UNIVERSITY OF FRIBOURG

BACHELOR THESIS

Empirical Comparison of Incremental Matrix Decomposition Techniques

Zakhar Tymchenko

from Ukraine

September 2017

SUPERVISORS	Dr. Mourad Khayati
	Prof. Dr. Philippe Cudré-Mauroux
RESEARCH GROUP	University of Fribourg
	Department of Informatics
	eXascale Infolab (XI)

Abstract

Data analysis often relies on matrix decomposition techniques, such as Centroid Decomposition, or Singular Value Decomposition. These techniques have been used extensively to solve batch time series tasks, e.g., recovery of missing values, noise reduction, compression, etc. The application of these techniques to streaming data requires efficient incremental decomposition algorithms. Moreover, most streaming implementations of SVD and CD (and their variants) are in different high-level programming languages and, to the best of our knowledge, there does not exist any direct empirical comparison between them.

In this thesis, we describe a few selected matrix decomposition techniques, provide a C++ implementation for them and conduct an empirical comparative analysis of those techniques using real-world data sets containing up to 1 million of values. More specifically, we compare different incremental CD implementations and compare them against a variant of incremental SVD. We show that, thanks to the caching procedure, the incremental CD algorithm outperforms all other variants of incremental SVD in performance on both rank-1 and rank-k incremental updates.

Chapter 1

Introduction

1.1 Context

Matrix decomposition techniques are extremely powerful tools and have been applied in various domains, such as the data analysis of time series. Decomposition techniques have different computational time and memory consumption, yielding the need to analyze them in order to assign each technique to its specific task. In this thesis, we are interested in the comparison of two matrix decomposition techniques: the Centroid Decomposition and the Singular Value Decomposition.

Centroid Decomposition (CD) [1] has been successfully applied to recover missing values in time series (cf. ReVival tool [2]). CD gives statistical information about a decomposed matrix, which is leveraged to perform the analysis task at hand. CD factorizes input matrix into a loading matrix and relevance matrix. Every element of the input matrix is a linear combination of loading values weighed by their relevance.

The Singular Value Decomposition (SVD) [3] technique is one of the most powerful matrix decomposition techniques that has been used to perform various data analysis tasks. SVD factorizes input matrix into two orthogonal matrices and a diagonal matrix containing singular values. Orthogonal matrices conjugate input to obtain a diagonal one. For instance, SVD has been used for the recovery of missing values, e.g., GROUSE technique [4], REBOM [5], etc.

Data analysis has been on the rise in recent years due to increase in the amount of generated data and the capabilities to process it. Streaming data, i.e. when the original time series is updated from time to time, is an important part of this process since data analysis is not only needed as a retrospect, but also in real-time processing. The extensive use of streaming processing, calls for the need of online decomposition techniques that can achieve real-time responsiveness.

Thus, we are interested in exploring different incremental implementations of the CD and SVD techniques. Most of the existing implementations of CD and SVD, both batch and incremental variants, are in different programming languages, making it difficult to conduct a direct empirical comparison between them. Moreover, the existing implementations are also using high level languages creating an overhead of automatic memory management and particularities of managed languages (virtual machine, just-in-time compiler, garbage collection etc.) and safety runtime checks, like array bounds checks. All of the previous reasons greatly affect the performance of algorithms that rely on heavy matrix computations.

1.2 Contributions

The main contributions of this thesis are to i) describe the properties of the main matrix decomposition techniques, ii) provide low-level implementations for batch and incremental variants of the Singular Value Decomposition and the Centroid Decomposition techniques in C++ and iii) evaluate all the implemented algorithms on real-world streams of time series. In this thesis, we are interested only on full decomposition techniques since the approximated techniques are less accurate then the full ones yielding a worse data analysis.

All the studied algorithms, unless mentioned otherwise, are implemented from scratch based on their specifications or implementations in other programming languages. Such algorithms are tested with static and dynamic code analysis and different data to ensure correctness. In order to have a good precision, all the computations and storage of matrix elements are in double-precision floating-point numbers.

The empirical evaluation of the performance of the algorithms implemented in C++ is performed on real-world data to measure time complexity. Additionally, the Centroid Decomposition technique is analyzed from the point of view of its sign vector search algorithm.

1.3 Outline

Chapter 2 introduces the notations used in this thesis. Chapter 3 studies the Centroid Decomposition technique and describes available algorithms for batch and incremental options. Chapter 4 studies available algorithms for Singular Value Decomposition and focuses on an incremental technique. Chapter 5 presents the evaluation of the algorithms using real-world data sets to study the behavior of the algorithms with different data. Chapter 6 finalizes the results and concludes the thesis.

Chapter 2

Notations

In this Chapter, we introduce all the notations used throughout this thesis.

Bold uppercase letters are matrices, e.g., $\mathbf{X}$. Normal uppercase letters are vectors, e.g., Z, matrix rows and columns are normal font uppercase letters with corresponding indices (X_{i*} for rows, X_{*i} for columns). Matrix elements are normal font lowercase letters, e.g., x_{ij}. A transpose of a matrix $\mathbf{X}$ is denoted as $\mathbf{X}^T$. All vectors use the column representation.

Vector norm $||V||$ is the Euclidean Norm of $V = \{v_1, v_2, \cdots, v_n\}$ defined as $||V||_2 := \sqrt{\sum_1^n v_i^2}$. Multiplications that involve scalars are denoted using the symbol $\times$, otherwise the symbol is $\cdot$.

In this thesis, we are interested in the application of matrix decomposition techniques to long streams of time series. Thus, we assume as an input a rectangular matrix $\mathbf{X}$ of n rows and m columns where the number of rows is much bigger than the number of columns, i.e., $n \gg m$ and contain only real numbers ($\mathbf{X} \in \mathbb{R}^{n \times m}$, $n, m \in \mathbb{N}$).

Chapter 3

Centroid Decomposition

In this Chapter, we describe the properties of CD and discuss different batch and incremental implementations of this technique.

Definition 1 (Centroid Decomposition) *Let* $\mathbf{X} \in \mathbb{R}^{n \times m}$ *be an input matrix, then CD(*$\mathbf{X}$*)* $:= \{ \mathbf{L}, \mathbf{R} \}$ *such that* $\mathbf{L} \cdot \mathbf{R}^T = \mathbf{X}$ *and where:*

- $\mathbf{L} \in \mathbb{R}^{n \times m}$ *a matrix of loading values*
- $\mathbf{R} \in \mathbb{R}^{m \times m}$ *a matrix of relevance values*

This decomposition computes centroid values, loading vectors (matrix $\mathbf{L}$) and relevance vectors (matrix $\mathbf{R}$) to approximate respectively eigenvalues, left singular vectors and right singular vectors of $\mathbf{X}$.

3.1 Batch Centroid Decomposition

3.1.1 Batch Algorithms

There exists a number of algorithm variants to perform a batch Centroid Decomposition of an input matrix [6]. The most efficient algorithm to compute CD is called Scalable Sign Vector (SSV) [1]. In this thesis, we use the SSV algorithm to implement the batch CD. The SSV algorithm computes the decomposition by iterating over the number of column in the matrix (m times) and performing at each iteration a two-step process. First, the algorithm determines a sign vector $Z \in \{+1.0, -1.0\}^n$ which maximizes $||\mathbf{X}^T \cdot Z||$. Then, it computes the corresponding loading and relevance vectors which form $\mathbf{L}$ and $\mathbf{R}$ matrices.

