competitor (Torch).

- When varying the number of tuples m of the query result matrix $A \in \mathbb{R}^{m \times n}$ (using Housing 1-20 data sets), **only the computation of $A^T A$ performed by LMFAO [12] was significantly affected** due to its dependence on the data set size $|\mathcal{D}|$. The execution of FED stayed the same as n is fixed throughout the Housing data sets. The execution of FAM differed only slightly in the number of iterations due to the different actual values of $A^T A$.

- When varying the number of attributes n of the query result matrix $A \in \mathbb{R}^{m \times n}$, **FAM became more efficient than FED**. FAM is **quadratic** in n (and linear in the chosen low rank and iterations) while FED is **cubic** in n. A cubic function grows faster than a quadratic function, thus it is indeed expected of FED to become slower than FAM as n increases.

- The numerical precision of FAM algorithm has been tested on the Housing and Retailer data sets. Empirical tests provided evidence that factoring out a constant α from $A^T A$ in the iterative algorithm played to our advantage for the Housing data sets. The main reason is that FAM used smaller intermediary results which are more accurately represented [28].

6.2 Data Sets

6.2.1 Housing Data Set

Housing data set ($\mathcal{D}_{\text{housing}}$) has the relational schema presented in Figure 6.1. There are 20 versions of this data set, from housing-1 up to housing-20, each of increasing number of rows m of the query result $A = Q_{\text{housing}}(\mathcal{D}_{\text{housing}}) \in \mathbb{R}^{m \times n}$. Q_{housing} is the natural join of all the relations of the housing data set. All the data sets have

the same number of columns $n = 28$. Housing data set was artificially created and has only continuous numerical values. The table below illustrates the sizes of the data matrix A given different versions of the data set:

#housing	1	4	6	7	14	20
rows	25000	1600000	5400000	14700000	102900000	400000000
columns	28	28	28	28	28	28
size (GiB)	0.006	0.417	1.126	3.836	26.856	104.4

The matrix A can be materialised using the query Q_{housing} as follows:

```
CREATE TABLE joinres AS (
  SELECT *
  FROM Shop NATURAL JOIN House        NATURAL JOIN Restaurant
            NATURAL JOIN Demographics NATURAL JOIN Institution
            NATURAL JOIN Transport
);
```

where *House, Demographics, Transport, Restaurant, Shop and Institution* are relations which differ with the size of the data set (from Housing-1 to Housing-20).

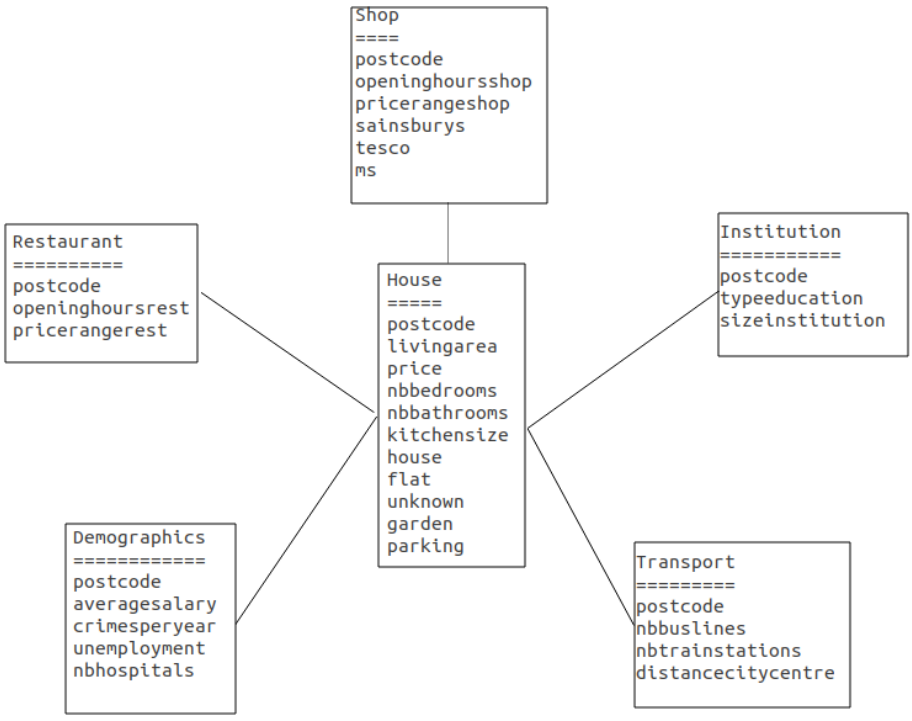

Figure 6.1: Structure of the Relational Database for Housing data set. This structure is **completely thrown away** by today's usual workflow which creates a data matrix out of it.

6.2.2 Retailer Data Set

Retailer data set ($\mathcal{D}_{\text{retailer}}$) has the relational schema presented in Figure 6.2. There are 4 versions of this data set, from retailer-1 up to retailer-4, each of increasing number of columns n of the query result $A = Q_{\text{retailer}}(\mathcal{D}_{\text{retailer}}) \in \mathbb{R}^{m \times n}$. All the data sets have the same number of rows $m = 84055800$. Retailer data set is used in real life by a large US Retailer for forecasting user demands and sales. The table below illustrates the sizes of the data matrix A given the versions of the data set:

#retailer	1	2	3	4
rows	84055800	84055800	84055800	84055800
columns	33	158	3804	6433
size (GiB)	20.666	98.949	2382.310	4028.759

The matrix A can be materialised using the query Q_{retailer} as follows:

```
CREATE TABLE joinres AS (
  SELECT *
  FROM Inventory NATURAL JOIN Location NATURAL JOIN Census
                 NATURAL JOIN Item     NATURAL JOIN Weather
);
```

where *Inventory, Location, Census, Item, Weather* are relations which differ with the size of the data set (from Housing-1 to Housing-20).

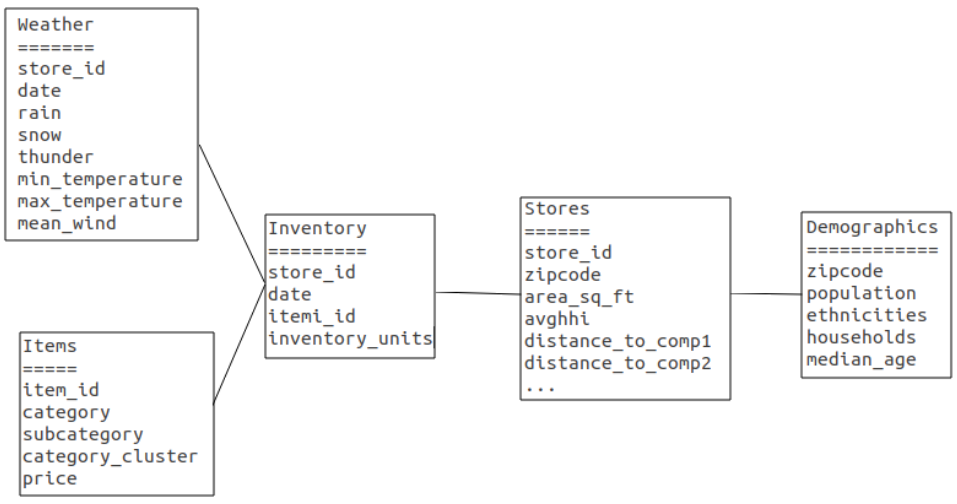

Figure 6.2: Structure of the Relational Database for Retailer data set. This structure is **completely thrown away** by today's usual workflow which creates a data matrix out of it.

