Technische Universiteit Delft
Faculteit Elektrotechniek, Wiskunde en Informatica
Delft Institute of Applied Mathematics

Large-scale SVD algorithms for Latent Semantic Indexing, Recommender Systems and Image Processing

Verslag ten behoeve van het
Delft Institute of Applied Mathematics
als onderdeel ter verkrijging

van de graad van

BACHELOR OF SCIENCE
in
TECHNISCHE WISKUNDE

door

Y.M. van der Vlugt

Delft, Nederland
Juli 2018

BSc verslag TECHNISCHE WISKUNDE

Large-scale SVD algorithms for Latent Semantic Indexing, Recommender Systems and Image Processing

Y.M. van der Vlugt

Technische Universiteit Delft

Begeleider

Martin van Gijzen

Overige commissieleden

Juan-Juan Cai　　Bart van den Dries

Juni, 2018　　Delft

Abstract

Over the past three decades, the singular value decomposition has been increasingly used for various big data applications. As it allows for rank reduction of the input data matrix, it is not only able to compress the information contained, but can even reveal underlying patterns in the data through feature identification. This thesis explores algorithms for large-scale SVD calculation, and uses these to demonstrate how the SVD can be applied to a variety of fields, including information retrieval, recommender systems and image processing.

The algorithms discussed are Golub-Kahan-Lanczos bidiagonalization, randomized SVD and block power SVD. Each algorithm is implemented in Matlab and both error and time taken by each algorithm are compared. We find that the block power SVD is very effective, especially when only the truncated SVD is required. Due to its simplicity, speed and relatively small error for low-rank matrix approximation, it is an ideal method for the applications discussed in this thesis.

We show how the SVD can be used for information retrieval, through Latent Semantic Indexing. The method is tested on the Time collection and we find that the SVD removes much of the noise present in the data and solves the issues of synonymy and polysemy. Then, SVD-based algorithms for recommender systems are presented. We implement a basic SVD algorithm called Average Rating Filling, and a (biased) stochastic gradient descent algorithm, which was developed for the Netflix recommender-system prize. These are tested on the Movielens 100k dataset, resulting in the best performance by biased stochastic gradient descent. Finally, the SVD is used for image compression and we find that, while not very useful for face recognition, the SVD could provide a time- and space-efficient method for searching through an image database for similar images.

Contents

1 Introduction **1**

I The Singular Value Decomposition **2**

2 Properties of the Singular Value Decomposition **2**

3 Golub-Kahan-Lanczos Bidiagonalization **4**

4 Power Method **7**
- 4.1 Power Method for Eigenvalue Problems . . . 7
- 4.2 Power Method for SVD . . . 8

5 Randomized SVD **11**

6 Comparison of methods **13**
- 6.1 Error . . . 14
- 6.2 Time . . . 16

II SVD for Information Retrieval **17**

7 Latent Semantic Indexing method **17**
- 7.1 Semantic effects of reduction . . . 17
- 7.2 Queries . . . 18
- 7.3 Folding-in . . . 18

8 Data representation **19**
- 8.1 Term frequency . . . 19
- 8.2 Inverse document frequency . . . 19

9 Small-scale LSI examples **20**
- 9.1 Example 1 . . . 20
- 9.2 Example 2 . . . 21

10 Text representation **22**

11 Large-scale implementation of LSI search **23**

III SVD for Recommender Systems **26**

12 The Method **26**

13 Average rating filling **27**

14 Stochastic gradient descent **28**

15 Biased stochastic gradient descent **30**

IV SVD for Image Processing **32**

16 Image compression **32**

17 Face recognition **34**

18 Further applications of the SVD **36**
18.1 The pseudoinverse and linear least squares problems 36
18.2 Principal component analysis 36

19 Conclusion **37**

References **39**

Appendices **41**

A Matlab Code for Part 1 **41**
A.1 GKLB 41
A.2 Randomized SVD 41
A.3 Block power SVD 42

B Matlab code for Part 2 **42**
B.1 Preprocess documents 42
B.2 LSI search and testing 43

C Matlab code for Part 3 **43**
C.1 Average rating filling (ARF) 43
C.2 Stochastic gradient descent (SGD) 44
C.3 Biased stochastic gradient descent 45

D Matlab code for Part 4 **45**
D.1 Image compression 45
D.2 Face recognition 46

E Material for Example 2 **46**
E.1 Book titles for Example 2 46
E.2 Term-document matrix for Example 2 47
E.3 Term-document plot for Example 2 48

1 Introduction

It is becoming increasingly important and challenging to make sense of, store, and efficiently search through huge amounts of data. This challenge has led to the development of many complex algorithms, each suited to its own specific application, in the fields of big data and machine learning. The objective of this thesis is to show how the singular value decomposition (SVD) can be broadly used for a variety of big data applications. This objective can be split into three sub-objectives:

1. To investigate large-scale applications of the SVD, such as information retrieval, recommender systems and image processing.
2. To show how the SVD can be computed efficiently, considering the structure of the data and outcomes required by the large-scale applications discussed.
3. To identify why the SVD can lead to an improvement in performance, increase in efficiency and reduction of noise with respect to other, more traditional, algorithms.

This report is structured in four parts. We begin in Part 1 by providing some background information about the SVD, and present three algorithms with which the SVD can be computed. Both the performance and time taken by the algorithms are compared, in order to predict how they will perform on the large-scale applications.

We then go on to discuss these applications, and the SVD algorithms are applied to each test problem discussed. Part 2 shows how the SVD can be used for information retrieval. Specifically, the latent semantic indexing (LSI) method for text retrieval is explained and demonstrated through both small- and large-scale examples. In Part 3, the application of SVD for recommender systems is discussed and we demonstrate another SVD-like approach applied to movie rating prediction. Finally, in Part 4 we apply the SVD to a different type of data: images. The possibilities for image compression and face recognition using the SVD are discussed.

Part I
The Singular Value Decomposition

The singular value decomposition (SVD) is a very useful tool from linear algebra. It is the factorization of a matrix into a product of two orthogonal matrices, containing the singular vectors of the input matrix, and one diagonal matrix, containing the singular values of the input matrix. The large-scale applications discussed in this project require computation of the SVD for very large, often sparse matrices representing huge datasets. Finding the SVD of such massive matrices is a numerically intensive process and cannot be done analytically or using algorithms that change the structure of the matrix. Many simple algorithms also rely on properties such as symmetry, or the matrix in question being square, that cannot be assumed in most real-life cases. This part will first introduce the concept of the SVD and some properties that will be relevant in later sections. Then, we will discuss and compare three algorithms suitable for finding the SVD of such large matrices: Golub-Kahan-Lanczos bidiagonalization, (block) power SVD and randomized SVD.

2 Properties of the Singular Value Decomposition

In this section, we first define the SVD, and present a number of useful properties. We end with the definition of the truncated SVD, which is the form that will be most relevant to later sections.

Definition 2.1. (Golub & van Loan, 1996) The singular value decomposition, or SVD, of an $m \times n$ matrix A of rank r is given by

$$A = U\Sigma V^T$$

where:

(i) $U = [u_1, u_2, \ldots, u_m] \in \mathbb{R}^{m\times m}$ is an orthogonal matrix of which the columns define the m orthonormal eigenvectors of AA^T, also referred to as the left singular vectors.

(ii) $V = [v_1, v_2, \ldots, v_n] \in \mathbb{R}^{n\times n}$ is an orthogonal matrix of which the columns define the n orthonormal eigenvectors of A^TA, also referred to as the right singular vectors.

(iii) $\Sigma \in \mathbb{R}^{m\times n}$, $\Sigma = \text{diag}(\sigma_1, \sigma_2, \ldots \sigma_p)$, where $p = \min(m, n)$ and $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > \sigma_{r+1} = \ldots = \sigma_p = 0$. The σ_i, $i = 1, \ldots, p$ are the singular values of A.

The triplet (σ_i, u_i, v_i) will be referred to as the i-th singular triplet of A.

The decomposition can be inferred from the following observation:

$$\begin{aligned} AA^T &= U\Sigma V^T V\Sigma^T U^T \\ &= U\Sigma I\Sigma^T U^T \\ &= U(\Sigma\Sigma^T)U^T \end{aligned}$$

This is the eigendecomposition of the matrix AA^T, in which U holds the eigenvectors of AA^T and $\Sigma\Sigma^T$ holds the eigenvalues. Orthogonality of U follows from AA^T being symmetric and thus having an orthonormal basis of eigenvectors. Furthermore, from $\Sigma\Sigma^T = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_p)$ it follows that $\sigma_i = \sqrt{\lambda_i}$ for $i = 1, 2, \ldots, p$. Note that the same can be done for the matrix V, where we use $A^TA = V\Sigma^T\Sigma V^T$.

Theorem 2.1. *Let the SVD of A be given by Definition 2.1. We observe the following properties:*

1. $A = \sum_{i=1}^{r} u_i \cdot \sigma_i \cdot v_i^T$ *is the dyadic decomposition of A*
2. $Row(A) = span\{v_1, \ldots, v_r\}$
3. $Col(A) = span\{u_1, \ldots, u_r\}$
4. $Nul(A) = span\{v_{r+1}, \ldots, v_n\}$

5. *$Nul(A^T) = span\{u_{r+1}, \ldots, u_m\}$*

6. *The Frobenius norm of A is given by $\|A\|_F^2 = \sigma_1^2 + \sigma_2^2 + \ldots + \sigma_r^2$*

7. *The 2-norm of A is given by $\|A\|_2^2 = \sigma_1$*

One way to understand the SVD is by examining what role each component (U, Σ and V) plays in the transformation given by the matrix A. An $m \times n$ matrix A maps a unit sphere in $\mathbb{R}^n$ to an ellipsoid in $\mathbb{R}^m$. For a 2×2 matrix this is visualized in Figure 1. Since U and V are orthogonal, applying V^T and U results in two rotations without distorting the shape, while application of Σ stretches the circle along the coordinate axes to form an ellipse. For higher dimensions, with $rank(A) = r$, the unit sphere is transformed to an r-dimensional ellipsoid with semi-axes in the direction of the left singular vectors u_i of magnitude σ_i (Kalman, 1996).

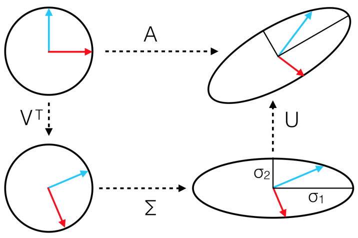


Figure 1: Geometric interpretation of SVD

The SVD allows for construction of a low-rank approximation of any matrix, as we see in the following result:

Theorem 2.2 (Eckart & Young)**.** *Let the SVD of A be given by Definition 2.1 with $r = rank(A) \leq p = \min(m, n)$, and define*

$$A_k = \sum_{i=1}^{k} u_i \cdot \sigma_i \cdot v_i^T \tag{1}$$

then

$$\min_{rank(B)=k} \|A - B\|_F^2 = \|A - A_k\|_F^2 = \sigma_{k+1}^2 + \ldots + \sigma_p^2$$

Moreover, A_k is the best k-rank approximation for A for any unitarily invariant norm, hence

$$\min_{\text{rank}(B)=k} \|A - B\|_2 = \|A - A_k\|_2 = \sigma_{k+1}$$

Equivalent to Equation 1 is the following form, which will be referred to as the truncated SVD:

$$A_k = U_k \cdot \Sigma_k \cdot V_k^T \tag{2}$$

where

$$\begin{aligned} U_k &= [u_1, u_2, \ldots, u_k] \in \mathbb{R}^{m \times k} \\ V_k &= [v_1, v_2, \ldots, v_k] \in \mathbb{R}^{n \times k} \\ \Sigma_k &= \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_k) \in \mathbb{R}^{k \times k} \end{aligned}$$

Now, remember that the singular values σ_i^2 are the eigenvalues of the matrix $A^T A$, which (in data applications) happens to be the covariance matrix of the data represented by A (Prokhorov, 2011). The covariance matrix defines both the spread and variance of the data. The eigenvector of $A^T A$ corresponding to the largest eigenvalue of $A^T A$ lies in the direction of the most variance of the data. In other words, the singular vectors corresponding to the largest singular values of A explain most of the data; it is even quite likely that the singular vectors corresponding to the smallest singular values only represent noise in the data. Since the singular values are ordered in decreasing order in Σ, the k left and right singular vectors contributing most to the variance of the data represented by A are the first k columns of U and V, respectively. The truncated SVD is very useful for many of the applications that will be discussed, as it not only reduces the amount of space required to store data but can also remove associated noise, as will be shown in later sections.

3 Golub-Kahan-Lanczos Bidiagonalization

Golub-Kahan-Lanczos bidiagonalization (GKLB) is an iterative algorithm that approaches the truncated SVD of a matrix A (Golub & Kahan, 1964). An advantage of GKLB as opposed to the more traditional SVD based on Householder reflections (Golub & Reinsch, 1970) is that GKLB does not change the structure of the matrix, making it more suitable for very large matrices. Furthermore, (Golub & Kahan, 1964) also states that while in general, the method involving Householder reflections is faster than GKLB, this is not the case when the matrix in question is sparse.

Say we want to compute the k-dimensional truncated SVD A_k. First, the matrices P_{k+1} and Q_k are computed such that the product $P_{k+1}^T AQ_k$ is in bidiagonal (rather than diagonal) form, where P_{k+1} and Q_k are unitary matrices with orthonormal columns, i.e.

$$P_{k+1}^T AQ_k = B_k = \begin{bmatrix} \alpha_1 & & & \\ \beta_2 & \alpha_2 & & \\ & \beta_3 & \ddots & \\ & & \ddots & \alpha_k \\ & & & \beta_{k+1} \end{bmatrix} \tag{3}$$

Multiplying both sides of Equation 3 with P_{k+1} gives

$$A(q_1 q_2 \dots q_k) = (p_1 p_2 \dots p_{k+1}) \begin{bmatrix} \alpha_1 & & & \\ \beta_2 & \alpha_2 & & \\ & \beta_3 & \ddots & \\ & & \ddots & \alpha_k \\ & & & \beta_{k+1} \end{bmatrix}$$

Expanding the k^{th} column on both sides gives

$$\begin{aligned} Aq_k &= \alpha_k p_k + \beta_{k+1} p_{k+1} \\ \implies \beta_{k+1} p_{k+1} &= Aq_k - \alpha_k p_k \end{aligned}$$

In the same way, $A^T P_{k+1} = Q_k B_k^T$ gives

$$A^T(p_1 p_2 \dots p_{k+1}) = (q_1 q_2 \dots q_k) \begin{bmatrix} \alpha_1 & \beta_2 & & \\ & \alpha_2 & & \\ & \ddots & \ddots & \\ & & \alpha_n & \beta_{n+1} \end{bmatrix}$$

Here, expanding the k^{th} column on both sides gives

$$\begin{aligned} A^T p_k &= \beta_k q_{k-1} + \alpha_k q_k \\ \implies \alpha_{k+1} q_{k+1} &= A^T p_{k+1} - \beta_{k+1} q_k \end{aligned}$$

From this, Algorithm 1 is obtained.

Algorithm 1: GKLB Algorithm

```
1 Choose $w \in \mathbb{R}^n$ randomly
2 $\beta_1 p_1 = w$
3 $\alpha_1 q_1 = A^T p_1$
4 for $i = 1, 2, \dots$ do
5     $\beta_{i+1} p_{i+1} = Aq_i - \alpha_i p_i$
6     $\alpha_{i+1} q_{i+1} = A^T p_{i+1} - \beta_{i+1} q_i$
7 end
```

where $\alpha_i > 0$, $\beta_i > 0$ such that $\|q_i\| = \|p_i\| = 1$. Now

$$AQ_k = P_{k+1}B_k \tag{4}$$
$$A^T P_{k+1} = Q_k B_k^T + \alpha_{k+1} q_{k+1} e_{k+1}^T \tag{5}$$

where $\alpha_{k+1} q_{k+1} e_{k+1}^T$ is the error.

A singular value $\tilde{\sigma}$ and corresponding left and right singular vectors $\tilde{u}$ and $\tilde{v}$ of B_k are related by

$$B_k \tilde{v} = \tilde{\sigma}\tilde{u},$$
$$B_k^T \tilde{u} = \tilde{\sigma}\tilde{v}$$

Substituting the above in Equations 4 and 5 gives:

$$AQ_k \tilde{v} = \tilde{\sigma} P_{k+1}\tilde{u},$$
$$A^T P_{k+1}\tilde{u} = \tilde{\sigma} Q_k \tilde{v} + \alpha_{k+1} q_{k+1} e_{k+1}^T \tilde{u}$$

Comparing the above equations to $AV = U\Sigma$ and $A^T U = V\Sigma$, we see that the singular values of B_k converge to the singular values of A, and $P_{k+1}\tilde{u}$ and $Q_k\tilde{v}$ converge to the left and right singular vectors of A, respectively. When we stop the algorithm at $k < \min(m, n)$, we obtain an approximation of the first k singular triplets, forming the truncated SVD of A. However, only the first few singular triplets (relative to k) will have converged, and it is usually necessary to perform far more than k iterations when the true first k singular triplets are required.

Algorithm 1 is constructed to make the columns of P and Q orthonormal. However, orthogonality may be lost due to round-off error, which can lead to large errors in the resulting SVD. To prevent this, we can re-orthogonalize P and Q during the algorithm. It has been shown that it is only necessary to re-orthogonalize one of the bases, say Q, to preserve orthogonality (Simon & Zha, 2006). This is called one-sided re-orthogonalization and results in Algorithm 2.