To compute sign vectors, the algorithm uses the equivalence of maximization problem of $||\mathbf{X}^T \cdot Z||$ with the maximization of $Z^T \cdot V$, where V (a weight vector) is defined as $V := \text{diag}^{=0}(\mathbf{X} \cdot \mathbf{X}^T) \cdot Z$ and $\text{diag}^{=0}(\mathbf{A}) := \mathbf{A}$, s.t. $a_{ii} = 0 \, \forall i \in \{1 \ldots n\}$.

Scalable Sign Vector algorithm avoids explicitly creating $\mathbf{X} \cdot \mathbf{X}^T$ matrix which has the size of $n \times n$. $V = diag^{=0}(\mathbf{X} \cdot \mathbf{X}^T) \cdot Z$ is equivalent to the following formula: $v_i = z_i \times (z_i \times X_{i*} \cdot Z^T \cdot \mathbf{X} - < X_{i*}, X_{i*} >)$ where $Z^T \cdot \mathbf{X}$ can be stored in a single vector S. If the sign of a weight vector element v_i doesn't match the sign of $\_i$, we have to flip the sign of z_i and calculate the new weight vector.

Example 1 (Scalable Sign Vector) *Let $\mathbf{X}$ be an input matrix and Z_0 be a starting sign vector:*

$$\mathbf{X} = \begin{bmatrix} -1 & 4 \\ 2 & -3 \\ 3 & 0 \end{bmatrix}, Z_0 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \textit{then corresponding } V = \begin{bmatrix} -17 \\ -8 \\ 3 \end{bmatrix}.$$

Next, the sign in Z on the position $i = 1$ is flipped, because $v_i \times z_i < 0$ and among all the negative values it has the highest absolute value.

$$\textit{With the new } Z_1 = \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix} \textit{ new weight vector becomes } V = \begin{bmatrix} -17 \\ 20 \\ 9 \end{bmatrix}$$

Now $\forall i = 1...3$ we have $v_i \times z_i > 0$, so Z_1 is the optimal sign vector.

When the signs of the weight vector and the sign vector match, this ensures maximization of $Z^T \cdot V$, which is equivalent to maximizing $||X^T \cdot Z||$.

The SSV algorithm was implemented from scratch according to the aforementioned paper. In addition to using SSV as a base case for tests, an improvement to the first stage (finding maximizing sign vector) at every iteration was applied - incremental weight vector computation [7]. The computation of the incremental weight vectors improves the efficiency by avoiding the recalculation of weight vector from scratch while keeping the same memory complexity. In the rest of the thesis we refer to Batch decomposition (batchCD) as the decomposition based on the SSV algorithm using either the iterative or the incremental weight vectors.

Example 2 (Centroid Decomposition) *First step of the decomposition at iteration $m = 1$ is to find the maximizing sign vector.*

$$\textit{Let } \mathbf{X} = \begin{bmatrix} -1 & 4 \\ 2 & -3 \\ 3 & 0 \end{bmatrix}, \textit{then maximizing sign vector is } Z = \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix}$$

Now it is possible to calculate R_{*1} and L_{*1}

$$R_{*1} = \frac{\mathbf{X}^T.Z}{||\mathbf{X}^T.Z||} = \begin{bmatrix} 0.65 \\ -0.76 \end{bmatrix} \textit{ and then } L_{*1} = \mathbf{X} \cdot R_{*i} = \begin{bmatrix} -3.69 \\ 3.58 \\ 1.95 \end{bmatrix}$$

Performing the same process for $m = 2$ and $\mathbf{X} = \mathbf{X} - L_{*1}^T \cdot R_{*1}$ yields:

$$\mathbf{L} = \begin{bmatrix} -3.69 & 1.84 \\ 3.58 & -0.43 \\ 1.95 & 2.28 \end{bmatrix} \quad and \quad \mathbf{R} = \begin{bmatrix} 0.65 & 0.76 \\ -0.76 & 0.65 \end{bmatrix}$$

3.1.2 Complexity

The runtime complexity of this algorithm is quadratic, as was shown in the cited paper. This stems from the fact that one iteration of sign vector search is O(n) and the number of sign vector iterations is at most n, on average $\frac{n}{2}$ per column.

The highest memory usage of the SSV algorithm occurs during the search of the sign vectors. The data structures loaded into the heap are i) the input matrix $\mathbf{X}$ of size $n \times m$, ii) current sign vector Z of size n, iii) the weight vector V of size n and iv) vector S of size m. In addition, we store a list of n scalar products $< \mathbf{X}_i*, \mathbf{X}_i* >$ which are needed more than once. The summation of all the data structures gives a total size of $nm + 3n + m$.

Incremental sign vector search can avoid simultaneous storage of S and Z, but vector Z is best allocated first since it is re-used after the search to reduce heap fragmentation. Thus, there is no difference in memory usage of iterative and incremental sign vector search algorithms.

3.2 Incremental Centroid Decomposition

We are going to explore two variants of incremental CD:

- Updating CD (updateCD), a method which stems from an incremental technique originally developed for SVD [8] and then adapted to Centroid-based algorithm;
- Cached CD (cachedCD), a method which relies on re-using the old sign vectors to calculate new decomposition.

3.2.1 Updating CD

Algorithm Description

Updating CD (updateCD) is an update technique, which is an adaptation of an incremental SVD technique described in [9] to Centroid method. The algorithm adds new information to $\mathbf{L}_0$ matrix ($\mathbf{L}$ matrix of old decomposition) to form an update matrix $\mathbf{S}$ and recalculate its Centroid Decomposition.

This method exploits one of the properties of CD - that matrix $\mathbf{L}$ is the stationary point of decomposition, i.e. CD($\mathbf{L}$) = { $\mathbf{L}$, $\mathbf{I}$ }. After CD($\mathbf{S}$) = { $\mathbf{L}_S, \mathbf{R}_S$ } is calculated, the new $\mathbf{L}$ of the decomposition is copied from the $\mathbf{L}_S$ and new $\mathbf{R}$ is a result of multiplication of $\mathbf{R}_S$ and $\mathbf{R}_0$.

The algorithm considers the possibility of new data increasing the rank of the matrix, making the rank-k update to the decomposition more complicated. This problem is partially avoided if we assume that the rank of the matrix is always maximal (i.e., $m = r$), which should never be a problem for long time series. While there is no analytic solution given for the case when the rank is not increased by new information and we have a rank-k update with $k \geq 2$, the empirical tests show that applying naive approach produces correct result. We replace a vector of new information by a matrix and adapt all data structures that follow from it. Then, we append a transformed matrix to $\mathbf{L}_0$ in the same way as if it was a rank-1 update.

The basis of this implementation for both rank-1 and rank-k algorithms is the Matlab implementation for updateCD [10].

Complexity

The updateCD has to run a full batch Centroid Decomposition, which makes it at least as slow as a batch decomposition. Additional complexity from updating process is limited to a few primitive matrix operations which are at most linear on n, since there are no matrix multiplications of $n \times n$ matrices. In case a rank-k update is applied, the amount of added rows just mimics the complexity of n.

The algorithm requires to keep in memory the modified input matrix $\mathbf{L}$, which has size of $(n + k) \times m$ and the input matrix $\mathbf{R}$ of $m \times m$ size. Thus, the total required memory is $m^2 + (nm + km) + (3n + 3k + m)$. It is not required to keep augmented $\mathbf{L}$, so Step 2 of batchCD can overwrite $\mathbf{X}$ when performing $\mathbf{X} = \mathbf{X} - L_{*1}^T \cdot R_{*1}$. This is why input matrix for batch CD doesn't count into total memory complexity of updateCD.

