Parallel and Distributed Computing: Singular Value Decomposition in the context of massive online Latent Semantic Analysis

Darío Bahena, Ricardo Zavaleta

August 9, 2015

Contents

Chapter 1

Introduction: Getting to know the SVD

1.1 What is SVD?

1.1.1 Matrix decompositions

We can think of Matrix decomposition by creating an analogy to what a product factorization is in the context of polynomials or as an analogy to what the factorization of a real number in terms of other numbers is. The idea is to generate a series of more elementary components that when multiplied produce the original object. The same idea applied to matrices correspond to obtaining a new set of matrices that when multiplied result in our original matrix.

When thinking in terms of a real number decomposition, we observe that this factorization exhibits perhaps some interesting properties about the original number. For example, we could discover whether it is the result of applying power 2, or perhaps we discover that it can not be decomposed in which case the number turns out to be a prime number. In some case, it is even more convenient to treat the original number as series of multiplications of simple ones.

Something similar happens with Matrix Decompositions. We might be able to apply a factorization to simplify some further operation or to reveal some hidden inherent characteristics of it. Some of the most famous matrix decompositions are shown in table 1.1

It is evident that explanation of each of these types of factorizations is far beyond the scope of this work. The user is kindly invited to take a look

Decomposition	**Expression**	**Components**
LU Decomposition	$A = LU$	L is a lower triangular matrix; U is a upper triangular matrix
Rank Factorization	$A = CF$	C is an $m \times r$ full column rank matrix; F is a $r \times n$ full row rank matrix
Cholesky Decomposition	$A = U^T U$	U is upper triangular with positive diagonal entries
QR Decomposition	$A = QR$	Q is an orthogonal matrix mxm; R is an upper triangular matrix $m \times n$
Eigendecomposition	$A = Q\Lambda Q^{-1}$	D is a diagonal matrix formed from eigenvalues of A; V are the corresponding eigenvectors
SVD Decomposition	$A = U\Sigma V^T$	U and V are orthogonal matrices; Σ is a diagonal matrix

Table 1.1: Some famous matrix decompositions

at [Str88] if interested.

1.1.2 SVD

The objective of this work is to study some particular variations of one of the matrix factorizations that appears in the previous subsection. It is the *SVD* (*Singular Value Decomposition*). We beg the reader for patience to discover which specific variation of this decomposition we are trying to study. For now, what we want to do now is to briefly explain, in informal terms, what the SVD is.

As pointed out in table 1.1 the SVD is a factorization that will convert a matrix A into a product of other three matrices U, Σ and V. The matrix A can be a real or a complex matrix. For the purposes of this work, we will only focus on the real case. Subsequent chapters will justify why this case is enough for our purposes.

Matrix U and matrix V will have the property of orthogonality. What this means is that the product $UU^T = I$ and that $VV^T = I$ too. On the other hand, the matrix Σ is a diagonal matrix. The contents of this diagonal are the famous *singular values of* A. In relation to this name, matrix U is called the *left singular vectors of* A and V is the *right singular vectors of* A. The reader might be intrigued by such a definition. We will not go very in deep regarding these terms here. Such a discussion will be left for a later

chapter: here we are only trying to attract the reader to this subject.

The SVD is a very important kind of factorization and the reader should not be surprised when he or she finds the paper by Kalman [Kal96] that talks about the SVD as "A singularly valuable decomposition". Strang [Str88] did something similar when he mentioned that the SVD was not nearly as famous as it should be. Golub [GVL12] also says about his book that the most recurring theme is the practical and theoretical value of the SVD. Another example is [MP12] where the authors talk about "The Extraordinary SVD".

1.2 The importance of SVD

1.2.1 Available material

The reader, apart from being intrigued by the definition of the SVD, might be also wondering whether this particular form of factorization is useful at all. He would be shocked by the quantity of information available in books and on-line regarding this topic. *Google Scholar* shows at least $797K$ matches for the term, including scientific papers, books, lecture notes, etc. The more generic *Google* search matches even more content: $1.34M$ matches! Considering that SVD has existed since the 19th century, it might come as no surprise such a huge amount of published work.

The SVD is a generic mathematical tool that can be applied in many fields of science and engineering. Fortunately, we are restricting ourselves to the field of computer science. In that sense, a search in the $ACMLibrary$, shows $11,501$ hits for the keyword "SVD".

1.2.2 Implementations in software

Several variations of the SVD have been implemented in software to the date. The authors of this work did some research about this and found out the implementations showed in table 1.2. Let us talk a little bit about some of them. Let us start with two of the most famous packages that implement some sort of SVD algorithm: SVDLIBC and SVDPACK.

SVDPACK [BDO$^+$93] was a result of Berry's research [Ber92a]. It consists on a package that implements four iterative methods for computing the SVD of large and sparse matrices. The package includes what is called the Lanczos method [Lan50a] and the Subspace Iteration method. It contains some well-known functions such as *las2()*, which is an instance of Lanczos algorithm. As mentioned in the *readme* file, the package was motivated by Information Retrieval tasks. The specific task will be discussed in a later

chapter. Initially, it was written in Fortran and ported to several types of machines, from supercomputers to workstations.

Another commonly used library is SVDLIBC [Roh07]. This one was written in C and it is based on Berry's SVDPACK. The documentation states that it provides a cleaned-up version of its predecessor. However, it clearly mentions that it only implements the *las2()* routine because it is evidently the fastest. Nevertheless, this routine is useful only for calculating the largest singular values. One of its problems is that lower singular values may be imprecise. Its latest is 1.4 and was uploaded in December 2012.

Other available libraries are SVDLIBJ (which is the Java implementation of SVDLIBC), PROPACK, LAPACK, etc. PROPACK [pro] it is also used for large and sparse matrices and uses the Lanczos algorithm too with the variant of partial re-orthogonalization. There is an implementation in Fortran and another one in Matlab. The most recent version is 2.1 which was uploaded in April, 2005.

LAPACK stands for Linear Algebra Package and is written in Fortran. It provides routines for solving systems of linear equations, least-squares, eigenvalue and singular values problems. In doing so, matrix factorizations are also provided. One of the targets of this package is to run efficiently on shared memory vector and parallel processors. It achieves higher efficiency because it takes into account the memory hierarchy of the machine. Internally, LAPACK depends on BLAS (Basic Linear Algebra Subprograms). The latest version is 3.5.0 and was released in November, 2013.

From table 1.2 it is pretty much clear that these packages mentioned above are the most popular. In table 1.2 we also show some other popular software or libraries that either implement an SVD algorithm or that make use of SVD as an internal tool for an specific purpose.

Implementations that deserve special attention are Gensim, Apache Mahout, SLEPc, LingPipe and GraphLab. We will talk about Gensim in later chapters. So let us dedicate some lines to the other softwares.

Apache Mahout[mah] is a framework that provides Machine Learning tools for building scalable performant applications. The latest stable release (0.11.0) was released recently. Regarding SVD implementations, it provides three: a stream-oriented one that uses what is called the Asymmetric Generalized Hebbian Algorithm [GW05]; another one that uses Lanczos; and there is a work-in-progress implementations of Stochastic SVD [Hal12].

SLEPc [sle] stands for Scalable Library for Eigenvalue Problem Computations and consists in a library for the solution of large scale sparse matrices on parallel computers. In particular, the web page mentions that it can be used for computing a partial SVD of a large, sparse, rectangular matrix.

Internally, it uses a generic lanczos solver [HRTV07a] that is used for their particular implementation for the SVD solver[HRTV07b].

LingPipe [lin] is a toolkit for processing text using computational linguistics. Specifically, it allows to find names and locations in news, classify twitter search results into categories and suggest correct spellings of queries. Regarding SVD implementations it uses a version that uses Stochastic Gradient Descent. The web page points out that it is based on a C code from Timely Development which in turn is based on [GW05], the one used by Mahout. The latest LingPipe is 4.1.0.

Finally, GraphLab [gra] is a machine learning platform that allows to create intelligent apps. Apparently, for SVD it uses a Lanczos solver as well [LBG$^+$12] [LGK$^+$14].

1.2.3 SVD variations

Another topic that deserves some lines is a classification of the available SVD techniques. We need a mental map of how the universe of this decomposition is arranged. This shall be brief since we only want to discern the specific problem that we will be investigating.

First, we need to establish that the SVD is used for decomposing one single matrix. The case where the factorization is done for two or more matrices is called GSVD (Generalized Singular Value Decomposition). Also, we are dealing with 2-dimension matrices. The case for multidimensional matrices (or Tensors) is not studied in this work.

As it has been mentioned previously, we are dealing with real matrices. It was also mentioned before that SVD could be applied for the complex case. We still have not mentioned why we are doing so but it will be clear in the next chapters.

The specific technique for obtaining the SVD depends also on the shape of the matrix. As it could be implied by table 1.2, we can distinguish between the sparse and the dense case. An special case for the sparse matrices are those matrices with a bidiagonal or tridiagonal form. Some of the algorithms that will be mentioned in later chapters perform some transformations in order to arrive to a similar matrix with this special shape and from there, the factorization is obtained in an easier way.

Another distinction is whether the SVD is *thin*, meaning that only some of the column vectors of the matrix U are obtained; *compact*, meaning that only the columns of U and V corresponding to non-zero singular values are calculated; or *truncated*, meaning that only columns of U and V corresponding to the largest singular values are obtained. This latter version is the one

Software/Library	**Algorithm**	**Usage**
SVDPACK	Lanczos and Subspace Iteration Algorithm	Library that implements some popular SVD algorithms for large, sparse matrices.
SVDLIBC SVDLIBJ	Lanczos Algorithm	"Modern" version of SVDPACK in C/Java
PROPACK	Lanczos Algorithm	Software package that contains functions for computing SVD of large, sparse matrices.
LAPACK	Divide and Conquer Algorithm	Library that provides routines for solving systems of linear equations.
Sense Clusters	SVDPACK	Word sense discrimination system
S-Space	SVDLIBC Matlab (LINPACK, predecesor of LAPACK) GNU Octave JAMA COLT	A collection of algorithms for constructing Semantic Spaces and Semantic Algorithms
Semantic Vectors	SVDLIBJ	A package to create semantic WordSpace models from free natural language text.
Infomap	SVDPACK	Build semantic models from free-text corpus and allows IR tasks.
Text To Matrix Generator	PROPACK	It is a Matlab toolbox used for text mining
Gensim	Distributed algorithm	It is a Python library designed to extract semantic topics from documents.
Apache Mahout	SSVD	It is a library of Machine Learning algorithms focused on collaborative filtering, clustering and classification.
SLEPc	Restarted Lanczos bidiagonalization	It is a software library for parallel computation of eigenvalues and eigenvectors of large, sparse matrices.
LingPipe	Stochastic Gradient Descent	It is a toolkit for processing text using computational linguistics.
GraphLab	Generic Lanczos	It is a framework for Machine Learning and data-mining tasks.