Algorithm 2: GKLB algorithm with one-sided re-orthogonalization

```
1  Choose $w \in \mathbb{R}^m$ randomly
2  $\beta_1 p_1 = w$
3  $\alpha_1 q_1 = A^T p_1$
4  for $i = 1$ to $k$ do
5      $p_{i+1} = Aq_i - \alpha_i p_i$
6      $\beta_{i+1} = \|p_{i+1}\|$
7      $p_{i+1} = p_{i+1}/\beta_{i+1}$
8      $P_{i+1} = [P_i \; p_{i+1}]$
9      $q_{i+1} = A^T p_{i+1} - \beta_{i+1} q_i$
10     $h_i = Q_i^T q_{i+1}$
11     $q_{i+1} = q_{i+1} - Q_i h_i$
12     $\alpha_{i+1} = \|q_{i+1}\|$
13     $q_{i+1} = q_{i+1}/\alpha_{i+1}$
14     $Q_{i+1} = [Q_i \; q_{i+1}]$
15 end
16 $[U_B, \Sigma_B, V_B] = \mathrm{SVD}(B)$
17 $U \leftarrow P_{k+1} U_B$
18 $\Sigma \leftarrow \Sigma_B$
19 $V \leftarrow Q_k V_B$
```

Another way to efficiently calculate the SVD is by using Matlab's SVDS method. The command `svds(A,k)` returns the first k singular triplets of A without having to calculate the full SVD of A, which is precisely what is required for the applications discussed. Older versions of Matlab used ARPACK

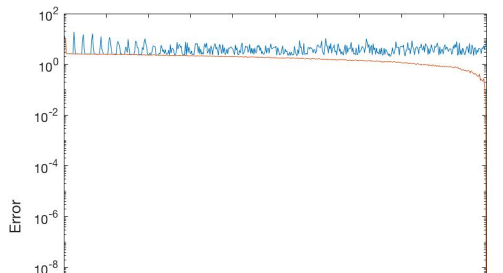
(ARnoldi PACKage) for the SVDS, which relied on the Implicitly Restarted Arnoldi Method (IRAM). However, more recent versions, including the version used for this paper, also use GKLB. Matlab also has a standard `svd(A)` command, which performs the full SVD of A. However, this command is not suited for large and sparse matrices.

We can implement the GKLB algorithm in Matlab and test its efficiency and error with and without one-sided re-orthogonalization. The corresponding Matlab code can be found in Appendix A.1. The time taken is also compared to Matlab's SVDS method. This is done for a random sparse 5000×500 matrix A with 4% density, as we will later see is often typical for the applications discussed. The error is computed using Equation 5 as:

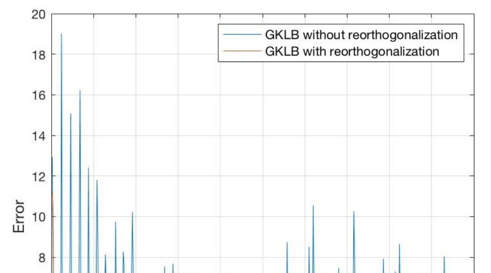
$$\begin{aligned} error &= \|A^T P_{k+1} - Q_k B_k^T\| \\ &= \|\alpha_{k+1} q_{k+1} e_{k+1}^T\| \\ &= \alpha_{k+1}\|q_{k+1}\| \\ &= \alpha_{k+1} \end{aligned}$$

The process of computing the time and error for each method is repeated ten times, and the average of these trials is shown in the results.

In Figure 2 the error is shown for GKLB with and without one-sided re-orthogonalization for increasing values of k, where k ranges from 1 to 500, the rank of A. We can see that for the GKLB algorithm without re-orthogonalization, the size of the error oscillates wildly showing that the method is very unstable. Furthermore, the error does not converge to 0, which is be caused by the lack of orthogonality of P and Q. The GKLB algorithm with one-sided re-orthogonalization has a much smoother performance, and the error eventually converges to 0 as desired. However, the most significant drop in error is not reached until the last possible iteration: for $k = 499$ the error is 0.0511, while for $k = 500$ the error drops to $5.8009 \cdot 10^{-14}$.



(a) Logarithmic scale



(b) Decimal scale

Figure 2: Error of GKLB method with and without one-sided re-orthogonalization

Before looking at the time taken by the algorithm, it is interesting to analyse the theoretical time complexity. We see in Algorithm 2 that there are k iterations; in each iteration, 2 matrix-vector products are performed which take $\mathcal{O}(mn)$ time. For the re-orthogonalization step, a matrix-vector multiplication with the matrices Q_i and Q_i^T is performed, of which the size increases at every iteration, thus taking $\mathcal{O}(nk^2)$ time. In total, the for-loop takes $\mathcal{O}(mnk + nk^2)$ time. Computing the SVD of the $(k+1) \times k$ matrix B takes $\mathcal{O}(k^3)$ time. The matrix products to obtain U and V take $\mathcal{O}(mk^2)$ and $\mathcal{O}(nk^2)$ time respectively. This leaves us at $\mathcal{O}(mnk + nk^2 + k^2(k + m + n))$ time. For the applications discussed in this project, k will often be significantly smaller than n and m, which puts us at $\mathcal{O}(mnk)$ time.

The time taken in practice for the three different methods (GKLB with and without one-sided re-orthogonalization, and SVDS) for increasing values of k is shown in Figure 3. It is interesting to note that the time taken by the SVDS method increases rapidly until $k = 167$, then drops and stabilizes to 0.5 seconds, regardless of the value of k. Furthermore, as one might expect, the GKLB algorithm with re-orthogonalization is slower than without, although this is a small price to pay for the significant increase in numerical stability. We can also see that, when k is increased and m and n are kept constant, the time required for the GKLB methods seems to increase quadratically, as we would expect from our previous analysis.

The reason for the sudden drop in time taken by the SVDS algorithm is that, from $k > \frac{1}{3}\min\{m, n\}$ (in this case, $k > 167$), Matlab calculates the full SVD using a different method than SVDS. The full SVD method takes a constant amount of time regardless of the value of k, as the only difference is the number of singular triplets returned.

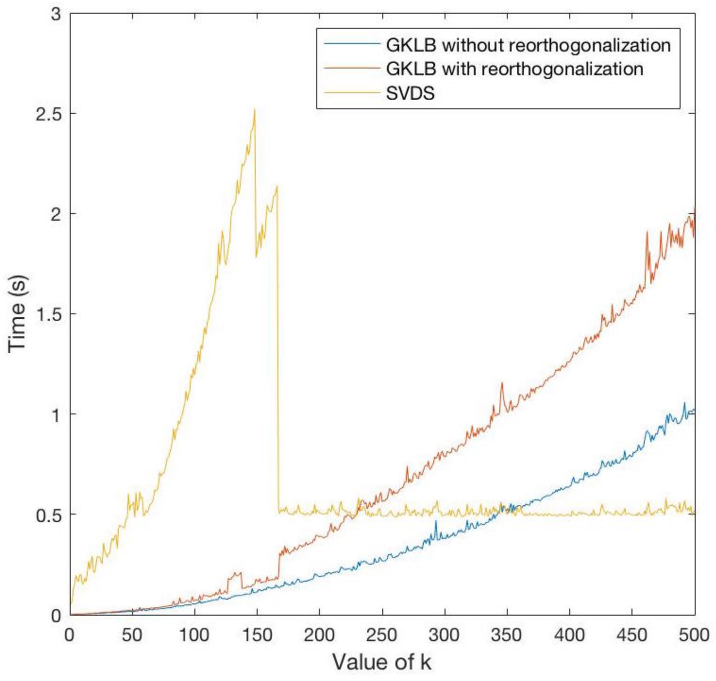


Figure 3: Time taken by GKLB and SVDS methods

For the remainder of this report, when GKLB is used for comparisons to other algorithms and large-scale applications, the reader may assume that one-sided orthogonalization is performed.

4 Power Method

The power method, also known as power iteration or Von Mises iteration, is an algorithm that iteratively approaches the largest eigenvalue λ_1 and corresponding eigenvector ξ_1 of a diagonalizable matrix A (von Mises & Pollaczek-Geiringer, 1929). This section will discuss this method and extend it to a power method for computing the truncated SVD.

4.1 Power Method for Eigenvalue Problems

If A is a symmetric $n \times n$ matrix with eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ and corresponding eigenvectors $\xi_1, \xi_2, \ldots, \xi_n$ such that $|\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_n|$, then A is diagonalizable as $A = V\Lambda V^T$ where $V = [\xi_1, \xi_2, \ldots, \xi_n]$ is orthogonal and $\Lambda = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$. Since V is orthogonal we have $V^T V = I$ and thus

$$\begin{aligned} A^k &= (V\Lambda V^T)(V\Lambda V^T)\dots(V\Lambda V^T) \\ &= V\Lambda^k V^T \end{aligned}$$

Now, if we choose a random unit vector $x_0 \in \mathbb{R}$ and we iteratively multiply by A so that $x_n = Ax_{n-1}$ then

$$\begin{aligned} x_1 &= Ax_0 = V\Lambda V^T x_0, \text{ and} \\ x_k &= A^k x_0 = V\Lambda^k V^T x_0 \\ &= \sum_{i=1}^{n} \lambda_i^k \xi_i \xi_i^T x_0 \end{aligned}$$

Setting $\xi_i^T x_0 = c_i$ where c_i is some constant, and dividing λ_1 out of the sum, we obtain:

$$x_k = \lambda_1^k \Big(c_1 \xi_1 + \sum_{i=2}^{n} c_i \Big(\frac{\lambda_i}{\lambda_1}\Big)^k \xi_i \Big)$$

Under the assumption that $|\lambda_1| > |\lambda_2|$ (i.e. there is one dominant eigenvalue) then it follows that $\big(\frac{\lambda_i}{\lambda_1}\big)^k$ converges to zero for increasing values of k. If this is the case, then we are left with

$$\begin{aligned} x_k &= A^k x_0 \\ &\approx \lambda_1^k c_1 \xi_1 \end{aligned}$$

If we normalize the vector x_i at each step such that $\|x_i\| = 1$, then the method eventually converges to a unit vector that lies in the same direction as the dominant eigenvector ξ_1, since λ_1 and c_1 are constants. We can then use the Rayleigh quotient to determine the dominant eigenvalue as $\lim_{k\to\infty} x_k^T A x_k = \lambda_1$.

The further away $|\lambda_1|$ lies from $|\lambda_2|$, the faster the convergence of the method. In the case that $|\lambda_1| = |\lambda_2|$, the method will converge to a vector that lies in the span of the first two eigenvectors.

If we wish to calculate the first m eigenvalues of A, then method can be extended to the block power iteration. This can be done in almost the same way, where we initialize a random $n \times m$ matrix X and apply the iteration rule $X_{k+1} = orth(AX_k)$. Here, normalization is replaced by orthogonalization.

4.2 Power Method for SVD

The power method for SVD, as proposed in (Bentbib & Kanber, 2015), is a simple extension to the method described above. Let A be an $m \times n$ matrix; we wish to approach the decomposition $A = U\Sigma V^T$. Now,

$$\begin{aligned} A^T A &= V\Sigma^T U^T U \Sigma V \\ &= V\Sigma^2 V^T \text{ and so,} \\ (A^T A)^p &= V\Sigma^{2p} V^T \end{aligned}$$

where we use Σ^i to denote $\text{diag}(\sigma_1^i, \sigma_2^i, \dots, \sigma_r^i, 0, \dots, 0)$ such that $\Sigma^i \in \mathbb{R}^{n\times n}$. Now, say we were to start with a random vector $x_0 \in \mathbb{R}$ and apply the iteration $x_j = A^T A x_j$, then

$$\begin{aligned} x_j &= (A^T A)^j x_0 \\ &= (V\Sigma^{2j} V^T) x_0 \\ &= \sum_{i=1}^{j} \sigma_i^{2j} v_i v_i^T x_0 \end{aligned}$$

Again, we can replace $v_i^T x_0$ by a constant c_i and factor out σ_1, resulting in

$$x_j = \sigma_1^{2j} \left(c_1 v_1 + \sum_{i=1}^{j} \left(\frac{\sigma_i}{\sigma_1} \right)^{2j} c_i v_i \right)$$

Again, under the assumption that $|\sigma_1| > |\sigma_2|$, the factor $\left(\frac{\sigma_i}{\sigma_1}\right)^{2j}$ will converge to 0 for $j \to \infty$, and so we are left with $x_j \approx \sigma_1^{2j} c_1 v_1$, i.e. x_j is a vector in the same direction as the first right singular vector v_1. Since v_1 is a unit vector, normalizing x_j such that $\|x_j\| = 1$ will result in $\lim_{j\to\infty} x_j = v_1$.

On its own, v_1 is not of much use to us; we also want σ_1 and u_1. For this, we consider that for the i-th singular triplet, $Av_i = \sigma_i u_i$. Since u_i are unit vectors and all σ_i are constants, we find

$$\sigma_1 = \|Av_1\|, \text{ and}$$
$$u_1 = \frac{1}{\sigma_1} A v_1$$

Combining the previous with the relation $A^T u_1 = \sigma_1 v_1$ results in Algorithm 3.

Algorithm 3: Power method for first singular triplet of matrix A

```
1 Choose $v_{1_0} \in \mathbb{R}^n$ randomly with $\|v_0\| = 1$
2 for $i = 1$ to $j$ do
3     $\sigma_{1_i} = \|Av_{1_{i-1}}\|$
4     $u_{1_i} = \frac{Av_{1_{i-1}}}{\sigma_{1_i}}$
5     $v_{1_i} = \frac{A^T u_{1_i}}{\|A^T u_{1_i}\|}$
6 end
```

As with the eigenvalue power method, this method can be simply extended to calculate the first singular triplets of an $m \times n$ matrix A, resulting in Algorithm 4.

Algorithm 4: Block power method for k singular triplets of matrix A

```
1 Choose orthogonal matrix $V_0 \in \mathbb{R}^{n\times k}$ randomly
2 for $i = 1$ to $j$ do
3     $U_{k_i} = orth(AV_{i-1})$
4     $V_{k_i} = orth(A^T U_i)$
5 end
6 $\Sigma_k = U_{k_j}^T A V_{k_j}$
7 $S = sign(\Sigma_k)$
8 $\Sigma_k \leftarrow S\Sigma_k$
9 $U_k \leftarrow U_{k_j} S$
```

Normalization is again replaced by orthogonalization, as we know that the columns of U and V should be orthonormal. The algorithm eventually converges to $U_k \approx U_{k_j}$, $V_k \approx V_{k_j}$ and Σ_k. The resulting SVD is not strictly correct, especially considering the definition of Σ in Section 2. We find that the singular values are only somewhat in decreasing order, and some of the singular values are negative. For this reason, the last step on lines 7-9 of the algorithm is added as it ensures that the singular values in Σ_k are positive. It does so by locating the negative singular values, and changing the sign of the corresponding columns of U.

The method is tested for a sparse 5000×500 matrix A with density 4%. The corresponding Matlab code can be found in Appendix A.3. The results shown are the average of 5 runs for the same matrix, due to random initialization of V_0. We will first look at the convergence of the method, measured by $\|A_k - \hat{A}_k\|$, where A_k is the true k-rank truncated SVD and $\hat{A}_k$ is the approximation achieved by the block power SVD. Figure 4 shows that the difference converges to 0 when the number of iterations is increased, as desired, for $k = 50$.

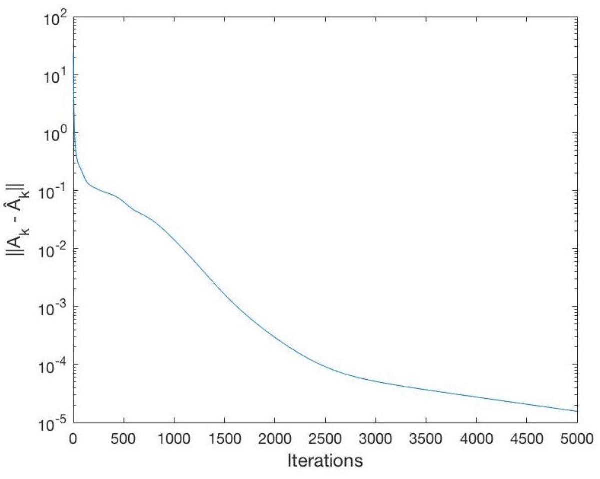


Figure 4: Convergence of block power method

The error of the k-rank approximation is given by $\|A - A_k\|_2 = \|A - U\Sigma V\|_2$. As we have seen in Section 2, it holds that

$$\min_{\text{rank}(X)\leq k} \|A - X\|_2 = \|A - A_k\| = \sigma_{k+1}$$

where σ_k is the k-th largest singular value of A. This provides us with a lower error bound, which is achievable by a perfect SVD computation.

We first measure the error $\|A - A_k\|_2$ for increasing values of k, when 5, 20, 50 and 100 iterations are performed, as shown in Figure 5. The plot also shows the minimum reachable error σ_{k+1}. We see that the method has very unstable behaviour, especially for larger values of k, when only 5 iterations are performed. However, the performance improves quickly when more iterations are performed, and there is quite minimal difference between the error for 50 and 100 iterations, and the minimum possible error.

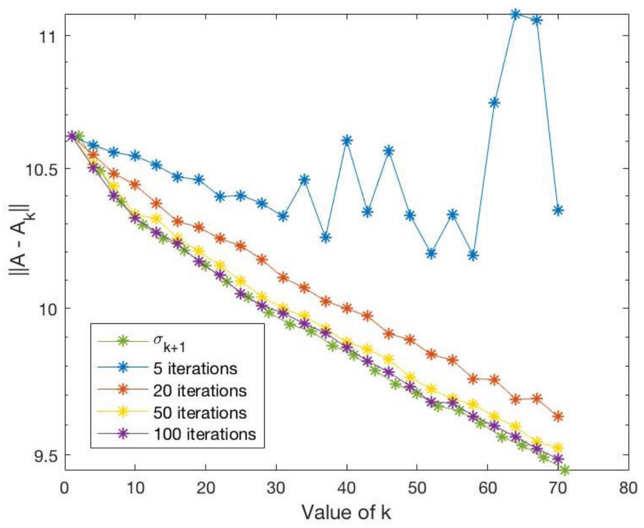


Figure 5: Performance of block power method

We can also see that the time increases (almost) linearly with the number of iterations in Figure 6, which makes sense since each iteration should take the same amount of time. Now, considering that the errors for 50 and 100 iterations were so similar, and 100 iterations would take twice as long as 50, we see that it may not be worth performing more than 50 iterations in this case when on a time limit. For the remainder of this report, when block power SVD is used for comparisons to other algorithms and large-scale applications, the reader may assume that 50 iterations are performed.

