

Research Internship

April 23, 2007 – September 14, 2007

Zürich, Switzerland

Large scale Singular Value Decomposition and applications in Machine Learning

Latent Semantic Analysis – Collaborative Filtering

Aurélien BOFFY

Engineer student

Supervisors: Gökhan BAKIR – Olivier BOUSQUET



École Normale Supérieure de Cachan
61 avenue du Président Wilson
94235 Cachan cedex
France
+33 1 47 40 20 00



Google Switzerland GmbH
Freigutstrasse 12
8002 Zurich
Switzerland
+41 44 668 1800

Confidentiality

A significant part of the work I did at Google involves confidential information that cannot be included in such a report. Thus, this document does not contain all the work I did during this internship. I am not allowed to describe the context of the internship and the concrete applications of my work (an important part of the code is already used in production). This report only contains public methods, algorithms and applications.

Contents

Introduction	5
Host company	6
Google	6
Zürich office	6
Work environment	6
I Large scale Singular Value Decomposition	8
1 What is SVD?	9
1.1 Definition	9
1.2 Properties	10
1.2.1 Truncated SVD	10
1.2.2 Relation to Eigenvalue Decomposition	10
1.3 Characteristics of our problem	11
1.3.1 Large matrix	11
1.3.2 Sparse matrix	11
1.3.3 Partial SVD	12
1.3.4 Parallel implementation	12
2 Methods to compute SVD	13
2.1 LAPACK (full dense SVD)	13
2.1.1 QR decomposition	13
2.1.2 Bidiagonalization	14
2.1.3 Analysis	15
2.2 Power Method	17
2.2.1 Standard Power Method	17
2.2.2 Adaptation to our problem	18
2.2.3 Analysis	19
2.3 Generalized Hebbian Algorithm	21
2.3.1 Oja Model	21
2.3.2 Stanger Model	22
2.3.3 Analysis	23

2.4	Sampling methods	23
2.4.1	Overall view	24
2.4.2	Results	24
2.5	Lanczos algorithm	25
2.5.1	Overview	25
2.5.2	Parallel computation	26
2.5.3	Experimental results	26
II	Applications	28
3	Latent Semantic Analysis	29
3.1	Motivation	29
3.2	Principle	30
3.3	Usage of the SVD for Latent Semantic Analysis	30
3.3.1	Vector Space Model	30
3.3.2	Interpretation of the SVD model	31
3.3.3	Interests of this low-rank approximation	32
3.3.4	Visualization purpose	33
3.4	Preprocessing of the document-term matrix	34
3.4.1	Tf-idf scheme	34
3.4.2	Other normalization methods	35
3.5	Relatedness measure	35
3.5.1	Comparing two terms or two documents	35
3.5.2	Comparing a term and a document	36
3.5.3	Considering new queries	36
3.5.4	Other similarity measures	37
3.6	Evaluation – Word List Expander	38
3.6.1	Precision-Recall trade-off	38
3.6.2	Experimental results	39
4	Collaborative Filtering	42
4.1	Presentation	42
4.1.1	The Netflix database	42
4.1.2	Common approaches	43
4.2	Missing values	44
4.2.1	Shifted columns	45
4.2.2	Gradient descent	45
4.3	Other approaches	50
4.3.1	Using a movie relatedness matrix	50
4.3.2	Taking user profiles into account	51
	Conclusion	53

List of Figures

2.1	<i>Computational time of the Lapack method to compute the SVD for a Latent Semantic Analysis application.</i>	15
2.2	<i>Lapack method running time is $O(\min(mn^2, m^2n))$.</i>	16
2.3	<i>The power method is much faster than Lapack, because we do not need to compute the full SVD.</i>	20
2.4	<i>Computational time of the Lanczos algorithm on a single machine.</i>	27
3.1	<i>Projection of a corpus of 10,000 labelled documents on the resulting 5-dimensional LSA-space.</i>	33
3.2	<i>Typical trade-off between precision and recall.</i>	39
3.3	<i>Quality of the generated list with respect to the number of dimensions of the concept-space and to the relatedness measure.</i>	40
4.1	<i>The estimated ratings are clipped to the range 1-5.</i>	51
4.2	<i>Functions used to take user rating profiles into account during the learning stage.</i>	52

Introduction

Singular Value Decomposition (SVD) based algorithms are for example used to recognize data in noisy environments or to reduce data storage requirements. They are used in many different areas, especially in Image Processing and in Machine Learning. In particular, Google could use SVD in different applications of data modeling/mining, like *Latent Semantic Analysis* (LSA), *Recommender Systems*, or other information retrieval algorithms. Unfortunately, SVD algorithms typically have cubic complexity. As Google is usually working on very large corpora of text documents (of the order of billions) where each document is represented as a very large feature vector, performing SVD is a challenge. The initial goal of this internship was to come up with a method to perform SVD for very large matrices using the massively parallel Google infrastructure.

During the first part of the internship, I did some literature review and tried different algorithms to compute SVD, focusing on the LSA application. Once we had a satisfying method, the next phase was to implement a word list generator (whose goal is to generate automatically a list of words that are related to some input seed words), based on the results of the LSA procedure. This tool is particularly useful for the people that were working in my team. This stage consisted in many experiments and tuning to maximize the quality of the generated list.

Then, we tried to apply our SVD algorithm to another important application, which is *Collaborative Filtering*. This problem has different properties that make our original approach not very efficient. We tried to alleviate these difficulties by using new techniques.

Please note that this report is of course mainly focused on the mathematical and algorithmic aspects of my work. It also describes the different applications and the experimental results. However, even if this represents an important part of my work, I should mention that this internship has also been a great opportunity for me to improve my programming skills. Software development is of course crucial for me because I'm planning to carry on working in the industry (I got an offer for a full-time position at Google). Software architecture, design patterns, or unit testing, are very important for an engineer and I think this internship was really a good complement to the courses I have had so far.

Host company

Google

Google is an American company specializing in Internet search and online advertising. It was founded in 1998 and has about 14,000 employees. In 2006, its revenue was over 10.6 billion dollars.

67% of queries worldwide went through Google in April 2007. Google's main competitors are Yahoo and Microsoft. Although Internet search remains its core business, Google has developed many other applications, ranging from an email service (GMail), to an automated news aggregator (Google News), via a web mapping application (Google Maps).

Google generates revenue mostly by delivering online advertising. Businesses use its AdWords program to promote their products and services with targeted advertising, and third-party websites use the AdSense program to deliver relevant advertisements.

Zürich office

Google is growing very rapidly. The company is hiring about 25 employees every week and receives more than 1,000 resumés a day. Google has many offices all over the world. The first engineering center in Europe has been opened in Zürich in 2004. More than 350 engineers are working in this office on many different projects. I am not allowed to give any further details about the projects that are currently being worked on in Zürich or about the team I was working with.

Work environment

Google is known for its overall quality of life. Offices are offering a lot of work facilities, ranging from the direct connectivity to the Google network to the amazing video-conferencing coverage which allows *Googlers* from Zürich to work easily with employees of Mountain View. The company just makes sure that employees do not have to worry about any practical issues. Thus many benefits are offered to employees, like the famous free gourmet food,

on-site massages, along with numerous social events that happen along the year (*i.e.*, ski trips, cinema nights...).

Furthermore, employees work in a very diverse and multi-cultural environment. In the Google Zürich office, there are people from more than 35 countries.

From an engineering point of view, Google is also a great place to work. First because all the people you work with are really smart and it's a great feeling because whatever question coming to your head, you can be sure to get a good answer. For instance, the authors of Vim and GZip, Bram Moolenaar and Jean-Loup Gailly, are working at Google Zürich. I was supervised by very good people (Gökhan Bakır and Olivier Bousquet), and several other well-known people were working in the same office as me, like Thomas Hofmann. These engineers try to stay close to the academic community. For instance, I was taking part to a Machine Learning Reading Group every week, were people interested in Machine Learning simply met to discuss about new papers that could be interesting for Google applications. Technical talks on diverse topics (including non-technical or non-Google-related lectures) are also held in a weekly basis.

Another advantage of working at Google for an engineer is the resources that Google is offering, in terms of computational capacity. It becomes easy to run job on parallel and to work on huge data.

In order to get consistent code, and to optimize the engineering work, all the projects share the same codebase. Moreover, Google wrote or wrapped tools to provide a simple and reliable development environment: build tools, unit testing and performance frameworks, bug reporting, version control systems...

Part I

Large scale Singular Value
Decomposition

Chapter 1

What is SVD?

1.1 Definition

Let's start with the definition of the *Singular Value Decomposition* (SVD). We will see some of its properties afterwards and explain why it is so powerful.

Suppose X is an $m \times n$ matrix of real-valued data. The Singular Value Decomposition of X consists of writing X as the product of 3 matrices:

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

where:

- Σ is a diagonal matrix whose diagonal entries are non-negative and are called *singular values*. By convention, they are usually arranged in descending order.
- U and V are matrices whose columns are orthonormal: the norm of each is 1 and the inner product between 2 different columns is 0 (*i.e.*, $U^T U = I$ and $V^T V = I$). The columns of U and V are called the *left* and *right singular vectors*, respectively, for the corresponding singular values.

More schematically:

$$\begin{array}{cccc} X & & U & & \Sigma & & V^T \\ \left(\begin{array}{ccc} x_{1,1} & \cdots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{m,1} & \cdots & x_{m,n} \end{array} \right) & = & \left(\begin{array}{c} [u_{1,1}] \\ \vdots \\ [u_{m,1}] \end{array} \right) \cdots \left(\begin{array}{c} [u_{1,r}] \\ \vdots \\ [u_{m,r}] \end{array} \right) & & \left(\begin{array}{ccc} \sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_r \end{array} \right) & & \left(\begin{array}{ccc} [v_{1,1}] & \cdots & v_{1,n}] \\ \vdots & & \\ [v_{r,1}] & \cdots & v_{r,n}] \end{array} \right) \end{array}$$

where r is the rank of X ($r \leq \min(m, n)$).

Note that an entry $x_{i,j}$ of the original matrix X can be computed with this equation:

$$x_{i,j} = \sum_{k=1}^r u_{i,k} \sigma_k v_{k,j} = \mathbf{u}_{i,\cdot} \Sigma \mathbf{v}_{j,\cdot}^T$$

where $\mathbf{u}_{i,\cdot}$ represents the i th row of the matrix U .

1.2 Properties

1.2.1 Truncated SVD

One of the main interests of the SVD is that it provides a simple method to get an optimal approximation of X . When you want to approximate a given matrix with a "simpler" one, it is very common to use the *rank* as a measure of "complexity". Thus, we aim at finding the matrix X_k whose rank equals $k < r$, which is the best approximation of X for a certain measure. The most common measure of discrepancy is the sum-squared error, or the *Frobenius distance* (Frobenius norm of the difference) between X and Y :

$$\|X - Y\|_F^2 = \sum_{i,j} (X_{i,j} - Y_{i,j})^2$$

When multiplying all three matrices U , Σ and V^T together to reconstruct the original matrix, the singular values act as weights of the singular vectors:

$$X = \sum_{t=1}^r \sigma_t \mathbf{u}_{\cdot,t} \mathbf{v}_{\cdot,t}^T$$

The matrix is seen as a sum of r rank 1 matrices of decreasing importance. A singular vector with a large corresponding singular value has a large impact on the reconstruction, while small value indicates a singular vector with almost no impact on the result. Thus small values and their vectors may be discarded without affecting the result noticeably.