Table 1.2: Available software/libraries containing an SVD implementation/application

that we will be working with.

Yet another classification can be whether the algorithm is serial, parallel or distributed. In this work we will emphasize the parallel and distributed versions but we will obviously have to correlate the parallelization steps with the serial version. Finally, the SVD algorithm can work in streaming fashion, depending on whether the whole matrix A is available at all times.

1.2.4 Applications

Now let us guide you through some of the real world applications of the SVD. This will allow us to give yet another dimension of the importance of this extraordinary tool. We shall mentions specifically the case of compression and PCA. Another application is that of LSI but we will postpone such treatment to a later chapter.

In numerical analysis, one important task consists on determining whether a problem is well-conditioned or not. For example, given the problem of solving the linear equation $Ax = b$ we can obtain the so called *condition number* of the matrix A. This condition number is a metric of the sensitivity of the solution to changes or errors in the entries of A. SVD comes into play because this condition number is given by ratio of the largest and the smallest singular values [Str88].

An use of SVD is in what is called Principal Component Analysis (PCA) [Lay12]. If we suppose that a matrix A represents customer $\times$ product information in the form of, for example, the probability of a customer i buying a product j, or the utility of product j to customer i, etc. The idea behind the use of SVD is to find the underlying basic factors in this information: age, income, family size, etc. Hence, a customer's behavior is determined by a weighted combination of those factors. In other words, the behavior should be characterized by a k-dimensional vector. At the same time, there should be another set of vectors that determine the probability of buying the product based on a particular factor. All this means is that the matrix A can be expressed as the product of U, the matrix of factor weights per customer, and V, the matrix of purchase probabilities of products based on the basic factors. Through best rank approximation(which will be explained in a subsequent chapter) that is based on SVD, it is possible to eliminate noise factors present in the original A matrix and allow to obtain a new matrix A_k.

In image compression, the SVD is used in the following way[Str88]: an image is supposed to be represented by an array of intensity of pixels of $n \times n$ A. If we would like to transmit A, it would require to send through

the wires $O(n^2)$ real numbers. Instead of sending that, one could send the matrices U, Σ and V. Again, thanks to best rank approximation, one could instead send three smaller matrices that account for $O(nk)$ real numbers. If $k \ll n$, this results in space saving. Evidently, such compression implies the reduction of resolution as well. An even more advanced technique use a fixed basis represented by some of the singular vectors to represent any picture approximately. The space spanned by these vectors is similar to the space spanned by all the singular vectors. By doing this, it is enough to send the matrix of singular vectors only once and then transmit the required top singular vectors. As visual example, we show in fig. 1.1 a picture that has 16 singular values. A compressed version with only 5 singular values is shown in fig. 1.2. [Kal96] shows that rounding its entries generates fig. 1.3 which is almost perfectly restored.

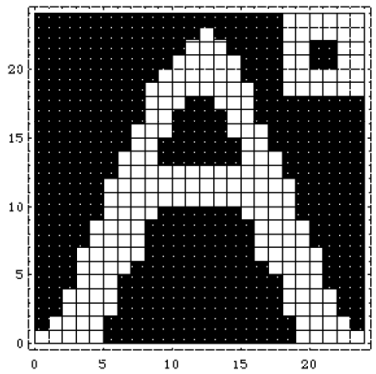

Figure 1.1: Original image with 16 singular values[Kal96]

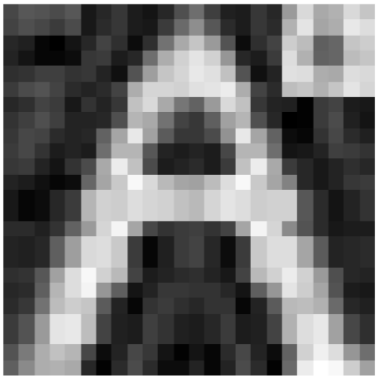

Figure 1.2: Compressed image with only 5 singular values[Kal96]

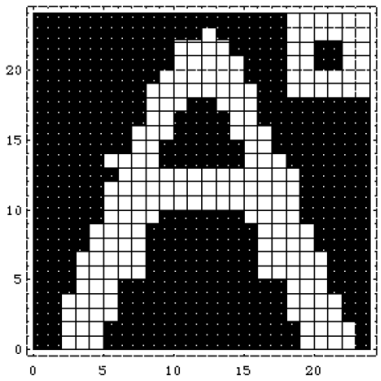

Figure 1.3: Compressed image with rounded entries[Kal96]

A bunch of other possible applications are available in the literature. For a quick review, the reader is encouraged to take a look at[Str88] and [Lay12].

1.3 The idea behind this work and scope of the research

At this point, the reader should be convinced that the SVD is both an important and an interesting mathematical tool used for analysis and mod-

ification of a matrix A. What we want to do now is to justify the current work that we are presenting. We wish to provide a sensible explanation of how we decided to restrict the vast literature to a smaller but still huge set of papers and books which were reviewed in this project.

Since some time, we have been interested in a multi-diciplinary area known as Natural Language Processing. From this common interest, we came up with the idea of presenting the state of the art of some particular task used in NLP. At the same time, and due to an implicit requirement of coming up with a Master thesis that could be useful for the company we both work at (or at least related to any of its products), we thought of finding a product that could obtain some benefit from this work.

In a preliminary study presented by us before, we showed interest on how documents written in free natural language can be processed to obtain useful information from it. One possible task we considered initially was the construction of *thesauri*, which is a tool used in Information Retrieval (IR), for word disambiguation. A particular step within the construction of a thesaurus consist on obtaining some indication of how similar two ore more terms are. As pointed out in that document, recent work has focused on using tensors [1] for that matter [SBST11]. So we wanted to understand how tensors are used. Specifically how tensor decompositions are applied for this task.

Soon after, we figured out that for understanding the work that has been done with tensors and tensor decompositions, we should first understand how this works in matrices. Hence we did some preliminary research and found out that SVD is used in IR, specifically in the form of the so called Latent Semantic Indexing (LSI). So due to time constraints and because it seemed interesting we chose to do our research on this topic. Matrices in this context represent documents and is a mathematical representation of a given corpus. The idea is to be able to perform retrieval tasks using this mathematical representation. Now, the matrices used in LSI are large and sparse. This fact restrict our focus to a particular type of SVD: the *Truncated SVD*. Roughly speaking, our interest turned out to be how we can increase the performance of IR when using LSI.

At the same time we found out an *Oracle* product that performs LSI. On June 5, 2012, Oracle announced that *Collective Intellect, Inc.* was going to be acquired. *Collective Intellect, Inc.* used to be a provider of cloud-based social intelligence solutions to enable organizations to deal with consumer's conversations on social media platforms. These tools are aimed to monitor,

[1] Tensors are a generalization of matrices

understand and respond to these conversations. According to a press note [oraa], its analytics engine processes tens of millions(!) of conversations daily and it is able to eliminate redundant or irrelevant data. It also captures intentions, interests, identify emerging trends, etc. This engine was integrated to *Oracle's Software-as-a-Service* initiative [orab]. Unfortunately, due to the agenda of the people working on this product, we could not get more information for now, but we hope to continue conversations with this group in the near future, in order to focus our thesis project on that product.

As a related alternative, we found out another product (which was mentioned in table 1.2) called *Gensim* [ŘS11] which introduces a novel distributed algorithm that allows to process a comparable amount of documents. This is particularly important because some of the other frameworks that we found have not (apparently) be used for these types of corpora. In the future, the study of this software may be useful for the purposes of the Oracle's initiative.

With this brief background we are now able to establish the scope of this project. We will narrow down our work to the following:

1. Focus will be on the efficient computation of SVD, in the context of LSI.

2. A current industry need is to be able to process "hundreds of millions" of documents daily. An example is the Oracle's Software-as-a-Service initiative.

3. Special attention will be paid to the pitfalls of distributed/parallel algorithms used to find SVD decompositions of matrices that meet the requirements of LSI.

This definition of the scope of the problem will serve to filter out the vast universe of literature related to the SVD. At the same time, it will allow us to keep focus on the problem that the industry needs to solve. This problem still remains to be formulated in a more strict way.

1.4 Definition of the problem

With this background, now we are able to establish our problem and research question. It can be stated as follows:

Given the fact that modern LSI applications require end to end processing of hundreds of millions of documents every day, in a streamed updatable

fashion, how can the particular SVD step be optimized through parallel or distributed computing?

Note that this problem definition includes all the restrictions we had imposed in the previous section. It mentions the use of LSI which implies the objective of Information Retrieval which also implies that we are dealing with large and sparse matrices; it requires online, streamed processing and the volumn of information is described as the one that is currently found in real world applications. One additional observation is that we will prefer algorithms that fit in comodity hardware.

The final objective of this research project, is to find what the last step that computer science research has taken to solve this problem is. The route that we will follow to figure that out is described in the next section.

One of the points that we want to emphasize is that we give a lot of importance to the understanding of the underlying problems and solutions. For this reason, we tried to be as detailed as possible in each section. We want to make this clear because it should come as a surprise that during the work we describe more problems that needed to be fixed to reach the current state of the art. In other words, the problem definition established here might have smaller subproblems that will be explained in the following chapters.