6.3 Varying the Number of Rows in A

The first experiment involves varying the number of rows in the result data matrix. This is done by using the Housing data set. To simplify the analysis, we split the computation of the low rank model in two phases.

The first phase consists of the computation of the data matrix in case of SkLearn/TensorFlow/Torch, or the computation of the covariance matrix $A^T A$ in case of FAM and FED.

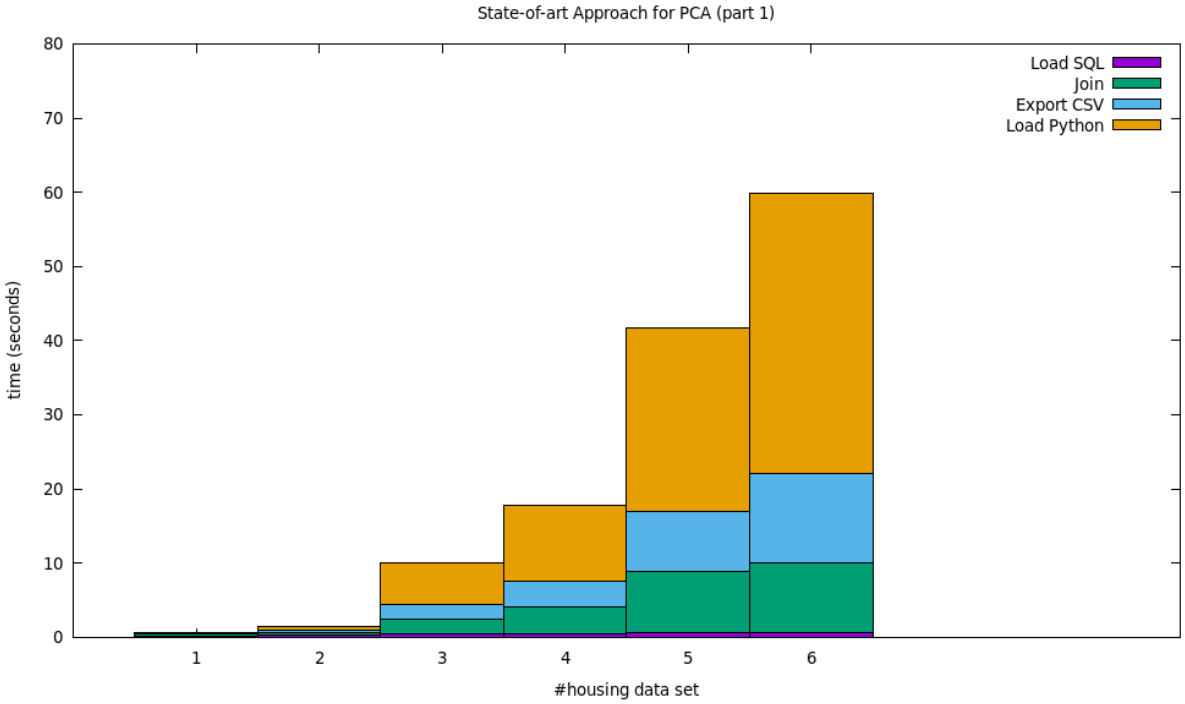

Figure 6.3: The materialisation of the data matrix, required by the prior state-of-art approaches.

As Figures 6.3 and 6.4 illustrate, there is already a time gap of three orders of magnitude between the precomputation required by the current state of art algorithms for PCA implemented by SkLearn/TensorFlow/Torch (which require A materialised) compared to FAM and FED which only require $A^T A$. As presented in Figure 6.3 to compute A one needs to load the database $\mathcal{D}$ into PostgreSQL, compute the natural join query $Q(\mathcal{D})$ and export the data in a CSV format. The data then would get loaded into Python such that it is ready to be processed by

state-of-art PCA libraries. PostgreSQL ran out of 12GiB available RAM when computing Housing 7-20 and Retailer data sets.

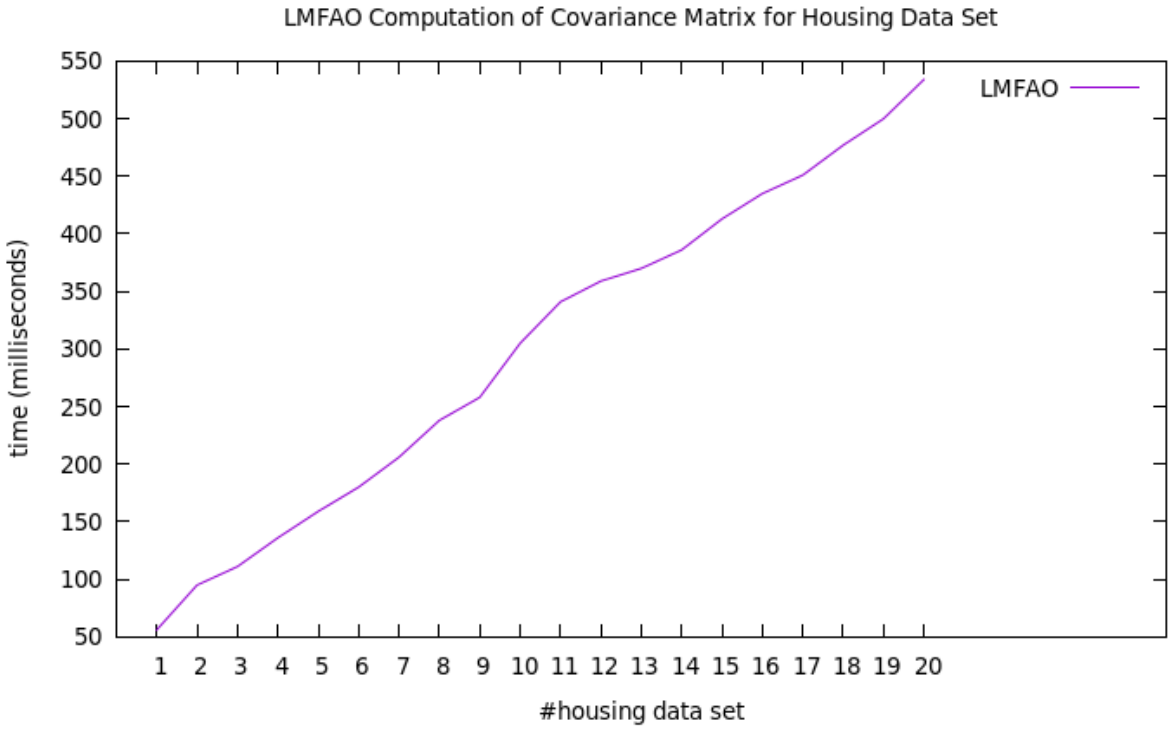

Figure 6.4: The computation of the covariance matrix, required by FAM and FED.

Note here that the size of A^TA can be exponentially smaller than the size of A. Thus, LMFAO [12] successfully computed A^TA for all Housing data sets (Figure 6.4). We can already see a time gap of more than three orders of magnitude between the precomputation required by the current state-of-art approach and the precomputation required by the approach proposed by this thesis.