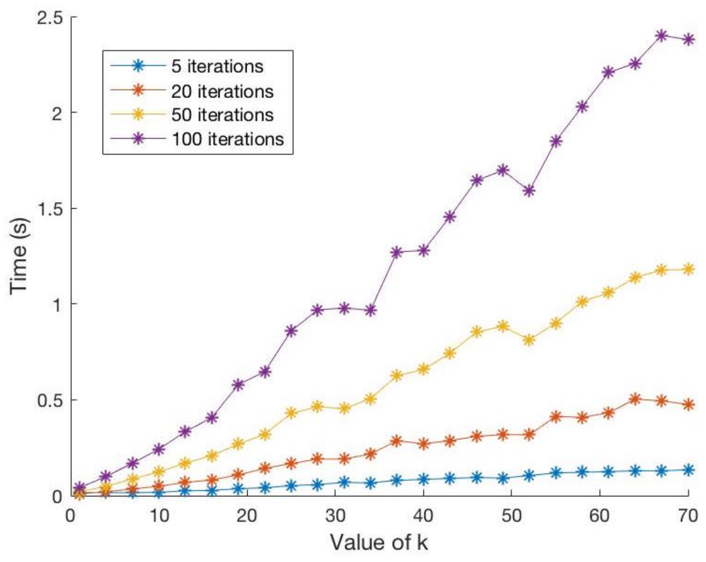


Figure 6: Time taken by block power method

5 Randomized SVD

Another method for computing the truncated SVD is the randomized SVD algorithm proposed in (Halko, Martinsson, & Tropp, 2011). The method computes the decomposition $A_k = U_k \Sigma_k V_k^T$ by using random sampling to approximate the range of A. The algorithm can be split into two distinct stages:
Stage A: Compute an approximate basis for the range of input matrix A, i.e. a matrix Q for which Q has orthonormal columns and $A \approx QQ^T A$. Q should have less columns than A.
Stage B: Given matrix Q, use $B = Q^T A$ to compute the SVD factorization of A.
Stage A is done using random input, while Stage B is completely deterministic.

First, some intuition will be provided concerning the use of randomness for Stage A. In this stage, we desire a basis for the range of matrix A with rank$(A) = k$. Now, draw random vectors ω_i, $i = 1, 2, \ldots, k$ and form the product $y_i = A\omega_i$, $i = 1, 2, \ldots, k$. Due to randomness, it is likely that $\{\omega_i : i = 1, 2, \ldots, k\}$ is a linearly independent set, and so the sample vectors y_i are also linearly independent. Also, each y_i is in the range of A, and so we can conclude that $\{y_i : i = 1, 2, \ldots, k\}$ spans the range of A. To produce an orthonormal bases we simply need to orthonormalize the vectors y_i.

Now say we want to approximate the range of a lower-dimensional subspace with dimension $k <$ rank(A). If we generate $k + p$ sample vectors, we obtain the enriched set $\{y_i : i = 1, 2, \ldots, k + p\}$, which has a higher probability of spanning the required subspace. (Halko et al., 2011) find that typically, small values of p already work quite well for this purpose.

Using this intuition, we obtain Algorithm 5 to compute the desired k-rank matrix Q for an $m \times n$ matrix A, with oversampling parameter p.

Algorithm 5: Stage A

```
1 Choose $\Omega \in \mathbb{R}^{n\times(k+p)}$ randomly
2 $Y \leftarrow A\Omega$
3 $Q \leftarrow orth(Y)$
4 **return** $Q$
```

However, when the singular value spectrum of A decays too slowly, so will the approximation error. To overcome this problem, (Halko et al., 2011) propose performing a small number of power iterations on AA^T, as shown in Algorithm 6. The motivation behind this is that if the singular values of A are Σ, then the singular values of $(AA^T)^q$ are Σ^{2q+1}, which drives down the spectrum, and with it the error, exponentially. In practice, $q = 1$ or 2 works sufficiently.

Algorithm 6: Stage A with power iteration

```
1 Choose $\Omega \in \mathbb{R}^{n\times(k+p)}$ randomly
2 $Y \leftarrow (AA^T)^q A\Omega$
3 $Q \leftarrow orth(Y)$
4 **return** $Q$
```

The method for Stage B is shown in Algorithm 7. When $B = Q^T A$ we obtain the low-rank approximation $A \approx QB = Q\tilde{U}\Sigma V^T$. Note that B is a $(k+p)\times m$ matrix, while A is $m \times n$ and $(k+p) \ll n$, so the SVD of B is much easier to compute than that of A.

Algorithm 7: Stage B

```
1 $B \leftarrow Q^T A$
2 Compute the SVD: $B = \tilde{U}\Sigma V^T$
3 $U \leftarrow Q\tilde{U}$
4 **return** $U, \Sigma, V$
```

As with GKLB, the time complexity for randomized SVD is $\mathcal{O}(mnk)$, although it can be reduced asymptotically to $\mathcal{O}(mn \log k)$ by using a specially structured random sampling matrix, such as the subsampled random Fourier transform. Furthermore, only a constant number of passes over A are required, as opposed to $\mathcal{O}(k)$ passes for GKLB. Especially in the case of huge input matrices, this can make a significant difference in the time required for computing the SVD.

We again measure the error using $\|A - A_k\|_2$, of which the lower bound is σ_{k+1}. The expected error of randomized SVD, for computing the k-rank approximation of matrix A with oversampling parameter p and q power iterations, is (Erichson, Voronin, Brunton, & Kutz, 2016):

$$\mathbb{E}\|A - U\Sigma V^T\| \leq \left[1 + \sqrt{\frac{k}{p+1}} + \frac{e\sqrt{k+p}\sqrt{\min\{m,n\}-k}}{p}\right]^{\frac{1}{2q+1}} \sigma_{k+1}$$

We can test the implementation on a sparse 5000×500 matrix of 4% density, as in Section 3. The Matlab code for randomized SVD can be found in Appendix A.2. The results shown are the average of 5 runs for the same matrix. Figure 7 shows the effect of using oversampling, where we use $q = 1$ as the baseline for power iterations. We see that the error is indeed bounded from below by the singular values. We also see a slight improvement in performance as p is increased, especially for low values of k.

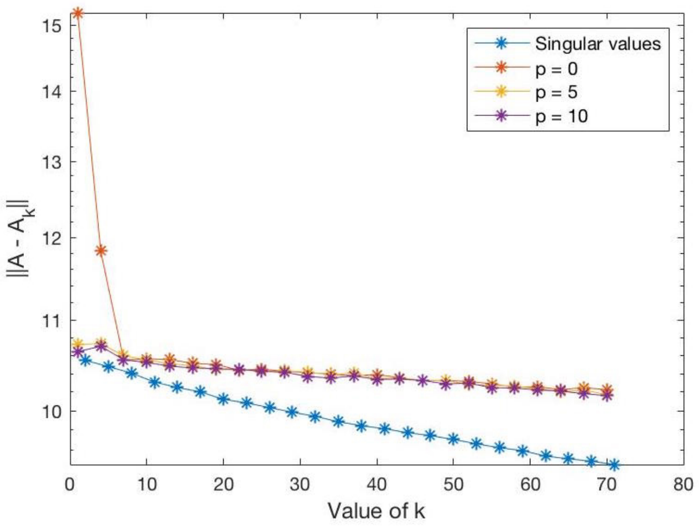


Figure 7: Performance of randomized SVD for different values of p

Figure 8 shows us the importance of the power iteration, where $p = 5$ is used as the baseline for oversampling. When the power iteration is not performed at all, the error is significantly higher than when it is performed once or twice. The difference between $q = 1$ and $q = 2$, however, is quite small, and may not be worth it due to the substantial increase in time taken by the algorithm, caused by two more multiplications of the large and sparse matrix A.

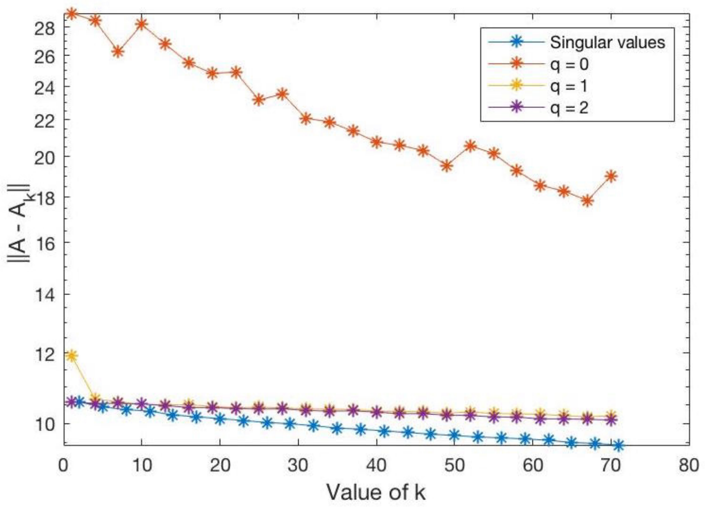


Figure 8: Performance of randomized SVD for different values of q

For the remainder of this report, when randomized SVD is used for comparisons to other algorithms and large-scale applications, the reader may assume that $q = 1$ power iteration is performed and we use the oversampling parameter $p = 5$.

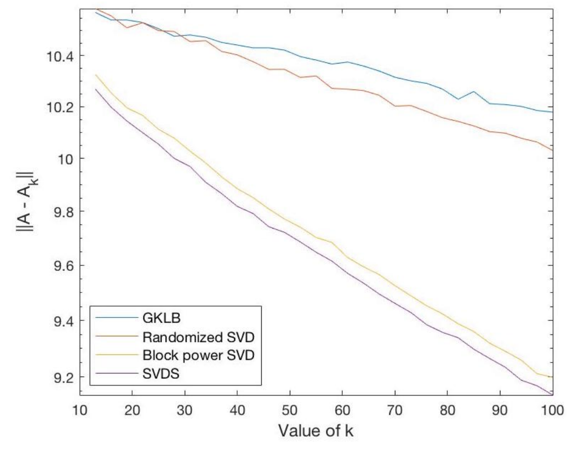
6 Comparison of methods

In this section, the GKLB, randomized SVD and block power algorithms will be compared to the Matlab SVDS algorithm in order to test which is most suitable for further use in large-scale applications. For testing, both a 5000×500 matrix of 4% density, as in the previous sections, and a full 5000×500 matrix are used, as some applications that will be discussed also require the SVD of a full matrix, such as image processing or the ARF algorithm from Section 13.

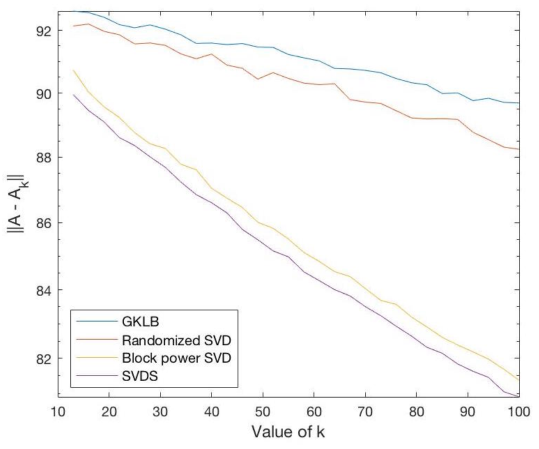
For GKLB, one-sided re-orthogonalization is used, and for randomized SVD the oversampling parameter p is set to 5, and $q = 1$ power iteration is performed. For block power SVD, 50 iterations are used in order to balance error with time taken. The methods are all run 5 times for values of k between 5 and 100. We have seen that some methods, such as GKLB, show instability for very low values of k. This skews the axes and doesn't allow for closer comparison of the methods for higher values of k. The reason we stop at $k = 100$ is that the applications discussed further on use the truncated SVD, so we are not interested in relatively higher dimensions.

6.1 Error

In previous sections, different error measures have been used to measure performance of the various algorithms. In order to compare the methods, we will measure error using $e_k = \|A - A_k\|_2$, remembering that $\min_{rank(X)\leq j} \|A - X\|_2 = \sigma_{j+1}$. Figure 9 shows the error of each method for the sparse and full matrices.



(a) Sparse matrix ($\|A\| = 32.19$)



(b) Full matrix ($\|A\| = 92.75$)

Figure 9: Performance of algorithms discussed

We see that the relative performance is very similar for both sparse and full matrices. The performance of the block power SVD almost matches that of SVDS; more iterations would have brought the difference even closer together. The randomized SVD and GKLB algorithms also have similar performance, although worse than the other two methods. It is interesting that the GKLB algorithm has much worse performance than SVDS, despite SVDS being based on GKLB. Closer inspection of Matlab's SVDS code proves that it contains many checks for convergence and two-sided reorthogonalization, which would explain the superior performance.

It is also important to realize why there is such a difference in performance between the algorithms. In randomized SVD, error can originate in the multiplication of AA^T. In practice, this multiplication of large and sparse matrices can lead to significant numerical errors and should be avoided.

Another portion of this difference lies in the distribution of the singular values, and in the way they are calculated. In GKLB and randomized SVD, the first k singular values of the input matrix A are found by computing the SVD of a smaller k-rank matrix B. This is not the case for the block power SVD, since there we compute $\Sigma_k = U_k^T A V_k$. The result of these different approaches is best seen by comparing the singular value spectra found by each algorithm. The singular values found by each algorithm for $k = 50$ for a random sparse 5000×500 matrix A are shown in Figure 10, compared to the true singular values found by Matlab's SVD.

At first sight, it seems as though the singular values found by the block power SVD are not shown in Figure 10; however, the line is simply difficult to see as it almost exactly follows the line of the true singular values, showing why block power SVD performed so well in our error tests. The randomized SVD method appears to maintain a rather constant difference between the singular values found and the true value. The singular values found by the GKLB method show rather different behaviour, as they first appear to match the true values, but decline quickly. The reason for this behaviour is that GKLB approaches the singular values of A by calculating the singular values of the matrix B, which is bidiagonal. We see that the first few singular values of B correspond closely to the true values.

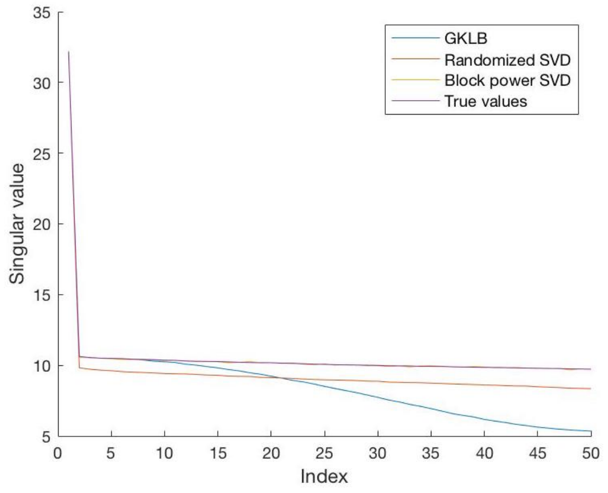


Figure 10: Singular values of A found by algorithms discussed for $k = 50$

We should also examine the difference between the true and the calculated singular values when a full SVD is performed using each algorithm. This is shown in Figure 11. Since we saw a significant drop in the error of GKLB between $k = 499$ and $k = 500$, both results are shown here to compare where this drop originates, as GKLB 500 and GKLB 499, respectively.

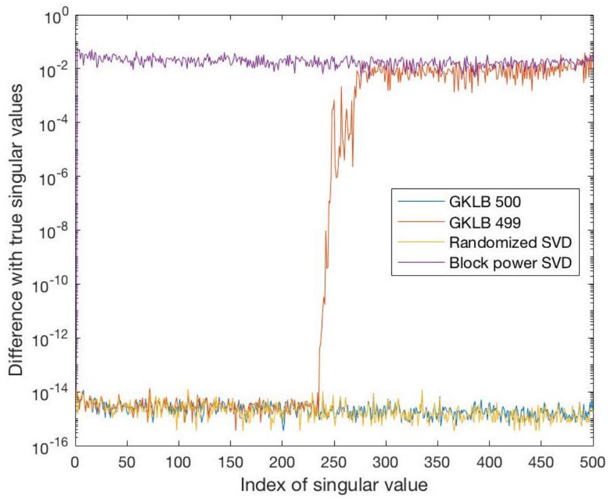
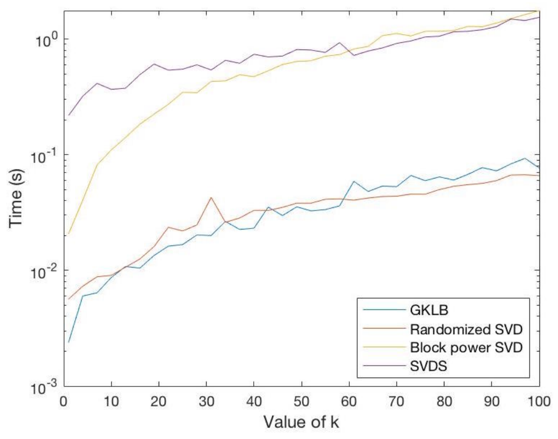


Figure 11: Difference between true singular values and singular values found by algorithms for full SVD

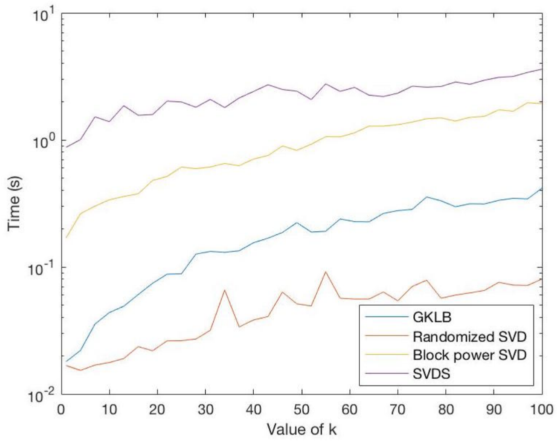
Contrary to the results we saw earlier, GKLB and randomized SVD now seem much more effective in calculating the SVD than the block power SVD. Also, we see that GKLB with 499 iterations performs very well for approximately the first half of the singular values, and the error suddenly increases for the second half, while GKLB with 500 iterations shows essentially no error in the singular values calculated, explaining the error drop. However, it is important to realise that in the case of both GKLB and randomized SVD, the SVD of a matrix B with $\text{rank}(B) = 500$ must be calculated as one of the last steps- which is what we were trying to do in the first place!

6.2 Time

For large-scale applications, it is of importance that the algorithm used is very efficient. For example, Facebook has suggested using SVD for their adjacency matrix of Facebook users to Facebook pages induced by likes, with size $\mathcal{O}(10^9) \times \mathcal{O}(10^8)$ (Tulloch, 2014), and it would not be practical to incur a waiting time of possibly hours for computation of the SVD only. The time taken for each algorithm is shown in Figure 12.



(a) Sparse matrix



(b) Full matrix

Figure 12: Time taken by algorithms discussed

There is more difference between the algorithms when it comes to time taken. As one would expect, the time taken for the truncated SVD of a full matrix is generally longer than that of a sparse one. While for a sparse matrix, the time taken by GKLB and randomized SVD is very similar, randomized SVD clearly outperforms the other algorithms when it comes to full matrices. We also see that for small values of k, the block power SVD takes much less time than SVDS, which may make it desirable to choose this method over SVDS for specific cases due to similarity in performance.