1.5 Roadmap

The following chapters of this work are organized as follows. First we do a quick review of the history of the SVD as a mathematical tool. The purpose is to recognize the age of this tool which can account as a measure of the available information available in the literature. This will allow us to determine some seminal articles or books written about this topic. Also, we will be able to recognize when the first computational algorithms were proposed. On the other hand we want to be able to understand what are the types of problems that used to be solved with this tool.

After that we deal with the mathematical notion of the SVD. There we will provide very sensible explanations of the underlying logic, theorems, demonstrations regarding this subject. The proof of the existence of a SVD decompositions of a matrix is provided in two different ways: through the Spectral Theorem and through a geometrical proof. This step is important because we need to understand the tool before trying to add some improvements. We consider that the first step is taken there.

Once we understand the SVD as a mathematical tool, we will focus in the

next chapter on the actual reason why we need such type of factorization. For this, we briefly study what Information Retrieval is and what Latent Semantic Indexing is. We also justify the usage of SVD as a tool for solving this problem. In doing so, we will discover that what we actually need is a modified version of the SVD: the Truncated SVD which is in turn justified by the so called Eckart-Young-Mirsky theorem.

An important fact that we need to establish is whether it is possible to find a numerical computational algorithm. So we will spend some time reviewing some concepts of Numerical Analysis in order to determine whether the SVD can be considered a well-conditioned problem. Otherwise, carrying on the project would not be worth it.

After having understood the general problem and the tool, combined with the industry need, we will turn to see what available implementations of the truncated SVD have been discovered or designed. The industry problem will force us to mainly focus on the distributed/parallel versions but in order to understand them we will have to also pay attention to the serial versions that originate those other versions.

To finish this work, we present some conclusions regarding the algorithms that we investigated. We point out some of the pitfalls in the proposed methods and also propose some immediate paths that have not been tried yet, and that we expect (in case this turn to be a thesis) investigate and propose deeper improvements. Since the literature regarding this topic is really vast, we will also try to mention some of the options that we should investigate in the near future to get a thorough vision of the state of the art.

Chapter 2

A Brief History of Singular Value Decomposition

The development of the theory and algorithms for Singular Value Decomposition has a long history but it can be divided in three important periods. The first of them covers the 19th century and approaches the problem from the linear algebra perspective. The work of Beltrami, Jordan and Sylvester belong to this era. At that time, one of the interesting problems was that of the bilinear forms. In particular, they wanted to determine whether a bilinear form could be transformed to another one using transformations of the two spaces involved in the system.

The second era corresponds to an approach from a different area of mathematics: the Integral equations. Schmidt and Weyl did the most important research in this area. This development occurred during the first part of the 20th century. Also, it is during this time that Picard (in 1910) first called the numbers σ_k used in the SVD decomposition as "singular values" (*valeurs singulieres* in French).

Some decades after that, during the second part of the 20th century, the development of SVD took place in the field of numerical analysis. Particularly, the introduction of computers aimed the development of more and more efficient algorithms to obtain the SVD.

In the following sections we provide a brief review of the main milestones in the development of SVD's theory. When considered pertinent, some mathematical treatment will be done to better explain the thoughts of the mathematicias. If the reader is interested in a more detailed explanation of the history, he or she can consult [Ste93]. In that paper, Stewart does a great job explaining the work of the mentioned mathematicians us-

ing the modern notation of matrices in a way that the development of the mathematical manipulation seems less obscure.

2.1 Beltrami's work

Eugenio Beltrami was a professor of Physics and Mathematics at the University of Pavia and worked on bilinear forms, foundations of geometry, theory of elasticity, electricity, hydrodynamics, etc. In 1873, Beltrami published his work on SVD in the "Journal of Mathematics for the Use of the Students of the Italian Universities". This is the first time the Singular Value Decomposition (without that specific name, though) was described in the literature. The purpose of the publication was to encourage the use and study of the bilinear forms since this functions were being studied by other eminent mathematicians like Kronecker and Christofel. Specifically, he mentions that some interesting problems arise when one eliminates restrictions (that some mathematicians had artificially imposed) such as identical or inverse substitutions to both series of variables. Let us briefly explain how the development of the idea was done. Note that we will be using modern mathematical notations for simplicity.

First, Beltrami focused his attention to bilinear forms, that is, linear systems that involves two series of variables like:

$$f = \Sigma_{rs} a_{rs} x_r y s$$

In this equation, x_r and y_s are the two groups of variables. In modern day mathematics, this bilinear form is equivalent to:

$$f(x, y) = x^T A y$$

That is, the series of variables can be considered as vectors. Then, by doing the substitution of $x = U\xi$ and $y = V\eta$, we get:

$$f(x, y) = (U\xi)^T A(V\eta) = \xi^T S\eta$$

Where $S = U^T AV$. Basically, he says that the series x and y are linear combinations of U and V, which are orthogonal. In other words linear combinations of the basis expresed by the columns of those matrices. To solve the problem, he tries to do a diagonalization of the matrix Σ. He

noticed that there are $n^2 - n$ degrees of freedom that can be manipulated in order to do so. Let us now call $S = \Sigma = diag(\sigma_1, ..., \sigma_n)$. In that case, by post-multiplying by V we get:

$$\Sigma = U^T AV \iff \Sigma V^{-1} = U^T AVV^{-1} \iff \Sigma V^T = U^T A \tag{2.1}$$

and also, by pre-multiplying by $(U^T)^{-1}$:

$$(U^T)^{-1}\Sigma = (U^T)^{-1}U^T AV \iff U\Sigma = AV \tag{2.2}$$

From 2.1 and 2.2, we can obtain the following equivalent equations:

$$\Sigma^2 U^T = U^T AA^T \tag{2.3}$$

and:

$$(A^T A)V = V\Sigma^2 \tag{2.4}$$

We can solve this system to find the values of σ by talking the determinants:

$$det(AA^T - \sigma^2 I) = 0 \tag{2.5}$$

$$det(A^T A - \sigma^2 I) = 0 \tag{2.6}$$

So, for getting the SVD one can first find the roots of 2.6. After that, one can obtain U from equation 2.3 and finally one can obtain V from equation 2.1.

We want to emphasize again that the development by Beltrami seemed more involved due to the notation he used. In that sense, we should mention that Stewart's paper does a great job explaining the details mentioned above.

2.2 Jordan's work

Camille Jordan was a French mathematician who worked mainly on group theory. He is well known because of the Jordan Normal Form and the Jordan matrix that are used in linear algebra. In 1874, he published "Mémoire sur les formes bilinéaires" where he tackled the problem of finding the singular values as an optimization problem. His problem can be formulated as follows: *Given the bilinear form:*

$$P = x^T Ay \tag{2.7}$$

how can we find the maximum and minimum of P with the restrictions of

$$\|x\|^2 = \|y\|^2 = 1 \tag{2.8}$$

?

To solve this, he took the derivative:

$$dP = dx^T Ay + x^T Ady = 0 \tag{2.9}$$

According to him, the expression above can be handled as a linear combination of dx and dy in the following way:

$$dx^T Ay + x^T Ady = \sigma dx^T x + \tau dy^T y \tag{2.10}$$

Then, by similarity of the terms he rewrites the above equation into another two:

$$Ay = \sigma x \tag{2.11}$$

$$x^T A = \tau y \tag{2.12}$$

Again, we can solve this system just as Beltrami did through the determinant:

$$D = \begin{vmatrix} -\sigma I & A \\ A^T & -\sigma I \end{vmatrix} = 0 \tag{2.13}$$

What he gets at this point is the value of the σ factors which correspond to a singular value of A because notice that here he does not have a complete matrix in the left side or in the right side of his original equation.

2.3 Sylvester's work

James Joseph Sylvester is yet another mathematician that worked on the SVD theory. He made fundamental contributions to matrix theory and number theory. His contributions to the theory of SVD was published in three different articles during 1889. In particular, in "The Messenger of Mathematics" he published "A new proof that a general quadric may be reduced to its canonical form by means of a real orthogonal substitution". What is interesting in this work is that he describes an iterative algorithm for reducing a quadratic form to a diagonal form. After that, he submits another note where he moves from reducing a quadratic form to a bilinear form. This later note corresponds to an iterative algorithm to decompose a matrix into its SVD. His induction procedure starts with a matrix A:

$$A = \begin{pmatrix} a & 0 & f \\ 0 & b & g \\ f & g & c \end{pmatrix} \tag{2.14}$$

And the problem is to eliminate elements f and g. The procedure for this is not quite clear for us and hence it will not be presented nor explained here. As [Ste93] mentions, Sylvester is not really rigorous in his mathematical reasoning. One interesting thing is that Sylvester used the term "canonical multipliers" for the singular values of the matrix A.

2.4 Approach from Integral Equations

Schmidt and Weyl developed a theory of SVD from the perspective o Integral equations. Schmidt published his work in 1907, whereas Weyl published his work in 1912.

Schmidt was a German mathematician advised by David Hilbert. He worked on integral equations and made important contributions to functional analysis. The problem that Schmidt tried to solve is to find the best approximation to A of the form:

$$A \cong \sum_{i=1}^{k} x_i y_i^T \tag{2.15}$$

so that

$$\|A - \sum_{i=1}^{k} x_i y_i^T\| = \min \tag{2.16}$$

In other words, he wants to find the best approximation of rank not greater than k. This approximation idea is important for our purposes and it will be further developed in the next chapter where we will describe how SVD is used for Latent Semantic Indexing.

Similarly, Weyl developed his theory of SVD using the problem of determining the rank of a matrix in the presence of error. In this case, if A is a matrix of rank k and $\tilde{A} = A + E$, then he found that the last $n - k$ singular values of $\tilde{A}$ satisfy:

$$\tilde{\sigma}_{k+1}^2 + ... + \tilde{\sigma}_n^2 \leq \|E\|^2 \tag{2.17}$$

In other words, Weyl's contribution to the theory of the SVD was to develop a general perturbation theory. This particular contribution is important because it can be considered a numerical analysis of the conditioning

of the SVD problem. We will not say anything more about this here because we will reserve a complete chapter of this work to this matter.