The second phase involves the actual model computation task. This requires A to be computed for the current state-of-art approaches and A^TA to be computed for this thesis's proposed approach.

As it can be seen in Figure 6.5, between the state-of-art approaches, Torch is the fastest at computing the rank-7 PCA model. Again, because of the 12GiB main memory constraint, only data sets housing-1 up to housing-6 could be processed. One more observation is that the time it took to compute the *join* is similar to the time it took to compute the model using Torch.

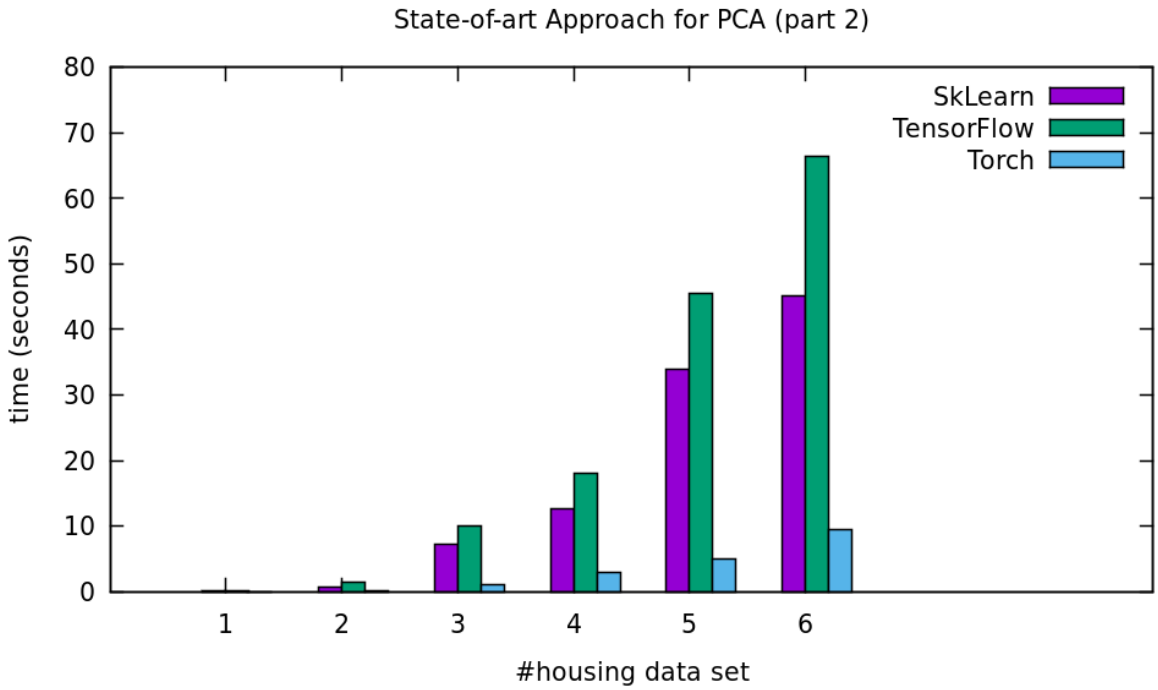

Figure 6.5: The computation of the simpler rank-7 PCA model by SkLearn/TensorFlow/Torch state-of-art libraries.

Figure 6.6 shows FAM and FED's performance when computing rank-7 L2-PCA over the housing data sets, contrasted with the performance of the main competitor (Torch) on the first 6 versions of the housing data sets. We see another four orders of magnitude gap between factorised computation approach (FAM and FED) and the main competitor, Torch.

Another key insight given by Figure 6.6 is that both FAM and FED's run times do not depend on the number of rows (m) of the data matrix $A \in \mathbb{R}^{m \times n}$. Hence, the run time is constant throughout the housing data sets. Furthermore, FAM is slower than FED by almost two orders of magnitude on housing data sets. This is as expected since the number of columns of A (n) is only 28 and FED depends solely on this. See Chapters 3 and 4 for formal complexity results.

Careful inspection of Figures 6.6 and 6.4 show that FAM and FED are actually orders of magnitude faster than LMFAO's computation of $A^T A$. Thus, the overall run time of the approach proposed by this thesis is spent computing the covariance matrix.

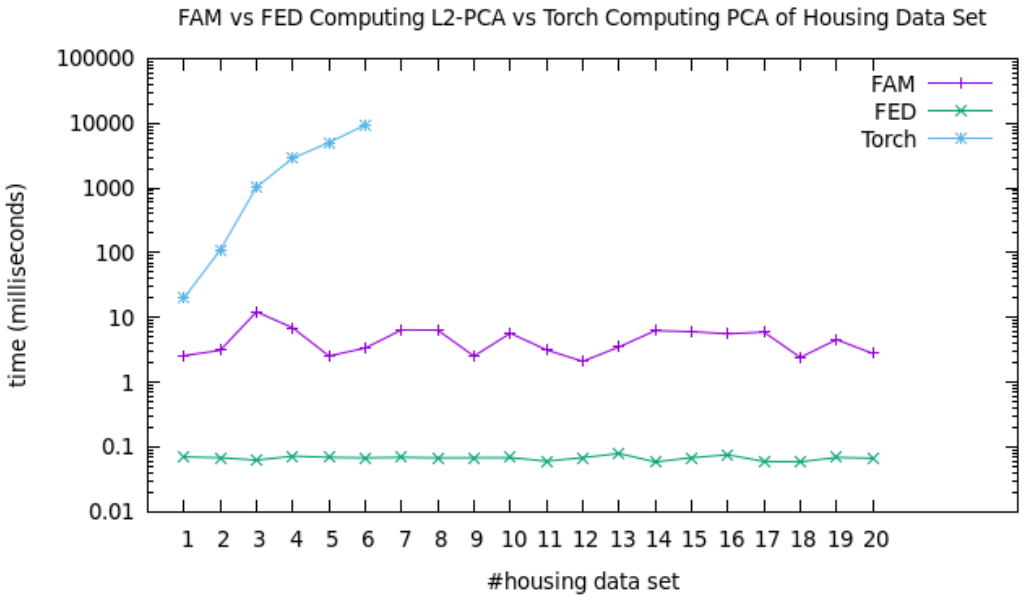

Figure 6.6: The competition factorised (FAM and FED) vs state-of-art (Torch) on learning rank-7 L2-PCA, respectively rank-7 PCA.

6.4 Varying the Number of Columns in A

The second experiment involves varying the number of columns in the result data matrix. This is done by using the Retailer data set which has categorical features. To simplify the analysis, the low rank model computation is split in two phases.