Example 3 (Updating CD) *We now expand on Example 2. Let* $\mathbf{L}_0$ *and* $\mathbf{R}_0$ *be Centroid Decomposition of X:*

$$\mathbf{L}_0 = \begin{bmatrix} -3.69 & 1.84 \\ 3.58 & -0.43 \\ 1.95 & 2.28 \end{bmatrix} \textit{ and } \mathbf{R}_0 = \begin{bmatrix} 0.65 & 0.76 \\ -0.76 & 0.65 \end{bmatrix}$$

Suppose we add a new row $B = \begin{bmatrix} -2 & 1 \end{bmatrix}^T$ *to* $\mathbf{X}$. *We have to check whether this update increases the rank of decomposition.*

$$N = \mathbf{R}_0^T \cdot B = \begin{bmatrix} -2.06 \\ -0.86 \end{bmatrix}$$

$$Q = B - R_0 \cdot N = \begin{bmatrix} -2 \\ 1 \end{bmatrix} - \begin{bmatrix} -2 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Since new information B *doesn't affect the rank of* $\mathbf{L}_0$ *we proceed to augment* $\mathbf{L}_0$ *with* N*:*

$$\mathbf{S} := \begin{bmatrix} L \\ N \end{bmatrix} = \begin{bmatrix} -3.69 & 1.84 \\ 3.58 & -0.43 \\ 1.95 & 2.28 \\ -2.06 & -0.86 \end{bmatrix}$$

Then running $\mathbf{S}$ *through CD gives us* $\mathbf{L}_S$ *and* $\mathbf{R}_S$ *as follows:*

$$\mathbf{L}_1 = \mathbf{L}_S = \begin{bmatrix} 3.54 & 2.12 \\ -3.54 & -0.70 \\ -2.12 & 2.12 \\ 2.12 & -0.70 \end{bmatrix} \textit{ and } \mathbf{R}_S = \begin{bmatrix} -1.00 & -0.08 \\ -0.08 & -1.00 \end{bmatrix}$$

And finally $\mathbf{R}_1 = \mathbf{R}_0 \cdot \mathbf{R}_S = \begin{bmatrix} -0.71 & 0.71 \\ 0.71 & 0.71 \end{bmatrix}$

3.2.2 Cached CD

Algorithm Description

The incremental updates to the decomposition can also be obtained by caching of the sign vectors Z of each column of the matrix that we obtain during the application of batch decomposition of $\mathbf{X}$ [11]. This is equivalent to running the normal CD algorithm on the new matrix with appended rows and recalculating everything again, except this time using the old Z_i vectors (padded with $+1.0$ to match the new row dimension of $\mathbf{X}$) as a starting point, instead of $\{+1.0\}^n$ as the algorithm would normally do.

Complexity

The Cached CD algorithm performs the same calculations as the batchCD, but since our starting sign vectors are supposed to be closer to the correct ones, we do a very low number of sign vector iterations, since cached Z is "closer" (in terms of Hamming distance) to the optimal Z, which was studied in detail in [11]. When we perform rank-k update, this distance increases depending on the amount of added rows. Since the average number of iterations of batchCD is $\frac{n}{2}$ per column [1], the number of iterations for batchCD for rank-k update is equal $\frac{n+k}{2} = \frac{n}{2} + \frac{k}{2}$. But, since the first n rows are already calculated during the previous decomposition, then calculating corresponding elements of sign vector is not needed anymore and the algorithm will perform $\frac{k}{2}$ iterations for the k added rows. With $\frac{k}{2}$ iterations per column, the whole update is $\frac{k \times m}{2}$ iterations (cf. Section 5.2.3).

This algorithm has to hold in memory all the sign vectors for m columns. All dimensions that use n are also increased by k for rank-k update. Sign vectors add additional $(n + k)(m - 1)$ elements. Thus, the total of elements that need to be stored in memory is $(2nm + 2km) + (2n + 2k + m)$.

Example 4 (Cached CD) *We now repeat the same* $\mathbf{X}$ *and* B *as in Example 3.*

Let $\mathbf{X} = \begin{bmatrix} -1 & 4 \\ 2 & -3 \\ 3 & 0 \end{bmatrix}$*, and cached sign vector for* $m = 1$ *be* $Z = \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix}$

Suppose we add a new row of $B = \begin{bmatrix} -2 & 1 \end{bmatrix}^T$ *to* $\mathbf{X}$ *and append 1.0 to Z. Then going through the SSV process yields us the following sequence of sign vectors:*

$$Z_0 = \begin{bmatrix} -1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, V = \begin{bmatrix} -11 \\ 13 \\ 3 \\ -19 \end{bmatrix} \longrightarrow Z_1 = \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix}, V = \begin{bmatrix} -23 \\ 27 \\ 15 \\ -19 \end{bmatrix}$$

And it leaves Z_1 *as optimal sign vector. Repeating the process for* $m = 2$ *yields the updated decomposition:*

$$\mathbf{L} = \begin{bmatrix} -3.54 & 2.12 \\ 3.54 & -0.71 \\ 2.12 & 2.12 \\ -2.12 & -0.71 \end{bmatrix} \textit{ and } \mathbf{R} = \begin{bmatrix} 0.71 & 0.71 \\ -0.71 & 0.71 \end{bmatrix}$$

Example 3 gives a result which is different, Centroid Decomposition is unique by signs, but not absolutely unique, since $||\mathbf{X}^T \cdot Z|| = ||\mathbf{X}^T \cdot (-1) \times Z||$.

Chapter 4

Singular Value Decomposition

In this Chapter, we describe the properties of SVD and discuss different batch and incremental implementations of this technique.

Definition 2 (Singular Value Decomposition) *Let* $\mathbf{X} \in \mathbb{R}^{n\times m}$ *be a matrix, then* $SVD(\mathbf{X}) = \{\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}\}$*, such that* $\mathbf{X} = \mathbf{U} \cdot \mathbf{\Sigma} \cdot \mathbf{V}^T$ *and where:*

- $\mathbf{U} \in \mathbb{R}^{n\times n}$ *is an orthogonal matrix containing left-singular vectors of* $\mathbf{X}$*. These vectors represent orthonormal Eigenvectors of* $\mathbf{X}{\cdot}\mathbf{X}^T$
- $\mathbf{V} \in \mathbb{R}^{m\times m}$ *is an orthogonal matrix containing right-singular vectors of* $\mathbf{X}$*. These vectors represent orthonormal Eigenvectors of* $\mathbf{X}^T{\cdot}\mathbf{X}$
- $\mathbf{\Sigma} \in \mathbb{R}^{n\times m}$ *is a matrix with singular values of* $\mathbf{X}$ *on the main diagonal and zeroes elsewhere. The singular values represent a set of square roots of eigenvalues of* $\mathbf{X}{\cdot}\mathbf{X}^T$*.*

There exists a variant of SVD called Truncated SVD (or Thin SVD, TSVD) for matrices where $n > m$.