2.5 Approach from Numerical Analysis

By the second half of the 20th century, it was obvious that computation of SVD is a difficult task that should be optimized. According to Stewart, the first practical methods for computing the SVD come from Kogbetliantz and Hestenes which resemble the Jacobi eigenvalue algorithm which will be quickly mentioned in a later chapter. After that, Golub and Kahan published faster algorithms. In particular, Golub published in 1970 his SVD algorithm [GR70] that has been popular during the last three decades. This algorithm will also be explained in the chapter reserved for serial algorithms. Let us just additionally say that [GR70] brought SVD to prominence [GCGO07] because it explained how it could be applied, what were the pitfalls of other methods and how to overcome this difficulties to obtain a numerically stable SVD algorithm.

Since then, many other algorithms have been proposed. And it is precisely these difficulties and the need for faster algorithms one of the reasons why we decided to study this topic. As it will become clearer in the following chapters there are still some pitfalls in the current algorithms that deserve some more effort in order to be understood and solved.

Chapter 3

Theory behind SVD

Let us begin by giving the formal definition of the SVD factorization, which is in indeed a theorem (we will restrict ourselves to real matrices, that is, to matrices whose entries belong to the field $\mathbb{R}$).

Theorem 3.1 (Singular Value Decomposition)**.** *Let A be a real matrix of $n \times m \Rightarrow \exists$ orthogonal matrices V $(n \times n)$ and U $(m \times m)$, and diagonal matrix Σ $(m \times n) \ni$:*

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

where Σ has the following properties:

$$\begin{aligned} &\Sigma = diag(\sigma_1, \ldots, \sigma_p), && for \;\; p = min(n, m) \\ &\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0, && for \;\; r = rank(A) \\ &\sigma_{r+1} = \sigma_{r+2} = \cdots = \sigma_p = 0 \end{aligned}$$

Note that in the theorem 3.1, we are considering diagonal matrices on its more generic form that does not require them to be square; the definition of diagonal matrix M can simply be that any element other than M_{ii} becomes zero.

Before presenting the proofs, is convenient to provide more context about the theory behind this matrix factorization (and probably also, part of the motivation behind).

3.1 The intuition behind SVD

In this section we will provide several informal ways of looking at the SVD factorization, aiming to ignite the formal discussion of next section (where we prove the SVD theorem).

3.1.1 SVD as a function composition

The first thing to remember, is that matrices are the operational representation of an special type of function between vector spaces[1] called linear transformations (also called linear mappings or linear functions). We say is an operational representation, in the sense that they provide an explicit recipe to apply the function. Furthermore, the functions they represent are special, as they have the nice property of preserve algebraic structure across domain and codomains. Such property can be summarized as:

$$f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$$

Where the addition and products mentioned above, are the vector addition and multiplication by an scalar; defined for vector spaces. For the specific case of this work, where we restrict our attention to real matrices, we can tell that they do represent functions $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$.

In this context of linear functions, the matrix multiplication is the operational representation of the composition of the associated functions. A matrix factorization is in essence, a way to understand what the underlying function does; the whole product can be seen as serial algorithm, where each matrix represents on particular step or transformation. Each of the four matrices that appear on the SVD factorization, has its own function as follows:

- A is a function $\mathrm{F_A} : \mathbb{R}^n \rightarrow \mathbb{R}^m$
- V is a function $\mathrm{F_V} : \mathbb{R}^n \rightarrow \mathbb{R}^n$
- Same goes for V^T, which is $\mathrm{F_{V^T}} : \mathbb{R}^n \rightarrow \mathbb{R}^n$

[1] The reader is invited to review any Linear Algebra textbook, to recall the definition of a vector space

- Σ is a function $\mathrm{F}_\Sigma : \mathbb{R}^n \rightarrow \mathbb{R}^m$
- U is a function $\mathrm{F}_\mathrm{U} : \mathbb{R}^m \rightarrow \mathbb{R}^m$

Thus, in the context of function compositions, the SVD factorization can be restated as:

$$\mathrm{F}_\mathrm{A}(\mathbf{x}) = \mathrm{F}_\mathrm{U}(\mathrm{F}_\Sigma(\mathrm{F}_{\mathrm{V}^\mathrm{T}}(\mathbf{x})))$$

3.1.2 SVD as a change of basis

Next thing to remember, are the specially nice properties of orthogonal matrices. By definition they are square matrices $(n \times n)$, and their columns form an orthonormal basis of $\mathbb{R}^n$; this property implies that they are invertible, but also, that the inverse is specially easy to compute: it is just the transpose. In addition, orthogonal matrices are an special case of change of basis matrices: if Q is an orthogonal matrix, then it can be seen as a function which takes vectors in the coordinates of its column basis, and that spits as result the coordinates in the canonical basis. On the same line, Q^{-1} does represent the opposite change of basis (from the canonical coordinates to those in terms of the columns of Q).

Having set the proper context, let us restate the SVD factorization as a sequence of successive transformations:

1. Start with a vector $\mathbf{x} \in \mathbb{R}^n$ in canonical coordinates.
2. Perform a change of basis using matrix V^T, from the canonical coordinates to those in terms of the columns of V.
3. Once expressed as coordinates of columns of V, apply the linear transformation Σ; this not only converts the vector from $\mathbb{R}^n$ to $\mathbb{R}^m$, but also expresses the coordinates in terms of the columns of U.
4. Once transformed, perform another change of basis using matrix U; from the coordinates of columns of U to the canonical ones We end then with a vector in $\mathbb{R}^m$.

3.1.3 SVD as a geometrical interpretation

And what was the advantage of the factorization then? In simple terms, decomposing the function behind matrix A, as a sequence of three simpler (easier to understand) transformations. Here the geometric interpretation helps to complete the picture, as orthogonal (change of basis) matrices do represent rigid transformation in space, that is, they do not alter the lengths of vectors (hence, preserve shapes). Strictly speaking, orthogonal matrices can be decomposed as a rotation and a reflection; but for geometric intuition, is often desirable to think in the rotation part only.

On the other hand, diagonal matrices are the simplest transformation possible, they do not change the basis but just expand or contract the coordinates along the axis given by the basis. Again, if we consider the generic case of diagonal matrices, a negative element D_{ii} would additional provoke a reflection in the axis i; but since the SVD decomposition produces only positive elements on the diagonal, we ignore this case and just think in terms of contractions or expansions along the axes.

Armed with this geometrical insight, we can enhance our understanding of the action of A through the SVD decomposition, by associating to the simpler operations the corresponding geometrical transformations. The geometric visualization usually requires a couple of simplifications: first of all, the dimensions of domain and codomain must be reduced; as it is easier to visualize things in $\mathbb{R}^2$ or $\mathbb{R}^3$, than in an arbitrary $\mathbb{R}^n$. Given two dimensions fit well in a screen, let us pick $\mathbb{R}^n = \mathbb{R}^m = \mathbb{R}^2$.

Secondly, we need to focus our attention in an specific set of points (as visualizing the effect of a linear transformation against "all" vectors in space, even for $\mathbb{R}^2$, is a quite abstract and complex task). The usual procedure is to pick the vectors in the unitary sphere in $\mathbb{R}^2$ (which contains in particular the columns of V, as they are unit orthogonal vectors).

Let us proceed now: let the matrix A be of dimensions 2×2, the matrices V and U be formed by unit column vectors $\{\mathbf{v_1}, \mathbf{v_2}\}$ and $\{\mathbf{u_1}, \mathbf{u_2}\}$, respectively; and let matrix Σ be $diag(\sigma_1, \sigma_2)$ such that $\sigma_1 > \sigma_2 > 0$. We will additionally assume that $\sigma_1 > 1$ and $\sigma_2 < 1$, in order to allow them represent an expansion and contraction, respectively. The previously described steps of the SVD factorization, can be now augmented with the corresponding geometrical transformations:

1. Start with unit sphere in $\mathbb{R}^2$, with the unit vectors $\mathbf{v_1}$ and $\mathbf{v_2}$ living inside of it.
2. Action of V^T: Rotate the space such that $\mathbf{v_1}$ and $\mathbf{v_2}$ become the new orthogonal basis (this transformation leaves the shape of the sphere intact).
3. Action of Σ: Once rotated, the unit sphere is expanded in the direction of $\mathbf{v_1}$ (per σ_1), and contracted in the direction of $\mathbf{v_2}$ (per σ_2).
4. Action of U: Once reshaped, the resulting ellipse is taken from the basis $\{u_1, u_2\}$ back to the canonical basis (this transformation changes the orientation of the ellipse, given that is not a symmetric figure; but still it preserves it shape).

These steps can be summarized in the following figure [2]:

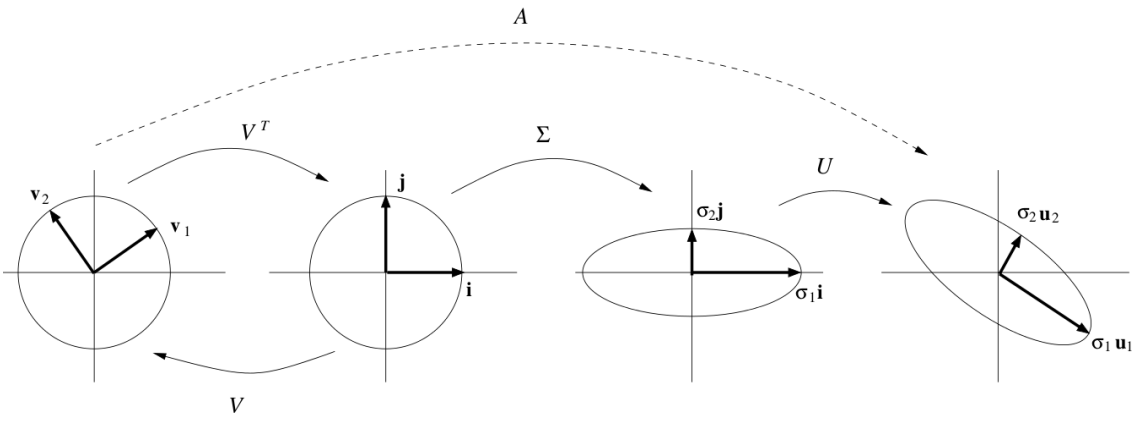

Figure 3.1: Geometrical interpretation of $A = U\Sigma V^T$, over the unit sphere.