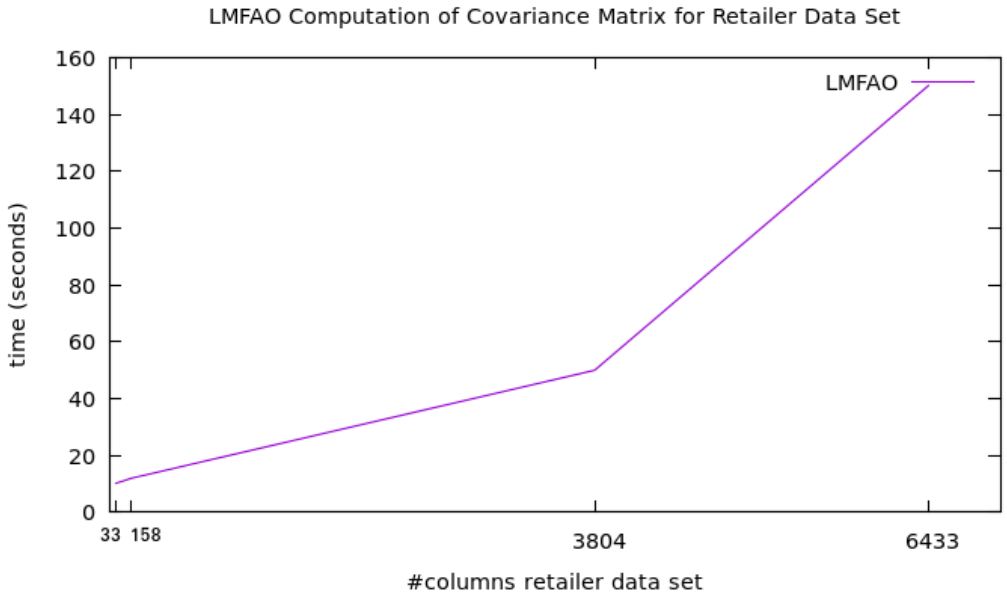

Figure 6.7: The computation of the covariance matrix, required by FAM and FED.

The first phase is the computation of the covariance matrix using LMFAO (see Figure 6.7). Due to 12GiB main memory constraint, none of the Retailer data set versions fit in the main memory for the state-of-art approaches to be tested. The full power of LMFAO is unleashed when computing the covariance matrix for the 4^{th} Retailer data set of size 84055800×6433 which materialised would occupy about 3.93 TiB representation space, while the covariance matrix is of size 6433×6433 which is only 315.73 MiB. Recall (Section 2.4) that the time complexities presented in Chapters 3 and 4 are actually over estimating by a factor of d where d is the size of the largest categorical active domain. This is because LMFAO uses a more succinct encoding than the presumed one-hot, which depends on the FAQ-width [9].

The second phase is performing the actual learning task. FAM and FED compete in learning rank-7 L2-PCA over the four Retailer data sets.

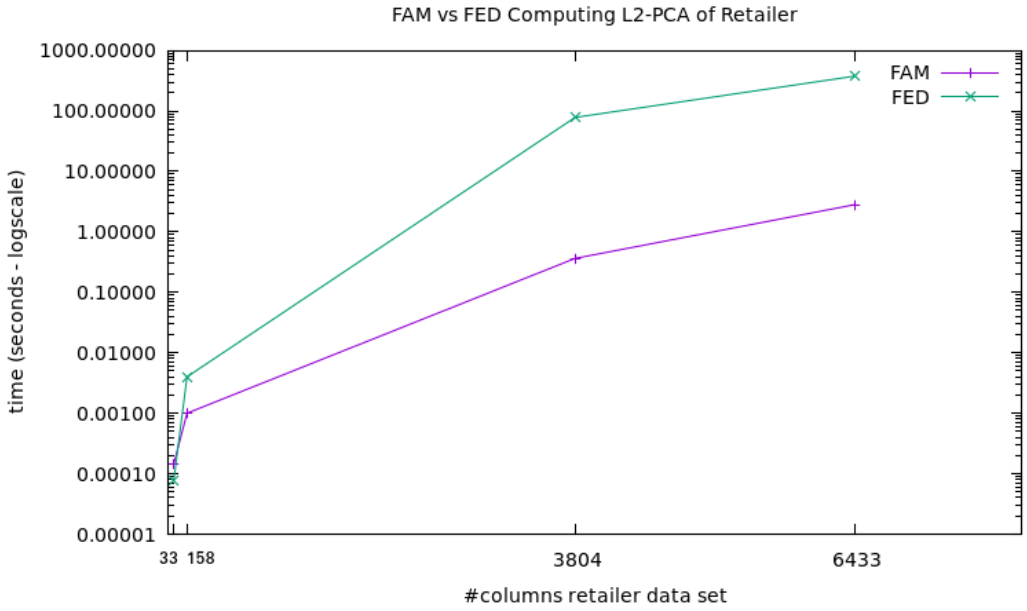

Figure 6.8: The factorised competition FAM vs FED on learning rank-7 L2-PCA over Retailer data set.

Analysing Figure 6.8 we immediately see that FAM is orders of magnitude faster than FED for retailer-2 up to and including retailer-4 data sets. In case of FAM, the dominant running time is spent on phase one (the computation of the covariance matrix) while in the case of FED, the time it took to learn arbitrary rank L2-PCA is proportional to LMFAO's covariance matrix computation (see Figure 6.7).

Note the fact that since there are infinitely many solutions to the minimisation problem (see Section 2.2.2), FAM and FED will not produce identical low-rank models.

6.5 Numerical Precision

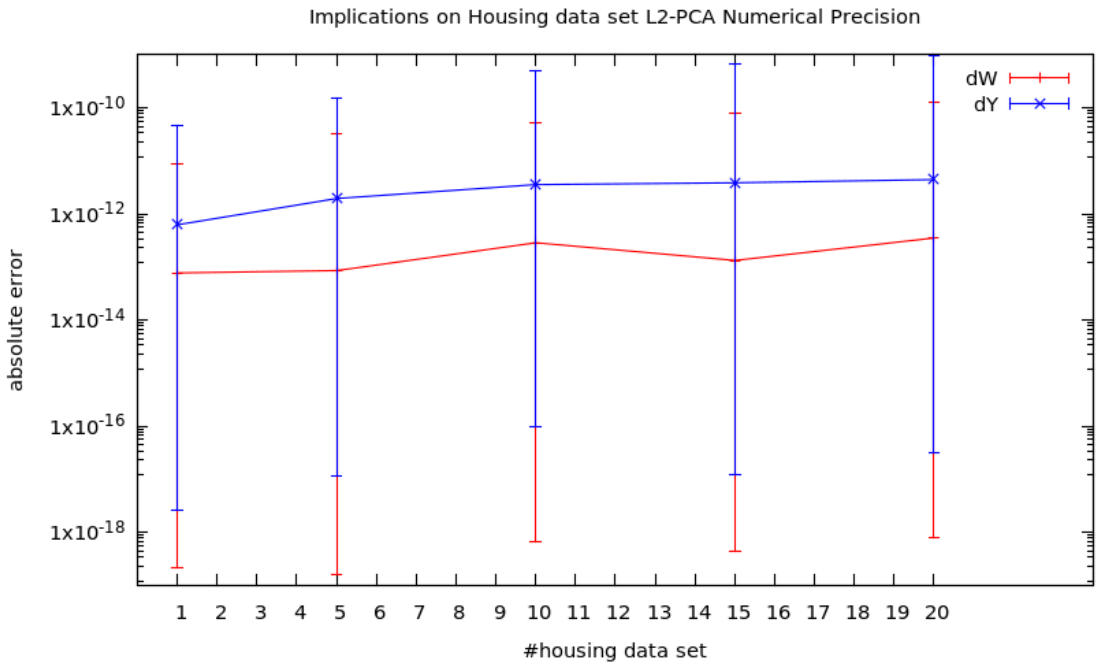

Figure 6.9: The average, minimum and maximum entry of ∂W and ∂Y for rank-7 L2-PCA of Housing data sets.