The *truncated SVD* consists of retaining only the largest k singular values and the corresponding columns of U and V . The truncated matrices can be multiplied together and it turns out that $U_k \Sigma_k V_k^T = X_k$: it is the best rank- k approximation (closest in the least square sense, *i.e.*, for the Frobenius norm) to the original matrix X :

$$U_k \Sigma_k V_k^T = X_k = \operatorname{argmin}_{\operatorname{rank}(Y)=k} \|X - Y\|_F$$

1.2.2 Relation to Eigenvalue Decomposition

We can notice that the SVD is closely related to the regular *Eigenvalue Decomposition* of a square symmetric matrix Y in $Y = W D W^T$ where W

is orthogonal and D is diagonal. As a matter of facts, U is the matrix of the eigenvectors of XX^T and V is the matrix of the eigenvectors of $X^T X$, Σ^2 being in both cases the matrix of eigenvalues:

$$XX^T = U\Sigma^2U^T \quad X^T X = V\Sigma^2V^T$$

1.3 Characteristics of our problem

The Singular Value Decomposition has many applications. We will consider some of them later in this report. Nevertheless, the different applications that were of interest for my supervisors at Google have some properties in common that I had to keep in mind during the beginning of the internship, particularly for the literature review.

1.3.1 Large matrix

One of the main constraints for this work was the size of the data. This is very common at Google since they are dealing with a huge amount of information. To have a first idea of this issue, let's simply address the case of a *document-term matrix*, because I was working with such matrices during a long part of my internship. These matrices are very common in natural language processing to represent documents as mathematical objects (matrices). Each row of such a matrix simply represents a document, while each column represents a word. If a given term appears in a document, the corresponding entry is non-zero¹, and the entry is zero otherwise.

As Google is dealing with the entire Web, the number of documents is of the order of billions. The number of terms that appears on Internet is also huge, mostly because of typographic mistakes and proper names. We will see later that it is often useful to get rid of infrequent terms and we can finally approximate the number of columns to one million.

To summarize, we can consider that we have to handle matrices that have of the order of a billion rows and a million columns.

1.3.2 Sparse matrix

Although matrices we have to deal with are very big, they are also particularly sparse. For instance for the document-term matrix², we can consider that a document contains on average 200 different terms. Thus, only $200/10^6 = 0.02\%$ of the entries are non-zeros.

This characteristic is of course very important in regards to the methods used to perform SVD. Handling such matrices is different from handling

¹It may be 1 in case of a binary matrix, it may be the number of occurrences of the term in the document, etc.

²It is roughly the same for the other matrices I had to deal with.

regular ones, since it is much more efficient to only store the non-zero entries. It is crucial to think about algorithm design that allows this sparsity to be preserved during execution. Otherwise, the space needed to store the data could be multiplied by a factor of several thousands and the subsequent computations would take much more time.

1.3.3 Partial SVD

The most important property of SVD used by the different applications is related to the truncated SVD described in section 1.2.1 page 10. We usually only need to compute the first k singular values and their corresponding singular vectors. This is a property of the problem that we really need to take into account because it reduces the amount of work quite significantly. Indeed, for a typical $10^9 \times 10^6$ matrix, performing the full SVD corresponds to computing about 10^6 singular values and corresponding singular vectors, while we usually only need to consider the first few hundreds.

1.3.4 Parallel implementation

Finally, as soon as we have some information about the problem and especially its size, we understand that we won't be able to solve it on a single machine, because of the required storage space and computational time. It is good to have this in mind from the start, so that we can quickly discard methods that are impossible to parallelize. Some computational tasks (like matrix multiplications) are easy to parallelize, while others are not.

Chapter 2

Methods to compute SVD

During the first few weeks, I spent a significant part of my time reading papers about SVD and the different ways that have been tried so far to compute it quickly. I have implemented several methods, in order to have more insights about the problems and to have some points of comparison.

We introduce some methods in the following sections, focusing on the characteristics of our problem.

2.1 LAPACK (full dense SVD)

The most common method used to perform SVD is the one implemented in *LAPACK* (Linear Algebra PACKage) and in most popular SVD algorithms. For instance, the `svd` command of Matlab uses LAPACK routines.

The basis of these methods lies in the reduction of the original matrix X to a bidiagonal form (*i.e.*, a matrix where only the main diagonal and first superdiagonal entries are non-zero) by using orthogonal transformations called *Householder reflections* [11]. It is then easy to compute the SVD of this bidiagonal matrix by using common methods for the computation of eigenvalues of symmetric matrices.

2.1.1 QR decomposition

Householder reflections are usually used to calculate *QR decomposition* of a matrix: $A = QR$ where Q is an orthogonal matrix and R is an upper triangular matrix.

A Householder reflection is an orthogonal matrix H used to zero selected components of a vector. For an arbitrary vector, there is a Householder

reflection H so that

$$H \begin{pmatrix} * \\ * \\ * \\ * \\ * \end{pmatrix} = \begin{pmatrix} * \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

This can be used to gradually transform an $m \times n$ matrix to upper triangular form. First we want to zero the first column (except the first element):

$$H_1 \begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{pmatrix} = \begin{pmatrix} * & * & * & * \\ 0 & \otimes & * & * \\ 0 & \otimes & * & * \\ 0 & \otimes & * & * \\ 0 & \otimes & * & * \end{pmatrix}$$

In the next step, we focus on the highlighted entries and determine a Householder matrix \hat{H}_2 so that

$$\hat{H}_2 \begin{pmatrix} \otimes \\ \otimes \\ \otimes \\ \otimes \end{pmatrix} = \begin{pmatrix} \otimes \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

If $H_2 = \text{diag}(I_1, \hat{H}_2)$, then

$$H_2 H_1 \begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{pmatrix} = \begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{pmatrix}$$

We repeat this process until we get an upper triangular matrix. As each Householder matrix is orthogonal, we have obtained a QR decomposition.

2.1.2 Bidiagonalization

The process is very similar to bidiagonalize the initial matrix. We use Householder transformations to alternatively zero parts of the columns and of the rows of the matrix:

$$\begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{pmatrix} \rightarrow \begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{pmatrix} \rightarrow \begin{pmatrix} * & * & 0 & 0 \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{pmatrix} \rightarrow \begin{pmatrix} * & * & 0 & 0 \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{pmatrix} \rightarrow \\
 \begin{pmatrix} * & * & 0 & 0 \\ 0 & * & * & 0 \\ 0 & 0 & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{pmatrix} \rightarrow \begin{pmatrix} * & * & 0 & 0 \\ 0 & * & * & 0 \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \\ 0 & 0 & 0 & * \end{pmatrix} \rightarrow \begin{pmatrix} * & * & 0 & 0 \\ 0 & * & * & 0 \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Note that it is not possible to diagonalize the matrix by using this method. Once the first column has been zeroed (except the first element), it is not possible to zero all the entries of the first row (except the first element), because it would un-zero the first column.

Once the matrix has been bidiagonalized, the SVD of a bidiagonal matrix can be computed very efficiently, as described by Dhillon and Parlett [8] for example. This last stage is not critical from a computational complexity perspective.

2.1.3 Analysis

This method (and other similar ones) is certainly the best one today to compute the SVD of a given arbitrary matrix. I have tried it for a document-term matrix (see section 1.3.1 page 11). Figure 2.1 shows the computational time in seconds with respects to the number of documents and with respect to the number of words (when a new document is added, it also adds new terms in the index, so both graphs are correlated).

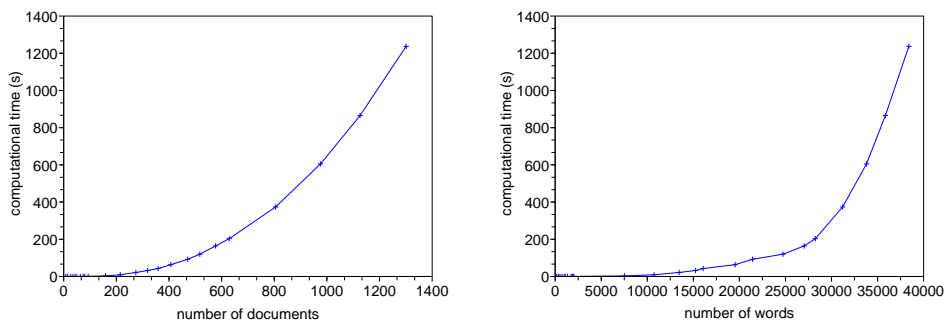


Figure 2.1: *Computational time of the Lapack method to compute the SVD for a Latent Semantic Analysis application.*

Actually, it can be shown (see [11]) that the computational complexity of the bidiagonalization step that we have seen before is $O(\min(mn^2, m^2n))$, where $m \times n$ is the size of the matrix. This is also the complexity of the complete SVD because subsequent steps have a lower cost. This is validated by Figure 2.2.

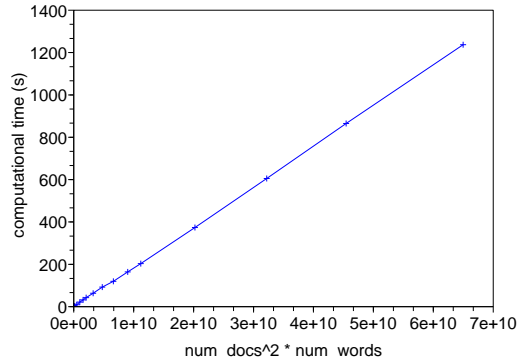


Figure 2.2: *Lapack method running time is $O(\min(mn^2, m^2n))$.*

As you can see on these figures, we are very far from an acceptable algorithm for our purposes: it takes about 20 minutes for a matrix with only 1,200 documents, while we want to be able to handle billions of documents. Of course, we are using a single machine but it is obvious that a cubic running time is not admissible for us. Different reasons can explain this inadequacy:

- First, this method computes the full SVD of the matrix, while we have explained that we only need a few hundreds singular values and associated singular vectors.
- Second, the bidiagonalization technique does not take into account the sparsity of the matrix. Even it is initially sparse, it becomes dense after the first iteration:

$$H \begin{pmatrix} * & * & 0 & 0 \\ * & 0 & 0 & 0 \\ 0 & 0 & * & 0 \\ * & * & 0 & * \\ * & 0 & * & 0 \end{pmatrix} = \begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{pmatrix}$$

Thus, not only this method is very slow, but it is also not usable from a data storage perspective.

Nevertheless, trying this common algorithm and testing it on real data allowed me to have a better understanding of the different issues, and to have an initial point of comparison.

2.2 Power Method

The next method we wanted to try to perform SVD is called the *Power Method*. It is one of the simplest methods for finding the largest eigenvalue and corresponding eigenvector of a square, diagonalizable matrix. We will first consider this technique, and then we will see how we can adapt it for our purposes.

2.2.1 Standard Power Method

Suppose A is an $n \times n$ diagonalizable matrix. The algorithm starts with a vector $\mathbf{b}_0 \in \mathbb{R}^n$, which may be a random vector, or an approximation of the dominant eigenvector if available. Then, the method is described by the iteration

$$\mathbf{b}_{k+1} = \frac{A\mathbf{b}_k}{\|A\mathbf{b}_k\|}$$

Under the following assumptions

- A has an eigenvalue λ_1 that is strictly greater in magnitude than the other ones.
- \mathbf{b}_0 has a nonzero component in the direction of the corresponding dominant eigenvector \mathbf{v}_1 .

then (\mathbf{b}_k) converges to an eigenvector associated with the dominant eigenvalue. Consequently, as $A\mathbf{v}_1 = \lambda_1\mathbf{v}_1$, the sequence $\left(\frac{\mathbf{b}_k^T A\mathbf{b}_k}{\mathbf{b}_k^T \mathbf{b}_k}\right)$ converges to λ_1 .

This can be proven easily: let $\lambda_1, \dots, \lambda_n$ be the n eigenvalues of A and let $\mathbf{v}_1, \dots, \mathbf{v}_n$ be the corresponding eigenvectors. Suppose that λ_1 is the dominant eigenvalue, *i.e.*, $|\lambda_1| > |\lambda_j|$ for $j > 1$. As the n eigenvectors are linearly independent, they form a basis for n -dimensional space. Hence, the starting vector \mathbf{b}_0 can be expressed as the linear combination

$$\mathbf{b}_0 = \sum_{i=1}^n c_i \mathbf{v}_i$$

We have assumed that $c_1 \neq 0$ ¹. Then it follows that

$$A^k \mathbf{b}_0 = \sum_{i=1}^n c_i A^k \mathbf{v}_i = \sum_{i=1}^n c_i \lambda_i^k \mathbf{v}_i = c_1 \lambda_1^k \underbrace{\left(\mathbf{v}_1 + \sum_{i=2}^n \frac{c_i}{c_1} \left(\frac{\lambda_i}{\lambda_1}\right)^k \mathbf{v}_i \right)}_{\rightarrow \mathbf{v}_1}$$

Since $\mathbf{b}_k \propto A^k \mathbf{b}_0$, we conclude that (\mathbf{b}_k) converges to (a multiple of) the eigenvector \mathbf{v}_1 . We can also observe that the convergence is geometric, with ratio $|\frac{\lambda_1}{\lambda_2}|$.