Part II
SVD for Information Retrieval

This part will introduce the first large-scale application of the singular value decomposition: information retrieval. To be more specific, we will be focussing on the Latent Semantic Indexing (LSI) method, which is used for searching documents in a large database. The method was originally described in (Berry, Dumais, & O'Brien, 1995).

In that time, traditional search engines, which were most commonly used for retrieving documents from scientific databases, relied on matching the words in a user's query to the words contained in a document. This method is very bad at handling two language-associated problems: *synonymy* and *polysemy*. Synonymy implies that there are many ways to describe the same object or concept. There is an enormous variation in the words people use to describe a document: it has been found that two people choose the same keyword for a well-known object less than 20% of the time (Furnas, Landauer, Gomez, & Dumais, 1983). Not using the correct words to describe a document when using such a traditional search engine leads to very poor recall, and often the user will not be able to find the document in question. Therefore, an ideal search engine would understand, for example, that *car* means the same as *automobile*.

Polysemy occurs when a word has two different meanings. A gardener searching for *tree* may not be so interested in results about graphs that don't contain cycles. Thus our ideal search engine should also be able to recognize the conceptual meaning of the query and documents. As we will see, LSI uses the truncated SVD to handle both of these problems quite well.

7 Latent Semantic Indexing method

This section will explain the LSI method described in (Berry et al., 1995). The method first requires construction of a data matrix A representing the documents and terms in the database we want to search through. Given such a database of documents D, containing a set of terms T, with $|D| = n$, $|T| = m$, we can construct a data input matrix A by setting $A_{t,d}$ to the frequency of word $t \in T$ in document $d \in D$. A is then factored into U, Σ and V using the SVD and subsequently the dimensionality is reduced to k dimensions by constructing the truncated SVD as in Equation 2. The resulting factors can be referred to as the word-feature matrix U_k, document-feature matrix V_k and truncated singular value matrix Σ_k. Here, k is the suspected number of features, or concepts, present in the data.

Each row of A represents a term vector, signifying the frequencies with which that term appears in each document. The corresponding rows of U_k represent term-feature vectors for each term. Similarly, each column of A represents a document vector, containing the frequencies with which each term appears in that document, and so the corresponding rows of V_k represent document-feature vectors for each document. The truncated SVD has mapped the terms and documents to the same k-dimensional space, which will be referred to as the *semantic space*.

For example, the word *phone* might be a represented as a combination of the following features:

$$\textit{phone} = 40\% \text{ technology} + 30\% \text{ communication} + 15\% \text{ social media} + 5\% \text{ photography} + \dots$$

Here, the percentages are indicative of the weights of the corresponding features in the term-feature vector of *phone*. Now, when searching for documents about phones, we are interested in documents of which the document-feature vectors have similar weightings. In practice, we will not know which dimension represents which feature, and this is also unnecessary in most applications.

7.1 Semantic effects of reduction

The dimension reduction from A to A_k removes much of the noise present in the original data. As we have seen in Section 2, the last $m - k$ columns of U and last $n - k$ columns of V contribute least to the variance of the data. Thus the removal of these columns can be seen as the removal of noise from the data, which can reveal the underlying latent semantic structure. Furthermore, for a very large database, m and n can be very high, reaching to hundreds of thousands, or even in the millions. Experimental results for the optimal value of k for such large databases, however, have often ranged in the mere hundreds. This results in much smaller quantities of data to be stored, searched through and manipulated.

This reduction has two effects desirable for natural language processing. The first is that similar terms will be close together in the k-dimensional space, even when they never occur in the same document. For example, one document may use the word *car* and another might prefer *automobile*. However, it is likely that both documents contain related words such as *steering wheel*, *drive*, *wheel* and so forth. Based on these similarities, the words *car* and *automobile* will be close together in k-dimensional space, so that a query for *car* will also return documents containing *automobile*. This handles the problem of synonymy.

The second effect is that documents with only a small overlap in the same terms will not be close together in k-dimensional space. It is desirable that a user looking for computer chips, for example, does not get results about Pringles. Given a large enough document database, LSI should be able to place a query for *computer chips* in a very different context than a query for *potato chips*, while traditional methods would consider these queries very similar. As a result, LSI is much more capable of handling polysemy than traditional methods. To summarize, LSI is based on conceptual information retrieval rather than literal matching of document terms with queries.

7.2 Queries

A query can be represented similarly to a document, as a vector $q \in \mathbb{R}^m$ where element q_t is the frequency of term $t \in T$ of the query. In order to represent the query in the semantic space, we apply the transformation

$$\hat{q} = q^T U_k \Sigma_k^{-1}$$

The component $q^T U_k$ represents a summation over the k-dimensional term vectors, and subsequent multiplication by Σ_k^{-1} applies weights to the different dimensions.

Now that the query is represented in the same semantic space as the documents, the most similar, and thus relevant, documents can be returned to the user. Similarity is expressed by comparing which documents are closest to the query, which is typically done using the cosine similarity measure, defined as

$$\text{similarity}(q, d) = \frac{q \cdot d}{\|q\| \|d\|}$$

7.3 Folding-in

When a new document is added to the database, it must be incorporated into the document matrix V_k, and any new terms must be incorporated into the term matrix U_k. Computing a new SVD from scratch, however, is very time-consuming and impractical, sometimes even impossible, for large databases. To solve this issue, (Berry et al., 1995) propose a method called *folding-in*, whereby the original document and term matrices are updated to hold the new terms and documents.

Folding in a document essentially follows the same process as posing a query. In order to fold in a new document vector $d \in \mathbb{R}^m$ into an existing LSI model, compute

$$\hat{d} = d^T U_k \Sigma_k^{-1}$$

Now, $\hat{d}$ is a weighted projection of d onto the span of the current term vectors, and can be added to the rows of V_k. In the same way, folding in a new term t is done by computing the projection

$$\hat{t} = t V_k \Sigma_k^{-1}$$

While having the benefit of avoiding the expensive SVD recomputation, folding-in is not an ideal process. This is because the new terms and documents have no effect on the existing term and document representations. Eventually, folding-in many documents can drastically decrease the quality of the LSI database.

8 Data representation

Previously, we have assumed that our data input matrix A contained the raw frequencies of words in documents, given by

$$A_{t,d} = f_{t,d}$$

where $A_{t,d}$ denotes the entry in the tth row and dth column of A, and $f_{t,d}$ denotes the frequency of term t in document d. However, this does not take into account that shorter documents will contain less words, or that some words, called *stop words*, occur much more than others in all documents, such as *the*, *and*, and *of*. To solve this problem, *term frequency-inverse document frequency*, or tf-idf, can be used in order to apply weightings to terms (Manning, Raghavan, & Schütze, 2008).

8.1 Term frequency

A number of methods can be discerned to apply a local weighting to the terms within a document, which will be discussed here.

1. Raw term frequency: $tf_{t,d} = f_{t,d}$
2. Boolean frequency: $tf_{t,d} = \begin{cases} 1 & \text{if term } t \in \text{document } d \\ 0 & \text{otherwise} \end{cases}$
3. Logarithmically scaled frequency: $tf_{t,d} = \begin{cases} 1 + \log f_{t,d} & \text{if } f_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$
   This method takes into account that a term appearing 20 times more often than another term in a document is not strictly 20 times more important.
4. Normalized frequency: $tf_{t,d} = \alpha + (1 - \alpha)\frac{f_{t,d}}{\max_{t \in d} f_{t,d}}$
   This method normalizes the term frequency weights over all terms occurring in the document. Here, α is a smoothing parameter between 0 and 1, typically set to 0.4.

8.2 Inverse document frequency

The idea of inverse document frequency is to scale down the term frequency of a term, based on how often this term appears in the entire document collection. In this way, terms which are very specific to certain documents will become more relevant, and the weightings of stop words are scaled down drastically. Thus we define

$$idf_t = \log \frac{N}{df_t}$$

where N is the number of documents in the database and df_t is the document frequency of term t, i.e. the number of documents containing t. Note that if a term is present in all documents in the database, then $N = df_t$ and so $idf_t = 0$.

Applying these methods now allows for construction of a weighted input data matrix A with

$$A_{t,d} = tf_{t,d} \times idf_t$$

Now, words (such as stop words) that are present in every document, while they may have a very high term frequency, will have a tf-idf of 0. This is very useful, since it removes the need to keep a "blacklist" of stopwords, a so-called *stop list*, that the search engine should not take into account. Furthermore, words which are present in only a few documents will have a relatively high weighting, which is desirable since these can carry much more information about document context and topic.

9 Small-scale LSI examples

In this section, two small-scale examples of LSI implementation will be shown. The first example was created for the purposes of this paper in order to demonstrate LSI, tf-idf and queries, and the second is copied from (Berry et al., 1995) in order to validate the implementation.

9.1 Example 1

We will consider the following documents:

- d_1 is about a smartphone with a navigation app.
- d_2 is about a mobile phone.
- d_3 is about a car with a navigation app.
- d_4 is about an automobile.

Furthermore, since d_1 and d_2 are about phones, they both contain the words *call* and *app*, while d_3 and d_4 both contain the words *transport* and *navigation* since they are both about cars. Using boolean term frequency, we obtain the following table:

Table 1: Term-document matrix corresponding to Example 2

	d_1	d_2	d_3	d_4
smartphone	1	0	0	0
mobile	0	1	0	0
call	1	1	0	0
app	1	1	1	0
car	0	0	1	0
automobile	0	0	0	1
transport	0	0	1	1
navigation	1	0	1	1

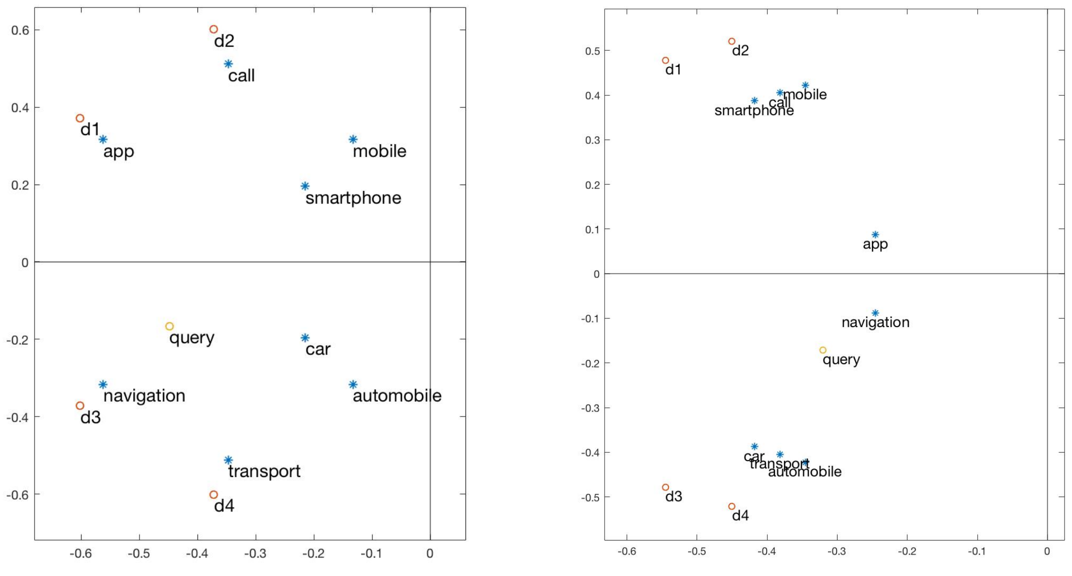
From the table, we can see two distinct categories: the first four terms are about phones, while the last four are about cars. There is a small amount of noise, due to two documents from different categories also containing information about a navigation app.

Computing the 2-dimensional truncated SVD of this matrix gives

$$A_2 = \begin{bmatrix} -0.215 & 0.195 \\ -0.133 & 0.316 \\ -0.347 & 0.512 \\ -0.562 & 0.316 \\ -0.215 & -0.195 \\ -0.133 & -0.316 \\ -0.347 & -0.512 \\ -0.562 & -0.316 \end{bmatrix} \begin{bmatrix} 2.803 & 0 \\ 0 & 1.902 \end{bmatrix} \begin{bmatrix} -0.602 & -0.372 & -0.602 & -3.72 \\ 0.372 & 0.602 & -0.372 & -0.602 \end{bmatrix} = U_2 \Sigma_2 V_2^T$$

Now, each word is represented by its corresponding row of U_2 and each document by its corresponding row in V_2. For example, *smartphone* corresponds to $(-0.215, 0.195)$, while d_3 corresponds to $(-0.602, -0.372)$. Thus each word and document is a coordinate (x, y) which can be plotted in order to visualize the semantic space, as shown in Figure 13a. The figure also contains the projection for the query *automobile navigation app.*

We see that the words from the first two documents about phones are all in the upper quadrant, while the words from the other documents are in the lower quadrant. Furthermore, the words *smartphone* and *mobile* are very close together, while those words do not occur in the same documents. The same goes for the words *car* and *automobile*. Remember that similarity of words and documents is determined by their cosine similarity, so it is important to compare words based on the angle they make relative to the origin, rather than the Euclidian distance between two words. Using this, we see that *transport* is also very similar to *car* and *automobile*.



(a) Without IDF

(b) With IDF

Figure 13: 2-dimensional plot of terms and documents corresponding to Example 1

Figure 13b shows the results of also applying inverse document frequency to the term-document matrix. We can see that this gives a much stronger clustering effect, placing the phone-related words very close to the first two documents, and the car-related words very close to the last two. There is also a significant change in the location of *app* and *navigation*, which are placed closer together since they co-occur in two documents. However, *app* lies more in the direction of the phone documents since it appears in both of those, while for the same reason *navigation* lies more in the direction of the car documents.

Another important thing to note is that the query for *automobile navigation app* is much closer to d_3 and d_4 than to d_1 and d_2, even though d_1, d_3 and d_4 each contain exactly two words from the query. We obtain the following cosine similarity ranking:

1. d_3 with similarity$(query, d_3) = 0.9738$
2. d_4 with similarity$(query, d_4) = 0.9332$
3. d_1 with similarity$(query, d_1) = 0.3528$
4. d_2 with similarity$(query, d_2) = 0.2206$

A traditional search method which only compares the overlap between words in the query and in the documents would return d_1, d_3 and d_4 with equal ranking. However, the LSI method appears to have given a higher weight to *automobile* than to *navigation* and *app*, thus giving a very high similarity score for the documents about cars, and a much lower score for the other documents.

9.2 Example 2

The example from (Berry et al., 1995) considers a database of 17 mathematical book titles, which can be found in Appendix E.1. The corresponding term-document matrix can be found in Appendix E.2.

From this matrix, we can compute the 2-dimensional reduced SVD $A_2 = U_2 \Sigma_2 V_2^T$ of the term-document matrix and plot the terms and documents as in Example 1.

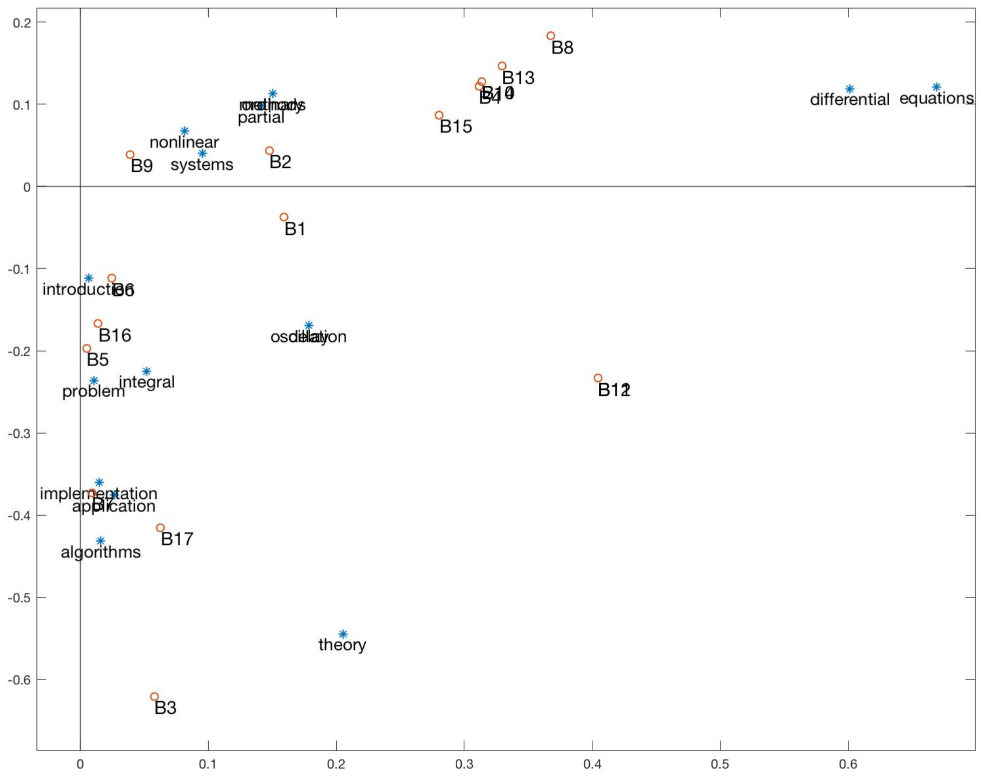


Figure 14: Term-document plot generated for Example 2

The plot from (Berry et al., 1995) can be found in Appendix E.3; we can see that the coordinates for the words and documents are the same, thus validating the implementation. Furthermore, we can see some interesting word groupings, such as *differential equations*, *nonlinear systems*, *oscillation delay* and *ordinary partial methods*.

10 Text representation

The LSI method uses the *bag-of-words* model for document representation. This means that it only takes the term frequencies into account and not the order in which they appear in the document. Thus it makes no distinction between the sentences *the cat ate the mouse* and *the mouse ate the cat*, or between *A not B* and *B not A*. For the purposes of comparing documents this will not hinder us too much, since it is safe to assume that documents about similar topics will contain a similar bag of words.