Chapter 3 proposed a numerical precision optimisation in Equation 3.20. The following experiment will provide some empirical evidence towards why Equation 3.20 benefits FAM algorithm. Both versions of FAM, the one which factors α out of S and the version which does not, are executed and (W_α, Y_α) respectively (W, Y) denote the resulting rank-7 models. Let ∂W and ∂Y be the entry-wise absolute value difference: $|W_\alpha - W|$ and $|Y_\alpha - Y|$.

Figure 6.9 shows the average, minimum and maximum entries of ∂W and ∂Y matrices for housing data sets $\{1, 5, 10, 15, 20\}$. Notice the fact the maximum and average absolute difference between the two versions of FAM grows as the entries in the covariance matrix become bigger. A close inspection of the matrix contents

showed that the entries of Y and W would become larger than the entries of Y_α and W_α, intuitively indicating a correlation with the fact that non scaled covariance matrix has large entries. The minimum differences seem random.

6.6 Covariance Matrix - numerical precision

Numerical precision tests have been conducted to compare the usage of LMFAO with the naive approach for computing the covariance matrix. The naive approach was feasible to test on the Housing data sets 1 up to 10. For the Housing data sets 11 up to 20 and the Retailer data sets 1 up to 4 there was not sufficient main memory available (recall the 12GiB constraint).

The results showed that for Housing data sets 1 up to 10, the values involved in calculations were integers in the range $[2^0, 2^{53})$ which as discussed in appendix C.1 can be fully represented by 64-bit *double* datatype in *C++*. Thus, regardless of the approach, the calculations would not include any rounding error.

However, for sufficiently large data sets, we expect LMFAO to be significantly more numerically accurate as it performs exponentially fewer floating point additions and multiplications thus exponentially fewer rounding errors are expected to occur.

7 Related Work

7.1 Factorised Learning

Previous research in Machine Learning (ML) model training using factorised computation paradigm covers a few of the classic ML algorithms. Let a data matrix be defined as the result of a natural join query over a multi-relational database. The following algorithms have been considered:

[17] considers Linear Regression models. The focus is on computing the least squares regression models over a non-materialised data matrix. This work is tangential to this thesis as Factorised Alternating Minimisation (FAM) computes in parallel (using matrix operations) Ridge Regression models for each iteration step (one regression per column of the data matrix) in a factorised fashion.

[9] shows an approach to optimise a general objective function which includes Principal Component Analysis (PCA). This proposes an alternative approach to the two presented in this thesis, an approach that uses Batch Gradient Descent (BGD) with Armijo line search rather than closed form solution or iterative approximations.

[10] focuses on functional aggregate queries (FAQs) and explaining how the following models can be learnt in a factorised fashion: Robust Linear Regression with Huber loss, linear support vector machine (SVM) and K-means clustering.

7.2 Factorised Linear Algebra

Particular attention has been offered to linear algebra performed over data matrices as previously defined:

[16] focuses on the Gram-Schmidt algorithm which produces the QR factorisation of the data matrix. This can be used effectively to learn Linear Regression, Cholesky Decomposition, Moore-Penrose pseudo-inverse and Singular Value Decomposition and thus it can be used to compute generalised low rank models in a factorised fashion.

[11] introduces MorpheusPy, a tool which uses manual re-writings of some linear algebra operations in a factorised computation fashion to optimise linear algebra on top of multi-relational databases using NumPy. This approach has, however, limited applicability given by the 'supported' re-written linear algebra operations. However, it can compute Linear Regression, Logistic Regression, K-means and Generalised Non-negative Matrix Factorisation.

7.3 Non-factorised Linear Algebra

There are competitors worthy of being mentioned even though they do not use the factorised computation paradigm:

MADlib supports a Low-Rank Matrix Factorisation [13] procedure and Singular Value Decomposition [14]. MADlib uses high performance mathematics libraries with special aggregate functions [15]. It successfully computed the Singular Value Decomposition of a truncated version of the Retailer data set for a subset of up to 100 one-hot encoded columns.

8 Conclusion

8.1 Summary

This thesis' goal was to present different approaches for computing generalised low rank models over multi-relational databases using the factorised computation paradigm. In particular, it focused on Quadratically Regularised Principal Component Analysis (L2-PCA) and showed two successful approaches: Factorised Alternating Minimisation (FAM) and Factorised Eigen-Decomposition (FED). The computational complexity analysis revealed a time complexity gap between the factorised approach (FAM and FED) and the current state-of-art approaches which materialise the data matrix. As a result, FAM and FED could be used to compute L2-PCA over multi-relational data sets orders of magnitude larger than what the current state-of-art can compute, without requiring extra computational power. The data matrices used were complete matrices produced by natural join queries, in order to check the run time performance. However, as previously mentioned, FAM and FED can be also executed on matrices with missing values. The time complexity speed-up brings new opportunities. One opportunity is that models can be kept up-to-date more often which directly affects the accuracy. Another implied opportunity is that more models can be computed within the time budget needed by

the competing systems to compute one model. This also directly affects the accuracy of the final model which can be a combination of generalised low rank models.

8.2 Future Work

If I had more time, I would have liked to investigate the Machine Learning (ML) accuracy of the models produced by FAM and FED. Furthermore, I would have analysed how the runtime performance can be used to improve the accuracy of ML models which use low rank models as inputs.

A direct application of L2-PCA is the prediction of missing values in matrices. In order to test runtime performance, our queries over *Housing* and *Retailer* data sets were *natural joins*, which produced complete data matrices. A direction of the thesis could be a data imputation application that can recover an unknown matrix $A \in \mathbb{R}^{m \times n}$ of rank k from as little as $\Omega(nk \log n)$ known entries [7].

Another direction in which I would have liked to delve into, given more time, is the numerical stability of FAM and FED algorithms. This is because speed and correctness of an algorithm are essential, as is numerical stability. Not to be confused with over-fitting, numerical errors can easily crop up and alter a low rank model to a state where its predictions are simply incorrect.

A fascinating direction which covers an even wider range of the generalised low rank models is to use gradient descent to minimise the objective function. This is particularly powerful as it does not require closed form solutions to exist and can be applied to L2-PCA as well.

These could be possible starting points for my future DPhil/PhD/Research work.

Appendices

A

Quadratically Regularised PCA

This chapter covers key proofs regarding the alternating minimisation algorithm for computing the Quadratically Regularised PCA.

A.1 Continuous Mathematics

Matrix calculus will be used in order to simplify some of the calculations. The following formulae build on top of the 1^{st}-Year Continuous Mathematics course. Throughout this section only (Section A.1), the bold lower case letters will refer to vectors which can be either in column or in row matrix form.