¹It is the case with probability 1 if \mathbf{b}_0 is chosen randomly.

2.2.2 Adaptation to our problem

Computing the dominant singular value and associated singular vectors

The power method was originally designed to compute the dominant eigenvalue of a diagonalizable square matrix. However, we are working with rectangular matrices and we want to calculate the singular values. We can use the relations between the Eigenvalue Decomposition and the SVD that we have seen in section 1.2.2 page 10.

In order to compute the left and right singular vectors associated with the dominant singular values, we now start with 2 vectors \mathbf{u}_0 and \mathbf{v}_0 , which may be random vectors as well. Then, the algorithm is described by these iterations:

$$\begin{aligned}\mathbf{u}_{k+1} &= \frac{A\mathbf{v}_k}{\|A\mathbf{v}_k\|} \\ \mathbf{v}_{k+1} &= \frac{A^T\mathbf{u}_{k+1}}{\|A^T\mathbf{u}_{k+1}\|}\end{aligned}$$

This is equivalent to

$$\begin{aligned}\mathbf{u}_{k+1} &= \frac{AA^T\mathbf{u}_k}{\|AA^T\mathbf{u}_k\|} \\ \mathbf{v}_{k+1} &= \frac{A^T A\mathbf{v}_k}{\|A^T A\mathbf{v}_k\|}\end{aligned}$$

Thus, (\mathbf{u}_k) converges to an eigenvector associated with the dominant eigenvalue of the symmetric square matrix AA^T and (\mathbf{v}_k) to an eigenvector associated with the dominant eigenvalue of the symmetric square matrix $A^T A$. Referring to section 1.2.2, we conclude that these are also the left and right singular vector associated with the dominant singular vector of A . Note that we never have to compute the matrices AA^T and $A^T A$ explicitly (they would be very large).

Computing several singular value and associated singular vectors

The standard power method can find only the dominant eigenpair $(\lambda_1, \mathbf{v}_1)$. For our purposes, we need to compute some subsequent ones². If A is symmetric³, it can be proven that if $\mathbf{u}_1 = \frac{\mathbf{v}_1}{\|\mathbf{v}_1\|}$, then $A' = A - \lambda_1\mathbf{u}_1\mathbf{u}_1^T$ has the same eigenvalues and eigenvectors as A , except that λ_1 has been replaced by 0. Thus, we just have to apply the power method to A' , and so on. This is called the *Deflation Method*.

²Actually, we need to compute singular values but we just saw that we can still use similar iterations.

³In our case, AA^T and $A^T A$ are of course symmetric.

We definitely do not want to compute the matrix $A - \lambda_1 \mathbf{u}_1 \mathbf{u}_1^T$ that is not sparse anymore. Thus, what we are actually doing in the code is maybe closer to the *Gram-Schmidt Process*, even if it is equivalent to the Deflation Method. The Gram-Schmidt process is a method for orthogonalizing a set of vectors. Practically speaking, to compute the i th dominant singular value and associated singular vectors, we restart the power method on A , but at each iteration, we orthogonalize the newly computed singular vectors with regards to the previously computed ones. Let \mathbf{u}^j and \mathbf{v}^j denote the singular vectors associated to the j th singular values. Let suppose that the first $i - 1$ singular values and corresponding singular vectors have already been computed and that we wish to compute the next ones. The algorithm is described by these iterations:

$$\begin{aligned}\mathbf{u}_{k+1}^i &= \frac{A\mathbf{v}_k^i}{\|A\mathbf{v}_k^i\|} \\ \mathbf{u}_{k+1}^i &\leftarrow \mathbf{u}_{k+1}^i - \sum_{j=1}^{i-1} \text{proj}_{\widetilde{\mathbf{u}}^j} \mathbf{u}_{k+1}^i \\ \mathbf{v}_{k+1}^i &= \frac{A^T \mathbf{u}_{k+1}^i}{\|A^T \mathbf{u}_{k+1}^i\|} \\ \mathbf{v}_{k+1}^i &\leftarrow \mathbf{v}_{k+1}^i - \sum_{j=1}^{i-1} \text{proj}_{\widetilde{\mathbf{v}}^j} \mathbf{v}_{k+1}^i\end{aligned}$$

where

- $\widetilde{\mathbf{u}}^j$ and $\widetilde{\mathbf{v}}^j$ represent the previously estimated singular vectors.
- $\text{proj}_{\mathbf{v}} \mathbf{u}$ is the projection operator defined by

$$\text{proj}_{\mathbf{v}} \mathbf{u} = \frac{\langle \mathbf{u}, \mathbf{v} \rangle}{\|\mathbf{v}\|^2} \mathbf{v} = \frac{\langle \mathbf{u}, \mathbf{v} \rangle}{\langle \mathbf{v}, \mathbf{v} \rangle} \mathbf{v}$$

Thus, to orthogonalize a vector, we simply project it orthogonally onto the subspace generated by the singular vectors that have already been estimated and the result is the difference between the original vector and this projection.

We can easily see that this technique is equivalent to the Power Method applied to $A - \lambda_1 \mathbf{u}_1 \mathbf{u}_1^T$, but it allows us to work only on sparse matrices.

2.2.3 Analysis

Compared to the method of Lapack, the Power Method has several advantages for our problems:

- It is adapted to sparse matrices. Extra storage is needed only for each singular vector and value.
- The Power Method is very easy to code and to adapt to a parallel implementation, because it only needs matrix-vector multiplications. On the contrary, Lapack is a library written in Fortran and could be more difficult to adapt to our purposes. Furthermore, it would certainly be more difficult to make this package work in parallel.
- This method is suitable when we do not need to compute the entire SVD of a matrix.

Thus, depending on the degree of sparsity of the matrix and of the number of singular values we need to compute, this method can be much faster than Lapack. Figure 2.3 shows the computational time in seconds with respects to the number of estimated singular values and associated singular vectors. This graph was obtained for a document-term matrix of 10,000 documents (*i.e.*, 10,000 rows), each one containing on average 280 different words (*i.e.*, 280 non-zero entries per row on average). The stopping criteria I used was the relative precision of the singular values. This graph shows the computational times for three different precision parameters.

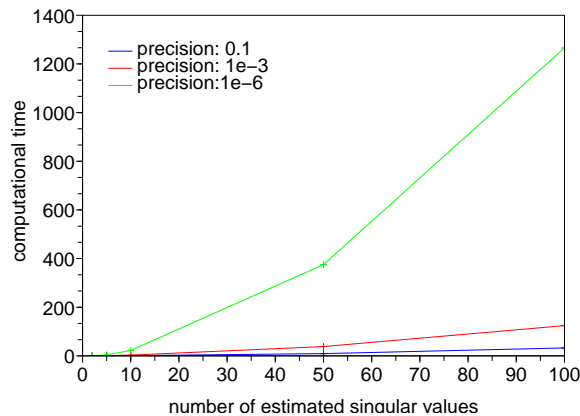


Figure 2.3: *The power method is much faster than Lapack, because we do not need to compute the full SVD.*

As you can see, to compute the largest 100 singular values of a document-term matrix that contains 10,000 documents, this method needs 1 minute for a precision of 10^{-3} and 20 minutes for a precision of 10^{-6} , while the technique implemented in Lapack needs about 10 minutes for a matrix of only 1,000 documents (see Figure 2.1). Note that these experiments have always been runned on a single machine.

However, this method is not sufficient because it becomes more inaccurate as we calculate more singular values. Error is introduced at each step and this error accumulates as the process continues. Beyond the first few singular vectors and values, rounding error reduces the accuracy below acceptable limits. Furthermore, even it is faster than Lapack, it is still too slow, considering the huge data that we need to process.

2.3 Generalized Hebbian Algorithm

In this section, we consider an approach which is quite different from the previous ones, even it has some relations with the power method, since it only requires matrix-vector multiplications. This method uses the *Generalized Hebbian Algorithm* (GHA), which was introduced by Oja and Karhunen in 1984 [16]. They demonstrated that Hebbian learning could be used to derive the first eigenvector of a dataset given observations (vectors) presented serially. The idea was extended in 1989 by Sanger [19] to allow further eigenvectors to be computed within the same framework. It has been applied by Gorrell for language processing applications in 2006 (see for example [12] and [13])⁴.

The term *Hebbian* is usually used to describe unsupervised learning algorithms that consist of a neural network whose weights are continuously adapted to the input data. The goal of Hebb rules is to reinforce what happens often: to simplify, the network is satisfied with what it has produced and reinforces it (*self-satisfaction* principle), without knowing if it was good or not.

2.3.1 Oja Model

GHA is originally used to compute the first eigenvector of a dataset (the relation between eigenvectors and singular vectors has been examined in section 1.2.2 page 10). This is related to *Principal Component Analysis* (PCA). The data consists of a set of m observations, which are represented by n variables. The observations are considered as row vectors of an $m \times n$ matrix A . In the LSA case, A is the document-term matrix: an observation is a document and n is the number of words in the dictionary.

PCA usually involves the computation of the eigenvectors of the covariance matrix $A^T A$. The Oja model allows to compute the first eigenvector of $A A^T$ by presenting the data (*i.e.*, the documents, the rows of A) serially, to a neural network. The n components of an observation are multiplied by connection weights and summed to generate an output. The weights of the

⁴As far as I am concerned, I have studied this kind of algorithms during J.-P. Nadal course *Statistical Modeling Inference in Neurosciences*.

network are then modified according to this output (*Hebb rule*):

$$\mathbf{u}(t+1) = \mathbf{u}(t) + \lambda \underbrace{\left(\mathbf{u}^T(t) \cdot \mathbf{a} \right)}_{\text{output}} \mathbf{a}$$

where:

- \mathbf{u} is the weight vector that converges to the dominant eigenvector of $A^T A$
- \mathbf{a} is an input vector (data observation)
- λ is the learning rate

Intuitively, the eigenvector \mathbf{u} is updated by adding the input vector \mathbf{a} weighted by the output $\mathbf{u}^T \cdot \mathbf{a}$, which represents the extent to which \mathbf{a} already resembles to the eigenvector \mathbf{u} . In this way, the strongest direction in the input comes to dominate.

2.3.2 Stanger Model

This foundation has been extended by Stanger to discover the next dominant eigenvectors. The idea is very similar to the one we have seen in the previous section (Gram-Schmidt process). The update needs to be made orthogonal to previously computed eigenvectors: they are removed from the training update in order to take them out of the picture. Let \mathbf{u}_p denote the p th eigenvector:

$$\mathbf{u}_p(t+1) = \mathbf{u}_p(t) + \lambda \left(\mathbf{u}_p^T(t) \cdot \mathbf{a} \right) \left(\mathbf{a} - \sum_{i \leq p} (\mathbf{u}_i^T \cdot \mathbf{a}) \mathbf{u}_i \right)$$

Note that we also remove the projection on the current eigenvector, in order to keep the vectors normalized.

To recover the original formulation of Stanger, let's define:

- $c_i = (c_{i,1} \ \dots \ c_{i,n})^T$, the i th eigenvector
- $x = (x_1 \ \dots \ x_n)^T$ the input vector
- $y_i = c_i \cdot x$ the activation.