Unfortunately, the process of turning a document into a bag of words is not simple. Two important steps must be made: tokenization and normalization (Manning et al., 2008). Tokenization is the process of splitting up a piece of text into its constituent terms, or tokens. A simple approach is to split up the words along spaces and punctuation such as , . : ; ! and ?. However, we can run into problems with apostrophes or hyphens: *state-of-the-art* should become *state of the art*, while *thirty-two* should become *thirtytwo*. Another difficulty is text containing dates or website links, as we would want *brightspace.tudelft.nl* to remain as one token while standard tokenization splits this into *brightspace tudelft nl.*

After having tokenized the text, the tokens must be normalized. This is especially important for determining term frequencies, since we do not want to count, for example, *run* and *Running* as different terms, even though these are very different strings. To do so, we must create equivalence classes of words, so that we can classify tokens into sets of words. For example, the equivalence class for *word* should also contain *Word, words, Words* and so forth. There are a number of approaches that should be considered:

1. Capitalisation: an obvious strategy is to reduce all the letters to lowercase, also called *case-folding.* This works well in most cases, although it may cause problems for names or organisations, which should be capitalized regardless of their place in a sentence or enthusiasm of the author.

2. Removing diacritics, such as classifying *naïve* and *naive* in the same equivalence class, is fairly intuitive for the English language. In some other languages, however, the meaning of a word can be completely dependent on an accent.

3. Stemming: a document can contain many different grammatical instances of the same word, such as *analyze*, *analysis* and *analytical.* Stemming reduces words to a common base form, often by removing the characters at the end of words according to a set of rules. For example, a word ending in *-ss*, such as *dress*, should not be altered, while a word ending in a single *s* should have that *s* removed. There are many difficulties with stemming, as this example rule will turn *leaves* into *leave* when it should become *leaf*, while *caves* should not become *caf.* The most commonly used stemming algorithm for English is the Porter Stemming algorithm.

Fortunately, both the Matlab Text Analytics toolkit as the Python Natural Language toolkit contain all text-standardization functions described above. The Matlab code constructed for preprocessing documents can be found in Appendix B.1.

11 Large-scale implementation of LSI search

In this section, we will demonstrate a large-scale implementation of Latent Semantic Indexing on the Time collection (Krovetz, 1988). This is a collection of 423 articles from Time magazine about a variety of subjects. The collection also includes a set of 83 queries and relevance judgments as to which documents are relevant to each query. These can be used to test the implementation.

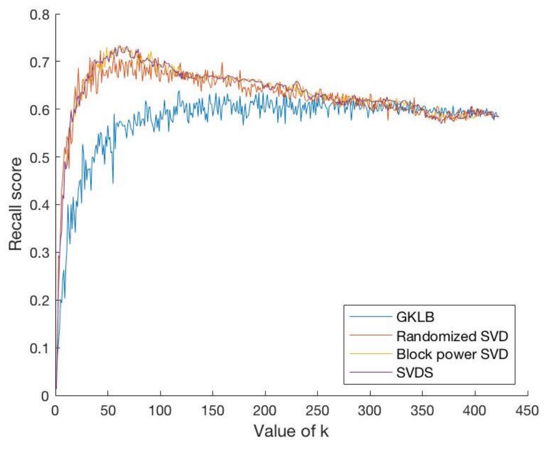
First, the documents are preprocessed according to the methods described in Section 10, using the Matlab code in Appendix B.1. From the resulting bag of words, a tf-idf matrix is created using logarithmic term frequency as described in Section 8. The queries, too, are preprocessed and the documents and query vectors are translated to k-dimensional space using the LSI method as described in Section 7. For each query, we can return the 20 most relevant documents using the k-nearest neighbors method. The returned documents are then compared to the expected relevant documents, and we can calculate the recall score of the method using:

$$\text{Recall} = \frac{\#\text{ Relevant documents retrieved}}{\#\text{ Relevant documents}}$$

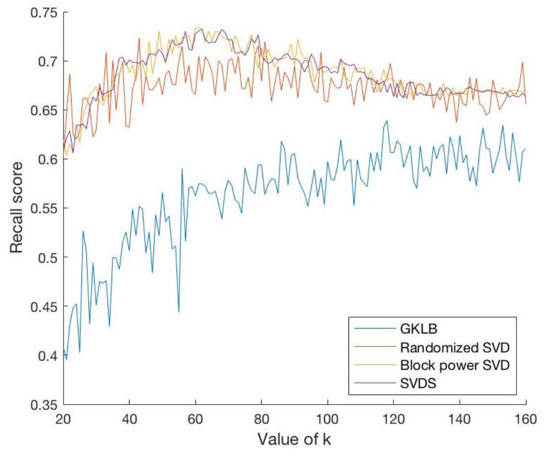
Note that, when the number of returned documents is increased, the recall score will also increase. For example, returning all the documents will result in a recall score of 100%. For the purposes of this experiment, the value of 20 is chosen so that a small number of documents is returned relative to the large size of the database, taking into account that the maximum number of relevant documents for a query in the Time collection is 18. The relevant Matlab code for LSI search and calculation of the recall score can be found in Appendix B.2.

Now we can calculate the recall score of the method for different values of k, in order to find the optimal value for this document collection. Here, k ranges from 1 to 423, the number of documents in the collection. For this, we can use the different algorithms discussed in Part 1, in order to test their performance on this real-life test problem rather than the random sparse matrix used previously. This may also show us what effect the errors incurred by the algorithms have on their usefulness in large-scale applications. After all, a small change in the document-feature vectors can strongly affect which documents are returned first.

The results are shown in Figure 15. The recall score shown is the average of the recall scores for each of the 83 queries posed for that value of k. We see that, in general, the recall score increases dramatically initially. All methods but GKLB then peak around $k = 60$ and slowly decrease afterwards. This decrease in performance shows that the dimension reduction provides us with more information about the latent semantic structure of the document database than the tf-idf matrix alone.



(a) Full plot



(b) Close up plot

Figure 15: Recall of LSI method for different algorithms and different values of k

In Section 6 we saw that the error incurred by the block power SVD and SVDS methods were very similar for relatively low values of k. Now, we also see that their performance for LSI barely differs. The block power SVD even performs ever so slightly better at its highest point, peaking at $k = 60$ with a score of 0.7338 whereas SVDS peaks at $k = 68$ with a score of 0.7321. What is especially interesting, is that GKLB performs so much worse than the other methods, while its error was comparable to that of the randomized SVD. The latter two methods also show much more unstable behaviour than the former.

The time taken by each algorithm for calculating the SVD of the tf-idf matrix is shown in Table 2. The times shown are the average of 5 runs, measured in seconds. We see somewhat different results than those in Section 6, remembering that the tf-idf matrix is sparse. Most notable is that the randomized SVD takes much longer than the other methods for low values of k, while in Section 6 we had seen that randomized SVD had quite efficient performance on both sparse and full matrices. We also see that the block power SVD is only faster than SVDS for $k = 5$. As we saw in Section 3, the time taken by SVDS significantly decreases after $k > \frac{1}{3}\text{rank}(A)$ as it switches to a different SVDS method. Since $\text{rank}(A) = 423$ here, this explains the time reduction seen for $k = 200$ and $k = 300$. It is hard to tell what the cause is of the sudden time reduction seen in randomized SVD for those same values of k, as the algorithm uses the built-in `qr` and `svd` commands of Matlab, of which the code is not available.

Table 2: Time taken for computing the SVD of the tf-idf matrix

Value of k	5	25	50	75	100	200	300
GKLB	0.0088	0.0429	0.1029	0.2370	0.4869	1.7988	3.5143
Randomized SVD	0.6986	1.6544	2.6949	3.7397	4.7898	1.0569	1.3006
Block power SVD	0.0594	0.3092	0.6751	1.1456	1.7027	4.0436	6.8957
SVDS	0.0967	0.1977	0.4459	0.8843	1.0091	0.4478	0.4066

The semantic space can be visualised by reducing the word matrix U to 2 dimensions and plotting the words at their respective coordinates, as in Section 9. The result is displayed in Figure 16. Remember that the similarity of words is determined by the cosine similarity metric, and thus similar words lie in the same direction relative to the origin. For example, here we can see the words *egypt*, *syria*, *syrian*, *cairo*, *iraq*, *egyptian*, *jordan* and so forth all in the same direction. Similarly, just below are the words *buddhust*, *viet*, *china*, *chinese*, *mao* and *communist*. However, we may not immediately see the connections between all words displayed. This is because we have reduced the dimension to $k = 2$, for which the recall score is very low as we can see in Figure 15, and thus is not the optimal dimension for the semantic space.

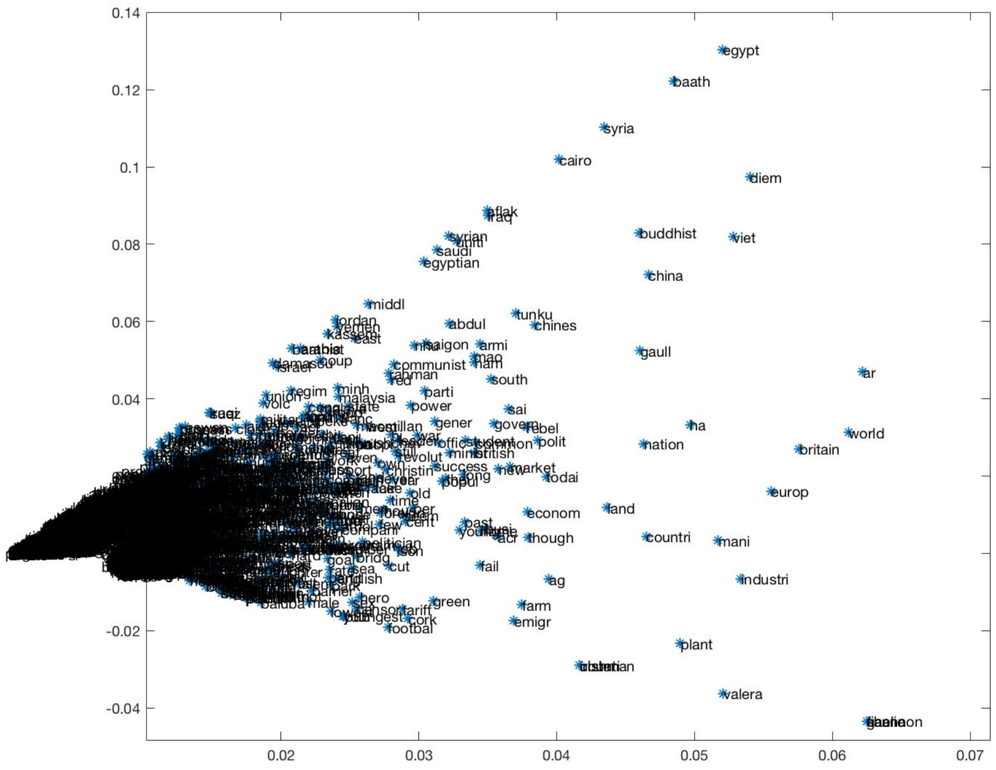


Figure 16: Visualisation of the semantic space of Time collection

Part III
SVD for Recommender Systems

With the unlimited options available to us when searching for a movie on Netflix, music on Spotify or products on online stores, recommender systems have become almost essential to us for making a selection in this huge amount of data. *Collaborative filtering* recommender systems rely on past feedback on items and user behaviour to infer new user-item associations (Ricci, Rokach, & Shapira, 2011). This can be based on explicit feedback, such as ratings, or implicit feedback, such as click counts or the amount of time spent viewing an item.

Traditional collaborative filtering systems often rely on finding a so-called neighbourhood of users who gave similar ratings for the same items, and predicting the ratings for unrated products for a user based on the ratings given by users in the neighborhood. However, this approach has many flaws. Firstly, for large item databases, most users will often only have rated less than 1% of the items, making it extremely difficult to find exact matches between users. Secondly, the number of computations required for calculating the correlation between each customer can grow extremely fast, rendering such algorithms useless for very large problems. Finally, different item names often refer to very similar items, which attempting to exactly match items across users cannot accommodate for. These issues are called *sparsity*, *scalability* and *synonymy* respectively.

In 2006, Netflix announced a contest to improve their movie recommender system, releasing a database of 100 million movie ratings for approximately 500,000 users and 17,000 movies. A prize of 1 million dollars was awarded to the algorithm that could predict unknown ratings for movies with 10% higher accuracy than Netflix's own CineMatch algorithm. Many of the best algorithms used matrix factorization algorithms such as SVD to explore the latent structure of the database (Koren, Bell, & Volinsky, 2009). This part discusses some of the winning SVD-based algorithms.

12 The Method

In the case of LSI for document searching, the truncated SVD was used to map the words and documents to the same semantic space, and the value of k was the dimension of that space. Now we wish to map users and items to the same k-dimensional feature space, so that these, too, can be compared. For example, when applying SVD to a movie database, each dimension may represent a different genre, and the weight of that dimension in the user-feature or item-feature vectors indicates how much the user likes that genre or to what extent the movie adheres to that genre.

The basic method, as described by (Sarwar, Karypis, Konstan, & Riedl, 2002), works as follows: a matrix is constructed with users as rows and items (such as movies) as columns. Each entry $A_{u,i}$ is the rating given by user u for item i. Calculating the truncated SVD now supplies us the matrices U_k, V_k and Σ_k. We now have a latent k-dimensional space where related items should be close together, and users should be close to items they like. We can now predict the rating $\hat{r}_{u,i}$ for user u of item i using

$$\hat{r}_{u,i} = U_k \sqrt{\Sigma_k}^T (u) \cdot \sqrt{\Sigma_k} V_k^T (i) \tag{6}$$

Now, the user-feature matrix is the product $U_k \sqrt{\Sigma_k}^T$, and the item-feature matrix is $\sqrt{\Sigma_k} V_k^T$. Thus the prediction is equal to the dot product of the corresponding user-feature and item-feature vectors.

As we have seen in previous sections, this technique is very good at handling both sparsity and synonymy, as it will not match items exactly, but discovers underlying features in the data. Furthermore, it solves the scalability issue because new users and items can be folded-in with very low computational cost. Adding on many users will require recomputation of the SVD, but this process can be done offline.

The difficulty with this approach lies in the construction of the user-item matrix A. If constructed in a similar fashion to the word-document matrices, we will have treated unrated items as having a rating of 0. This works for approaches where we do not analyse ratings, but the presence of items, in the same way as analysing the presence of words in a document. We can apply this, for example, when comparing groceries bought by consumers at a supermarket, or music in playlists. However, if a user has not yet watched a certain movie or used a certain item, we cannot assume a rating of 0, since there is still the possibility that the user will enjoy the item.

There are various solutions to this problem. A standard approach is to fill the empty entries with the average rating for that product or user (in most cases, average product ratings appear to yield better results). Another approach, first developed by Simon Funk for the Netflix contest, uses stochastic gradient decent to approach the SVD for the known values, without having to fill in or assume unknown values in the process. This part will discuss both approaches.

13 Average rating filling

First, we will implement the SVD-based algorithm in which we fill the zero entries in the user-item matrix with the average of each movie rating and calculate the truncated SVD of the user-movie matrix, as is done by (Sarwar et al., 2002). We will refer to this as the average-rating filling (ARF) algorithm.

The algorithm will be tested on the MovieLens 100k dataset (Harper & Konstan, 2016). This is a dataset comprising of 100,000 ratings by 943 users on 1682 movies. Ratings take on integer values in the range of 1-5. The dataset is split into a training set and a test set, where the training set comprises of 80% of the data and the test set of 20%. In this way, the ARF algorithm can be applied to the training set, and we can test the performance of the algorithm by predicting the ratings for the test set. The overall performance is calculated using the root mean squared error (RMSE):

$$\text{RMSE} = \sqrt{\sum_{(u,i)\in\kappa} \frac{(r_{u,i} - p_u \cdot q_i)^2}{N}}$$

where N is the total number of test cases, p_u is the u-th row of the user-feature matrix $U_k\sqrt{\Sigma_k}^T$, and q_i is the i-th row of the item-feature matrix $\sqrt{\Sigma_k}V_k^T$. The predicted rating $\hat{r}_{u,i} = p_u \cdot q_i$ is clipped to improve performance. This entails that when $\hat{r}_{u,i} < 1$ we set $\hat{r}_{u,i} = 1$, and when $\hat{r}_{u,i} > 5$ we set $\hat{r}_{u,i} = 5$ since we can be sure that none of the real ratings are outside the 1-5 range. Clearly, we wish to minimize the RMSE. The relevant Matlab code for ARF can be found in Appendix C.1.

Since we are working with the truncated SVD, we can test for which value of k, and for which of the SVD algorithms from Part 1, the ARF algorithm will work best. For consistency, and due to random initialization of some of the SVD algorithms, testing is done using 5-fold cross-validation. The results are shown in Figure 17. At first sight, there appears to be no yellow line for the block power SVD algorithm; however, this line is in fact present and almost exactly follows the plot line for SVDS. We can see that the error is minimized at $k = 13$ for both SVDS and block power SVD with an RMSE of 0.9815. This would indicate that the optimal number of features with which the users and movies in this database can be described and compared is 13, and that on average a predicted rating will differ approximately 1 point from the true rating.

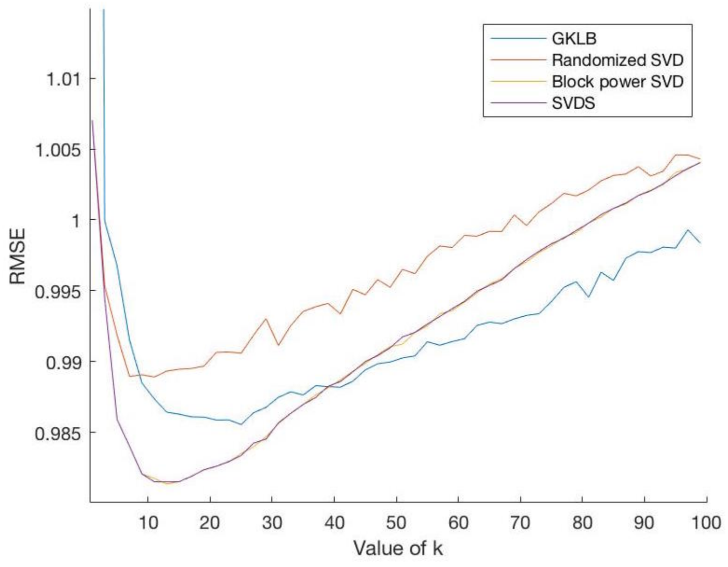


Figure 17: RMSE of ARF algorithm for different values of k

Table 3 shows how long each algorithm takes to perform the SVD of the ARF matrix. The times shown are the average of 5 runs, measured in seconds. We saw in Section 6 that the block power SVD has relatively faster performance for full matrices than sparse matrices, in comparison to SVDS. Since the ARF matrix is full, while the LSI matrix was sparse, this explains why the block power SVD shows better time performance than SVDS now. For this application, we would prefer the block power SVD over the SVDS since it takes less time while having very similar performance.