A.1.1 Matrix Calculus

Assuming $\mathbf{w} \in \mathbb{R}^N$, $f_i : \mathbb{R}^N \mapsto \mathbb{R}$ for $i \in \{1, 2, \ldots, M\}$, and $\mathbf{f} : \mathbb{R}^N \mapsto \mathbb{R}^M$, $\mathbf{f}(\mathbf{w}) = \Big(f_1(\mathbf{w}), \ldots, f_M(\mathbf{w})\Big)$ the following notation conventions are used:

$$\frac{\partial f_i(\mathbf{w})}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial f_i(\mathbf{w})}{\partial w_1} \\ \vdots \\ \frac{\partial f_i(\mathbf{w})}{\partial w_N} \end{bmatrix}, \tag{A.1}$$

$$\frac{\partial \mathbf{f}(\mathbf{w})}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial f_1(\mathbf{w})}{\partial \mathbf{w}} & \ldots & \frac{\partial f_M(\mathbf{w})}{\partial \mathbf{w}} \end{bmatrix}. \tag{A.2}$$

The following formulas will be used to simplify calculations. Their usage is easy but their proofs can be tedious:

$$\frac{\partial(X\mathbf{w})}{\partial\mathbf{w}} = X^T,\ for\ X \in \mathbb{R}^{m\times n}\ and\ \mathbf{w} \in \mathbb{R}^{n\times 1} \tag{A.3}$$

$$\frac{\partial(\mathbf{w}^T X)}{\partial\mathbf{w}} = X,\ for\ X \in \mathbb{R}^{m\times n}\ and\ \mathbf{w} \in \mathbb{R}^{m\times 1} \tag{A.4}$$

$$\frac{\partial(\mathbf{w}^T X\mathbf{w})}{\partial\mathbf{w}} = X^T\mathbf{w} + X\mathbf{w},\ for\ X \in \mathbb{R}^{n\times n}\ and\ \mathbf{w} \in \mathbb{R}^{n\times 1} \tag{A.5}$$

For more details, see Chapter 4.1.3.1 of Murphy's textbook: *Machine Learning: A Probabilistic Perspective* [2].

A.2 Alternating Minimisation Derivation

This section focuses on deriving the solution to Alternating Minimisation algorithm. Quadratically Regularised PCA was formulated as:

$$minimise \quad \|A - XY\|_F^2 + \gamma\|X\|_F^2 + \gamma\|Y\|_F^2, \tag{A.6}$$

where $X \in \mathbb{R}^{m\times k}$ and $Y \in \mathbb{R}^{k\times n}$.

Assume X is fixed and that we optimise Y. We rewrite the objective as:

$$\|A - XY\|_F^2 + \gamma\|X\|_F^2 + \gamma\|Y\|_F^2 = \gamma\|X\|_F^2 + \sum_{i=1}^{n}(\|\mathbf{a}_i - X\mathbf{y}_i\|_2^2 + \gamma\|\mathbf{y}_i\|_2^2), \tag{A.7}$$

where the vectors $\mathbf{a}_i$ and $\mathbf{y}_i$ with $i \in \{1,\ldots,n\}$ are the columns of A and Y. Since this is a sum of quadratic functions in $\mathbf{y}_i$, each of the terms is minimised (independent of the others) when the partial derivative with respect to each $\mathbf{y}_i$ is zero. We rewrite each term of the sum in matrix form:

$$\|\mathbf{a}_i - X\mathbf{y}_i\|_2^2 + \gamma\|\mathbf{y}_i\|_2^2 = (\mathbf{a}_i - X\mathbf{y}_i)^T(\mathbf{a}_i - X\mathbf{y}_i) + \gamma\mathbf{y}_i^T\mathbf{y}_i, \tag{A.8}$$

$$= \mathbf{a}_i^T\mathbf{a}_i - 2\mathbf{y}_i^T X^T\mathbf{a}_i + \mathbf{y}_i^T X^T X\mathbf{y}_i + \gamma\mathbf{y}_i^T\mathbf{y}_i, \tag{A.9}$$

and now take the partial derivative $\frac{\partial}{\partial\mathbf{y}_i}$ of the objective function A.6 for each $i \in \{1,\ldots,n\}$:

$$\frac{\partial}{\partial\mathbf{y}_i}(-2\mathbf{y}_i^T X^T\mathbf{a}_i + \mathbf{y}_i^T X^T X\mathbf{y}_i + \gamma\mathbf{y}_i^T\mathbf{y}_i) = -2X^T\mathbf{a}_i + 2X^T X\mathbf{y}_i + 2\gamma\mathbf{y}_i; \tag{A.10}$$

setting it to zero, gives (for each i):

$$\begin{aligned} 0 &= -2X^T\mathbf{a}_i + 2X^TX\mathbf{y}_i + 2\gamma\mathbf{y}_i, && \text{(A.11)} \\ \mathbf{y}_i &= (X^TX + \gamma I_k)^{-1}X^T\mathbf{a}_i. && \text{(A.12)} \end{aligned}$$

But now, the puzzle can be put together: let $\hat{W} = (X^TX + \gamma I_k)^{-1}X^T$. Hence:

$$\begin{aligned} Y &= \begin{bmatrix}\mathbf{y}_1 & \dots & \mathbf{y}_n\end{bmatrix}, && \text{(A.13)} \\ &= \begin{bmatrix}\hat{W}\mathbf{a}_1 & \dots & \hat{W}\mathbf{a}_n\end{bmatrix}, && \text{(A.14)} \\ &= \hat{W}A, && \text{(A.15)} \\ &= (X^TX + \gamma I_k)^{-1}X^TA. && \text{(A.16)} \end{aligned}$$

This is the closed form solution for the matrix $Y \in \mathbb{R}^{k\times n}$ which minimises the objective function A.6 when matrix $X \in \mathbb{R}^{m\times k}$ is fixed.

In order to find the matrix X which minimises the objective function when the matrix Y is fixed, we will use the property of Frobenius norm: $\|A\|_F = \|A^T\|_F$ and rewrite the objective as:

$$\|A - XY\|_F^2 + \gamma\|X\|_F^2 + \gamma\|Y\|_F^2 = \|A^T - Y^TX^T\|_F^2 + \gamma\|X^T\|_F^2 + \gamma\|Y^T\|_F^2. \quad \text{(A.17)}$$

We apply now the exact same approach as before, and obtain (mutatis mutandis):

$$\begin{aligned} X^T &= ((Y^T)^TY^T + \gamma I_k)^{-1}(Y^T)^TA^T, && \text{(A.18)} \\ &= (YY^T + \gamma I_k)^{-1}YA^T. && \text{(A.19)} \end{aligned}$$

Therefore,

$$\begin{aligned} X &= ((YY^T + \gamma I_k)^{-1}YA^T)^T, && \text{(A.20)} \\ &= AY^T(YY^T + \gamma I_k)^{-1}. && \text{(A.21)} \end{aligned}$$

The iterative algorithm will finally be given by the following:

$$\begin{aligned} X_{t+1} &= AY_t^T(Y_tY_t^T + \gamma I_k)^{-1}, && \text{(A.22)} \\ Y_{t+1} &= (X_{t+1}^TX_{t+1} + \gamma I_k)^{-1}X_{t+1}^TA. \ \blacksquare && \text{(A.23)} \end{aligned}$$

The proof however, depends on the inverse $(X^TX + \gamma I_k)^{-1}, \gamma > 0$ existing for any matrix $X \in \mathbb{R}^{m\times n}$ proof of this fact can be found in Appendix [B.3.2].