Definition 3 (Truncated SVD) *Let* $\mathbf{X} \in \mathbb{R}^{n\times m}$ *be an input matrix and SVD(*$\mathbf{X}$*) =* $\{\ \mathbf{U}, \mathbf{\Sigma}, \mathbf{V}\ \}$ *is a decomposition of* $\mathbf{X}$*. Then, TSVD(*$\mathbf{X}$*) :=* $\{\ \widetilde{\mathbf{U}}, \widetilde{\mathbf{\Sigma}}, \mathbf{V}\ \}$ *is a Truncated SVD such that:*

- $\widetilde{\mathbf{U}} \in \mathbb{R}^{n\times m}$ *is a matrix that contains the first* m *columns of* $\mathbf{U}$
- $\widetilde{\mathbf{\Sigma}} \in \mathbb{R}^{m\times m}$ *is a matrix that contains the upper square block of* $\mathbf{\Sigma}$

TSVD satisfies the equation $\mathbf{X} = \widetilde{\mathbf{U}} \cdot \widetilde{\mathbf{\Sigma}} \cdot \mathbf{V}^T$. Because $\mathbf{\Sigma}$ of SVD is always zero below the upper $m \times m$ block, so this lower $n - m \times m$ block also zeroes out the part of $\mathbf{U}$ that is truncated (columns $m + 1$ to n). Thus, with both of them truncated, the reconstruction of the original matrix is exactly the same, there's no loss of information.

4.1 Batch SVD

SVD is often calculated in a two-step process. First, a reduction of a matrix to a bi-diagonal form (all elements outside of main diagonal and one auxiliary diagonal are zero) using Householder reflections is applied. Second, a diagonalization of this bi-diagonal form using Givens rotations is performed [3].

Definition 4 (Householder reflections) *Householder matrix is an orthogonal and symmetric matrix of the form* $\mathbf{H} = \mathbf{I} - 2 \cdot V \cdot V^T$*, where* $|V| = 1$.

When multiplied with a vector, Householder matrices allow to zero out all its elements except one. Sequential application of those matrices on the left and on the right allow the transformation of the matrix into a bi-diagonal form by zeroing columns below main diagonal and rows to the right of upper second diagonal. For X, an input matrix, $\mathbf{B} = \mathbf{H} \cdot \mathbf{X} \cdot \mathbf{S}$ is a bi-diagonal transformation of $\mathbf{X}$, where $\mathbf{H}$ and $\mathbf{S}$ are products of householder matrices.

Definition 5 (Givens rotation) *Givens matrix* $\mathbf{G}$ *is an orthogonal matrix* $\mathbf{G}$ *with* $g_{ii} = g_{jj} = s$*,* $g_{ij} = -g_{ji} = c$*,* $g_{kk} = 1 \; \forall k \neq i, j$ *and zeroes elsewhere.* s *=sin(*ϕ*) and* c *=cos(*ϕ*) for some angle* ϕ*.*

Products of Givens matrices ($\mathbf{G}$ and $\mathbf{P}$) are then used to zero out second diagonal: $\mathbf{\Sigma} = \mathbf{G} \cdot \mathbf{B} \cdot \mathbf{P}$. Then, the obtained SVD is $\{ \mathbf{U}, \mathbf{\Sigma}, \mathbf{V} \}$ where $\mathbf{U} = \mathbf{G} \cdot \mathbf{H}$ and $\mathbf{V}^T = \mathbf{S} \cdot \mathbf{P}$.

4.1.1 Full and Truncated SVD

Full SVD is not suitable for long time series because it has quadratic memory complexity [1]. Matrix $\mathbf{U}$ which is received as an output, is $n \times n$. Assuming we work with double-precision numbers, the memory consumed by this matrix is $n^2 \times 8$ bytes. Using single-precision floating point numbers reduces the memory requirement by a factor of two, but it sacrifices numerical precision and is still not enough for bigger data samples.

Truncated SVD reduces both time and memory complexity of the decomposition, because last $n - m$ columns of the decomposition don't need to be formed explicitly and do not take up memory. For most practical applications in time series (i.e. where $n \gg m$), there is little interest in full SVD for those matrices, since it becomes too complex.

4.2 Incremental Singular Value Decomposition

We explored a number of different incremental techniques. But only one suitable to the task was selected. There is a very used compilation of incremental SVD techniques IncPACK [12], but all the algorithms which it contains aim at low-rank approximation of big matrices to find eigenvalues or singular values and not the actual full update of the decomposition. The only algorithm which was doing an update to the decomposition is Updating SVD.

4.2.1 Updating SVD

Working with data streams can require both rank-1 and rank-k updates to the decomposition. The Incremental SVD [8] performs an update to an existing decomposition, constructs an augmented $\mathbf{\Sigma}$ which is appended by new information that an update introduces to the matrix. Then, it re-diagonalizes the appended $\mathbf{\Sigma}$ by running it through truncated SVD.

The algorithm needs to find a component of a new data orthogonal to the input matrix $\mathbf{U}$, this is required to do a projection of new information onto the orthogonal basis $\mathbf{U}$ and to determine whether the update increases the rank of decomposition. The QR decomposition [3] is applied to find this component.

QR Factorization

Definition 6 (QR Decomposition/Factorization) *Let* $\mathbf{X} \in \mathbb{R}^{n \times m}$ *be an input matrix, then* $QR(\mathbf{X}) = \{\mathbf{Q}, \mathbf{R}\}$*, such that* $\mathbf{X} = \mathbf{Q} \cdot \mathbf{R}$ *and where:*

- $\mathbf{Q} \in \mathbb{R}^{n \times n}$ *is an orthogonal matrix*
- $\mathbf{R} \in \mathbb{R}^{n \times m}$ *is an upper triangular matrix*

QR factorization is performed using Householder reflections to zero out the input matrix below main diagonal. Since the matrix $\mathbf{Q}$ is a product of orthogonal transformations, it is orthogonal too. Also, $\mathbf{Q}$ spans the same subspace of $\mathbb{R}^n$ as $\mathbf{X}$. This property is used by the algorithm to find an orthogonal component of new information with relation to $\mathbf{U}$, and matrix $\mathbf{Q}$ can give it.

An implementation by [13] (updateSVD) was used as a basis for C++ implementation of the algorithm. This implementation works exclusively with truncated SVD - as an input, re-diagonalizing $\mathbf{\Sigma}$ and an updated output.

Complexity

We are interested in the runtime complexity on n and k (amount of added rows from rank-k update) since those are the only varying arguments. The number of columns is not supposed to change when augmenting the decomposition.

QR decomposition, which is used in the process, has a runtime complexity of approximately $2nm^2$ and quadratic memory complexity on n, the exact amount of consumed memory is up to implementation.

Assuming that we perform rank-k update and always have the worst case (rank mismatch in step 1 and step 2), the runtime complexity of updateSVD consists of:

1. Two calls of QR - one on $n \times (m + k)$ matrix, another on $m \times (m + k)$
2. Truncated SVD runs on augmented $\mathbf{\Sigma}$, which is $(m + n + k) \times (2m + k)$
3. Matrix copying, this process is always linear on any single dimension, unless there are square matrices
4. A few additional matrix multiplications - $m \times m$ by $m \times k$, $n \times m$ by $m \times m$, $n \times n$ by $n \times m$ and $m \times m$ by $m \times m$.

To finalize everything we have the following runtime complexity:

- Complexity on the amount of rows (n): QR is linear on rows and thus linear on n, matrix copying is at most linear, TSVD is linear on rows, matrix multiplication is quadratic in case of $n \times n$ by $n \times m$ matrix. The total complexity is quadratic on n.
- Complexity on the amount of added rows (k): QR is quadratic on rows if we form $\mathbf{Q}$ explicitly and thus quadratic on k, TSVD is quadratic on columns (if it needs an additional basis from Step 2, else it is linear), matrix multiplications involving k are linear. The total complexity is quadratic on k.