Then

$$c_{ij}(t+1) = c_{ij}(t) + \lambda(t) \left(y_i(t)x_j(t) - y_i(t) \sum_{k \leq i} c_{kj}(t)y_k(t) \right)$$

2.3.3 Analysis

I have implemented this method which has several advantages. The main difference between GHA and the other standard algorithms we have studied is that GHA is an incremental approach. It is not a batch algorithm where we process the whole matrix at once. This has the benefit that the full data matrix does not have to be held in memory. The only persistent storage requirement is the developing singular vectors themselves.

Thus, it is appropriate in a different range of circumstances. It allows larger datasets to be processed⁵. When the entire matrix is not available from the start, GHA is also a very useful algorithm. The incremental approach of GHA seems appropriate to prevent us from having to recompute the SVD each time a new document is added to the dataset.

As noticed by Gorrell, the method is appropriate in situations where the dataset is very large or unbounded and time is not a priority. These are not exactly the concerns for Google and for my supervisors. Yes, the dataset is huge, but Google also has the ability of storing it easily. Because we do not need to work on a single machine, the incremental approach is no longer that interesting. Besides, Google has such an important computing capacity that recomputing the SVD when new documents are added to the dataset is not a major problem.

Computational time is more important for our purposes, and our different experiments were very disappointing because this method is much too slow. We need to loop several times on the whole dataset and to restart the process for each new singular triplet (singular value and associated singular vectors). In addition, as for the Power Method, the error accumulates for each singular triplet.

I finally think that this method is particularly interesting and has many advantages compared to the Power Method, but it is not really adapted to our problem because Google has different constraints (memory is not an issue, no need for adaptivity).

2.4 Sampling methods

In the beginning of my internship, approximating the SVD of a large matrix by using sampling methods was the main idea of my supervisors. Thus, most papers I read during the literature review stage were using this principle. This family of algorithms is of great interest in many cases and for many applications, but we finally realized that using sampling was finally not necessary for Google because its computing resources are so important that standard linear algebra methods applied on the whole matrix are actually sufficient, as we will see in the next section.

⁵Memory becomes quickly the bottleneck, the limiting factor of a standard SVD algorithm, as we will see later.

In the following, we introduce the main principle and mathematical theorem behind these techniques.

2.4.1 Overall view

Instead of computing the SVD of the entire matrix, the main idea of this family of algorithms is to sample a constant number of rows and columns of the matrix, scale them appropriately to form a small matrix, say S , and then compute the SVD of S , which is a good approximation to the SVD of the initial matrix. Accuracy is not very important for our application (for example for LSA): the exact projection to the lower dimensional space is not necessary, since we are usually mainly interested in a nearest neighbor search.

The sampling process used to chose the rows (and the columns) of the sub-matrix is usually based on the norm of these rows.

2.4.2 Results

As in section 1.2.1 page 10, let $X_k = U_k \Sigma_k V_k^T$ be the best rank- k approximation of the original matrix X , and \widehat{X}_k be the rank- k approximation of X obtained by this family of sampling method. The results are usually of the form (see for example [9])

$$\|X - \widehat{X}_k\|_F \leq \|X - X_k\|_F + \epsilon \|X\|_F$$

Note that $\|X\|_F$ might be significantly larger than $\|X - X_k\|_F$.

Other algorithms achieve results of the form (see for example [7])

$$\|X - \widehat{X}_k\|_F \leq (1 + \epsilon) \|X - X_k\|_F$$

These bounds are usually based on the Johnson-Lindenstrauss Lemma [14], which shows that any set of n points in high dimensional Euclidean space could be embedded into an $O\left(\frac{\log n}{\epsilon^2}\right)$ dimensional space without distorting the distances between any pair of points by more than a factor of $1 \pm \epsilon$, for any $0 < \epsilon < 1$.

We also thought about performing SVD on such small sub-matrices several times (in parallel), and then merging the results to reduce the variance and lead to an improved approximation. However, merging different SVD is of course not straightforward because the resulting singular vectors need to stay orthogonal.

I do not go into too many details about these methods, since another algorithm solves the problem more easily. I think that it is important to know when to stop going into one direction when another one already meets the requirements.

2.5 Lanczos algorithm

We finally tried the SVD solvers implemented in SVDPACK and ARPACK (ARnoldi PACKage). Both algorithms are based on the *Lanczos algorithm* but ARPACK is easier to parallelize. I spent some time to adapt both methods to make them work in Google infrastructure, but I obviously didn't have to re-implement them. We explain below what are the main ideas, trying to emphasize the connections with the algorithms that we have seen so far.

2.5.1 Overview

The Lanczos method is a technique used to solve large, sparse, symmetric eigenproblems (we remind that SVD and Eigenvalue Decomposition are closely related. See section 1.2.2 page 10). Thus, we are using it to compute the eigenvalues of $Y = XX^T$ (or $X^T X$). The method involves partial tridiagonalizations of the symmetric matrix. Note that this is similar to the algorithm implemented in Lapack, whose first step was to bidiagonalize the initial rectangular matrix. We have seen that Householder transformations can be adapted for this purpose (see section 2.1.1 page 13). However, they destroy sparsity and intermediate large dense matrices arise during the reduction. This suggests that the elements of the tridiagonal matrix have to be computed directly. This is done during the so-named *Lanczos iteration* [11].

This iteration halts before complete tridiagonalization is obtained. This is the second main contribution of this technique. Information about Y 's extremal eigenvalues tends to emerge long before the tridiagonalization is complete. This makes the Lanczos algorithm very interesting in our case because we only need the largest singular values. More precisely, the method generates a sequence of $k \times k$ tridiagonal matrices T_k with the property that the extremal eigenvalues of T_k are progressively better estimates of the extremal eigenvalues of Y .

We do not enter in the details of the Lanczos iteration but we just mention that it is closely related to the Power Method that we have seen in section 2.3 page 20. Indeed, as for the Power Method, the multiplication by the original matrix (in this case Y) is the only large scale linear operation. Hence, it is easy to parallelize. During the iterations of the power method, whilst getting the ultimate eigenvector $A^n b_0$, we also got a series of vectors $A^i b_0$, $0 < i < n - 1$, which were eventually discarded. The Lanczos algorithm saves these information and uses the Gram-Schmidt process that we have seen in section 2.2.2 page 18, to reorthogonalize them into a basis⁶.

⁶They form the *Krylov subspace*, but we prefer not to enter into too much details.

2.5.2 Parallel computation

We are using *MPI* (*Message Passing Interface*) for the parallel implementation of this method. MPI is a protocol that allows synchronization and communication functionality between computers. Our MPI implementation consists of a specific set of routines callable from the C++ main program.

We already mentioned that the only significant operation of the Lanczos algorithm is the matrix-vector multiplication between $Y = XX^T$ and a given vector. Thus, this is very easy to parallelize. Let's consider the case of a $m \times n$ document-term matrix X . We assume that N machines are available (N could be of the order of hundreds). We simply decide to split the m documents in N parts and each machine $i \in \{1 \dots N\}$ is responsible for building its own document-term matrix X_i . We make sure that these matrices have the same number of columns, even if the words contained in each set of documents may be different.

$$X = \begin{pmatrix} X_1 \\ \vdots \\ X_N \end{pmatrix}$$

Then, implementing the multiplication between XX^T and a given vector is almost straightforward. To multiply the transpose X^T by a given vector \mathbf{u} , each machine i simply performs the multiplication between the transpose of the locally stored matrix X_i and \mathbf{u} , and all the results are then summed. To multiply X by the vector $\mathbf{u}' = X^T \mathbf{u}$, we naturally split \mathbf{u}' into N parts whose sizes correspond to the number of rows of the N local matrices. Each machine multiplies its matrix X_i by its associated part of \mathbf{u}' and the results are then merged.

Of course, we remind that we use a format adapted to sparse matrices to store X , that is, only the nonzero entries are stored.

I just mentioned in this section the parallelization of the matrix-vector product because it is related to the SVD solver. However, almost everything I did during my internship used a parallel implementation, since I was dealing with gigabytes or even terabytes of data almost on a daily basis. This is very common and Google has its own specific tools, like *MapReduce* [5], a framework for distributed computation, based on functional languages' concepts of Map and Reduce.

2.5.3 Experimental results

In order to compare with the previously tested methods, we first tested this algorithm on a single machine, without any distributed computation. Some results are shown on Figure 2.4.

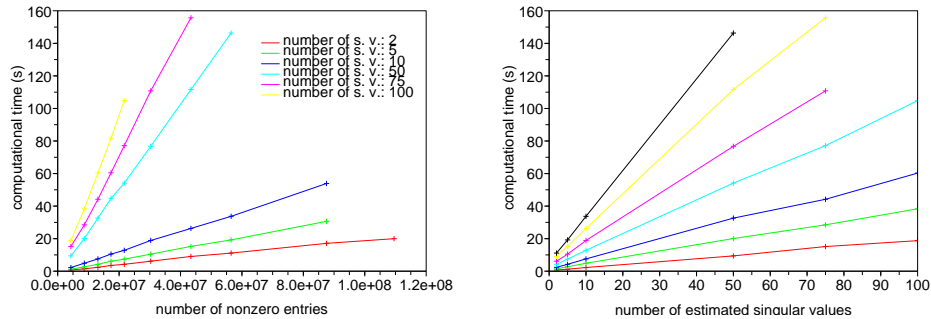


Figure 2.4: *Computational time of the Lanczos algorithm on a single machine.*

The left graph shows the computational time in seconds with respect to the number of nonzero entries in the original matrix X . Note that we still use document-term matrices. Each curve corresponds to a certain number of estimated singular triplets (singular values and associated singular vectors). On the contrary, the right figure represents the computational time with respect to the number of computed singular triplets and each curve corresponds to a certain matrix size.

The main purpose of these two figures is to show that computational time is linear in the number of nonzero entries and in the number of estimated singular triplets. This is of course incomparably better for our purposes than the initial algorithm, whose complexity was $O(\min(mn^2, m^2n))$, where $m \times n$ is the size of X . For instance, on a single machine, we can now compute the largest 100 singular values of a matrix containing 100,000 documents in about 2 minutes, while it would have taken about 6 years of computation with the method implemented in Matlab and Lapack (if we assume that we have enough memory to store the full matrix, which is of course impossible).

We can also notice that the limiting factor is now memory: Figure 2.4 has been obtained by running experiments with the biggest dataset I could use on a single machine before running out of memory. This explains why methods like Generalized Hebbian Algorithm (see section 2.3 page 21) or Sampling techniques (see section 2.4 page 23) are very interesting for many people because the main bottleneck is now memory and not time.

Of course the implementation is originally designed to be used in parallel. Without going into confidential details, let me just say that this method is usually used to perform SVD on a document-term matrix containing a number of documents which is usually of the order of the whole French Web. It then runs on a few hundreds of machines.

These results met the requirements of my supervisors and we could then think in more details about different applications of this tool.

Part II

Applications

Chapter 3

Latent Semantic Analysis

3.1 Motivation

Latent Semantic Analysis is one of the most famous *Information Retrieval* technique. Information retrieval is the science of searching for documents or for information in documents. Web search engines such as Google are certainly the most visible Information Retrieval applications.

The problem is very simple: a user enters a query (usually a few words), and the goal is to retrieve documents from the Web that are *relevant* for this query. Most approaches work by literally matching terms in documents with those of the query. Thus, documents that contain the most number of occurrences of the query terms are simply returned.

These methods have several important drawbacks. Users think about a certain conceptual content that they express with some words. The problem is that these words, considered individually, do not provide reliable evidence about what the user is thinking about. We usually distinguish two main difficulties:

Synonymy: There are usually many ways to express a given concept. Users in different contexts, knowledge or linguistic habits will describe the same idea using different terms. Thus, many relevant documents usually use a different vocabulary than the one used in the query. These documents can not be retrieved.

Polysemy: Most words have more than one distinct meaning. In different contexts or when used by different people, the same term (*e.g.* "surfing") takes on varying significances ("surfing the web", "surfing at Malibu beach"). Returning documents that contain all or at least some of the words in the query is usually not optimal, since many documents may contain some query terms, while dealing with totally different topics.

To summarize, some documents may be relevant without being returned because they do not contain the specific terms of the query, while non-relevant documents could be retrieved because they contain some query words, even if these words are used with two different meanings in the document and in the query.