As for the GKLB and randomized SVD algorithms, we again see that they show worse performance. However, in this case GKLB outperforms randomized SVD, while in the error analysis and LSI search implementation the opposite was true. Randomized SVD now has much faster performance than it did for the LSI implementation; this was to be expected as in Section 6 we saw that the algorithm is much faster for full than for sparse matrices.

Table 3: Time taken for computing the SVD of the ARF matrix

Value of k	5	25	50	75	100	200	300
GKLB	0.0125	0.0911	0.1845	0.2507	0.3462	1.1194	1.7612
Randomized SVD	0.1678	0.2831	0.3913	0.4943	0.5566	0.2017	0.2793
Block power SVD	0.2159	0.3229	0.4843	0.7268	0.9208	2.2860	3.3308
SVDS	0.2529	0.5795	1.0681	1.1472	1.6790	4.2612	7.8470

14 Stochastic gradient descent

This section discusses the stochastic gradient descent (SGD) method first proposed by Simon Funk for the Netflix prize (Funk, 2006). The goal of stochastic gradient descent is to approach the k-dimensional user-feature vectors p_u and item-feature vectors q_i. This is done using a learning algorithm, which minimizes the mean squared error on the set of known ratings:

$$\begin{aligned} f(p_*, q_*) &= \sum_{(u,i)\in\kappa} (r_{u,i} - q_i \cdot p_u)^2 \\ &= \sum_{(u,i)\in\kappa} f_{u,i}(p_u, q_i) \end{aligned}$$

where κ is the set of (u,i) pairs for which the rating $r_{u,i}$ is known, and $q_i \cdot p_u = \hat{r}_{u,i}$ is the predicted rating of user u for item i.

In order to minimize f, we can take the derivative $\frac{\partial f}{\partial \theta}$ and attempt to find a minimum iteratively using the rule $\theta \leftarrow \theta - \alpha \frac{\partial f}{\partial \theta}$. This lets us follow the error landscape in the direction of the steepest downward gradient, hence the name *gradient descent*. Now, for $f_{u,i}(p_u, q_i) = (r_{u,i} - q_i \cdot p_u)^2$, we have

$$\frac{\partial f_{u,i}}{\partial p_u} = -2q_i(r_{u,i} - q_i \cdot p_u)$$
$$\frac{\partial f_{u,i}}{\partial q_i} = -2p_u(r_{u,i} - q_i \cdot p_u)$$

This gives the learning rule:

$$error = r_{u,i} - q_i \cdot p_u$$
$$p_u \leftarrow p_u + \alpha \cdot error \cdot q_i$$
$$q_i \leftarrow q_i + \alpha \cdot error \cdot p_u$$

where α is the learning rate (the factor 2 is incorporated into α). A high learning rate makes the algorithm work faster but can be unstable and skip over true minima. A low learning rate is slower, but more accurate.

The SGD algorithm then initializes p_u and q_i as random vectors, and loops over each rating a fixed number of times, each time applying the learning rule. Eventually, it should converge to the user-feature and item-feature vectors, and we can calculate the predicted rating for user u of item i as $\hat{r}_{u,i} = q_i \cdot p_u$. In other words, the vectors p_u form the rows of the user-concept matrix $U_k \sqrt{\Sigma_k}^T$, and the vectors q_i form the rows of the item-concept matrix $\sqrt{\Sigma_k} V_k^T$ as in Equation 6.

As with the ARF algorithm, we will test the SGD algorithm on the MovieLens 100k dataset. The corresponding Matlab code can be found in Appendix C.2. In the implementation, there are a number of factors that can be fine-tuned. The first is k, the number of suspected features. The second is α, the learning rate. The value of α is closely tied to the number of *epochs*, which is the number of times the algorithm loops over the training set. For a very low value of α, a higher number of epochs is required since the algorithm learns at a much slower rate. However, the number of epochs should not be too high, since this will cause the model to overfit on the training data, which can cause worse performance on the test data. For the purposes of optimizing these values, the base values will be set at $k = 15$, $\alpha = 0.005$ and $epochs = 20$. The user-feature and item-feature vectors are initialized as random vectors drawn from the normal distribution. Again, 5-fold cross-validation is used for testing the algorithm, and the average of these trials is shown in the results.

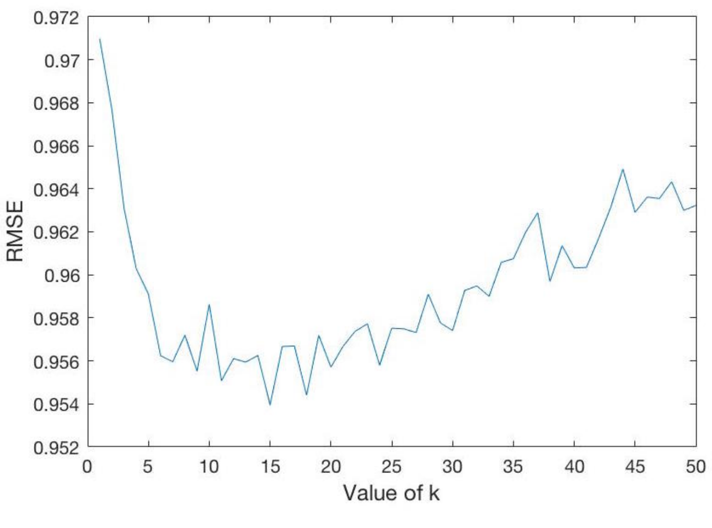


Figure 18: RMSE of SGD algorithm for different values of k

First, we will attempt to optimize for the number of features, k. The results are shown in Figure 18. We can see that the algorithm seems to perform best for $k = 15$, with an RMSE of 0.9539. However, the behaviour of the SGD algorithm is much more unstable than that of ARF, as the random initialization of the vectors can greatly influence the outcome.

Since the learning rate (α) and number of epochs required can influence each other greatly, these should be varied simultaneously in order to optimize both values. In Figure 19 we see that high values of the learning rate indeed lead to overfitting, especially when the number of epochs is increased. The error also increases rapidly when α is set very low and the number of epochs is decreased, as it will not have converged by this point. The minimum seems to occur around $\alpha = 0.005$ and $epochs = 35$. More precise analysis provides the optimal values $\alpha = 0.003$ and $epochs = 37$ for an RMSE of 0.9517. However, we must also consider that 37 epochs of training takes almost twice as long as 20 epochs, for an average improvement of only 0.0022.

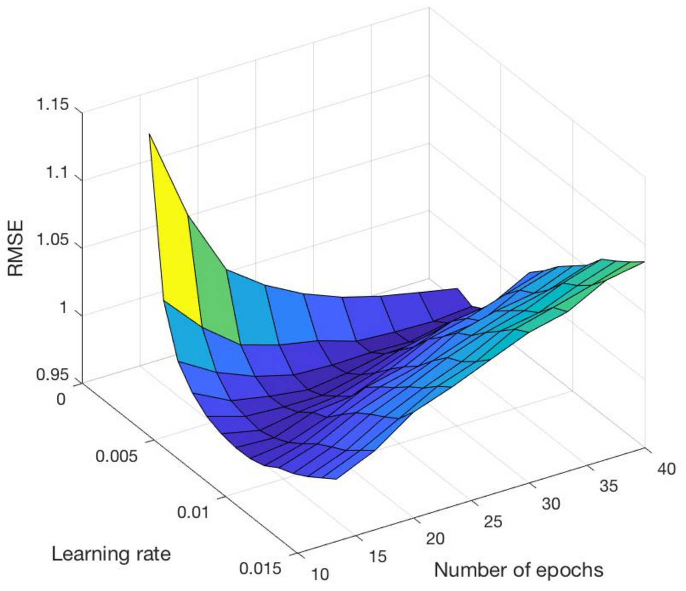


Figure 19: RMSE of SGD algorithm for different values of α

15 Biased stochastic gradient descent

In making the ARF and SGD models, we have assumed that users show no bias in their ratings and all movies are of equal quality. For example, the model will predict that a user who enjoys romantic comedies will give a high rating to all movies for which the "romance" and "comedy" weightings of the item-feature vector are high. However, this does not take into account the vast difference in (perceived) quality of movies, or that some users give harsher ratings than others.

These observations have been incorporated by (Koren et al., 2009) into the existing SGD model by using biases: a constant b_i for each movie, and b_u for each user. A positive b_u indicates that user u gives kinder ratings than average, while a positive b_i indicates that movie i is perceived to be better, regardless of the genres it contains. Say we would want to calculate Alice's rating of *The Godfather*, without knowing anything about Alice's movie preferences or movies *The Godfather* is similar to. Now, *The Godfather* is a better than average movie, and tends to receive ratings 0.7 higher than the average. However, Alice is a very critical user, who rates movies 0.5 lower than her average counterparts. If the average rating for all movies is 3.5, our estimated rating would be $3.5 + 0.7 - 0.5 = 3.9$.

Using biases, the predicted rating can be extended as follows:

$$\hat{r}_{u,i} = \mu + b_i + b_u + q_i \cdot p_u$$

where μ is the global average of all movie ratings. In (Koren et al., 2009) it is also proposed to minimize the regularized squared error, rather than the mean squared error:

$$f(p_*, q_*) = \sum_{(u,i)\in\kappa} (r_{u,i} - \hat{r}_{u,i})^2 + \lambda(\|p_u\|^2 + \|q_i\|^2 + b_u^2 + b_i^2)$$

where λ is the regularization constant. Now we have the following learning rules:

$$\begin{aligned} error &= r_{u,i} - \hat{r}_{u,i} \\ b_u &\leftarrow b_u + \alpha(error - \lambda b_u) \\ b_i &\leftarrow b_i + \alpha(error - \lambda b_i) \\ p_u &\leftarrow p_u + \alpha(error \cdot q_i - \lambda p_u) \\ q_i &\leftarrow q_i + \alpha(error \cdot p_u - \lambda q_i) \end{aligned}$$

Again, the user-feature and item-feature vectors are initialized as random vectors drawn from the normal distribution, and the biases are initialized as 0. The corresponding Matlab code can be found in Appendix C.3. For the values $k = 10$, $\alpha = 0.005$, *epochs* $= 38$ and $\lambda = 0.04$ the RMSE is minimized at an average value of 0.9440, indicating an improvement in performance compared to SGD without biases.

We have seen that the SGD algorithm, both with and without bias, shows much better performance than the ARF algorithm. However, SGD is much more difficult to finetune, since there are more variable factors that are reliant on each other, and the random initialization can lead to more unpredictable performance. The times taken for the algorithms for different values of k are shown in Table 4. For the SGD algorithms, the optimal values found for α, *epochs* and λ are used, and ARF is performed using block power SVD. The times shown are the averages of 5 runs, measured in seconds. We see that ARF is the fastest by far, performing 3-4 times faster than the SGD algorithms.

Table 4: Time taken to predict Movielens ratings

Value of k	5	10	20	50	100
ARF	0.2458	0.3001	0.3704	0.4972	0.9915
SGD	3.7268	4.0225	4.2213	5.1231	6.8440
Biased SGD	4.4049	4.8031	5.1526	5.7820	7.5401

Part IV
SVD for Image Processing

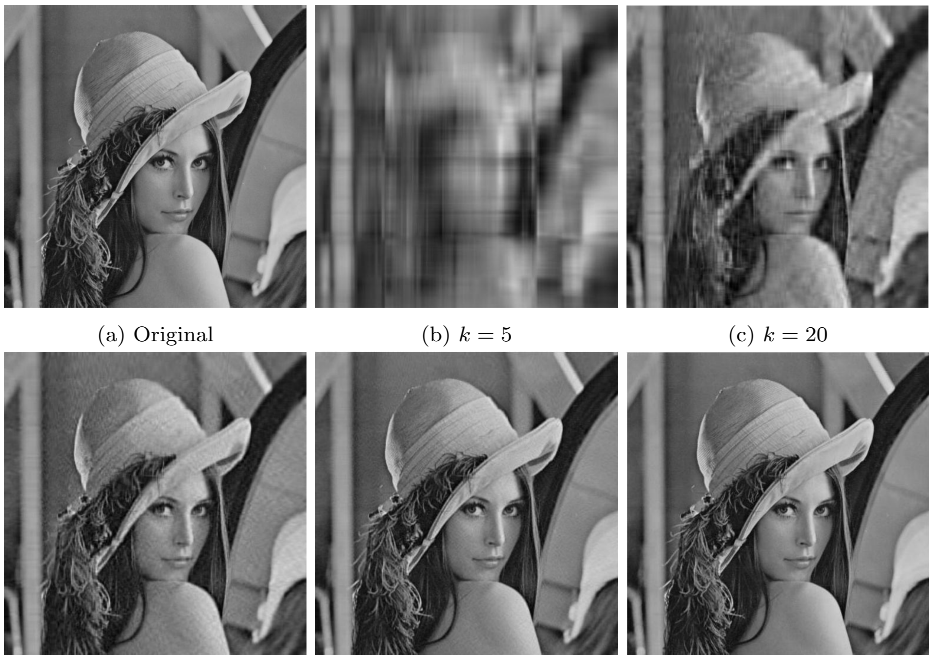
Previously, we have discussed applying the SVD in situations where our data can be represented as a two-dimensional matrix containing frequencies or ratings. Another data that can be represented in such a way is image data. An image can be represented as a matrix where each element represents the intensity of the pixel at that location. This part will discuss how the SVD can be used for image compression and face recognition.

16 Image compression

As we have seen in previous sections, the truncated SVD is very useful for drastically reducing the dimension of large data matrices. In the case of LSI and recommender systems, this was especially useful as it removed the noise present in the data. In this section, we will see that this can also be applied for image compression, as the truncated SVD can capture the most important information of the image using much fewer dimensions (Sadek, 2012).

A grayscale $m \times n$ pixel image can be represented as a $m \times n$ matrix A where $A_{i,j}$ is the intensity of pixel $p_{i,j}$. In most cases, the intensity lies in the range $[0, 255]$ where 0 is black and 255 is white. When the original image takes up $\mathcal{O}(mn)$ space, the truncated SVD of the image matrix takes up $\mathcal{O}(k(m+n+1))$ space. Thus the space needed is reduced when $k < \frac{mn}{m+n+1}$. Furthermore, in Section 2 we saw that the truncated SVD is the best k-rank approximation for the original matrix, minimizing the difference between the original and truncated matrix. Accordingly, we would expect this method to minimize the difference between the original and compressed image for k-rank compression.

Figure 20 shows the resulting image $A_k = U_k \Sigma_k V_k^T$ for increasing values of k when we compress the Lena image using SVD. The original image is 512×512, and so the space needed is reduced for all $k < 255$. For $k = 100$, we can barely notice any difference between the original image and the compressed image. An example implementation of SVD image compression in Matlab can be found in Appendix D.1.


(a) Original (b) $k = 5$ (c) $k = 20$

(d) $k = 50$ (e) $k = 100$ (f) $k = 200$

Figure 20: Image compression for Lena image

We can also use this form of image compression to visualize the performance of the algorithms discussed in Part 1. The Lena image is compressed using GKLB with one-sided re-orthogonalization, randomized SVD with $p = 5$ and $q = 1$, block power SVD with 50 iterations and Matlab's SVDS for $k = 15$. The results are shown in Figure 21. We see that GKLB clearly performs the worst, as the image bears least resemblance to the original. This was to be expected since we saw in Section 6 that GKLB had the highest error $\|A - A_k\|$ in comparison to the other methods, and thus the largest difference between the truncated SVD of the image and the original. Randomized SVD appears to perform much better than GKLB, although not as well as the block power SVD and SVDS, which show very similar performance.

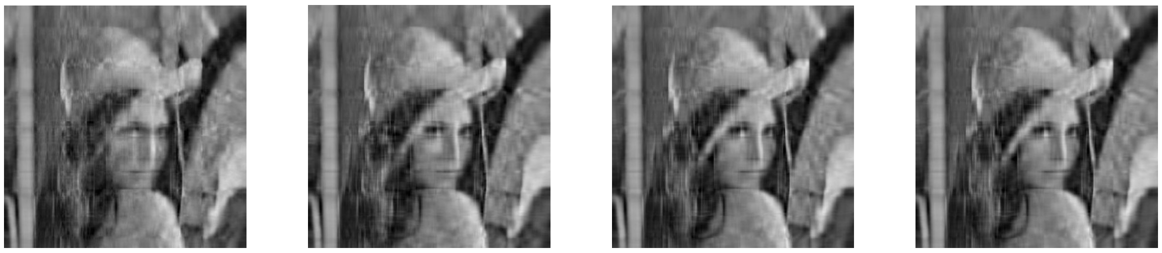

(a) GKLB (b) Randomized SVD (c) Block power SVD (d) SVDS

Figure 21: Lena image compressed using different algorithms, $k = 15$

Table 5 shows the times taken by each algorithm discussed for compressing the Lena image using the values of k shown in Figure 20. We see very similar performance of the block power SVD and SVDS; neither seems to perform consistently better than the other. For very low values of k, GKLB has a clear time disadvantage although this does not outweigh the much worse compression performance seen in Figure 21. As the image matrix is full, randomized SVD shows very good time performance relative to the other methods, especially considering higher values of k.

Table 5: Time taken for computing the SVD of the Lena image matrix

Value of k	5	20	50	100	200
GKLB	0.0020	0.0061	0.0179	0.0820	0.4328
Randomized SVD	0.0229	0.0170	0.0266	0.0306	0.0538
Block power SVD	0.0375	0.0738	0.1195	0.2784	0.6183
SVDS	0.0382	0.0590	0.1292	0.3778	0.0894

A colour image has three colour channels: R (red), G (green) and B (blue), and can be represented as an $m \times n \times 3$ matrix. For compression purposes, we can compute the truncated SVD for each channel separately, and recombine them to form the compressed image. An example is shown in Figure 22 for an image of Mondriaan's *Composition with Yellow, Blue and Black*, with original size 790×800. This example also shows that SVD compression works exceptionally well for images containing mostly vertical and horizontal lines, as the image is already recognizable for very low values of k.