With constant number of columns and rank-k update, memory peaks while the algorithm constructs a concatenated matrix $(\mathbf{U}|\mathbf{P})$ (see [10], Step 3). This matrix has to contain, worst-case, two matrices of $n \times n - r$ and $n \times n$. Required memory consists of $nm + n + m^2$ for input, $n^2 - nm$ for those two matrices and the results of augmented $\mathbf{\Sigma}$ decomposition - $(m+n+k)(2m+k)+(m+n+k)+(2m+k)^2$. This gives a total of $(n^2 + 7m^2 + 2k^2) + (2nm + nk + 7mk) + (2n + m + k)$ elements which need to be stored in the memory.

Example 5 (Updating SVD) *We take as input the same $\mathbf{X}$ and B as in Example 3.*

Let $\mathbf{X} = \begin{bmatrix} -1 & 4 \\ 2 & -3 \\ 3 & 0 \end{bmatrix}$, *$r$ = rank($\mathbf{X}$) and SVD($\mathbf{X}$):*

$$\mathbf{U} = \begin{bmatrix} -0.71 & 0.41 \\ 0.65 & 0.07 \\ 0.27 & 0.91 \end{bmatrix}, \mathbf{\Sigma} = \begin{bmatrix} 5.56 & 0 \\ 0 & 2.84 \end{bmatrix}, \mathbf{V} = \begin{bmatrix} 0.51 & 0.86 \\ -0.86 & 0.51 \end{bmatrix}$$

Let's assume that a new row of $B = \begin{bmatrix} -2 & 1 \end{bmatrix}^T$ is added to $\mathbf{X}$ and $\mathbf{U}$ is appended with a row of zeroes. According to the algorithm $A = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^T$ and $(U|A)$ together with $(V|B)$ are constructed to apply QR on them.

$$(U|A) = \begin{bmatrix} -0.71 & 0.41 & 0 \\ 0.65 & 0.07 & 0 \\ 0.27 & 0.91 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \mathbf{Q}_{UA} \cdot \mathbf{R}_{UA}$$
$$= \begin{bmatrix} -0.71 & -0.41 & 0 & 0.57 \\ 0.65 & -0.07 & 0 & 0.76 \\ 0.27 & -0.91 & 0 & -0.32 \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}$$

Since rank$(Q) \neq r$, an additional basis $\mathbf{P}$ is required.

$$(V|B) = \begin{bmatrix} 0.51 & 0.86 & -2 \\ -0.86 & 0.51 & 1 \end{bmatrix} = \mathbf{Q}_{VB} \cdot \mathbf{R}_{VB}$$
$$= \begin{bmatrix} -0.51 & 0.86 \\ 0.86 & 0.51 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 & 1.88 \\ 0 & 1 & -1.21 \end{bmatrix}$$

Since rank$(Q) = r$, no additional basis $\mathbf{Q}$ is required.

After the construction of $S_{aug} = \begin{bmatrix} 5.56 & 0 \\ 0 & 2.84 \\ 1.88 & 1.21 \\ 0 & 0 \end{bmatrix}$, *SVD*$(S_{aug}) =$

$$\mathbf{U}_S = \begin{bmatrix} 0.94 & 0.16 \\ 0.04 & -0.93 \\ 0.34 & -0.34 \\ -0 & 0 \end{bmatrix}, \boldsymbol{\Sigma}_S = \begin{bmatrix} 5.89 & 0 \\ 0 & 3.06 \end{bmatrix}, \mathbf{V}_S = \begin{bmatrix} 1 & 0.09 \\ 0.09 & -1 \end{bmatrix}$$

Afterwards, the update to original decomposition goes as follows:

$\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_S$, $\mathbf{U}_1 = (\mathbf{U}|\mathbf{P}) \cdot \mathbf{U}_S$, $\mathbf{V}_1 = (\mathbf{V}|\mathbf{Q}) \cdot \mathbf{V}_S$

And the updated decomposition looks like:

$$\mathbf{U}_1 = \begin{bmatrix} -0.65 & -0.50 \\ 0.61 & 0.04 \\ 0.30 & -0.80 \\ -0.34 & 0.34 \end{bmatrix}, \boldsymbol{\Sigma}_1 = \begin{bmatrix} 5.89 & 0 \\ 0 & 3.06 \end{bmatrix}, \mathbf{V}_1 = \begin{bmatrix} 0.58 & -0.81 \\ -0.81 & -0.58 \end{bmatrix}$$

Chapter 5

Evaluation

We evaluate the algorithms by running the implementations of different algorithms against each other [1]. First, we evaluate the memory allocation of the used data structures, then perform an evaluation on two implementations of Scalable Sign Vector algorithm – iterative and incremental methods of calculating weight vector. We show that the incremental weight vector computation is more efficient than the iterative one. We use the incremental computation as basis for batchCD and cachedCD implementations.

Then, we switch to the incremental algorithms and evaluate them on rank-1 and rank-k updates to the decomposition. We show that updateSVD is not scalable for long time series and we perform the remaining analysis on different Centroid Decomposition algorithms to measure their performance with respect to their key component - Sign Vector search.

5.1 Setup

The machine we used to run the experiments has the following specifications:

- CPU: Intel(R) Core(TM) i7-3770, 4 cores, 8 threads @ 3.40GHz;
- RAM: DDR3 16GB @ 1600 MHz;
- OS: Ubuntu 12.04.5 LTS (GNU/Linux 3.5.0-37-generic x86_64);
- C++ compiler: gcc 5.4.1 20160904;

[1] Code for implementations can be accessed through:
https://github.com/eXascaleInfolab/InCD_Benchmark

- Compiler options: -std=C++14 -Wall -Wextra -Werror -pedantic -O3.

The data sets used for the experiments are the following:

- Hydrological data [14], time series are randomly sampled from the data set and concatenated to create longer time series than available: 4 time series up to 1M elements;
- North America climate data [15], 4 columns taken – CO2, H2, WET, CLD to create 4 time series of up to 10K rows.

For the updateSVD algorithm, the QR implementation used is the one in GNU Scientific Library [16], with library being linked to the project. The C++ implementation for truncated SVD is based on [17].

5.2 Results

All experiments use hydrology data unless stated otherwise.

5.2.1 Memory analysis

Figure 5.1 shows the memory allocation of the algorithms at a rank-1 update with $m = 10$ and n varying rows from 100K to 1M.

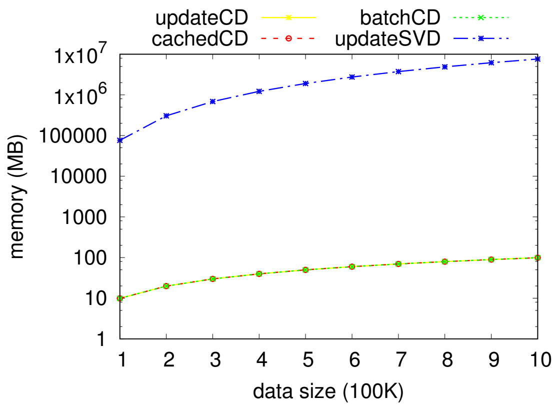


Figure 5.1: Memory consumption of incremental algorithms (log scale)

All Centroid Decomposition variants have a very low and near-identical memory usage. While updateSVD stands out with immense memory consumption in comparison. Reliance on full QR decomposition and its result to obtain an $n \times n$ matrix severely reduces the usability of this algorithm.