Although the geometrical interpretation works great for a simple example in $\mathbb{R}^2$, there are a couple of missing details in the action of matrix Σ which are worth mentioning. The first, is that the dimensions of Σ are those

[2] Which was taken from a MathStackExchange phorum post, we could not locate back the original source though.

of the original matrix ($n \times m$); therefore, its action is not only expanding or contracting, but also a migration of space (from $\mathbb{R}^n$ to $\mathbb{R}^m$). If there are more rows than columns ($m > n$), the transformation Σ will produce a bigger vector than its input (the diagonal entries beyond position n will be zeroes, which in turn will fill the new vector entries with zeroes as well; up to m entries). If on the contrary we have more columns than rows ($m < n$), then the effect will be a truncation of the input vector (resulting vector has as many entries as rows in Σ). In our example, this change of space was not perceived, as $n = m$.

The second omitted detail about the action of Σ in fig. 3.1, is even more subtle: along with the migration of space $\mathbb{R}^n$ to space $\mathbb{R}^m$, we are also changing the basis; from $\{\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}\}$ to $\{\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_m}\}$. This additional effect may not be evident at all, but is thanks to an additional property that is required on the two basis for the SVD factorization to hold:

$$A\mathbf{v_i} = \sigma_i \mathbf{u_i}, \quad \forall i = 1 \ldots r, \quad \text{where } r = rank(A).$$

The above property tells us that the vectors from the two basis were picked in a very special way: each vector $\mathbf{u_i}$ is parallel to the image under A of its associated $\mathbf{v_i}$; in other words, the transformation A maps the $\mathbb{R}^n$ basis $\{\mathbf{v_1}, \ldots \mathbf{v_n}\}$, into vectors which are parallel to the $\mathbb{R}^m$ basis $\{\mathbf{u_1}, \ldots \mathbf{u_m}\}$. Given that both basis are orthonormal, a consequence from this observation is that the orthogonality of the basis $\{\mathbf{v_1}, \ldots \mathbf{v_n}\}$ is preserved under A ([Kal96]). This is not a trivial property, and not every basis has it (given A is assumed to be given). This is actually the key problem of the SVD factorization, and the proofs provided in the next section, focus around the problem of finding such special basis.

3.2 The SVD proofs

The rest of this chapter provides more theoretical background about the SVD decomposition, in particular, it provides two different proofs of theorem 3.1:

- Algebraic proof using the Spectral Theorem.
- Geometric proof (implicitly using Compactness).

Each one of those proofs is intended to bring a different perspective, about such an important result as SVD is. The list is not exhaustive of course, there could be many more ways of proving it; but hopefully the short list presented here, will give the reader an idea about the rich theory behind this decomposition.

As with any mathematical theorem, proving is done based on previous results; since this is not a text book, we can not afford the luxury of proving every auxiliary theorem we use. However, we made an effort for at least mentioning explicitly the theorems; pointing to references, when possible, about their respective proofs. For some few cases (like the Spectral Theorem), we did include the proof of the auxiliary theorem as well.

3.2.1 Algebraic proof (using Spectral Theorem)

The proof in this section relies on the Spectral Theorem, which says that we can diagnonalize a symmetric matrix. Instead of jumping right away to the proof of SVD with such heavy machinery, we prefer a gradual approach consisting of the following steps:

- Emphasize that the main task of SVD factorization, is to find a basis for $\mathbb{R}^n$ whose orthogonality is preserved under A.
- Introduce Fundamental Theorem of Linear Algebra along the four subspaces related to an arbitrary matrix A.
- Motivated by the discussion about the four subspaces, bring to the picture the symmetric matrix $A^T A$ (aka the gramian), and state the properties we will need for the SVD proof.
- The symmetric nature of $A^T A$, will justify the usage of the Spectral Theorem; which we proceed to prove.
- Finally, we prove SVD theorem 3.1 itself using the Spectral Theorem as the main tool, but we also use the auxiliary theorems stated along the way.

3.2.1.1 The factorization properties

Let us resume the discussion from last section, where we claim that all we needed was that the vectors of the two bases had the property of $A\mathbf{v_i} = \sigma_i \mathbf{u_i}$. Let us prove such claim, and show that if that condition is met, then SVD factorization can be achieved.

Theorem 3.2 (SVD Part 1: the factorization)**.** *Let A be a real matrix of $n \times m$,if $\exists$ orthonormal basis $\{\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}\}$ for $\mathbb{R}^n$, and another orthonormal $\{\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_m}\}$ for $\mathbb{R}^m$ which hold the following property:*

$$A\mathbf{v_i} = \sigma_i \mathbf{u_i}, \quad \forall i = 1 \ldots r, \quad \text{where } r = rank(A).$$

Then we can factorize matrix A as $U\Sigma V^T$.

where

- *V is the orthogonal matrix formed by arranging vectors $\mathbf{v}$'s*
- *matrix U is defined similarly for vectors $\mathbf{u}$'s*
- *The only non zero entries of diagonal matrix Σ, are those $\Sigma_{ii} = \sigma_i > 0$ for $1 <= i <= r$.*

Proof. This theorem 3.2 is mentioned in [Kal96], though not explicitly proved. Let us do it here, following the advice from [Str88], that the trick is to think about a matrix multiplication AB, as the result of matrix-vector products $(A\mathbf{b_i})$, where the vectors are the columns of B. In our particular case, the matrix-vector products we have are $A\mathbf{v_i}$; if we arrange them as columns of a new matrix it would be equal to AV. That is:

$$\left[A\mathbf{v_1} \mid A\mathbf{v_2} \mid \cdots \mid A\mathbf{v_n}\right] = A\left[\mathbf{v_1} \mid \mathbf{v_2} \mid \cdots \mid \mathbf{v_n}\right] = AV$$

Since we do not have product AV in our target result, let us use the fact that V is orthogonal; which in particular implies that $V^{-1} = V^T$. That allows to focus on an target result, which involves AV:

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

So we can focus in proving that $AV = U\Sigma$. Let us work the left side first, which per our previous observation that $A\mathbf{v_i}$ are the columns of matrix product AV, and per hypothesis that $A\mathbf{v_i} = \sigma_i \mathbf{u_i}$ can be rewritten as follows:

$$\begin{aligned}
&AV && = \\
&A\left[\mathbf{v_1} \mid \mathbf{v_2} \mid \cdots \mid \mathbf{v_n}\right] && = \\
&\left[A\mathbf{v_1} \mid A\mathbf{v_2} \mid \cdots \mid A\mathbf{v_n}\right] && = \\
&\left[\sigma_1\mathbf{u_1} \mid \sigma_2\mathbf{u_2} \mid \cdots \mid \sigma_r\mathbf{u_r} \mid A\mathbf{v_{r+1}} \mid A\mathbf{v_{r+2}} \mid \cdots \mid A\mathbf{v_n}\right]
\end{aligned}$$

Let us now develop the left side $U\Sigma$ by thinking again in the result, as formed by columns of the form $U\mathbf{\Sigma_i}$ (where $\mathbf{\Sigma_i}$ is the ith column of diagonal matrix Σ):

$$\begin{aligned}
&U\Sigma && = \\
&\left[U\mathbf{\Sigma_1} \mid U\mathbf{\Sigma_2} \mid \cdots \mid U\mathbf{\Sigma_n}\right] && = \\
&\left[U\mathbf{\Sigma_1} \mid U\mathbf{\Sigma_2} \mid \cdots \mid U\mathbf{\Sigma_r} \mid U\mathbf{\Sigma_{r+1}} \mid U\mathbf{\Sigma_{r+2}} \mid \cdots \mid U\mathbf{\Sigma_n}\right] && = \\
&\left[U\mathbf{\Sigma_1} \mid U\mathbf{\Sigma_2} \mid \cdots \mid U\mathbf{\Sigma_r} \mid \underbrace{U\mathbf{0} \mid U\mathbf{0} \mid \cdots \mid U\mathbf{0}}_{n-r}\right] && = \\
&\left[U\mathbf{\Sigma_1} \mid U\mathbf{\Sigma_2} \mid \cdots \mid U\mathbf{\Sigma_r} \mid \underbrace{\mathbf{0} \mid \mathbf{0} \mid \cdots \mid \mathbf{0}}_{n-r}\right] && =
\end{aligned}$$

The last $n - r$ zero vectors were a consequence of the definition of Σ, which only has non-zeroes on diagonal up to position r. And the columns $U\mathbf{\Sigma_i}$ can be simplified further, as each column vector $\mathbf{\Sigma_i}$ has the only non-zero entry σ_i precisely at position i. Hence, only column i of U survives after multiplying it by $\mathbf{\Sigma_i}$, and the final effect is just the multiplication by scalar σ_i:

$$U\Sigma = \left[U\mathbf{\Sigma_1} \mid U\mathbf{\Sigma_2} \mid \cdots \mid U\mathbf{\Sigma_r} \mid \underbrace{\mathbf{0} \mid \mathbf{0} \mid \cdots \mid \mathbf{0}}_{n-r}\right] = \left[\sigma_1\mathbf{u_1} \mid \sigma_2\mathbf{u_2} \mid \cdots \mid \sigma_1\mathbf{u_r} \mid \underbrace{\mathbf{0} \mid \mathbf{0} \mid \cdots \mid \mathbf{0}}_{n-r}\right]$$

If we put together the developments for each side, we are almost done:

$$
\begin{aligned}
& AV && = \\
& \left[\sigma_1 \mathbf{u_1} \mid \sigma_2 \mathbf{u_2} \mid \cdots \mid \sigma_r \mathbf{u_r} \mid \underbrace{A\mathbf{v_{r+1}} \mid A\mathbf{v_{r+2}} \mid \cdots \mid A\mathbf{v_n}}_{n-r}\right] && = \\
& \left[\sigma_1 \mathbf{u_1} \mid \sigma_2 \mathbf{u_2} \mid \cdots \mid \sigma_1 \mathbf{u_r} \mid \underbrace{\mathbf{0} \mid \mathbf{0} \mid \cdots \mid \mathbf{0}}_{n-r}\right] && = \\
& U\Sigma &&
\end{aligned}
$$

□

It can be observed that the proof is not complete, though we are almost done; in order to achieve an equality between AV and $U\Sigma$, the only missing part is that the last $n - r$ items on each side are the same. This can be restated in an additional theorem:

Theorem 3.3 (SVD Part2: basis of null space)**.** *Assuming same definitions as theorem 3.2, it must be the case that:*

$$A\mathbf{v_i} = \mathbf{0}, \quad for \;\; (r+1) \leq i \leq n$$

which is equivalent to say that those vectors $\mathbf{v_i}$*, belong to the null space of A (they actually form a basis of it).*

We do not have yet the required machinery to proof theorem 3.3, but we will do it on the next section, when we introduce the subspaces associated with each matrix A.