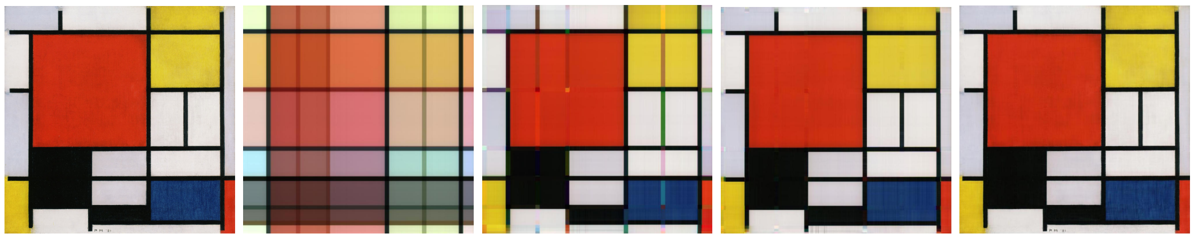

(a) Original (b) $k = 1$ (c) $k = 5$ (d) $k = 10$ (e) $k = 50$

Figure 22: Image compression for Mondriaan image

17 Face recognition

Just as the SVD allows us to search more efficiently for similar documents in a lower-dimensional space through LSI, we can also use the SVD to search for similar images in a lower-dimensional image space. This section will explore the possibility of using the SVD for face recognition (Zeng, 2007).

The method will be tested on the FEI faces database (Thomaz, 2005). The database contains images of 200 individuals, all between 19 and 40 years old, cropped to contain only their face. Of each individual, two images are used: one where the individual has a neutral expression, which is used to construct the database, and one where they are smiling, which is used for testing. Some examples from the database are shown in Figure 23 and 24.

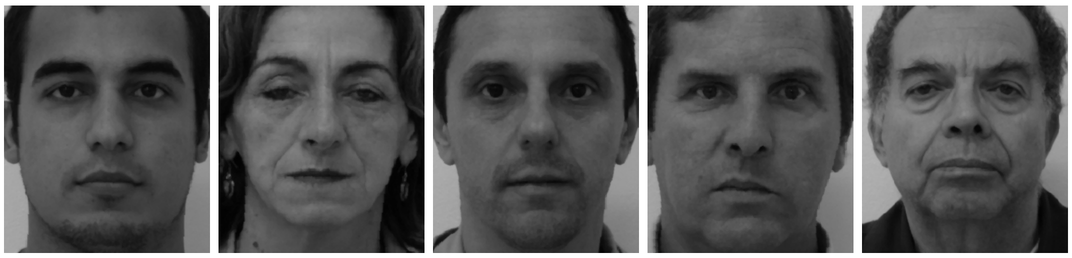


Figure 23: Neutral faces

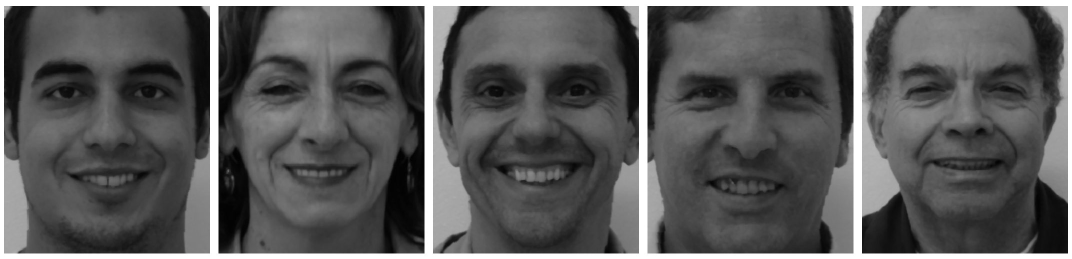


Figure 24: Smiling faces

The database is constructed as follows: each face image f_i, $i = 1, 2, \dots, n$ of size $a \times b$ is reshaped to a vector of length $m = ab$ by stacking the columns of the image. Then, we take the average $f_\mu = \frac{1}{n}\sum_{i=1}^{n} f_i$ of the faces. The "average face" is shown in Figure 25. The average face is subtracted from the other faces for normalization. We can then construct a matrix A, where each column is a normalized face vector. As with LSI search, we can take the truncated SVD $A = U_k \Sigma_k V_k^T$, where each row of V_k is a face-feature vector.

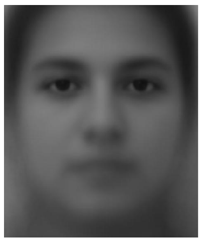


Figure 25: Average face of FEI database

To recognize a new face vector f, we first subtract the average face f_μ and then project the face onto the face-space. This is done in the same way as projecting a query for LSI, as $f' = f^T U_k \Sigma_k^{-1}$. The face closest to the projection f', using the cosine similarity metric, is the recognized face. The corresponding

Matlab code for face recognition can be found in Appendix D.2.

Figure 26 shows the performance of this method on the database for values of k between 1 and 200, the rank of the face matrix of the database. After $k = 13$, already 180 out of the 200 test faces are recognized correctly. The best recognition score is 190 for $k = 51$. The baseline shown in Figure 26 is the recognition score for simply finding the nearest face for each new face vector without applying SVD, which is 189/200. Although the face recognition works just as well with and without applying SVD, the space required to store the face vectors is significantly decreased: from $ab \times n$ to $n \times k$, where typically $k \ll ab$. Also, searching through the smaller database for a matching face is much faster than searching through the original full-size images. It should be noted that in this case, as opposed to what we have seen in Part 2 and 3, the performance of the method does not decrease for high values of k.

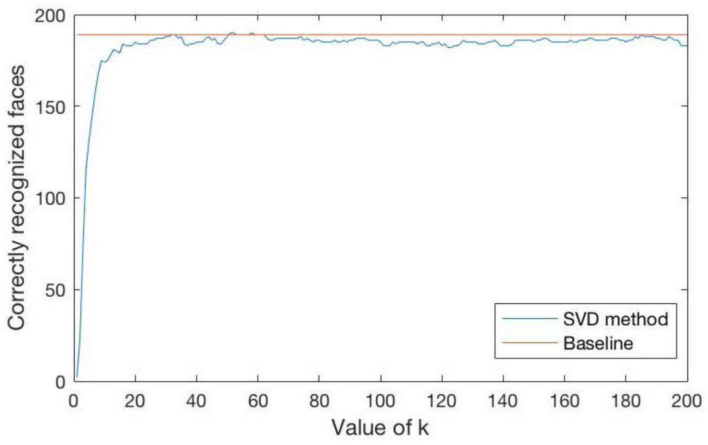


Figure 26: Faces correctly recognized using SVD method for different values of k

Some of the faces that were not recognized correctly are shown in Figure 27. We can see clearly why these were chosen as the nearest faces, as they are very similar.

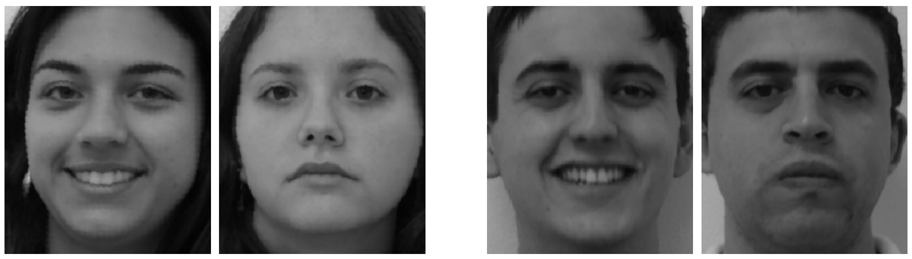


Figure 27: Incorrectly recognized faces

Compared to modern methods such as convolutional neural networks (Lawrence, Giles, Tsoi, & Back, 1997), the SVD face recognition method is far from ideal. A large disadvantage is that it is not shift-invariant: it only works when the face has already been detected and cropped out of the image so that it has exactly the same placing as the test face. We have seen that changes in facial expression are handled with relative ease, but if the head would be moved or rotated performance could greatly decrease.

An advantage to the method is that only one face is needed for training data, while most other methods only work when many examples of the same face are given. We have also seen that the method is very good at searching for similar images in a much lower-dimensional space. Although this may not be the answer for facial recognition, it is a good method for efficiently finding images similar to an input image, taking up much less time and space.

18 Further applications of the SVD

We have seen that the SVD is very useful for data compression, removal of noise and recognition of features in big data applications. In Section 2 it was also shown that the SVD also provides us with many important matrix properties; not only the singular triplets, but also the (effective) rank, rowspace, column space and many more. As we will see in this section, there are many other areas in which the SVD is useful.

18.1 The pseudoinverse and linear least squares problems

Definition 18.1. (Golub & Reinsch, 1970) Let A be an $m \times n$ matrix. Then an $n \times m$ matrix X is the pseudoinverse of A if X satisfies the following properties:

(i) $AXA = A$

(ii) $XAX = X$

(iii) $(AX)^T = AX$

(iv) $(XA)^T = XA$

The pseudoinverse of A is denoted by $A^\dagger$.

Now, when $A = U\Sigma V^T$, then $A^\dagger = V\Sigma^\dagger U^T$, where $\Sigma^\dagger = \text{diag}(\sigma_i^\dagger)$ and

$$\sigma_i^\dagger = \begin{cases} \frac{1}{\sigma_i} & \text{if } \sigma_i > 0 \\ 0 & \text{if } \sigma_i = 0 \end{cases}$$

With respect to the algorithms discussed in Part 1, it is important to note that in calculation of the pseudoinverse, the smallest singular values of A will become the largest values of $\Sigma^\dagger$. Now, while we have seen excellent performance by the block power SVD for calculation of the largest singular values, the method is much less precise for the smallest singular values, so one should exercise caution when using it for the pseudoinverse.

In practice, the pseudoinverse is only calculated using the singular values for which $\sigma_i > tol$, also referred to as the regularized pseudoinverse (Hansen, 1987). This is because the smallest singular values often only represent noise in the data (as we have seen ourselves), and inverting these will only cause the amount of noise to increase. The regularized pseudoinverse can, for example, be used to remove noise from images (Shim & Cho, 1981). In such cases, the block power SVD is again a good candidate for calculation of the SVD.

The pseudoinverse can also be used to solve linear least squares problems, which are used in regression analysis to solve overdetermined systems. Given an $m \times n$ matrix A and a vector $b \in \mathbb{R}^m$ with $m \geq n \geq 1$, then a least squares problem is given by

$$\min_x \|Ax - b\|_2$$

The unique solution to such a problem is given by $x = A^\dagger b$. As such, the SVD is a very useful tool for solving problems of this type.

18.2 Principal component analysis

In literature, the SVD is often mentioned alongside the term principal component analysis (PCA), one of the most important and wide-spread methods for multivariate data analysis first described by Karl Pearson (Pearson, 1901). The reason for this is that they are essentially the same thing. PCA is described as a method for simplifying data matrices in order to identify the *principal components*, i.e. essential patterns and features, of the data.

Consider a data matrix A of which the rows represent objects, and the columns represent variables. The findings for each object may be represented by a linear combination of these, possibly dependent, variables. PCA is a transformation of the data from this set of dependent variables to a set of linearly independent variables. These variables are the *principal components*: orthogonal vectors that lie in the direction of the greatest variance of the data. Calculating these allows for better explanation of the data (Wold, Esbensen, & Geladi, 1987).

The principal component vectors are calculated and stored in a matrix W, and a score matrix $T = AW$ is calculated which maps the original data to a feature space. The first k principal components are the eigenvectors of the covariance matrix A^TA belonging to the k largest eigenvalues of A^TA. Remembering from Section 2 that $A^TA = V\Sigma^2V^T$, we see that the principal components are the right singular vectors, and thus $W = V$. Also, from $T = AW = AV$ it follows that $T = U\Sigma$. Since calculating A^TA and subsequently finding its eigenvectors is a numerically unstable process, PCA is almost always performed using the SVD. Like SVD, PCA allows for dimension reduction which makes it a desirable technique for many data analysis applications.

19 Conclusion

This report has examined both the computation and some of the large-scale applications of the singular value decomposition. The algorithms discussed include Golub-Kahan-Lanczos Bidiagonalization, randomized SVD, block power SVD and Matlab's SVDS. We have seen that the SVD is useful for many applications, the main focus being on information retrieval, recommender systems and image processing.

The first algorithm discussed for computing the SVD was GKLB. Here we found that performance and numerical stability were significantly increased when one-sided re-orthogonalization was applied. Despite this, the algorithm seemed to perform worse than other algorithms when comparing error and in large-scale implementations. We saw that the spectrum of the singular values of the bidiagonal matrix B played a large role in this difference. Strong oversampling, i.e. allowing many more than k iterations, could solve this issue. However, we saw in the LSI implementation that an optimal value of $k = 60$ was found for $\text{rank}(A) = 483$. The amount of oversampling required for convergence of the first 60 singular values can be high, and leads us to calculation of the SVD for the bidiagonal matrix B of size similar to that of A, which can seem quite pointless.

Matlab's SVDS algorithm also implements GKLB and finds much better results. A part of this improvement comes from the convergence checks Matlab performs, which ensure that the error is minimized, but also cost much more time to perform. SVDS also uses strong oversampling, and even switches to a different SVD method when $k > \frac{1}{3}\text{rank}(A)$, as the amount of oversampling required after this point is as much work as calculating the full SVD.

The GKLB algorithm was often on par with randomized SVD. This algorithm has the advantage that it does not require iteration and is much simpler to follow than GKLB. For information retrieval and image processing, randomized SVD outperformed GKLB, while the opposite was true for the ARF recommender system.

However, neither algorithms performed as well as the block power SVD. This method showed smooth convergence to 0 in error as the number of iterations was increased and had incredibly similar performance to SVDS in all applications discussed. We also saw that the block power SVD was able to calculate the SVD much faster than SVDS for the recommender system application. The method is especially suitable for calculating the truncated SVD of full matrices when the value of k is small relative to the rank of the matrix. However, the block power SVD should only be used for low-rank approximations as it is not precise in calculating the full SVD.

The SVD is a very useful tool for many data applications where it is necessary to search through, and compare, large quantities of data. The reason for this is that the SVD identifies the component vectors contributing most to the spread, or variance, of the data, where the singular values can be seen as the weights assigned to these vectors. Furthermore, truncating the SVD not only reduces the space required to store the data, but can also reduce much of the noise present and uncover latent features and patterns that would not have been visible in the raw data.

We saw in Part 2 that this reduction of noise and exploitation of latent features made it possible to overcome the problems of polysemy and synonymy so often found in the fields of natural language processing and information retrieval. Representing words and documents in a lower-dimensional space provided us with much more information than only the raw data would have done. Because of this, the LSI search algorithm performs very well as opposed to literal word matching.

Part 3 subsequently showed that what can be done for words and documents, can also be done for items and users. This not only allows us to find items similar to the user's tastes, but also predict unseen ratings. The presence of unseen and biased ratings also led us to consider (biased) stochastic gradient descent, which provided a different approach to finding lower dimensional SVD-like vectors to represent the data. Considering this, further research could be done as to whether, and how, the SVD can be extracted from the results of SGD.

Lastly, we saw in Part 4 that the SVD can also be used for image compression, as image data is represented with minimal error compared to the original for k-rank compression. This also provided an opportunity to visualize the differences in performance between the algorithms discussed previously. Compared to modern methods, the SVD is not quite suited for face recognition, although it does provide a very (space-)efficient way to search through a large database for similar images.

A disadvantage to using the SVD for LSI and the ARF recommender system is that it does not provide us with information about how the data is structured, or why, for example, certain words and documents or users and items are placed close to each other in the reduced (semantic) space. We have often referred to word-feature, document-feature, user-feature and item-feature vectors, while not having any information about what these features may be. Especially in the case of users wanting to know why certain products have been recommended for them, this can be problematic. More research on the topic would be needed to investigate if, and how, it would be possible to extract more information on the nature of the identified features.

References

Bentbib, A. H., & Kanber, A. (2015). Block Power Method for SVD Decomposition. *Analele Universitatii Ovidius Constanta - Seria Matematica*, *23*(2), 45–58. doi: 10.1515/auom-2015-0024

Berry, M. W., Dumais, S. T., & O'Brien, G. W. (1995). Using Linear Algebra for Intelligent Information Retrieval. *SIAM Review*, *37*(4), 573–595. doi: 10.1137/1037127

Erichson, N. B., Voronin, S., Brunton, S. L., & Kutz, J. N. (2016). Randomized matrix decompositions using r. *Journal of Statistical Software*.

Funk, S. (2006). *Netflix update: Try this at home.*

Furnas, G. W., Landauer, T. K., Gomez, L. M., & Dumais, S. T. (1983). Human factors and behavioral science: Statistical semantics: Analysis of the potential performance of key-word information systems. *The Bell System Technical Journal*, *62*(6), 1753–1806.

Golub, G. H., & Kahan, W. (1964). Calculating the Singular Values and Pseudo-Inverse of a Matrix. *SIAM Series B Numerical Analysis*, *2*(2), 205–224. doi: 10.1137/0702016

Golub, G. H., & Reinsch, C. (1970). Singular value decomposition and least squares solutions. *Numerische mathematik*, *14*(5), 403–420.

Golub, G. H., & van Loan, C. F. (1996). *Matrix computations.* Baltimore, Maryland: Johns Hopkins University Press Baltimore.

Halko, N., Martinsson, P.-G., & Tropp, J. A. (2011). Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, *53*(2), 217–288.

Hansen, P. C. (1987). The truncatedsvd as a method for regularization. *BIT Numerical Mathematics*, *27*(4), 534–553.

Harper, F. M., & Konstan, J. A. (2016). The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, *5*(4), 19.

Kalman, D. (1996). A singularly valuable decomposition: the svd of a matrix. *The College Mathematics Journal*, *27*(1), 2–23.

Koren, Y., Bell, R., & Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, *42*(8).

Krovetz, R. (1988). *Time collection.* `ir.dcs.gla.ac.uk/resources/test_collections/time/`.

Lawrence, S., Giles, C. L., Tsoi, A. C., & Back, A. D. (1997). Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, *8*(1), 98–113.

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval.* Cambridge, England: Cambridge University Press.

Pearson, K. (1901). On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, *2*(11), 559–572.

Prokhorov, A. (2011). *Covariance matrix.* `http://www.encyclopediaofmath.org/index.php?title=Covariance_matrix&oldid=13365`.

Ricci, F., Rokach, L., & Shapira, B. (2011). Introduction to recommender systems handbook. In *Recommender systems handbook* (pp. 1–35). Springer.

Sadek, R. A. (2012). Svd based image processing applications: State of the art, contributions and research challenges. *International Journal of Advanced Computer Science and Applications*, *3*(7), 26-34.

Sarwar, B., Karypis, G., Konstan, J., & Riedl, J. (2002). Incremental singular value decomposition algorithms for highly scalable recommender systems. In *Fifth international conference on computer and information science* (pp. 27–28).