5.2.2 Runtime analysis

Sign vector search algorithm

Figure 5.2 compares two techniques of calculating a sign vector for Centroid Decomposition. The original matrix has $m = 4$ columns and n rows which vary from 10k to 100k.

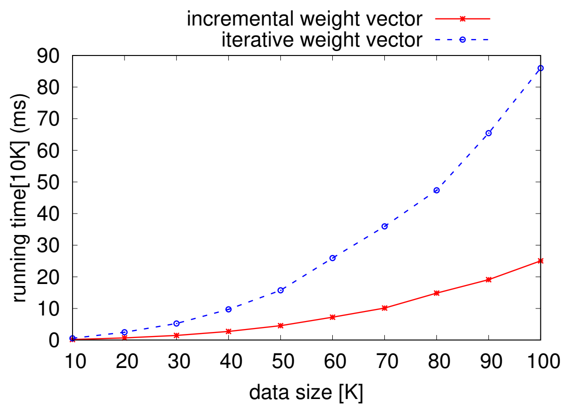


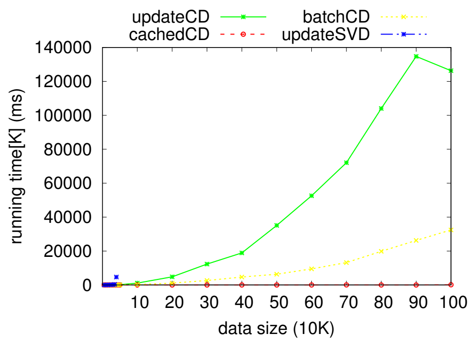
Figure 5.2: Runtime of sign vector computation methods

The results of this experiment show that both techniques have the same quadratic complexity but incremental algorithm is at least 3 times faster than the iterative one. This is explained by the fact that the former algorithm avoids a lot of calculations by reusing the old weight vector to calculate the new one. In the remaining experiments, we use the incremental weight vector algorithm as a basis for the batchCD and cachedCD algorithms.

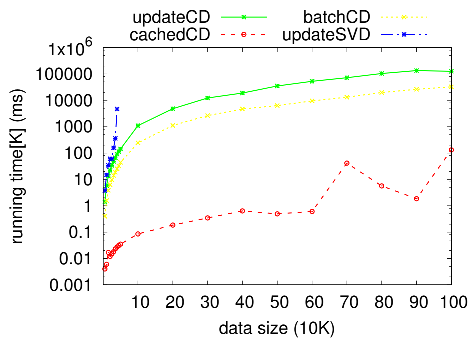
Incremental techniques

To evaluate the efficiency of the incremental techniques, we first perform a set of rank-1 update experiments where we vary the number on rows, then we conduct a set of rank-k update experiments where we fix the number of rows and vary the number of added rows.

In the first experiment evaluating the rank-1 update, the number of columns m is set to 4 while the number of rows n varies from 5K to 1M.



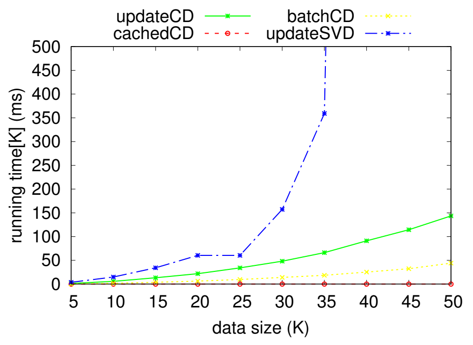
(a) Linear scale.



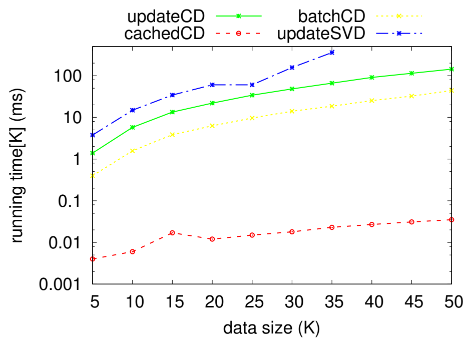
(b) Log scale.

Figure 5.3: Runtime of incremental algorithms, rank-1 update with varying rows

The results of Figure 5.3 show that updateCD and batchCD are quadratic with n, the last spike on updateCD is within the normal fluctuations of sign vector calculation caused by introducing new type of data (see later part on sign vector iterations). CachedCD is linear with n and remains at extremely low levels on the whole range.



(a) Linear scale.



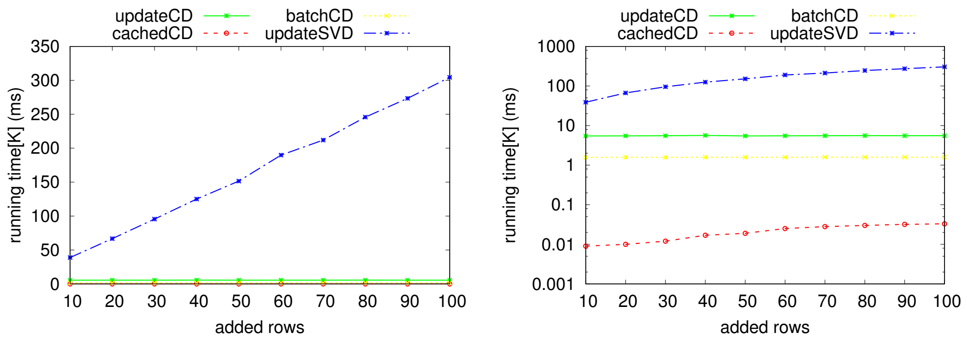
(b) Log scale.

Figure 5.4: Runtime of incremental algorithms, rank-1 update with varying rows; 40K value for updateSVD is off chart

Figure 5.4 shows that updateSVD cannot reasonably handle more than 35k rows due to memory and computational constraints. The value for 40K rows is approximately 13 times bigger than the one for 35K (off chart). This happened due to the fact that the required memory went beyond the physical capacity of the testing machine (16GB) and performance significantly went down beyond its normal complexity curve because of heavy Swap file usage.

In the next experiment, we evaluate the efficiency of the algorithms to perform a rank-k update. We set n to 10k, m to 4 and we vary the added rows from 10 to

100.



(a) Linear scale. (b) Log scale.

Figure 5.5: Runtime of incremental algorithms, rank-k update with varying added rows

Figure 5.5 shows that the runtime of the Centroid-based algorithms is almost unaffected by the rank of the update. In absolute value, the values are increasing trend-wise, which is not reflected on the graph, but the increase is extremely slow. UpdateSVD is significantly impacted by the rank of the update. As was discussed in Section 4.2, the runtime is quadratic, which is not reflected on the graph. This result could be explained by the fact that the execution of the algorithm shown on the graph doesn't branch into worst-case scenario and QR implementation by GSL might apply certain optimizations. This point is discussed in Section 6.

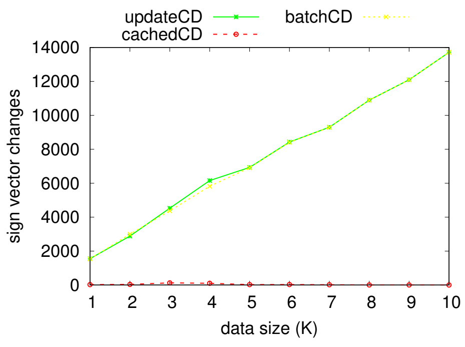
The previous experiments show that cachedCD clearly outperforms all other techniques. UpdateSVD has memory constraints that prevent it from being able to update matrices of 40K and above on our machine. The runtime complexity also becomes unreasonable when performing rank-k update. All Centroid-based techniques can operate on very big data sets, batchCD and updateCD are only limited by quadratic runtime, which means that they are limited in their usage. However, cachedCD can update even a big decomposition extremely quickly, both rank-1 and rank-k.