3.2.1.2 The Fundamental Theorem of Linear Algebra

In order to proof the pending theorem 3.3 from previous section, we need to present the four subspaces that an arbitrary matrix A introduces. But before that, a few important definitions and remarks:

- Matrix application $A\mathbf{x}$ can be seen as a linear combination of the columns of A:

$$A\mathbf{x} = \left[\mathbf{A_1} \mid \mathbf{A_2} \mid \cdots \mid \mathbf{A_n}\right] \mathbf{x} = \sum_{i=1}^{n} x_i \mathbf{A_i}$$

- Subspace: A subset of a vector space, which is itself a vector space (that is, contains the $\mathbf{0}$ and is closed under the addition and multiplication by an scalar).
- Dimension: Is the size of any basis of a vector space (an important result in Linear Algebra, shows that all the basis must have the same number of elements; hence, the dimension is a property of the space itself). The dimension of a vector space (or subspace) V is denoted as $\dim(V)$.
- Given vector space V and a subspace $W \subset V$, the subspace $W^{\perp} \subset V$ consists of all the vectors of V which are orthogonal to all the vectors of W. $W^{\perp}$ is called the orthogonal complement of W.

Now is right time to talk about the subspaces: we already established that each matrix of $m \times n$, can be seen as the operational representation of a linear transformation with signature $\mathbb{R}^n \rightarrow \mathbb{R}^m$. The action of A, in transforming the vectors from one space to the other, has an interesting effect on each side: both domain ($\mathbb{R}^n$) and codomain ($\mathbb{R}^m$), are broken in two orthogonal pieces. Those pieces actually, happen to be subspaces and their basis are contained on the matrices V and U of the SVD factorization! But let us explain piece by piece; a good start, is the column space.

The column space of a matrix A, is pretty much the concept of the image of a function; that is, the set of all vectors in $A\mathbf{x} \in \mathbb{R}^m \ni \mathbf{x} \in \mathbb{R}^n$, and it is denoted as $\mathrm{C}(A)$. Another way of looking at it (per one of the remarks above), is that each application of the linear transformation A (that is, each $A\mathbf{x}$), converts the input vector $\mathbf{x}$ into a linear combination of the columns of A; therefore, $\mathrm{C}(A)$ is the spanning set generated by the columns of A. It can be proved that this subset is actually a subspace of $\mathbb{R}^m$.

The next subspace is also clearly understood, is the so called null space of A. It consists of all the solutions to the homogeneous system $A\mathbf{x} = \mathbf{0}$ and is denoted as $\mathrm{N}(A)$. In the language of transformations, is the set of all vectors $\mathbf{x} \in \mathbb{R}^n$ that function A compresses into the zero vector of $\mathbb{R}^m$. This subset at least contains the vector $\mathbf{0}$, but in general it will contain much more vectors (only the non-singular matrices have $\mathrm{N}(A) = \{\mathbf{0}\}$). Again, it can be proved that this subset is also a subspace (though this one belongs to $\mathbb{R}^n$).

The next two subspaces, are not that intuitive to introduce; unless we think now in terms of the transformation represented by matrix A^T. This matrix represents a linear transformation that goes into the opposite direction of A, that is, from $\mathbb{R}^m$ to $\mathbb{R}^n$. If we think in the image of this function $\{ A^T \mathbf{y} \in \mathbb{R}^n \mid y \in \mathbb{R}^m \}$, an interesting realization comes to the picture: what if we apply the same idea as before, that $A^T \mathbf{y}$ is a linear combination of the columns of A^T:

$$A^T \mathbf{y} = \left[(\mathbf{A^T})_\mathbf{1} \mid (\mathbf{A^T})_\mathbf{2} \mid \cdots \mid (\mathbf{A^T})_\mathbf{m} \right] \mathbf{y} = \sum_{i=1}^{m} y_i (\mathbf{A^T})_\mathbf{i} = \sum_{i=1}^{m} y_i (\text{row } i \text{ of } A)$$

In order words, columns of A^T are the rows of A, therefore; the column space of A^T is precisely the row space of original matrix A; this is denoted as C(A^T) and it can also be proved that it is a subspace of $\mathbb{R}^m$. The last and fourth subspace, comes from considering the null space of A^T; that is, those vectors $\mathbf{y}$ in $\mathbb{R}^m$ which are compressed into the zero vector of $\mathbb{R}^n$. Is not hard to prove that this is a subspace as well; it is called the left null space of A, and denoted as N(A^T).

Summarizing, the four subspaces associated to any matrix A of $m \times n$ are the following (intuitive proofs that all of them are indeed subspaces can be found in [Str88]):

- C(A^T): row space, lives in $\mathbb{R}^n$
- N(A): null space, lives in $\mathbb{R}^n$
- C(A): column space, lives in $\mathbb{R}^m$
- N(A^T): left null space, lives in $\mathbb{R}^m$

The first thing we note is the intentional grouping of these subspaces; while C(A^T) and N(A) belong to $\mathbb{R}^n$ [the domain of A], C(A) and N(A^T) belong to $\mathbb{R}^m$ [the codomain of A]. These pairs of subspaces have more in common than merely sharing same hosting space, they are orthogonal with each other! This is the time to meet what Strang calls the Fundamental

Theorem of Linear Algebra (part II[3]):

Theorem 3.4 (Fundamental Theorem of Linear Algebra (part II))**.** *Let A be a real matrix of $n \times m$, then*

$$\mathrm{C}(A^T) = \mathrm{N}(A)^\perp \ \wedge \ \mathrm{C}(A) = \mathrm{N}(A^T)^\perp$$

Proof. Let $\mathbf{x} \in \mathrm{N}(A)$, then $\mathbf{x}$ satisfies the equation $A\mathbf{x} = \mathbf{0}$; but the resulting vector in $\mathbb{R}^m$ has as entries the dot product of $\mathbf{x}$ with the rows r_i of A, therefore, the equation $A\mathbf{x} = \mathbf{0}$ can be rewritten as m equations of the form:

$$r_i \cdot \mathbf{x} = 0, \ for \ 1 \leq i \leq m$$

which is essentially saying that the vector $\mathbf{x}$ is orthogonal to all the rows r_i of A; therefore, it is orthogonal to every linear combination of them. But those linear combinations are precisely the row space $\mathrm{C}(A^T)$; thus $\mathrm{C}(A^T) \perp \mathrm{N}(A)$, or, reusing previously introduced terminology (see remarks section), we can say that the row space is the orthogonal complement of the null space (which is written as $\mathrm{C}(A^T) = \mathrm{N}(A)^\perp$).

An analogous argument can be constructed for $\mathrm{C}(A)$ and $\mathrm{N}(A^T)$, using the equation $A^T\mathbf{y} = \mathbf{0}$ (details are in [Str88]). Thus, we can also conclude that the column space is the orthogonal complement of the left null space (which can be written as $\mathrm{C}(A) = \mathrm{N}(A^T)^\perp$). □

Having established the orthogonality of these subspaces, allow us to introduce a secret weapon that will finally help us prove the pending theorem 3.3. This weapon is another theorem that establishes a relationship between the dimension of any subspace and its orthogonal complement [4]:

Theorem 3.5 (Orthogonal Complement Dimension Theorem)**.** *Let W any subspace of $\mathbb{R}^n$, then is the case that*

$$\dim(W) + \dim(W^\perp) = n$$

[3]Strang presents the theorem parts in the opposite order in [Str88]; but we preferred to keep our own order, aiming to match better the flow of deductions presented in this work.

[4]The name was provided by us, as Lang does not name it in his book [Lan04].

An even more generic version of this theorem is proved by Lang in [Lan04] (theorem 2.3 in that book), and it basically says that if we take any subspace and its orthogonal complement together, they form the entirety of the host space! Another way of seeing this result, is saying that the host space V is the direct sum of the subspace W and its orthogonal complement (denoted as $V = W \oplus W^{\perp}$). Intuitively, the notion of a direct sum tells us that there is nothing out of the union of the subspace W and its orthogonal complement $W^{\perp}$; every vector in the original space can be expressed as a sum $x_1 + x_2$ (where each $\mathbf{x_1} \in W$ and $\mathbf{x_2} \in W^{\perp}$, and W), and $W^{\perp}$ do not share anything other than zero vector ($W \cap W^{\perp} = \{\mathbf{0}\}$).

Using this weapon, we can finally prove the pending theorem 3.3, which was about proving that the last $n - r$ vectors of V actually belong to $\mathrm{N}(A)$.

Theorem 3.3 (SVD Part2: basis of null space)**.** *Assuming same definitions as theorem 3.2, it must be the case that:*

$$A\mathbf{v_i} = \mathbf{0}, \quad for \;\; (r+1) \leq i \leq n$$

which is equivalent to say that those vectors $\mathbf{v_i}$*, belong to the null space of A (they actually form a basis of it).*

Proof. By hypothesis we know that

$$A\mathbf{v_i} = \sigma_i \mathbf{u_i}, \quad \forall i = 1 \dots r, \quad \text{where } r = rank(A).$$

Since the vectors $\mathbf{u_i}$'s form a basis of $\mathbb{R}^m$, that implies none of them can be zero; therefore $A\mathbf{v_i} \neq \mathbf{0} \;\; \forall i = 1 \dots r$. By definition, such condition implies that those vectors $v_i \notin \mathrm{N}(A)$, but since $\mathbb{R}^n = \mathrm{C}(A^T) \oplus \mathrm{N}(A)$, then the only other option for those vectors $\{\mathbf{v_1}, \mathbf{v_2}, \dots, \mathbf{v_r}\}$ is to belong to the row space $\mathrm{C}(A^T)$. Actually, since they are all orthogonal, they form a basis of $\mathrm{C}(A^T)$ (because $\dim(\mathrm{C}(A^T)) = \mathrm{rank}(A) = r$). Let us call this basis B_r.