3.2 Principle

Latent Semantic Analysis (LSA) is a method that was introduced by Deerwester *et al.* in 1990 [6]. The main principle of LSA is to assume that there is some underlying (*latent*) semantic structure in the data that is partially hidden by the randomness of word choice. The goal is to use statistical techniques to estimate this latent structure and get rid of the obscuring noise.

A document (or a query) is actually considered as being a sample of an ideal document (which is much longer) that would contain all the words related to the considered topic. The goal of LSA is to find out what are the terms that are not explicitly in the documents but are related to them yet (*i.e.*, the "latent semantic").

For instance, let's assume that a user query is simply "elevator". This user would also like documents containing the word "lift" to be returned, although documents rarely use both terms. LSA approach utilizes the fact that whilst, for instance, "elevator" and "lift" might not appear together, they will each appear with a similar set of words (for example, "building" and "floor"). Ideally, the method will automatically find out a superfeature that captures the concept of "liftiness" of a document.

3.3 Usage of the SVD for Latent Semantic Analysis

3.3.1 Vector Space Model

LSA is based on what is usually referred to as the *Vector Space Model* [17], where documents as well as queries are represented as feature vectors. Each feature takes a dimension in the vector space in which the data positions itself. The theory is that the relationships between the positions of the data points in the space tell us something about their similarity. A dataset takes the form of a set of vectors, which can be represented as a matrix.

In our case, each dimension corresponds to a term that occurs in the document collection. The matrix that represents the dataset is the $m \times n$ *document-term matrix* X , that we have introduced in section 1.3.1 page 11. Note that in this representation, the position and the order of the words in

a document are not taken into account. A document is simply represented by the words it contains. This is usually called the *Bag-of-Words* model.

As similarity can be measured using the positions of the points in the Vector Space, we could use the Euclidean distance between 2 points to determine if the corresponding documents are similar¹. This finally means that we just compare the rows of the document-term matrix that correspond to the 2 documents that we want to compare². However, we have already explained that some terms may appear in a document and not in another one, even if both documents are dealing with the same concepts. On the contrary, some terms can be in two different documents but with a different meaning. LSA uses the Singular Value Decomposition to transform the document-term matrix so that distances in the Vector Space make more sense.

3.3.2 Interpretation of the SVD model

LSA replaces the full document-term matrix X with a low-rank approximation X_k . The downsizing is of course achieved using truncated SVD. We only keep the k largest singular values and corresponding singular vectors (k is typically empirically selected in the range of 100 to 300).

$$\begin{array}{cccc}
 X & & U_k & & \Sigma_k & & V_k^T \\
 \left(\begin{array}{ccc} x_{1,1} & \cdots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{m,1} & \cdots & x_{m,n} \end{array} \right) & = & \left(\begin{array}{c} [u_{1,1} \cdots u_{1,k}] \\ \vdots \\ [u_{m,1} \cdots u_{m,k}] \end{array} \right) & \left(\begin{array}{ccc} \sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_k \end{array} \right) & \left(\left(\begin{array}{c} v_{1,1} \\ \vdots \\ v_{k,1} \end{array} \right) \cdots \left(\begin{array}{c} v_{1,n} \\ \vdots \\ u_{k,n} \end{array} \right) \right)
 \end{array}$$

SVD can be viewed as a technique for deriving a set of k uncorrelated factors. These factors may be thought of as artificial semantic concepts. The hope is that they represent extracted common meaning components of many different words and documents. Each document (resp. term) is then represented by its vector of factor values. These values can be thought of as weights indicating the strengths of association between the document (resp. term) and the underlying concepts. That is, the meaning of a particular term or document can be expressed by k factor values, or equivalently, by the location of a vector in the corresponding k -dimensional space, which is usually referred to as the *LSA-space* or the *concept space* (while the original n -dimensional space is referred to as the *term-space* because each axis corresponds to one term). The vectors that represent documents (resp. terms)

¹We will see later that other distance measures can be used, like the dot-product between the 2 vectors, or the cosine.

²We could compare columns of the matrix to estimate the similarity between words.

are of course the rows of the truncated singular vector matrix U_k (resp. V_k) (weighted by the singular values).

3.3.3 Interests of this low-rank approximation

Let's summarize the different reasons that explain why the SVD is a tool adapted to the problem that we have described in this chapter.

Data compression

SVD allows to store an approximation of the huge original document-term matrix as the product of 3 smaller matrices. In particular, it removes redundancy because data are represented on orthogonal axis. Then, in case we want to use any Machine Learning algorithm on the data (for instance for classification), it will be much easier to use them (and they will certainly work better) if they are applied to data that are represented by vectors of about 100 components than to data with hundreds of thousands components.

Noise reduction

The initial document-term matrix is presumed noisy. The SVD can reduce the weight of occurrences of accidental or anecdotal terms in some documents.

Smoothing

As we already discussed, the original document-term matrix is overly sparse. The dot-product between 2 rows is often meaningless when the vocabularies used in the documents are dissimilar. When multiplying all three matrices U_k , Σ_k and V_k^T together to reconstruct an approximation of the original matrix, the zeros are replaced by values that can be thought of as estimation of the semantic relation between the considered document and each word. This allows to handle the synonymy problem.

A single vector space

LSA maps both documents and words in the same k -dimensional space, so it is much easier to compare them. A new query can for example be represented in the LSA-space by the centroid of the words it contains, or can even be considered as a small document. In both cases, it becomes straightforward to look for the documents that are the nearest neighbors of the query.

3.3.4 Visualization purpose

Even if visualization was not the main concern of my supervisors, it is often an important interest of this kind of dimension reduction methods. I have tried to compute the first five singular triplets of a corpus of 10,000 documents which are labelled "pornographic" or "not pornographic". The projection of the documents on the resulting 5-dimensional concept-space is presented on Figure 3.1.

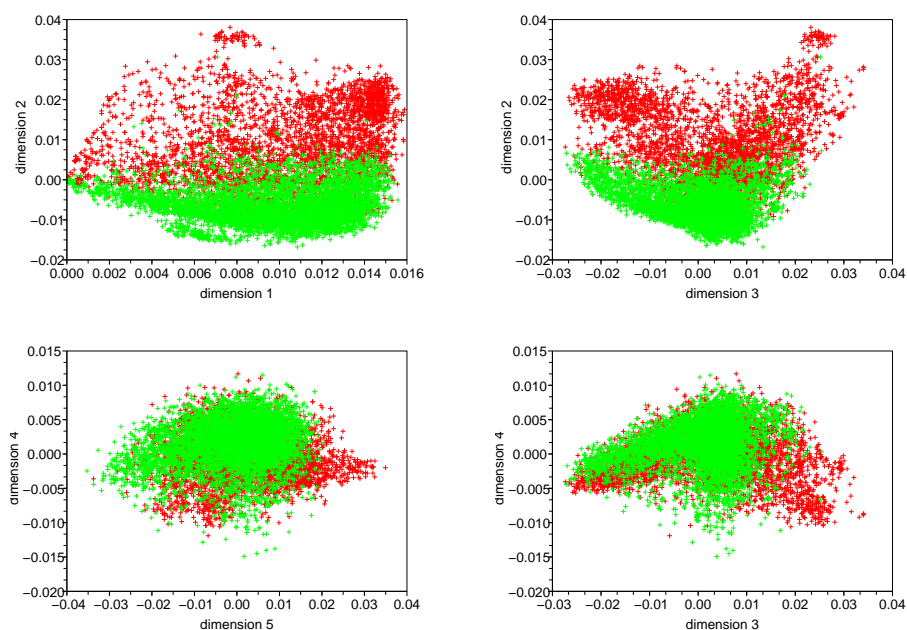


Figure 3.1: *Projection of a corpus of 10,000 labelled documents on the resulting 5-dimensional LSA-space.*

Red points represent documents with some pornographic contents. As we can notice on the two figures on the top, the second axis seems to distinguish porn and non-porn documents. This could be thought of as the concept embedded by the second dimension. Of course, the concepts are corpus-dependent. For this experiment, "pornography" is one of the most important concepts which emerge during LSA because the considered corpus contains a significant part of porn documents.

3.4 Preprocessing of the document-term matrix

In this part, we will focus on the original document-term matrix X . We have explained in section 1.3.1 page 11 that the elements of X are the number of occurrences of each word in a particular document, i.e., $X_{i,j}$ denotes the frequency in which term j occurs in document i .

Instead of directly performing SVD on this matrix, different techniques are available to preprocess it in such a way as to further increase the effectiveness of the method. Indeed, words contribute to varying extents to the semantic profile of a document. Very common words are usually referred to as *Stop-words* and some systems even filter them out prior to other operations. For instance, the word "the" has little impact on the meaning of passages in which it appears. More generally, a word which distributes itself evenly among the documents in a collection is of little value in distinguishing between them. LSA can therefore be made more effective by reducing the impact of such words and increasing the impact of less evenly distributed terms³.

3.4.1 Tf-idf scheme

Tf-idf [18] is certainly the most popular weighting scheme used in Information Retrieval to evaluate how important a word is to a document. Tf-idf stands for "Term frequency - inverse document frequency". It modifies each value of the document-term matrix. An $X_{i,j}$ entry with high Tf-idf score implies a strong relationship between the word j and the document i .

The Tf-idf weight is the product of two terms

$$\text{tf-idf}(i, j) = \text{tf}(i, j) * \text{idf}(j)$$

Term frequency: $\text{tf}(i, j)$ is simply the number of times term j appears in document i . This value is actually often normalized to prevent a bias towards longer documents:

$$\text{tf}(i, j) = \frac{n(i, j)}{\sum_k n(k, j)}$$

where $n(i, j)$ is the number of occurrences of word j in document i . The intuition is that the more frequently a given word occurs in a document, the more important it is for this document.

Inverse document frequency: This is a measure of the general importance of term:

$$\text{idf}(j) = \frac{|D|}{|\{d : j \in d\}|}$$

³We also get rid of the most infrequent terms that are usually typographic mistakes.

where $|D|$ is the size of the corpus and $|\{d : j \in d\}|$ equals the number of documents in which term j appears. Hence, the more documents a given word occurs in, the less discriminating this word is and the smaller its idf score is. Hence, this weight tends to filter common terms.

3.4.2 Other normalization methods

There is a multitude a such preprocessing techniques (See for example [4]). I have tried and compared several of them but let's just give the usual structure of most normalization schemes. The term weight is usually given by

$$L_{i,j}G_jN_i$$

where

- $L_{i,j}$ is the local weight for term j in document j
- G_j is the global weight for term j
- N_i is the normalization for document i to compensate for discrepancies in the lengths of the documents

For example, choosing $L_{i,j} = 1 + \log n(i, j)$, $G_j = 1$ and $N_i = \frac{1}{\sqrt{\sum_k L_{i,k}}}$ (cosine normalization) turned out to be an effective scheme for our purposes.

3.5 Relatedness measure

We have seen in section 3.3.1 page 30 about the Vector Space Model that similarity between data can be measured by using the locations of the points in the Vector Space. For instance, the dot-product of two rows (resp. columns) of the document-term matrix X is a common measure to determine if the two corresponding documents (resp. terms) are similar. There are basically three sorts of comparisons of interest: those comparing two terms, those comparing two documents, and those comparing a term and a document. Let's see how to perform these comparisons once we have performed SVD on the original document-term matrix (possibly after a preprocessing step).

3.5.1 Comparing two terms or two documents

The dot product between two column vectors of X reflects the extent to which two terms have a similar pattern of occurrences across the set of documents. The matrix $X^T X$ contains all these dot-products. After the SVD, X has been approximated by $X_k = U_k \Sigma_k V_k^T$ and

$$X_k^T X_k = V_k \Sigma_k U_k^T U_k \Sigma_k V_k^T = V_k \Sigma_k^2 V_k^T$$

Thus, the i, j cell of $X_k^T X_k$ can be obtained by taking the dot product between the i th and j th rows of the matrix $V_k \Sigma_k$. That is, if one considers the rows of $V_k \Sigma_k$ as coordinates for terms, dot products between these points give the comparison between terms.