5.2.3 Sign Vector iterations and different data sets

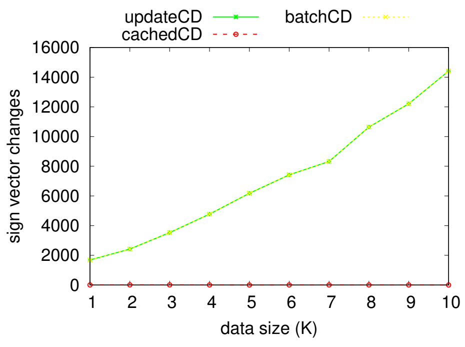
The results of the previous experiments show that Centroid-based techniques are more time and space efficient than the incremental SVD. Thus, the next experiments aim to study the incremental CD performance with respect to the key aspect of the algorithm – the search of maximizing sign vector. The performance metric we use is the number of flips of the sign of the elements of Z_i. All the general properties of the sign vector remain the same as described by [1].

To evaluate the stability of the Centroid-based algorithms, we conduct sign vector experiments on two different data sets. For updateCD, the number of sign vector iterations is taken from batch CD that is performed on augmented $\mathbf{L}_0$ during the updating process. First we evaluate the number of sign vector iterations in a rank-1 update with varying number of rows, then we evaluate a rank-k update with fixed number of rows.

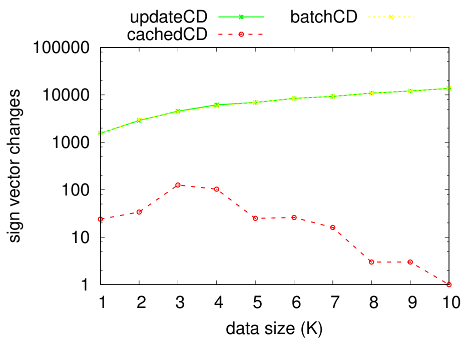
The first experiment on rank-1 update has a fixed number of columns $m = 4$ and a varying number of rows from 1K to 10K.



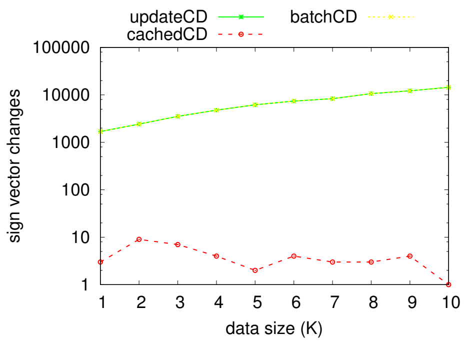
(a) Hydrology, Linear scale.



(b) Climate, Linear scale.



(c) Hydrology, Log scale.



(d) Climate, Log scale.

Figure 5.6: Sign vector iterations of incremental CD techniques, rank-1 update with varying rows

Figure 5.6 shows that the amount of sign vector iterations for batchCD and updateCD follows a linear trend. The number of iterations on average is approximately equal to $\frac{n}{2}$ per sign vector calculated, thus $\frac{n \times m}{2}$ for the whole matrix and the figure reflects this fact.

BatchCD and updateCD techniques have almost the same number of sign vector iterations. UpdateCD doesn't perform better than batchCD on the update because it has to perform a full CD in the process. Even though matrix $\mathbf{L}$ is a stationary point of decomposition (cf. Section 3.2.1), running CD on augmented $\mathbf{L}$ gives no computational advantage to the algorithm, because all sign vectors have to be re-calculated.

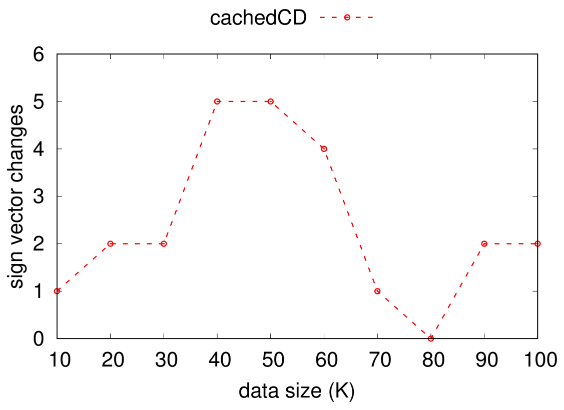


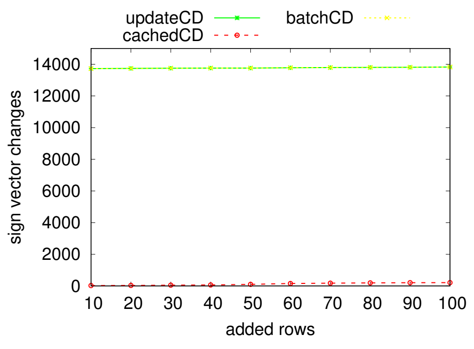
Figure 5.7: Sign vector iterations of cachedCD, rank-1 update, hydrology data

CachedCD outperforms both batchCD and updateCD and performs less sign flips when the number of rows increases. This is explained by the fact that rank-1 update has bigger impact on a smaller matrix, because of weight vector calculation (cf. Section 3.1). The runtime has a trend of increasing for rank-1 updates with increased number of rows, because one iteration in the search of a sign vector has linear complexity. Figure 5.7 extends the experiment on cachedCD to 100K rows. This experiment alongside the experiment in Figure 5.6 show that the required number of iterations is essentially constant for rank-1 updates, because the impact of one row is extremely limited if the data is not too different from what's already in the matrix.

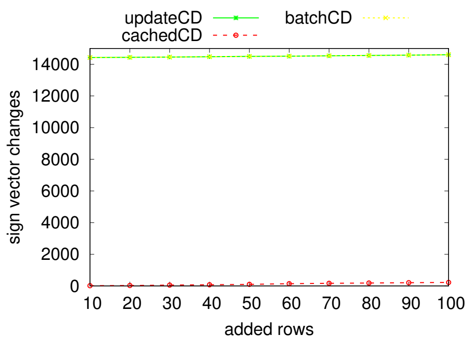
Next experiment is performing rank-k update with k varying from 10 to 100 on a fixed matrix with $m = 4$ and $n = 10\text{K}$.

Figure 5.8 shows a linear trend for all techniques when k varies. CachedCD requires more iterations than previously for rank-1 updates. For rank-k update, Z_i is supposed to be on average $\frac{k}{2}$ elements away from the optimum (cf. Section 3.2.2). Figure 5.9 shows that the number of iterations of cachedCD is approximately $\frac{k \times m}{2}$ for all values of k. Thus, the trend on k is linear on the added rows, same as on rows (n).

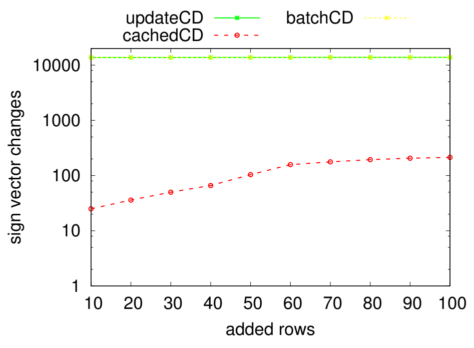
Figures 5.6 and 5.8 show that incremental CD techniques are stable and do not show any significant differences in their performance when applied to different real-world data sets. All observed trends can be traced in both data sets.