Let $\mathbf{v_i} \in \mathbb{R}^n$, which also belongs to $\{\mathbf{v_{r+1}}, \mathbf{v_{r+2}}, \dots, \mathbf{v_n}\}$; since $\mathbf{v_i}$ is orthogonal to every vector in B_r (as all the $\mathbf{v}$'s form an orthonormal basis of $\mathbb{R}^n$), then $\mathbf{v_i}$ can not belong to the subspace generated by B_r, which happens to be $\mathrm{C}(A^T)$. Using again the fact that $\mathbb{R}^n = \mathrm{C}(A^T) \oplus \mathrm{N}(A)$, we

can tell that the only other option for $\mathbf{v_i}$ is to belong to the nullspace $\mathrm{N}(A)$. And by definition of nullspace:

$$A\mathbf{v_i} = \mathbf{0}, \quad \forall i = (r+1) \dots n$$

which completes the proof. Mirroring the reasoning about basis B_r of the row space, we can also tell that the vectors $\{\mathbf{v_{r+1}}, \mathbf{v_{r+2}}, \dots, \mathbf{v_n}\}$ form a basis of the null space $\mathrm{N}(A)$. □

Since we established already the orthogonality between the four subspaces of matrix A, we can simply apply theorem 3.5 to them in pairs (depending on whether they are hosted on same space), and derive the following equations:

1. In $\mathbb{R}^n$: $\mathrm{C}(A^T) = \mathrm{N}(A)^\perp \implies \dim(\mathrm{N}(A)) + \dim(\mathrm{C}(A^T)) = n$
2. In $\mathbb{R}^m$: $\mathrm{C}(A) = \mathrm{N}(A^T)^\perp \implies \dim(\mathrm{N}(A^T)) + \dim(\mathrm{C}(A)) = m$

These two equations are what Strang calls the Fundamental Theorem of Linear Algebra (Part I); further references are [Str88] and [Str93]. The two parts of such theorem together, basically describe what are the subspaces generated by matrix A, what is the relationship among them (orthogonality) and what are their dimensions. The fig. 3.2 below (taken from [Str88]), summarizes both parts of this important theorem:

Strang goes further in contextualizing the SVD factorization in the above diagram ([Str93]), by noting that the columns of the matrices V and U, actually contain the basis of these four subspaces:

- The orthogonal matrix V contains a basis for the row space $\mathrm{C}(A^T)$ in the first r columns, and a basis for the null space $\mathrm{N}(A)$ in the last $n - r$ columns. We showed this already while proving theorem 3.3.
- The orthogonal matrix U contains a basis for the column space $\mathrm{C}(A)$ in the first r columns, and a basis for the left null space $\mathrm{N}(A^T)$ in the last $m - r$ columns.

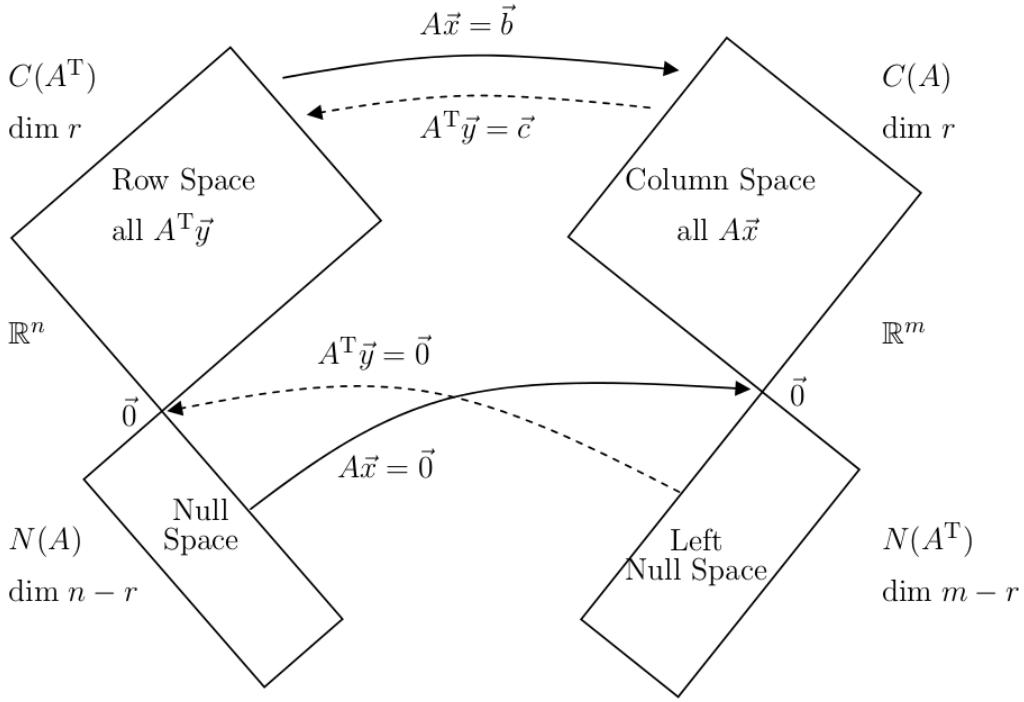

Figure 3.2: Visualization of the Fundamental Theorem of Linear Algebra

The last observation makes the SVD factorization even more astonishing and intriguing: not only it allows one to understand the true nature of an arbitrary matrix A, by explicitly giving the two change of basis that make A a positive diagonal matrix Σ (having just compressions and expansions). Also, if we consider a basis as a representation of a vector space; then the matrices V and U of the SVD factorization can be considered a representation of the four subspaces generated by that particular matrix A. Putting together the three matrices as in $A = U\Sigma V^T$, gives the truly complete picture about the effects of transformation A.

3.2.1.3 The gramian matrix $A^T A$

We brought the fig. 3.2 not only to illustrate the Fundamental Theorem of Linear Algebra, but also to justify the introduction of the matrix $A^T A$ (also called the gramian matrix of A). From the figure, it can be seen that the

matrix A takes the row space C(A^T) into the column space C(A); and we know that both subspaces have the same dimension $r = \text{rank}(A)$. As Strang explains in [Str88], the dimensions of the domain $\mathbb{R}^n$ and codomain $\mathbb{R}^m$ do not tell the real story about the linear transformation behind A; it is rather the dimensions of C(A^T) and C(A).

If we just pay attention to those subspaces, then the matrix A behaves like a bijection (this is proved in [Str88]); that is, if we took the submatrix of dimensions $r \times r$ that results from eliminating dependent rows and columns in A, such matrix would be invertible and the inverse would take the column space C(A) into the row space C(A^T). Thus, the real information of matrix A lies in the one-to-one transformation of C(A^T) into C(A); the null spaces in both sides (N(A) and N(A^T)) do not contribute much to the the transformation A, as they just get compressed to the zero vector.

Alright, putting aside the null spaces and focusing only in the bijection that A performs between the row and column spaces, one may be tempted to think from fig. 3.2 that the transpose of A (denoted as A^T), is actually the inverse of A in the context of those subspaces. But as Strang promptly clarifies in [Str88], that honor belongs only to the actual inverse of A. We refer to the inverse here, not in the regular sense, but restricted to the subspaces C(A^T) and C(A) (that is, we are talking about the inverse of the submatrix of $k \times k$ that we mentioned above). The effect of A^T is correct at the level of the whole subspace C(A); it takes it back to C(A^T). But the vectors that were originally mapped by A, are not necessarily recovered after applying A^T to that image $A\mathbf{x}$.

One particular way of reinforcing the fact that A^T is not the inverse, is by an indirect measure. If we take the dot product between the starting point $\mathbf{x}$ in $\mathbb{R}^n$, and the result of applying A^T to its image $A\mathbf{x}$, that would give us an indication of how close or distant they are (in the end, the dot product and the orthogonal projection are intimately related). If the starting and final vectors happen to be the same, the cited dot product shall be $\|x\|_2^2$ (where $\|.\|_2$ represents the known euclidean distance). Let us confirm ourselves that is not the case:

$$
\begin{aligned}
\mathbf{x} \cdot (\, A^T A\mathbf{x}) &= \\
\mathbf{x}^T (\, A^T A\mathbf{x}) &= \\
(\, \mathbf{x}^T \, A^T)(A\mathbf{x}) &= \\
(A\mathbf{x})^T (A\mathbf{x}) &= \\
(A\mathbf{x}) \cdot (A\mathbf{x}) &= \\
\|A\mathbf{x}\|_2^2 &\leq \|A\|_2^2 \, \|\mathbf{x}\|_2^2
\end{aligned}
$$

From the above development, we can see indeed that $\mathbf{x} \cdot (\, A^T A\mathbf{x}) \neq \|x\|_2^2$; actually, in the last step we used a general property of matrix norms, which in particular applies to the extension of the vector norm $\|.\|_2$ to matrices. We will not define it formally, but it suffices to keep the intuition that norm of A, denoted as $\|A\|_2$, is a measure of the distortion that the linear transformation A does on the space (is actually the maximum distortion on the unit sphere). In the last step we can see that such measure of distortion, is precisely one of the factors that prevents that simply taking the transpose A^T as a way back, sends us to the starting point in the row space.