The analysis for comparing two documents is similar, except that in this case we need to consider the dot product between two rows of the document-term matrix. We naturally need to consider the rows of $U_k \Sigma_k$ as coordinates for documents.

3.5.2 Comparing a term and a document

Originally, the fundamental comparison between a term and a document is the value of an individual cell of X . After the SVD, $X \approx X_k = U_k \Sigma_k V_k^T$, so the i, j cell of X_k is obtained by taking the dot product between the i th row of the matrix $U_k \Sigma_k^{1/2}$ and the j th row of the matrix $V_k \Sigma_k^{1/2}$. Thus, saying that the same space can be used to represent documents and terms was not exactly true. Yes, the vectors that represent documents and terms have the same number of components, but it is not possible to make a single configuration of points in a space that will allow all kind of comparisons. The spaces we need to use differ only by a stretching of the axes by a factor of $\Sigma_k^{1/2}$ yet.

3.5.3 Considering new queries

It is important to be able to compute appropriate comparison quantities for objects that did not appear in the original analysis. Indeed, we need to be able to compare a completely novel query, with the documents and terms of the corpus. A new query is also considered as a bag-of-words and can be thought of as a *mini-document* or *pseudo-document*. Let Q^{new} denote its vector in the original n -dimensional term-space. We need to derive its representation U_k^{new} that we will be able to use just like other rows of U_k in the previous comparison formulas. If the query is a real document (*i.e.*, a row of X), we should recover the corresponding row of U . Hence, $Q^{new} = U_k^{new} \Sigma_k V_k^T$, and we get directly $U_k^{new} = Q_k^{new} V_k \Sigma_k^{-1}$.

Thus, to compute the similarity of a new query Q^{new} and a term i , we can now refer to the previous section and directly compute the dot-product between $U_k^{new} \Sigma_k^{1/2}$ and a row of $V_k \Sigma_k^{1/2}$:

$$\begin{aligned} \langle U_k^{new} \Sigma_k^{1/2}, V_k^i \Sigma_k^{1/2} \rangle &= U_k^{new} \Sigma_k V_k^{iT} \\ &= Q_k^{new} V_k \Sigma_k^{-1} \Sigma_k V_k^{iT} \\ &= Q_k^{new} V_k V_k^{iT} \end{aligned}$$

3.5.4 Other similarity measures

So far, we have always compared two data-points by computing the dot-product between their corresponding vectors. Instead of the dot-product, the cosine of the angle between the vectors is also frequently used. The only difference is of course the normalization by the norms of the vectors. As we will see later, this has a major influence on the results because this allows not to give a higher weight to long documents and frequent words. We have also tried to simply use the Euclidean distance as a measure of relatedness. Furthermore, note that, depending on the papers, authors sometimes do not scale each component by the corresponding singular value.

3.6 Evaluation – Word List Expander

LSA can be used for several applications. We already spoke about *Document Retrieval* because it was the original goal of the method. In this section, we evaluate its performances for an application that we can refer to as a *Word List Expander*. Basically, the purpose is to create a list of words related to a specific topic. The user provides a small list of seed words as input and the program generates a new list of words that are related with this input list. Hence, this is a straightforward application of Latent Semantic Analysis: the list of input words is considered as a new query and we look for the terms that are its nearest neighbors in the *concept-space*, by using the relatedness measures defined in the previous section.

This application is closely related to *Query Expansion*, whose goal is to add a few words to the initial query to improve the quality of search results. However, our approach is more general, in the sense that we are looking for a long list of words (several hundreds) related to a given topic, but which are not necessarily synonyms of the seen words. They simply often co-appear with them.

Note that this tool is used in production code and is already helpful for people in the department I was working for.

3.6.1 Precision-Recall trade-off

To evaluate the quality of the words that belong to the automatically generated list, we consider each of them as being the unique feature of a binary classifier. For instance, suppose we run the program in order to generate a list of words related to sport. The quality of the generated list will be measured as the average score of each term of the list. To attribute a score to a given term (*e.g.*, "rugby"), we consider a binary classifier that retrieves every documents that contains the word "rugby". Once each term is associated to its corresponding classifier, we just need to measure the performance of these classifiers by using standard measures, like *Precision* and *Recall*.

The following table (also called *Confusion Matrix*) represents the relationship between actual classification and predicted classification:

	relevant	not relevant
retrieved	a (true positive)	b (false positive)
not retrieved	c (false negative)	d (true negative)

The two main measures of performance which reflect the effectiveness of a classifier are:

Precision: It is the proportion of retrieved documents that are relevant to the search:

$$\text{precision} = \frac{a}{a + b} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

Recall: It is the proportion of relevant documents that are retrieved (out of all relevant documents available):

$$\text{recall} = \frac{a}{a + c} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

It is trivial to achieve precision of 100% (by never returning any document) or recall of 100% (by returning all documents in response to any query). Therefore, one of these two measures alone is not enough. A trade-off is necessary. Figure 3.2 represents the typical relation between precision and recall.



Figure 3.2: *Typical trade-off between precision and recall.*

The so-called *F-measure* is commonly used to combine and balance precision and recall. It is defined as the weighted harmonic mean of precision and recall:

$$F_{\alpha} = \frac{(1 + \alpha)(\text{precision} * \text{recall})}{\alpha * \text{precision} + \text{recall}}$$

3.6.2 Experimental results

We have experimented our Word List Expander application by applying Latent Semantic Analysis on a significant part of the French Web. The

Tf-idf scheme (plus cosine normalization) is used to preprocess the original document-term matrix. The quality of the resulting lists is measured thanks to a set of labelled documents. For a given list of seed words, we generate a new list of 200 terms. Figure 3.3 shows the Precision and Recall scores with respect to the number of dimensions of the LSA space.

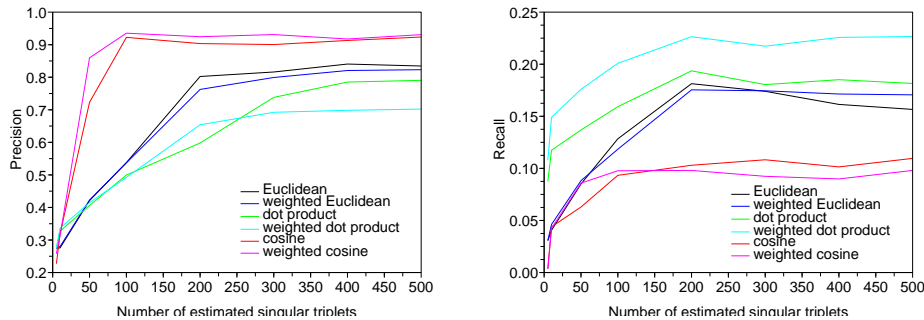


Figure 3.3: *Quality of the generated list with respect to the number of dimensions of the concept-space and to the relatedness measure.*

For this experiment, we tried the three main similarity measures that we discussed in section 3.5 page 35, *i.e.*, Euclidean distance, dot-product and cosine. For each of them, we also tried not to weight by the singular values.

We can first see that the influence of weighting by the singular values is not clear in this experiment. It depends on the used relatedness measure and on the number of dimensions of the concept space.

We can also note that a 200-dimensional space is usually enough to achieve good results. Adding too many components may lead to over-fitting.

Finally, we see that the results vary significantly depending on the relatedness distance used. If the user is mainly interested in precision, he or she should use cosine distance, while dot-product is more adapted if recall matters. This makes sense because common words have a high norm in the LSA space, so the dot-product between these words and any other term tend to be higher. Thus, these words (which have obviously a high recall score because they appear often in documents) have a higher probability to be contained in a generated list. On the other hand, term vectors are normalized with the cosine distance, in order to prevent such bias towards frequent words.

Tables 3.1 shows some examples of lists generated from two different sets of seed words. We can again notice that the list obtained with dot-product distance include more generic words, while documents that contains the terms generated with cosine measure are very likely to deal with the topic associated to the seed words. The choice of the adapted similarity measure clearly depends on the application.

seed words: bmw, renault, peugeot, audi		seed words: zidane, barthez, ronaldo, maradona	
cosine	dot-product	cosine	dot-product
volkswagen	auto	okocha	football
lexus	annonces	juninho	radio
audi	occasion	boumsong	coupe
subaru	km	ibrahimovic	ligue
koenigsegg	moto	barthez	championnat
kia	voiture	inzaghi	france
saab	renault	luyindula	sport
chrysler	automobile	bommel	club
renault	peugeot	robinho	om
wiesmann	petites	savidan	uefa
maserati	voitures	ronaldo	inter
venturi	annonce	dindane	saison
royce	bmw	rivaldo	champions
peugeot	véhicule	matuidi	euro
dacia	sport	kovacevic	real
bentley	véhicules	cafu	fc
fiat	honda	pauleta	match
hummer	tuning	sagnol	vidéo
lamborghini	vendre	aruna	psg
daihatsu	ford	lafourcade	attaquant
nissan	audi	beye	foot
mazda	volkswagen	elmander	classement
mercedes	assurance	villareal	résultats
rolls	automobiles	maradona	italie
rover	mercedes	maldini	angleterre
bmw	occasions	coupet	sports
picanto	diesel	messi	madrid
chevrolet	motos	pancrate	milan
toyota	accessoires	uefa	finale
bugatti	toyota	puel	juventus

Table 3.1: Lists of 30 terms generated from two different sets of seed words.

Chapter 4

Collaborative Filtering

4.1 Presentation

The other main application of SVD I worked on during the internship is called *Recommender Systems*. These are techniques to suggest/recommend products (like movies, music, books) to users, mostly based on ratings that users have previously given to some products. Recommender systems employ different techniques that can be classified into two main categories:

Content-based approach: the user is recommended products that have commonalities with products he or she has rated highly in the past.

Collaborative Filtering approach: the user is recommended products that other users with similar preferences also like.

We focus here on *Collaborative Filtering* [10], although the technique presented in section 4.3.1, page 50, can also be thought of as a content-based approach. More precisely, rather than only suggesting products to users, our task is to predict how a user will rate unseen items, given a history of the user's ratings of other items, as well as other users' ratings of items.

4.1.1 The Netflix database

Recommender systems have achieved widespread success in E-commerce, especially with the advent of the Internet. Vendors have sought to mine customers' purchase and rating information in order to provide product recommendations catering to users' individual tastes. For example, *Amazon.com* uses recommender systems for book purchases, and *Netflix* for movie rentals. Netflix is an on-line movie subscription rental service, which encourages customers to rate the movies that they watch. The company uses a recommender system to analyze the accumulated movie ratings and to make personalized predictions to subscribers based on their particular tastes. In October 2006, in the interest of improving their current algorithm, Netflix

released an anonymized subset of their rating database and offered a \$1 million prize to anyone who is able to make a 10% improvement over their current prediction algorithm [2]. This dataset is comprised of the ratings of over 480,000 individual users for a collection of 17,770 movies, a total of about 100 million ratings. Each rating is an integral value between 1 and 5. We have decided to work with this database, because it is one of the largest available and many recent papers compare their results based on it.

The dataset is distributed along with a *probe set* of user-movie pairs upon which algorithms can be tested. The true ratings for these pairs are known, so an algorithm's output for the set can be compared to the actual ratings in order to measure the error rate. The performance of an algorithm can be measured using *root mean squared error* (or RMSE). For a vector of actual ratings of movies θ , and a vector of our predictions for these ratings $\hat{\theta}$,

$$\text{RMSE} = \sqrt{E((\hat{\theta} - \theta)^2)}$$

Netflix reported the RMSE of their current algorithm against the testing dataset as 0.9514, which represents roughly a 10% improvement over simply predicting individual movie averages.

4.1.2 Common approaches

Many collaborative filtering algorithms have been described in the literature. Most of them are based on two basic approaches.