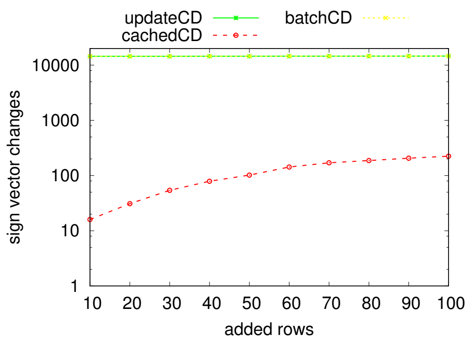
(a) Hydrology, Linear scale.



(b) Climate, Linear scale.

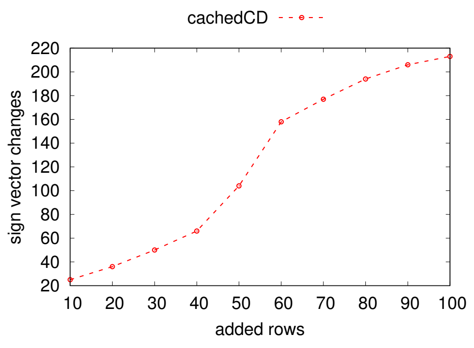


(c) Hydrology, Log scale.

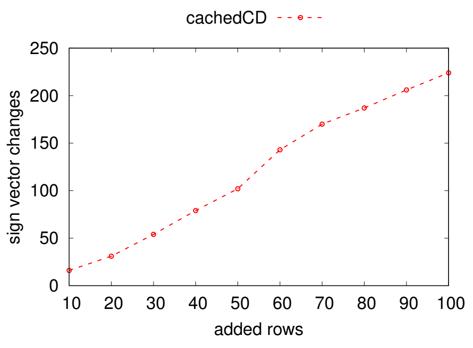


(d) Climate, Log scale.

Figure 5.8: Sign vector iterations of incremental CD techniques, rank-k update with varying added rows



(a) Hydrology, Linear scale.



(b) Climate, Linear scale.

Figure 5.9: Sign vector iterations of cached CD, rank-k update with varying added rows

Chapter 6

Conclusions and future work

In this thesis, we explored the properties of four incremental matrix decomposition algorithms - updateCD, batchCD, cachedCD and updateSVD. We provided their C++ implementations and used them to conduct an empirical evaluation of those algorithms.

Evaluation showed that updateSVD is not efficient enough to be used on bigger data sets, it falls short on both computational complexity (for rank-k updates) and memory usage (for matrices with big number of rows). CD variants are much more memory efficient and almost not impacted by the size of introduced data, which lets them work with much bigger data sets.

UpdateCD and batchCD fall short on computational complexity. It is not feasible to recalculate the whole decomposition every time new data is introduced because it can take hours for big matrices. Caching Sign Vectors proved to be extremely efficient - it drastically reduces the number of sign vector iterations and allows cachedCD to update the decomposition with linear complexity on both the size of matrix and the size of introduced data.

In the future, we plan to improve the implementations of the used algorithms. First of all, improving the existing implementation - optimization, better documentations, making code more universal, usable and improving overall quality. Second, exploring the possible expansion of the math mini-framework created for this task and implementing new algorithms based on it. It would be also of interest to investigate the optimizations that updateSVD applies to reduce the quadratic complexity (coming from QR) to linear.

Bibliography

[1] M. Khayati, M. H. Böhlen, and J. Gamper, “Memory-efficient centroid decomposition for long time series,” in *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, 2014, pp. 100–111. [Online]. Available: https://doi.org/10.1109/ICDE.2014.6816643

[2] O. Stapleton and M. Khayati, c-ReVival, Recovery of missing values using Centroid Decomposition, homepage: http://revival.exascale.info/, 2017.

[3] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd Ed.).* Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[4] L. Balzano and S. J. Wright, “On GROUSE and incremental SVD,” in *5th IEEE International Workshop on Computational Advances in Multi-Sensor Adaptive Processing, CAMSAP 2013, St. Martin, France, December 15-18, 2013*, 2013, pp. 1–4. [Online]. Available: https://doi.org/10.1109/CAMSAP.2013.6713992

[5] M. Khayati and M. H. Böhlen, “REBOM: recovery of blocks of missing values in time series,” in *Proceedings of the 18th International Conference on Management of Data, COMAD 2012, 2012, Pune, India*, 2012, pp. 44–55. [Online]. Available: http://comad.in/comad2012/pdf/khayati.pdf

[6] M. T. Chu and R. Funderlic, “The centroid decomposition: Relationships between discrete variational decompositions and svds,” *SIAM J. Matrix Analysis Applications*, vol. 23, no. 4, pp. 1025–1044, 2002. [Online]. Available: https://doi.org/10.1137/S0895479800382555

[7] M. Khayati, P. Cudré-Mauroux, and M. Böhlen, “Revival: Scalable recovery of missing values using centroid decomposition,” *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[8] M. Brand, “Incremental singular value decomposition of uncertain data with missing values,” in *Proceedings of the 7th European Conference on Computer Vision-Part I*, ser. ECCV ’02. London,

UK, UK: Springer-Verlag, 2002, pp. 707–720. [Online]. Available: http://dl.acm.org/citation.cfm?id=645315.649157

[9] J. R. Blevins and M. T. Chu, "Updating the centroid decomposition with applications in lsi," 2004, unpublished manuscript.

[10] J. Blevins, "cdupdate.m v1.11," 2004/08/04 21:32:25, jrb11@duke.edu. [Online]. Available: https://jblevins.org/research/centroid/cdupdate.m

[11] O. Stapleton, "Real-time centroid decomposition of streams of time series," 2017. [Online]. Available: https://exascale.info/assets/pdf/students/2017-Oliver_RealTimeCDStreamsTS.pdf

[12] C. Baker, K. Gallivan, and P. Van Dooren, IncPACK Overview, homepage: https://www.math.fsu.edu/ cbaker/IncPACK/, 2012.

[13] J. Blevins, "updatesvd.m v1.4," 2004/12/08 21:23:47, jrb11@duke.edu. [Online]. Available: https://jblevins.org/research/centroid/updatesvd.m

[14] Swiss Federal Office for the Environment, Water discharge time series, homepage: http://www.hydrodaten.admin.ch/en, 2016.

[15] A. C. Lozano, H. Li, A. Niculescu-Mizil, Y. Liu, C. Perlich, J. Hosking, and N. Abe, "Spatial-temporal causal modeling for climate change attribution," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '09. New York, NY, USA: ACM, 2009, pp. 587–596, data file NA-1990-2002-Monthly-high.csv from: "http://www-bcf.usc.edu/ liu32/data.html". [Online]. Available: http://doi.acm.org/10.1145/1557019.1557086

[16] GNU Project, "GNU scientific library v2.4, QR decomposition." [Online]. Available: https://www.gnu.org/software/gsl/doc/html/linalg.html#qr-decomposition

[17] D. Cook, "svd.c c code for computing a grand tour," 2009, dicook@iastate.edu. [Online]. Available: http://www.public.iastate.edu/ dicook/JSS/paper/code.html