A.3 Alternating Minimisation Convergence

Recall the objective function:

$$F(X,Y) \quad = \quad \|A - XY\|_F^2 + \gamma\|X\|_F^2 + \gamma\|Y\|_F^2, \gamma > 0. \tag{A.24}$$

One can fix X and rewrite it as a sum of convex functions in $(\mathbf{y}_i)$:

$$F(X,Y) \quad = \quad \gamma\sum_{i=1}^{k}\|\mathbf{x}_i\|_2^2 + \sum_{i=1}^{n}(\|\mathbf{a}_i - X\mathbf{y}_i\|_2^2 + \gamma\|\mathbf{y}_i\|_2^2), \gamma > 0, \tag{A.25}$$

symmetrically, one can rewrite it as a sum of convex functions in $(\mathbf{x}_i)$. We know that the square function is convex, and by the definition of a convex function, a sum of convex functions is trivially convex. Therefore, by taking the derivatives with respect to $\mathbf{y}_i$ and setting them simultaneously to zero (as we previously did), we find the Y (and symmetrically the X) which is the global minimum of the objective function F when X (and symmetrically Y) is fixed. This means that with every step, we decrease the value of the objective function.

The objective function is bounded below by 0 as it is a sum of squares. Also, at every step we decrease the objective function, therefore by Weierstrass's Theorem for Convergence, the function converges. This is not a powerful enough result, as we claim that the algorithm converges to the global optimum not to a local optimum. The full proof is not trivial and can be found in Appendix A.1.2 of Generalised Low Rank Models paper by M. Udell [7]. ■

B
Linear Algebra

B.1 Inverse of a Matrix

Often, calculating the inverse of a matrix is required. More about positive definite matrices can be found in Strang's textbook [1].

B.1.1 Sparse Matrices

Theorem 5 *The inverse of a sparse matrix is not necessarily sparse.*

Proof 3 *Take matrix $A_n \in \mathbb{R}^{n\times n}$ to be the bi-diagonal matrix which has ones on the principal diagonal and on the next diagonal above:*

$$A_n = \begin{bmatrix} 1 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & 1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 & 1 \end{bmatrix}$$

Observe that the ratio of nonzero entries in A_n tends to zero for large n:

$$\lim_{n\to\infty} \frac{n+(n-1)}{n^2} = 0. \tag{B.1}$$

thus, A_n is sparse. However, its inverse A_n^{-1}:

$$A_n^{-1} = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i \leq j \text{ and } j-i \text{ even} \\ -1 & \text{if } i \leq j \text{ and } j-i \text{ odd} \end{cases}$$

has a non zero ratio for large n:

$$\lim_{n\to\infty} \frac{n^2 - \binom{n}{2}}{n^2} = \frac{1}{2}. \blacksquare \tag{B.2}$$

B.2 Frobenius Norm

B.2.1 Submultiplicativity

Proof of submultiplicativity of Frobenius Norm ($\|AB\|_F^2 \leq \|A\|_F^2 \|B\|_F^2$):

Assume $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times k}$ arbitrary fixed:

$$\begin{aligned}
\|AB\|_F^2 &= \sum_{i=1}^{m}\sum_{j=1}^{k}(A_{i,:}B_{:,j})^2 && \text{(B.3)}\\
&= \sum_{i=1}^{m}\sum_{j=1}^{k}|A_{i,:}B_{:,j}|^2 && \text{(B.4)}\\
(CBS) &\leq (\sum_{i=1}^{m}\|A_{i,:})\|_2^2)(\sum_{j=1}^{k}\|B_{:,j}\|_2^2) && \text{(B.5)}\\
&= \|A\|_F^2\|B\|_F^2 && \text{(B.6)}
\end{aligned}$$

CBS is the remarkable Cauchy-Buniakovsky-Schwartz inequality. ■

B.3 Spectral Theorems

Spectral Theorems(Gilbert Strang[1] page 318 chapter 6.4):

- The eigenvalues of a real symmetric matrix are real and the eigenvectors of a real symmetric matrix are real.
- Eigenvectors of a real symmetric matrix (when they correspond to different λ's) are always perpendicular.
- Every symmetric matrix S has a complete set of orthogonal eigenvectors. Thus, $S = L\Lambda L^{-1}$ becomes $S = L\Lambda L^T$.
- All symmetric matrices are diagonalizable - there are always enough eigenvectors to diagonalize $S = S^T$.

B.3.1 X^TX-like Matrix Properties

Let $X \in \mathbb{R}^{m\times n}$ arbitrary fixed. This subsection will analyse some properties of a symmetric matrix $X^TX \in \mathbb{R}^{n\times n}$ and corollary, the same properties will follow for the symmetric matrix $XX^T \in \mathbb{R}^{m\times m}$ by replacing X with X^T as proofs remain unchanged.

Theorem 6 *X^TX (corollary - XX^T) is positive semi-definite.*

Proof 4 *Let $\boldsymbol{x} \in \mathbb{R}^{n\times 1}$ be an arbitrary fixed column vector:*

$$\boldsymbol{x}^T(A^TA)\boldsymbol{x} \;=\; (A\boldsymbol{x})^T(A\boldsymbol{x}) = \|A\boldsymbol{x}\|_2^2 \;\geq\; 0 \;\blacksquare \tag{B.7}$$

Theorem 7 *X^TX (corollary - XX^T) has non-negative eigenvalues.*

Proof 5 *Let (λ, v) be an arbitrary fixed eigenvalue-eigenvector pair of X^TX. Using the result above, we have:*

$$\boldsymbol{v}^T(X^TX)\boldsymbol{v} \geq 0 \tag{B.8}$$

$$\iff \quad \lambda\boldsymbol{v}^T\boldsymbol{v} \geq 0 \tag{B.9}$$

$$\iff \quad \lambda\|\boldsymbol{v}\|_2^2 \geq 0 \tag{B.10}$$

But by the spectral theorem $\boldsymbol{v}$ is a real eigenvector, thus $\|\boldsymbol{v}\|_2^2 \geq 0$. But then λ has to be non-negative.

B.3.2 $(X^TX + \gamma I)$-like Matrix Properties

Recall $X \in \mathbb{R}^{m\times n}$ arbitrary fixed.

Theorem 8 *If $\gamma > 0$ then $X^TX + \gamma I_n$ is invertible (corollary for $XX^T + \gamma I_m$).*

Proof 6 *We now check what happens to the eigenvalues of X^TX (which we already know that are non negative) when we add γI_n with $\gamma > 0$:*

$$(X^TX + \gamma I_n)\boldsymbol{v}_i \;=\; (X^TX)\boldsymbol{v}_i + \gamma\boldsymbol{v}_i \tag{B.11}$$

$$\;=\; \lambda_i\boldsymbol{v}_i + \gamma\boldsymbol{v}_i \tag{B.12}$$

$$\;=\; (\lambda_i + \gamma)\boldsymbol{v}_i. \tag{B.13}$$

The eigenvalues shifted by $\gamma > 0$. Thus the eigenvalues are all strictly positive which means, $X^TX + \gamma I_n$ is invertible (corollary, $XX^T + \gamma I_m$ is invertible). $\blacksquare$

C
Numerical Stability

C.1 Floating Point Standard

C.1.1 64-bit Precision (*double*)

Figure C.1 represents the IEEE754 [28] standard convention on how 64-bit precision floating point numbers (*double* in *C++*) are represented.