Nearest-Neighbors approach

Most collaborative filtering based recommender systems build a neighborhood of likeminded customers. They find similar users by finding correlation between their ratings. For instance, the distance between two users can be the average squared difference of their ratings for movies they both have evaluated. The basic assumption is that if two people give similar ratings to most of the movies they rated, then in the future their ratings will continue to be similar. One can then predict ratings for a user u by using ratings of other users who are sufficiently similar. Ratings given by its neighbors for movies that u has not rated yet can be taken to be predictions for those movies.

SVD

The assumption is that each movie can be described by k parameters saying how much that movie exemplifies k different aspects like "action", "comedy", etc. Correspondingly, each user is described by k values saying how much he or she prefers each aspect. To combine them together, we just multiply

each user preference by the corresponding movie aspect, and then add those k values to obtain a final opinion of how much that user likes that movie. Thus, this is a simple linear model. We have seen in the first section of this report that the Singular Value Decomposition is exactly the mathematic tool for finding those k aspects and the corresponding values for each user and each movie, which minimize the resulting approximation error, specifically the Frobenius norm of the error.

The dataset that consists of triplets (user, movie, rating) can be represented as an $m \times n$ user-movie matrix X , where m is the number of users and n is the number of considered movies. $X_{i,j}$ is the rating of user i for movie j . We can then perform the truncated SVD:

$$X = \sum_{t=1}^k \sigma_t \mathbf{u}_{\cdot,t} \mathbf{v}_{\cdot,t}^T$$

$\mathbf{u}_{\cdot,t}$ and $\mathbf{v}_{\cdot,t}$ are the left and right singular vectors, respectively. As for Latent Semantic Analysis, we can think of each dimension as being a concept or aspect of movies. For example let's assume that the p th dimension corresponds to action movie. The j th value of the right singular vector $\mathbf{v}_{\cdot,p}$ measures "how much Action does movie j have?", while the i th value of the corresponding left singular vector $\mathbf{u}_{\cdot,p}$ measure "how much does user i like Action movies?". We just multiply those together to get our estimate for how much user i would like movie j based only on the amount of Action. The model further says that the contributions from the different attributes are just added up linearly (weighted by the singular values) to make a final prediction.

The formulation of the problem is very similar to Latent Semantic Analysis. Basically, we just need to compute the truncated SVD of a large sparse matrix. Ideally, the obtained low rank approximation would *complete* the matrix and fill in the unknown values. Predictions are efficiently computed by taking the appropriate linear combination of factors.

I tried the tools that I have implemented for LSA but the results were disappointing, since the RMSE computed on the test data was 1.18, which is even worse than simply predicting movie averages. The main problem of the usual SVD for this application concerns the missing values.

4.2 Missing values

The main reason that explains the poor results of the usual SVD on the user-movie matrix is related to the missing values. As for LSA applications, the data matrix is very sparse. As a matter of fact, each user has rated about 200 movies on average, while the database contains a collection of 17,770

movies. Thus, the matrix only has ratings for approximately $\frac{200}{17,700} \approx 1\%$ of the elements. The rest of the elements are unknown.

These entries are not equivalent to the zero values of a document-term matrix. For a document-term matrix, a zero entry means that a word does not appear in a document. It gives us some information and we should take it into account. By applying a sparse SVD algorithm to a user-movie matrix, we assume that the missing values are actually zeros. This is clearly an inadequate assumption because it would mean that the user did not like the concerned movie. Hence, as the SVD minimizes the Frobenius distance, it tends to estimate the missing values by ratings close to zeros.

The following sections describe different approaches that we tried to alleviate this limitation.

4.2.1 Shifted columns

One straightforward way to deal with the missing values is to replace them by the average ratings for a customer or the average ratings for a movie. This approach has for example been used by Sarwar *et al.* in [20]. However, this method is not useable in our case, because our data matrix is huge and we have to keep it sparse.

Kleeman *et al.* have proposed to use a zero-mean method [15]. Each existing user's rating is shifted relative to each movie's average rating. Thus, missing values are still at zero but they now represent the average rating of the associated movie, instead of being a bad rating. This effectively initializes all unrated entries to the movie's mean before continuing with the SVD, which can be performed easily because the data matrix is still sparse. Then, we just need to shift back each column to get ratings' estimation.

We have tried this method but the RMSE obtained on the test data (0.97) was still not as good as the result currently achieved by the Netflix algorithm. If we look at the predicted ratings with this method, we can see that they are often very close to the movie's means. The main reason is that the matrix is so sparse (approximately 99% of the entries are zero), that for each column, almost every ratings correspond to the corresponding movie average and the actual known ratings may be considered as noise by the SVD algorithm.

4.2.2 Gradient descent

Principle

After having tried different techniques to handle missing values, we finally re-considered the problem from the beginning. We have seen in section 1.2.1 page 10, that SVD is especially used to find the best rank- k approximation matrix X_k to the original $m \times n$ matrix X , with respect to the Frobenius

norm, *i.e.*

$$X_k = \operatorname{argmin}_{\operatorname{rk}(Y)=k} \sum_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}} (X_{i,j} - Y_{i,j})^2$$

In the case of Collaborative Filtering, we would like to only take into account the known entries and ignore the unknown error on the 99% empty slots. Ideally, the problem we would like to solve is rather

$$\widehat{X}_k = \operatorname{argmin}_{\operatorname{rk}(Y)=k} \sum_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n \\ X_{i,j} \neq 0}} (X_{i,j} - Y_{i,j})^2$$

This problem cannot be solved by the usual SVD procedure. We simply propose to use a gradient descent algorithm to minimize the error on the known entries. Let say we are looking for a rank-1 matrix $\widehat{X}_0 = u^0 v^{0T}$. In the original SVD framework, u^0 and v^0 can be thought of as the first left and right singular vectors, respectively. The only difference is that we do not normalize these vectors, so we do not have any singular value. We will refer to them as *pseudo-singular vectors* in the following.

The error we want to minimize is

$$e^0 = \sum_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n \\ X_{i,j} \neq 0}} (X_{i,j} - u_i^0 v_j^0)^2$$

To minimize this error function, one takes steps proportional to the negative of the gradient. The derivatives of e^0 are

$$\frac{\partial e^0}{\partial u_i^0} = -2 \sum_{\substack{1 \leq j \leq n \\ X_{i,j} \neq 0}} (X_{i,j} - u_i^0 v_j^0) v_j^0 \quad \forall 1 \leq i \leq m$$

$$\frac{\partial e^0}{\partial v_j^0} = -2 \sum_{\substack{1 \leq i \leq m \\ X_{i,j} \neq 0}} (X_{i,j} - u_i^0 v_j^0) u_i^0 \quad \forall 1 \leq j \leq n$$

Thus, the components of the pseudo-singular values are updated using the formulas:

$$u_i^0 \leftarrow u_i^0 + \lambda \sum_{\substack{1 \leq j \leq n \\ X_{i,j} \neq 0}} (X_{i,j} - u_i^0 v_j^0) v_j^0 \quad \forall 1 \leq i \leq m$$

$$v_j^0 \leftarrow v_j^0 + \lambda \sum_{\substack{1 \leq i \leq m \\ X_{i,j} \neq 0}} (X_{i,j} - u_i^0 v_j^0) u_i^0 \quad \forall 1 \leq j \leq n$$

where λ is the learning rate.

We repeat this process for each new pair of pseudo-singular vectors. Once we have computed u^0 and v^0 , we can repeat the process with the leftovers $X - u^0 v^0 T$ to get a second pair of vectors, and so on, such that the target matrix is approximated by $X \approx u^0 v^0 T + u^1 v^1 T + \dots u^k v^k T$. This is clearly very similar to the usual SVD, the only difference being that we only take the known entries into account. If the original matrix X was full, it's worth noting that we would have computed exactly the SVD¹.

Hence, once $p - 1$ pairs of pseudo-singular vector have been computed, the error measure used to compute the p th one is:

$$e^p = \sum_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n \\ X_{i,j} \neq 0}} \left(X_{i,j} - \sum_{k=1}^{p-1} \tilde{u}_i^k \tilde{v}_j^k - u_i^p v_j^p \right)^2$$

where \tilde{u}^k denotes the estimation of u^k that has been computed in a previous step. The gradient and the updating steps are then straightforwardly derived from this formula.

Gradient Descent

Standard Batch Gradient Descent Given these formulas, we now just have to run a standard gradient algorithm. To perform one iteration of the gradient descent algorithm, one needs to compute all the derivatives of the error function, which form the full gradient. Then, we can update the pseudo-singular vectors. Therefore, standard gradient descent requires one sweep through the whole training set (*i.e.*, about 100 millions ratings), before the components of the pseudo-singular vectors can be changed.

The properties of this optimization algorithm are well-known: When the learning rate λ is small enough, the algorithm converges towards the global minimum of the error function (because this function is convex). In practice, we did not manage to get any interesting result with this method, because the convergence speed is much too slow. We tried to vary the training rate λ and the way we initialize a new pair of pseudo-singular vectors but it was not sufficient.

Stochastic Gradient Descent The *Stochastic Gradient Descent* is usually a good alternative to the standard gradient descent for Machine Learning applications when data sets are particularly large (See for example [3]). The idea is to update the model parameters after each training example. Thus, we still loop over the 100 millions ratings of the dataset, but instead of computing the gradient by taking all of them into account, we update some parameters after each training example. More precisely, when we consider

¹The singular values could be trivially extracted from the vectors.

the rating given by user i to movie j , we directly update the i th component of the left pseudo-singular vector and the j th component of the right one, by using the simplified formulas:

$$\begin{aligned} u_i^{0'} &\leftarrow u_i^0 + \lambda (X_{i,j} - u_i^0 v_j^0) v_j^0 \\ v_j^0 &\leftarrow v_j^0 + \lambda (X_{i,j} - u_i^0 v_j^0) u_i^0 \\ u_i^0 &\leftarrow u_i^{0'} \end{aligned}$$

The online gradient descent simplification relies on the hope that the random noise introduced by this procedure won't perturbate the average behavior of the algorithm.

Parallel implementation We have tried to parallelize the computation of the stochastic gradient descent procedure. As for the standard SVD algorithm (see section 2.5.2, page 26), we distribute the original data matrix over N machines, each of them being responsible of about $\frac{480,000}{N}$ rows. For example, for $N = 4$:

$$\left(\begin{array}{c} 1 \\ \hline 2 \\ \hline 3 \\ \hline 4 \end{array} \right)$$

Thus, each machine only considers the ratings that have been given by users that correspond to its associated rows. Let's assume that the first machine processes a rating $r_{i,k}$ given by user i to movie k . This machine will modify the i th component of the left pseudo-singular vector and the j th component of the right one. However, each machine stores its own version of the current right pseudo-singular vector. If later a different machine has to process a rating $r_{j,k}$ given by user j to the same movie k , this machine should know how the k th component of the right pseudo-singular vector has been updated by the first machine previously. This implies each machine having to communicate to all the other ones to tell them how they did modify the right pseudo-singular vector at each step. Furthermore, there could be a conflict if 2 machines have to process a rating of the same movie at the same time.

To handle this difficulty, we also divide the columns into N groups. During a first stage, each machine only considers the ratings that concern a

certain part of the movies, according to this scheme:

$$\begin{pmatrix} 1 & & & \\ \hline & 2 & & \\ \hline & & 3 & \\ \hline & & & 4 \end{pmatrix}$$

Thus, we are sure that a conflict can not happen, and the machines do not have to communicate. Once all the movies that correspond to this splitting have been handled, the machines have to communicate to share their results. Then, the considered movies by each machine shift according to the following figure:

$$\begin{pmatrix} & 1 & & \\ \hline & & 2 & \\ \hline & & & 3 \\ \hline 4 & & & \end{pmatrix}$$

This process continues until each machine has considered all the movies, *i.e.*, after N stages. Hence, the machines have to communicate only N times, instead of after each considered rating.

Despite this technique to parallelize the computation, the communication time between the machines remains too long for this rating dataset. It is faster to run the stochastic gradient descent on a single machine. Nevertheless, this parallelization strategy can be interesting for bigger datasets.