Shim, Y., & Cho, Z. (1981). Svd pseudoinversion image reconstruction. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *29*(4), 904–909.

Simon, H. D., & Zha, H. (2006). Low-Rank Matrix Approximation Using the Lanczos Bidiagonalization Process with Applications. *SIAM Journal on Scientific Computing*, *21*(6), 2257–2274. doi: 10.1137/S1064827597327309

Thomaz, C. E. (2005). *Fei faces database.* `fei.edu.br/~cet/facedatabase.html`.

Tulloch, A. (2014). *Fast randomized svd.* `research.fb.com/fast-randomized-svd/`.

von Mises, R., & Pollaczek-Geiringer, H. (1929). Praktische Verfahren der Gleichungsauflösung. *ZAMM*, *9*(2), 152–164. doi: 10.1002/zamm.19290090206

Wold, S., Esbensen, K., & Geladi, P. (1987). Principal component analysis. *Chemometrics and intelligent laboratory systems*, *2*(1-3), 37–52.

Zeng, G. (2007). Facial recognition with singular value decomposition. In *Advances and innovations in systems, computing sciences and software engineering* (pp. 145–148). Springer.

Appendices

A Matlab Code for Part 1

A.1 GKLB

```
 1  function [U,S,V] = GKLBR(A,k)
 2  % Returns the k-dimensional truncated SVD of A using Golub-Kahan Lanczos ...
        Bidiagonalization; include lines 28 and 29 for one-sided re-orthogonalization
 3
 4  % initialization empty matrices
 5  [m,n] = size(A);
 6  B = zeros(k+1, k);
 7  P = zeros(m, k+1);
 8  Q = zeros(n, k+1);
 9  rand('state',0);
10  w = randn(m,1);
11  alpha = zeros(k+1,1);
12  beta = zeros(k+1,1);
13
14  % initialization starting values
15  beta(1) = norm(w);
16  P(:,1) = (1/beta(1)) * w;
17  Q(:,1) = transpose(A)*P(:,1);
18  alpha(1) = norm(Q(:,1));
19  Q(:,1) = (1/alpha(1)) * Q(:,1);
20
21  % Bi-diagonalization algorithm
22  for i = 1:k
23      p = (A*Q(:,i)) - (alpha(i)*P(:,i));
24      beta(i+1) = norm(p);
25      P(:,i+1) = (1/beta(i+1)) * p;
26
27      q = transpose(A)*P(:,i+1) - beta(i+1)*Q(:,i);
28      h = (q'*Q(:,1:i))'; % re-orthogonalization of V
29      q = q - Q(:,1:i)*h; % re-orthogonalization of V
30      alpha(i+1) = norm(q);
31      Q(:,i+1) = (1/alpha(i+1)) * q;
32
33      B(i,i) = alpha(i);
34      B(i+1,i) = beta(i+1);
35  end
36  Q = Q(:, 1:k);
37  [Ub, S, Vb] = svd(B); % Use SVD of bidiagonal B to approach SVD of A
38  S = S(1:k,1:k); % Singular values of A
39
40  % Compute left and right singular vectors of A
41  U = zeros(m,k);
42  V = zeros(n,k);
43  for i = 1:k
44      U(:,i) = P * Ub(:,i);
45      V(:,i) = Q * Vb(:,i);
46  end
```

A.2 Randomized SVD

```
function [U,S,V] = randomSVD(A,k,p,q)
% Returns the k-dimensional truncated SVD of A using randomized SVD

[¬,n] = size(A);
ohm = randn(n,k+p); % initialize random matrix
Y = A*ohm;
% perform power iteration
```

```
if q > 0
    for i = 1:q
        Y = A*A'*Y;
    end
end
[Q,¬] = qr(Y,0); %orthogonalize Y
B = Q'*A;
[U2,S,V] = svd(B); % perform SVD of smaller matrix B
U = Q*U2;
U = U(:,1:k);
V = V(:,1:k);
S = S(1:k,1:k);
```

A.3 Block power SVD

```
function [U,S,V] = powerblocksvd(A,k,iter)
% Returns the k-dimensional truncated SVD of A using block power SVD

[¬,n] = size(A);
V = randn(n,k); % initialize random matrix
% perform block power method
for i = 1:iter
    [U,¬] = qr(A*V,0);
    [V,¬] = qr(A'*U,0);
end
S = diag(diag(U'*A*V)); % keep only diagonal values; other values are close to 0
sgn = sign(S);
S = sgn*S; % ensure singular values are positive
U = U*sgn;
```

B Matlab code for Part 2

B.1 Preprocess documents

```
function A = preprocess(filename)
% Returns term-document matrix of texts contained in 'filename'

str = extractFileText(filename);
str = strsplit(str,'*TEXT'); % split file into separate documents
str(1) = []; % first element is an empty string

% Normalize text in files and make bag of words
cleantext = erasePunctuation(str); % remove all punctuation
cleantext = lower(cleantext); % make text lowercase
cleantext = tokenizedDocument(cleantext); % tokenize text
cleantext = removeShortWords(cleantext,2); % remove tokens shorter than 2 characters
cleantext = removeLongWords(cleantext,15); % remove tokens longer than 15 characters
cleantext = normalizeWords(cleantext); % normalize words using Porter stemmer
bag = bagOfWords(cleantext); % create bag of words
bag = removeInfrequentWords(bag,3); % remove words that appear less than 3 times
bag = removeWords(bag,stopWords); % remove stop words
bag = removeEmptyDocuments(bag); % remove documents that no longer contain words
A = tfidf(bag,'TFWeight','log'); % Make term-document matrix A
end
```

B.2 LSI search and testing

```matlab
function [average] = LSIsearch(A,Q,k)
% Performs LSI on term-frequency matrix A and returns average recall score for query ...
    matrix Q

% Initialization
[U,S,V] = svds(A,k);
[¬,n] = size(Q);
Q2 = zeros(k,n);

% Transform queries to document space
for i = 1:n
    query = Q(:,i);
    newquery = transpose(query)*U*inv(S);
    Q2(:,i) = newquery;
end

% Process query relevance test file
filename = 'time/TIME.REL';
reldocs = extractFileText(filename);
reldocs = strsplit(reldocs,'\n');
reldocs(84) = [];

% Calculate relevance scores
founddocs = knnsearch(V, transpose(Q2),'K',20,'Distance','cosine'); % find 20 most ...
    relevant documents for each query using k-nearest neighbours
scores = zeros(1,83);
for i = 1:length(reldocs)
idx = str2num(reldocs(i));
idx(1) = [];
score = length(intersect(idx,founddocs(i,:))); % number of relevant documents found by LSI
scores(i) = score/length(idx);
end
average = sum(scores)/83; % return average of relevance scores
```

C Matlab code for Part 3

C.1 Average rating filling (ARF)

```matlab
function rmse = ARF(trainset,testset,total_users,total_movies,k)
% Performs ARF using k-dimensional SVD on training set; returns RMSE when tested on ...
    testset.

% Initialize matrix with ratings from training set
A = zeros(total_users,total_movies);
for i = 1:length(trainset)
    user = trainset(i,1);
    movie = trainset(i,2);
    rating = trainset(i,3);
    A(user,movie) = rating;
end

% Fill zero values of A with average ratings
for i = 1:total_movies
    n = nnz(A(:,i));
    if n ≠ 0
        avg = sum(A(:,i))/n;
        for j = 1:total_users
            if A(j,i) == 0
                A(j,i) = avg; % fill with average movie rating
            end
        end
    else
```

```
        for j = 1:total_users
            avg = sum(A(j,:))/total_movies;
            A(j,i) = avg; % if movie has not been rated, fill with average user rating
        end
    end
end

[U,S,V] = svds(A,k);
P = U*transpose(sqrt(S)); % compute user-feature matrix
Q = sqrt(S)*transpose(V); % compute movie-feature matrix

% compute RMSE for data in test set
errors = zeros(1,length(testset));
total = 0;
for i = 1:length(testset)
    user = testset(i,1);
    movie = testset(i,2);
    rating = testset(i,3);
    predrating = dot(P(user,:),Q(movie,:));
    if predrating < 1 % clip ratings lower than 1
        predrating = 1;
    elseif predrating > 5 % clip ratings higher than 5
        predrating = 5;
    end
    errors(i) = rating - predrating;
    total = total + (errors(i)^2);
end
rmse = sqrt(total/length(testset));
```

C.2 Stochastic gradient descent (SGD)

```
function [rmse] = SGD(trainset,testset,total_users,total_movies,k,alpha,e)
% Performs SGD with k-dimensional vectors, learning rate alpha and e epochs on training ...
    set; returns RMSE when tested on testset.

P = 0.1*randn(total_users,k); % randomly initialize user-feature vectors
Q = 0.1*randn(total_movies,k); % randomly initialize movie-feature vectors

% learn user-feature and item-feature vectors using training set
for i = 1:e % loop over training set e times
    for j = 1:length(trainset)
        user = trainset(j,1);
        movie = trainset(j,2);
        rating = trainset(j,3);
        p = P(user,:);
        q = Q(movie,:);
        error = rating - dot(p,q);
        P(user,:) = p + alpha*error*q; % learning rule for p_u
        Q(movie,:) = q + alpha*error*p; % learning rule for q_i
    end
end

% calculate RMSE on testset
errors = zeros(1,length(testset));
total = 0;
for j = 1:length(testset)
    user = testset(j,1);
    movie = testset(j,2);
    rating = testset(j,3);
    predrating = dot(P(user,:),Q(movie,:));
    if predrating < 1 % clip ratings lower than 1
        predrating = 1;
    elseif predrating > 5 % clip ratings higher than 5
        predrating = 5;
    end
    errors(j) = rating - predrating;
    total = total + (errors(j)^2);
end
rmse = sqrt(total/length(testset));
```

C.3 Biased stochastic gradient descent

```
function rmse = BiasedSGD(trainset,testset,total_users,total_movies,k,alpha,e,lambda)
% Performs biased SGD with k-dimensional vectors, learning rate alpha and e epochs on ...
    training set; returns RMSE when tested on testset.

P = 0.1*randn(total_users,k); % randomly initialize user-feature vectors
Q = 0.1*randn(total_movies,k); % randomly initialize movie-feature vectors
b_i = zeros(1,total_movies); % randomly initialize user biases
b_u = zeros(1,total_users); % randomly initialize movie biases
mu = sum(trainset(:,3))/nnz(trainset(:,3)); % calculate global average

for i = 1:e
    for j = 1:length(trainset)
        user = trainset(j,1);
        movie = trainset(j,2);
        rating = trainset(j,3);
        p = P(user,:);
        q = Q(movie,:);
        error = rating - (mu + b_i(movie) + b_u(user) + dot(p,q));
        b_u(user) = b_i(user) + alpha*(error - lambda*b_u(user)); % learning rule for ...
            user bias
        b_i(movie) = b_i(movie) + alpha*(error - lambda*b_i(movie)); % learning rule ...
            for movie bias
        P(user,:) = p + alpha*(error*q - lambda*p); % learning rule for p_u
        Q(movie,:) = q + alpha*(error*p - lambda*q); % learning rule for q_i
    end
end

% calculate RMSE on testset
errors = zeros(1,length(testset));
total = 0;
for j = 1:length(testset)
    user = testset(j,1);
    movie = testset(j,2);
    rating = testset(j,3);
    predrating = mu + b_i(movie) + b_u(user) + dot(P(user,:),Q(movie,:));
    if predrating < 1 % clip ratings lower than 1
        predrating = 1;
    elseif predrating > 5 % clip ratings higher than 5
        predrating = 5;
    end
    errors(j) = rating - predrating;
    total = total + (errors(j)^2);
end
rmse = sqrt(total/length(testset));
```

D Matlab code for Part 4

D.1 Image compression

```
img = imread('lena.jpg'); % read in Lena image
img = im2double(img);
k = 100;
[U,S,V] = svds(img,k); % compute k-dimensional truncated SVD of image
newimg = U*S*V';
imshow(newimg); % display compressed image
```

D.2 Face recognition

```
function score = facerec(A,k,avg)
% Performs k-dimensional SVD face recognition on face matrix A with average
% face avg

[U,S,V] = svds(A,k);
score = 0;
for i = 1:200
    name = strcat('faces/',num2str(i),'b.jpg'); % name of smiling faces
    img = imread(name);
    img = im2double(img);
    img = reshape(img,[75000 1]); % reshape face to vector
    img = img - avg; % normalize face
    img = img'*U*inv(S); % map face to face-space
    res = knnsearch(V,img,'K',1,'Distance','cosine'); % find nearest neutral face
    if ismember(i,res)
        score = score + 1; % increment score by 1 if face is correctly recognized
    end
end
```

E Material for Example 2

E.1 Book titles for Example 2

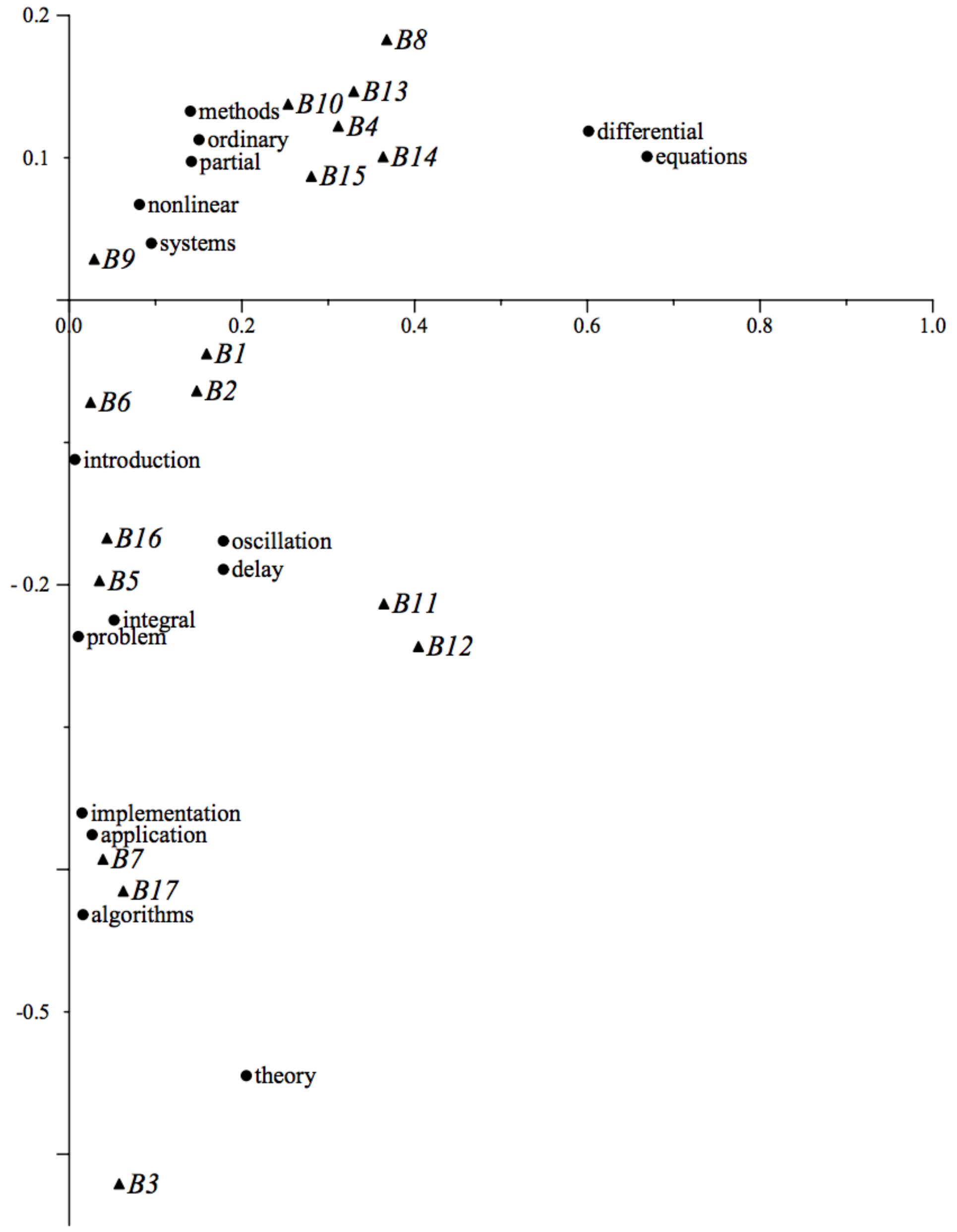
1. B_1: A Course on Integral Equations
2. B_2: Attractors for Semigroups and Evolution Equations
3. B_3: Automatic Differentiation of Algorithms: Theory, Implementation, and Application
4. B_4: Geometrical Aspects of Partial Differential Equations
5. B_5: Ideals, Varieties, and Algorithms An Introduction to Computational Algebraic Geometry and Commutative Algebra
6. B_6: Introduction to Hamiltonian Dynamical Systems and the N-Body Problem
7. B_7: Knapsack Problems: Algorithms and Computer Implementations
8. B_8: Methods of Solving Singular Systems of Ordinary Differential Equations
9. B_9: Nonlinear Systems
10. B_{10}: Ordinary Differential Equations
11. B_{11}: Oscillation Theory for Neutral Differential Equations with Delay
12. B_{12}: Oscillation Theory of Delay Differential Equations
13. B_{13}: Pseudodifferential Operators and Nonlinear Partial Differential Equations
14. B_{14}: Sinc Methods for Quadrature and Differential Equations
15. B_{15}: Stability of Stochastic Differential Equations with Respect to Semi-Martingales
16. B_{16}: The Boundary Integral Approach to Static and Dynamic Contact Problems
17. B_{17}: The Double Mellin-Barnes Type Integrals and Their Applications to Convolution Theory

E.2 Term-document matrix for Example 2

Table 6: Term-document matrix corresponding to Example 1

	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}	B_{16}	B_{17}
algorithms	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0
application	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
delay	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
differential	0	0	0	1	0	0	0	1	0	1	1	1	1	1	1	0	0
equations	1	1	0	1	0	0	0	1	0	1	1	1	1	1	1	0	0
implementation	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
integral	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
introduction	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
methods	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0
nonlinear	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
ordinary	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
oscillation	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
partial	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
problem	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0
systems	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0
theory	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1

E.3 Term-document plot for Example 2


B8
B13
B10
B4
B14
B15
methods
ordinary
partial
differential
equations
nonlinear
systems
B9
B1
B2
B6
introduction
B16
oscillation
delay
B5
B11
integral
problem
B12
implementation
application
B7
B17
algorithms
theory
B3