Figure C.1: 64-bit floating point number representation.

The formula for a 64-bit floating point (normal) number n given by $s/e/f$ bit pattern where s is bit 63 and f contains bits 51 to 0 (from left to right) is:

$$n := (-1)^s(1.f)2^{e-b} \tag{C.1}$$

where $b = 1023$ is the bias. Observe now the following inequality:

$$1.f \leq 2 \ \forall f \in \{0,1\}^{52}.$$

Let us fix $s = 0$. Then immediately by equation C.1 $n \geq 0$. Furthermore, for all e such that $e-b \leq -1 \iff e \leq b-1 \iff e \in \{0, \ldots, 1022\}$ we get $n < 1$. But then:

$$n \in [0, 1) \; \forall f \in \{0, 1\}^{52} \wedge \forall e \in \{1, \ldots, 1022\}$$

which means that $n \in [0, 1)$ for $2^{52} \times (2^{10} - 2) = 2^{62} - 2^{53}$ combinations of bit patterns. For $e = 0$ there are another 2^{52} numbers which are *denormalized*, all between $[0, 1)$. Thus, there are $2^{62} - 2^{52}$ numbers in $[0, 1)$. By symmetry, we get another $2^{62} - 2^{52}$ numbers in $(-1, 0]$. Therefore, there are about $2^{63} - 2^{53}$ numbers in $(-1, 1)$ out of all the 2^{64} possible bit patterns. Thus, approximately $\frac{1}{2} - \frac{1}{2048}$ of the numbers that can be represented using *double C++* data type, are in the interval $(-1, 1)$, i.e. about 50%. The same calculation can be done for *long double* data type and still 50% is obtained. This gives strong evidence that working with small numbers (in absolute value) is essential to avoid introducing rounding errors.

The gap between x and $succ(x)$ for $x \in [2^{52}, 2^{53})$ is $2^{52} \times 2^{-52} = 1$. This is one way of defining $\epsilon_{\text{machine}} = 2^{-52} \approx 10^{-16}$ which represents the machine precision when using *double* data type. It is also the distance $|1 - succ(1)|$. Note that this implies that for $x \approx 2^{84}$ the gap $|x - succ(x)| \approx 2^{32}$. Hence, one rounding when working with large x can have catastrophic effect on the result.

Bibliography

[1] Strang, Gilbert: *Introduction to Linear Algebra*, 3^{rd} edn.
(Cambridge: Massachusetts Institute of Technology, 2003)

[2] Murphy, Kevin P.: *Machine Learning: A Probabilistic Perspective*
(Cambridge: Massachusetts Institute of Technology, 2012)

[3] C. Eckart, G. Young: *The Approximation of One Matrix by Another of Lower Rank* (University of Chicago, Chicago, Illinois, Psychometrika - VOL. 1, No 3, September, 1936)

[4] J. Worrell, *Machine Learning - Michaelmas Term 2017 - Lectures 17, 18 : Principal Component Analysis* (https://www.cs.ox.ac.uk/teaching/materials17-18/ml/lectures/lecture17.pdf)

[5] C. Haase, V. Kanade, *Machine Learning - Michaelmas Term 2017 - Lecture 6: Regularization* (https://www.cs.ox.ac.uk/teaching/materials17-18/ml/lectures/lecture06.pdf)

[6] G. H. Golub, C. F. Van Loan: *Matrix Computations Third Edition*
(The John Hopkins University Press, 1996)

[7] Udell, Madeline et al.: *Generalized Low Rank Models*
(Foundations and Trends in Machine Learning, Vol.9, No. 1 (2016))

[8] D. Olteanu, M. Schleich: *Factorised Databases*
(SIGMOD Record (Database Principles Column), vol. 45, no. 2, June 2016)

[9] M. A. Khamis, H. Ngo, X. Nguyen, D. Olteanu, and M. Schleich: *Learning Models over Relational Data using Sparse Tensors and Functional Dependencies* (ACM Principles of Database Systems (PODS), Houston, June 2018; arXiv report 1703.04780, March 2017)

[10] M. Abo Khamis, R. Curtin, B. Moseley, H. Ngo, X. Nguyen, D. Olteanu and M. Schleich. *On Functional Aggregate Queries with Additive Inequalities* (In ACM Principles of Database Systems (PODS), Amsterdam, July 2019. Extended version in arXiv report 1812.09526, December 2018).

[11] Side Li, Arun Kumar *MorpheusPy: Factorised Machine Learning with NumPy* (https://adalabucsd.github.io/papers/TR_2018_MorpheusPy.pdf)

[12] M. Schleich, D. Olteanu, M. A. Khamis, H. Q. Ngo and X. Nguyen: *A Layered Aggregate Engine for Analytics Workloads* (In 2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3324961)

[13] Apache MADlib Low-Rank Matrix Factorisation (https://madlib.apache.org/docs/v1.15.1/group___grp___lmf.html)

[14] Apache MADlib Singular Value Decomposition (https://madlib.apache.org/docs/v1.15.1/group___grp___svd.html)

[15] N. Srebro and T. Jaakkola. *Weighted Low-Rank Approximations.* (In: ICML. Ed. by T. Fawcett and N. Mishra. AAAI Press, 2003, pp. 720–727. isbn: 1-57735-189-4).

[16] Bas A.M. van Geffen, *QR Decomposition of Normalised Relational Data* (A dissertation submitted for the degree of: Mater of Science in Computer Science Trinity 2018)

[17] M. Schleich, D. Olteanu, and R. Ciucanu: *Learning linear regression models over factorised joins* (SIGMOD, pages 3-18, 2016)

[18] M. A. Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich: *In-database learning with sparse tensors.* (CoRR, abs/1703.04780, 2017.)

[19] http://eigen.tuxfamily.org/

[20] https://eigen.tuxfamily.org/dox/group___TopicLinearAlgebraDecompositions.html

[21] *compute()* method complexity (https://eigen.tuxfamily.org/dox/classEigen_1_1SelfAdjointEigenSolver.html)

[22] A. Atserias, M. Grohe, and D. Marx: *Size bounds and query plans for relational joins.* (SIAM J. Comput., 42(4):1737-1767,2013.)

[23] D. Olteanu, J. Zavodny: *Size bounds for factorised representations of query results.* (TODS, 40(1):2, 2015)

[24] *The State of Data Science & Machine Learning 2017, Kaggle, October 2017* (based on 2017 Kaggle survey of 16,000 ML practitioners)

[25] *Kaggle. 201. Kaggle ML & DB Survey, Kaggle, 2018* (https://www.kaggle.com/kaggle/kaggle-survey-2018)

[26] *Intel 64 and IA-32 Architecture Software Developer's Manual* (https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf)

[27] (J. Widom - 2^{nd}-year Database Course 2016, lecture 1, p. 17)

[28] IEEE, *IEEE 754-2008: IEEE Standard for Floating-Point Arithmetic* (Publisher: IEEE (2008), ISBN-10: 0738157538, ISBN-13: 978-0738157535)

[29] *This thesis' github repository* (https://github.com/gabrielinelus/LinearAlgebra)

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

[31] H. Q. Ngo, E. Porat, C. Re, and A. Rudra. *Worst-case optimal join algorithms* (In PODS pages 37–48, 2012)