Experimental results

On a single machine, one training pass through the entire data set of about 100 millions ratings takes about 2 seconds. We have tried different stopping criteria for the gradient descent, but simply doing a fixed number of iterations over the whole data set was satisfying. We also empirically determined that computing around 150 pairs of pseudo-singular vectors gives the best results on the testing data set. Eventually, it takes about 5 hours to train the system (*i.e.*, to estimate 150 pseudo-singular vector pairs).

With this method, we achieved an RMSE of about 0.91, which is much better than the performance of a standard SVD algorithm (1.18). It is even better than the score obtained by the algorithm currently used by Netflix (0.95).

4.3 Other approaches

4.3.1 Using a movie relatedness matrix

Integrating data attributes is another important issue of Recommender System algorithms. The goal is to use attributes that characterize the users or the items in order to improve further the estimation results. So far, we always assumed that nothing is known about the users and the movies, apart from the ratings expressed so far. It could be useful to take advantage of other information, like the gender or the age of the users. Such a framework has been studied in different papers, like [1].

For the application of the user-movie database, Netflix also provide movie titles. We use the *Internet Movie Database* (IMDb) — which is an online database of information about movies and actors — in order to retrieve the genres used by the website to categorize movies. Indeed, movies can easily be described with certain umbrella terms, such as Western, dramas or comedies. We form an $n \times p$ binary matrix, where $n = 17,770$ is the number of movies in the database, and $p = 25$ is the number of different genres. As certain genres are closely related (like *Comedy* and *Family*, or *Crime* and *Film-Noir*), we perform SVD on this matrix, and then use the similarity measures that we have introduced in section 3.5 page 35, in order to build an $n \times n$ movie relatedness matrix M , where $M_{i,j}$ is a normalized measure of the similarity between movies i and j .

This similarity matrix is a prior knowledge that we use as a regularization term. The components of the right pseudo-singular vectors that correspond to movies that are closely related in terms of genres, are encouraged to be close. The benefit of adding such a regularization term is that we may avoid overfitting the training data.

Hence, once $p - 1$ pairs of pseudo-singular vectors have been computed, the error measure used to compute the p th one is now:

$$e^p = \sum_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n \\ X_{i,j} \neq 0}} \left(X_{i,j} - \sum_{k=1}^{p-1} \tilde{u}_i^k \tilde{v}_j^k - u_i^p v_j^p \right)^2 + \alpha \sum_{i,j=1}^n \left(v_i^p - v_j^p \right)^2 M_{i,j}$$

The experimental results of this approach are not very convincing. The RMSE computed on the testing data is slightly better than before but the difference is not significant. Our assumption is that genres are not sufficiently informative attributes. On average, we already know about 5,500 ratings per movie and this information is certainly much more explicative than genres.

4.3.2 Taking user profiles into account

The users have different rating habits. Some users tend to always rate movies with 3s, while other users only give 1s and 5s. Each user can be characterized by its ratings distribution. We have tried to take advantage of this information in different ways:

- Once we have estimated ratings with the pseudo-SVD algorithm described above, we tried to adjust the ratings distribution of each user to the one we have by considering only the known ratings. This can be done easily by using standard techniques that are for example used to fit image histograms to given distributions.
- We also tried to use a prior for the ratings. We can consider the known rating distribution of each user as a multinomial distribution whose parameters are the number of ratings given by the user and the frequency of each rating (proportion of 1s, of 2s, ..., of 5s). While estimating a new rating, we simply use this information as a prior, by using the Dirichlet distribution, which is the conjugate prior of the multinomial distribution in Bayesian statistics.
- Of course, we always clip the prediction to the range 1-5. This can be thought of as applying the function of Figure 4.1 to the estimations computed by the gradient descent procedure.

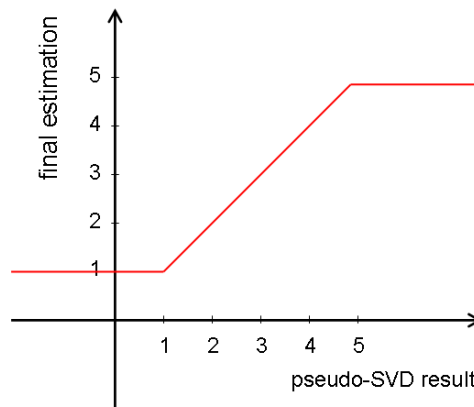


Figure 4.1: *The estimated ratings are clipped to the range 1-5.*

As we already corrupted the matrix analogy with our pseudo-SVD approach, we are not really restricted to linear model anymore. Thus, we tried to use different functions to alter the estimations, as shown in Figure 4.2. We fit a piecewise linear function that aims to take user rating habits into account. Thus, the left graph is adapted to a user

who gives low ratings, while the right function corresponds to a user who always give 1 or 5. Of course, this requires modifying the learning rule slightly to include the slopes of these functions.

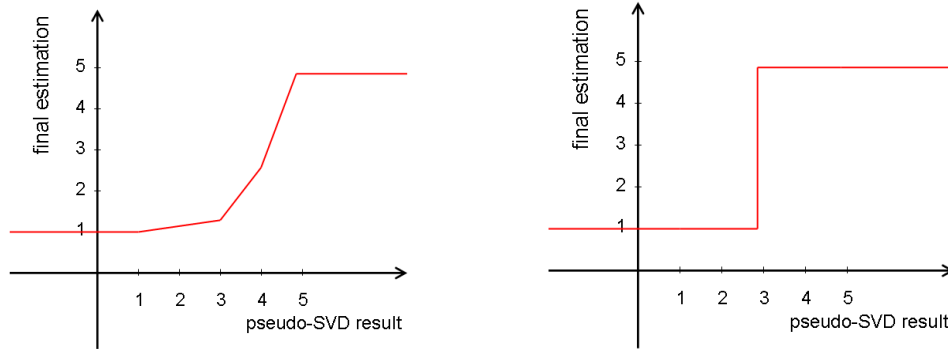


Figure 4.2: *Functions used to take user rating profiles into account during the learning stage.*

Unfortunately, none of these methods turned out to be really effective. They only lead to a slight improve of the RMSE computed on the testing dataset. Several reasons may explain these disappointing results. First, SVD should already take into account the fact that some users tend to give high ratings while other ones are choosier. A row of the left singular vector matrix corresponds to a single user and it may contain higher values if this user is easy to please and lower ones if the user is pickier.

Furthermore, it is not obvious that the distribution of the estimated ratings has to be adjusted to the distribution of the known ratings. Users tend to see (and to rate) movies that conform to their tastes. Thus, we can expect the average rating of unrated movies to be lower than the average of the known ratings. Secondly, even if a user always rate movies with the worst rate (1) or the best rate (5), it does not mean that we should always predict new ratings to be 1 or 5. For example, always predicting 3 for new movies may lead on average to a better RMSE than predicting only 1 or 5 and doing many mistakes.

Conclusion

As explained in the beginning of this report, we could not address every parts of the work I did during the internship in this document, because of confidentiality issues. However, it describes several methods we have tried to address a given mathematical problem (computing the *Singular Value Decomposition* of a large sparse matrix), from purely linear algebra methods, to machine learning oriented techniques like the *Generalized Hebbian Algorithm*.

While this report mainly focuses on the theoretical issues, I spent a significant part of my work time to adapt these methods to the Google environment, to make them work in parallel and with the Google specific data formats. This work was also very interesting and I think that it will help me a lot in the future. I also learned to produce quality code that is shared and can be re-used, maintained and extended easily by other engineers.

Once we were able to compute the SVD and met the requirements defined in the beginning of the internship, we could try different applications that were useful for the team I was working with. Even if I am not allowed to explain how my code is used by Google employees, I think that this report gives a good overview of two important applications of the SVD in text processing and data mining: *Latent Semantic Analysis* and *Collaborative Filtering*.

Furthermore, I think that it was also very useful to discuss with my two supervisors about machine learning techniques and diverse applications of my work. I learned a lot only by exchanging points of view, and even by just listening to all their ideas. Machine Learning at Google is fairly specific, especially because of the huge amount of data that engineers have to deal with, and this internship helped me to have a new perspective.

Bibliography

- [1] Jacob Abernethy, Francis Bach, Theodoros Evgeniou, and Jean-Philippe Vert. Low-rank matrix factorization with attributes. Technical report N-24/06/MM, École des Mines de Paris, 2006.
- [2] James Bennett and Stan Lanning. The netflix prize. *Proceedings of KDD Cup and Workshop*, 2007.
- [3] Léon Bottou. Stochastic learning. In *Advanced Lectures on Machine Learning*, number LNAI 3176 in Lecture Notes in Artificial Intelligence, pages 146–168. Springer Verlag, Berlin, 2004.
- [4] Erica Chisholm and T. G. Kolga. New Term Weighting Formulas for the Vector Space Method in Information Retrieval. Technical report ORNL/TM-13756, Computer Science and Mathematics Division, Oak Ridge National Laboratory, 1999.
- [5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 137–150. Google, Inc., 2004.
- [6] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.
- [7] Amit Deshpande and Santosh Vempala. Adaptive sampling and fast low-rank matrix approximation. In *RANDOM*, pages 292–303, 2006.
- [8] Inderjit S. Dhillon and Beresford N. Parlett. Orthogonal eigenvectors and relative gaps. *SIAM J. Matrix Anal. Appl.*, 25(3):858–899, 2003.
- [9] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. Fast monte carlo algorithms for matrices ii: Computing a low-rank approximation to a matrix. *SIAM J. Comput.*, 36(1):158–183, 2006.

-
- [10] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.
 - [11] G. H. Golub and C. F. van Loan. *Matrix Computations*. Johns Hopkins University Press, 3rd edition, 1996.
 - [12] Genevieve Gorrell. Generalized hebbian algorithm for dimensionality reduction in natural language processing. 2006.
 - [13] Genevieve Gorrell. Generalized hebbian algorithm for incremental singular value decomposition in natural language processing. In *EACL*, 2006.
 - [14] W. Johnson and J. Lindenstrauss. Extensions of lipschitz maps into a hilbert space. *Contemporary Mathematics*, 26:189–206, 1984.
 - [15] Alex Kleeman, Nick Hendersen, and Sylvie Denuit. Matrix factorization for collaborative prediction. *ICME*, 2006.
 - [16] E. Oja and J. Karhunen. On stochastic approximation of the eigenvectors and eigenvalues of the expectation of a random matrix. 106:69–84, 1985.
 - [17] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.
 - [18] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513–523, 1988.
 - [19] Terence D. Sanger. Optimal unsupervised learning in a single-layer linear feedforward neural network. *Neural Networks*, 2(6):459–473, 1989.
 - [20] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Application of dimensionality reduction in recommender systems – a case study. *ACM WebKDD Workshop*, 2000.

Index

- ARPACK, **25**
- Bag-of-Words, **31**
- Batch Gradient Descent, **47**
- Collaborative Filtering, **42**
- Confusion Matrix, **38**
- Deflation Method, **18**
- Dirichlet distribution, **51**
- Document-term matrix, **11, 30**
- Eigenvalue Decomposition, **10**
- F-measure, **39**
- Frobenius norm, **10**
- Generalized Hebbian Algorithm, **21**
- Gram-Schmidt Process, **19**
- Hebb rule, **22**
- Householder reflection, **13**
- Information retrieval, **29**
- Johnson-Lindenstrauss Lemma, **24**
- Lanczos algorithm, **25**
- LAPACK, **13**
- Latent Semantic Analysis, **29**
- MapReduce, **26**
- Matrix bidiagonalization, **14**
- Message Passing Interface, **26**
- Multinomial distribution, **51**
- Netflix, **42**
- Oja Model, **21**
- Power Method, **17**
- Precision, **38**
- Principal Component Analysis, **21**
- QR decomposition, **13**
- Query Expansion, **38**
- Recall, **38**
- Recommender System, **42**
- Singular Value Decomposition, **9**
- Stanger Model, **22**
- Stochastic Gradient Descent, **47**
- Stop-words, **34**
- Tf-idf scheme, **34**
- Truncated SVD, **10**
- Vector Space Model, **30**