Above reasoning is a further argument for $A^T \neq A^{-1}$; such special property only applies to orthogonal matrices (like the U and V, which appear in the SVD factorization). For orthogonal matrices, $A^T A = I$, where I is the identity matrix; and though that does not occur in general for an arbitrary matrix A, the function composition represented by $A^T A$ is quite interesting; it may not be the identity function, but at least it has one of its properties:

$$(\, A^T A)^T \, = \, A^T \, (\, A^T)^T \, = \, A^T A$$

The matrix $A^T A$ (called gramian) is equal to its transpose, which is the definition of a symmetric matrix. This particular symmetric matrix is quite important for us, as it represents the bridge between the SVD factorization and the Spectral Theorem; in short, the Spectral Theorem guarantees that any symmetric matrix is diagonalizable, and applying such factorization to our special matrix $A^T A$, give us the two bases that we need to build the SVD factorization (which happen contain, the bases of the four subspaces

of A). These details will be developed in the next two sections. For the moment, we will just finish the current one by establishing a few more facts about $A^T A$, which show its close connection with A.

Is not hard to show that $A^T A$ and A share the same null space; and that actually implies that the rank of both matrices is the same. Let us state that in a theorem and prove it:

Theorem 3.6 (Rank of the Gramian Matrix)**.** *Let A be a real matrix of rank r $\implies$ its gramian matrix $A^T A$ has the same rank r.*

Proof. Let us begin proving that $\mathrm{N}(A) = \mathrm{N}(A^T A)$. Let $\mathbf{x} \in \mathrm{N}(A)$, then:

$$A\mathbf{x} = \mathbf{0} \iff A^T(A\mathbf{x}) = A^T\mathbf{0} \iff A^T A\mathbf{x} = \mathbf{0}$$

Above just proves that $\mathrm{N}(A) \subset \mathrm{N}(A^T A)$, but the other contention can also be deduced. Let $\mathbf{x} \in \mathrm{N}(A^T A)$, then:

$$\begin{aligned}
&A^T A\mathbf{x} = \mathbf{0} && \iff \\
&\mathbf{x}^T A^T A\mathbf{x} = \mathbf{x}^T\mathbf{0} && \iff \\
&(A\mathbf{x})^T A\mathbf{x} = 0 && \iff \\
&(A\mathbf{x}) \cdot (A\mathbf{x}) = 0 && \iff \\
&\|A\mathbf{x}\|_2^2 = 0 && \iff \\
&A\mathbf{x} = \mathbf{0} && \iff \\
&\mathbf{x} \in \mathrm{N}(A)
\end{aligned}$$

The two developments above show that $\mathrm{N}(A^T A) = \mathrm{N}(A)$, that in particular means that $\dim(\mathrm{N}(A^T A)) = \dim(\mathrm{N}(A))$. If we apply the theorem 3.5 theorem to each matrix, we get the same dimension for the row space (the row spaces of both matrices live in $\mathbb{R}^n$, which has dimension n of course):

$$\dim(\mathrm{C}(A^T)) = n - \dim(\mathrm{N}(A)) = n - \dim(\mathrm{N}(A^T A)) = \dim(\mathrm{C}((A^T A)^T))$$

Knowing that the dimension of the row spaces is the same, we just need to recall that $\dim(\mathrm{C}(A^T)) = \mathrm{rank}(A)$, and then we can conclude that $\mathrm{rank}(A) = \mathrm{rank}(A^T A)$. □

Another useful result about the matrix $A^T A$, that we will need when proving the SVD theorem, talks about the qualities of its eigenvalues.

Theorem 3.7. *Let A be a real matrix of rank r, then its gramian matrix $A^T A$ has r positive eigenvalues.*

Besides reusing theorem 3.6, the key step in proving this result, has to do with the previously used fact that $\mathbf{x}^T A^T A\mathbf{x} = \|A\|_2^2 \geq 0$; which implies that $A^T A$ is not only symmetric but also semipositive-definite (by definition). And it turns out, that semipositive-definite matrices have the desired property of having r positive eigenvalues. We will not prove this theorem here, but [Str88] can be consulted for further details.

3.2.1.4 The Spectral Theorem

With the introduction in previous section of the gramian matrix $A^T A$, it is time for the heavy machinery that will allow us to prove the SVD theorem 3.1; we are talking of course, about the Spectral Theorem. We will not only present the theorem, but also include one of the many possible proofs. The one we chose was published by Wilf in [Wil81], and we did so because of its brevity and elegance. It actually makes an interesting connection between the Linear Algebra world where have been moving so far, and the world of Topology; the link is created by using the properties of compact sets. But before defining what a compact set is, let us motivate this interesting usage.

A common problem that many people are familiar with, specially if they took single-variable calculus courses at college; is that of finding the minima or maxima of a given function $f : \mathbb{R} \to \mathbb{R}$. There is a mechanical part about how to calculate those critical points, but a crucial part is to verify on the first place, if they actually exist. A common requirement is for the function f to be continuous, but further requirements are also needed on its domain. The reader may recall the famous Extreme Value Theorem, which

establishes what are those conditions on the domain for single-variable functions:

Theorem 3.8. *Let $f : \mathbb{R} \to \mathbb{R}$ be a continuous function over the closed (and bounded interval) $[a, b]$, then f reaches its maximum and its minimum over the same interval.*

The theorem 3.8 tells us what the required condition is on the domain of a continuous function f, in order to guarantee that its critical points (minimum/maximum) actually exist. The condition is simply to have a closed interval, which though not evident, has the following two properties:

1. Closed: It contains its limit points [5] (like 0 and 1).
2. Bounded: There are real numbers which serve as lower and upper bounds for all the elements of the interval [6].

A set which has these properties of being closed and bounded, is said to be compact. Being compact, is a generalization of the closed intervals on the real line. Why do we need such generalization? Well, simply because we may be interested in calculating minima and maxima for functions defined over more complex sets than $\mathbb{R}$; for example, vector spaces or matrix spaces. Another important property of compact sets, is that continuous functions preserve its "compactness"; that is, if S is a compact set and f a continuous function, then f(S) is also a compact set.

Armed with this brief, but hopefully enough understanding of what compact sets are; let us proceed with the proof. The first required artifact is a function called Od, which intuitively measures how close is an square matrix of $n \times n$, from having a diagonal form:

$$\mathrm{Od}(A) = \sum \sum_{i \neq i} A_{ij}^2$$

[5] The intuitive idea for limit points, is that a "limit" process can approximate them by using points inside the set.

[6] In the case of a closed interval in $\mathbb{R}$, these bounds happen to be the limit points and are inside the interval; but for more general spaces, that may not be the case.

The next artifact we need is the set of all orthogonal matrices (denoted $\mathrm{O}(n)$ [7]). Using product multiplication, this set has the algebraic structure of a group (though we do not really need such property here).

The next tool is the following theorem, about Jacobi's method for finding eigenvalues, which tells us that it is always possible to perform rigid transformations (change of basis), that take one non-diagonal matrix into a new one that is closer to a diagonal form (per the metric defined by function Od).

Theorem 3.9. *If A is a real non-diagonal matrix, $\implies$ there is an orthogonal matrix J such that* $\mathrm{Od}(J^T AJ) < \mathrm{Od}(A)$.

An informal proof is given in [Wil81], and a more detailed discussion is found in [GVL12]. This is the last tool we needed for presenting the proof of the Spectral Theorem from [Wil81], which follows below:

Theorem 3.10. *If A is a symmetric real matrix, $\implies$ there is a real orthogonal matrix Q such that $Q^T AQ$ is diagonal.*

Proof. Let f be the function that, given the fixed matrix A, maps every orthogonal matrix P into the product $P^T AP$. This function is continuous over the compact set $\mathrm{O}(n)$; hence, $\mathrm{f}(\mathrm{O}(n))$ is also compact. The set $\mathrm{f}(\mathrm{O}(n))$ contains all the possible products of fixed matrix A with orthogonal matrices, that may or may not give a diagonal as result. Then, we basically use brute force: search for the best candidate in that set of options. And we do that, by using the metric we define specifically for that purpose: the function Od. Thus, we want to search for the product of the form $P^T AP$ (an element of $\mathrm{f}(\mathrm{O}(n))$ which give us the minimum value of function Od. Here is where the compactness property comes into play; if the domain $\mathrm{f}(\mathrm{O}(n))$ was not compact, we could not even talk about the minimum of the (continuous) function Od.

Knowing that, the existence of the minimum of function Od in the set $\mathrm{f}(\mathrm{O}(n))$ is granted, the next thing to realize is that such minimum must be zero. This is easily seen by using reduction to the absurd: let us suppose that

[7] Not to be confused with the big-O notation for algorithms complexity.

the minimum reached at matrix $D = Q^T AQ$, is not zero. That would mean the matrix D is not diagonal yet and then, by theorem 3.9 we know that there must exist another matrix $D^* = Q^T DQ$, such that $\text{Od}(D^*) < \text{Od}(D)$. But that contradicts the assumption that the minimum was reached at D. Therefore, the minimum of Od must be zero.

If the minimum of Od is zero, it means that is reached on a matrix which is diagonal (the square of all its off-diagonal[8] elements is zero, which means they are all zero). The existence of such diagonal matrix of the form $Q^T AQ$ proves the theorem.

□

There are a couple of useful corollaries that derive from the Spectral Theorem just proved:

- In the search of the minimum, there was an implicit iterative process of applying multiplications, where the matrix at step $i + 1$, was obtained from the previous matrix at step i, in the following way: $D_{i+1} = J^T D_i J$ (the matrices J are obtained by the Jacobi's method of rotations [9]). If we think in the series of transformations done by this iteration, from the original matrix A, we would realize that all we did was to apply rigid transformations; therefore, the resulting entries in the diagonal must be real (there is no way that applying a rotation to real matrix, produces another matrix with complex entries).

- The second observation is that the result $D = Q^T AQ$, where D is a diagonal matrix; implies that $AQ = QD$, which in turn can be broken into individual equations of the form $A\mathbf{q_i} = d_i\mathbf{q_i}$. This tells us that the columns of the orthogonal matrix Q are the eigenvectors of A, and that the diagonal of D contains its eigenvalues. We started the other way around, but in practice, the usual motivation for diagonalizing a symmetric matrix (or for applying the Jacobi's method), is to actually find the eigenvalues and eigenvectors.

[8] The off-diagonal elements of a matrix, are those which do not lie on the diagonal.

[9] This algorithmic perspective, is actually the main inspiration of this proof, which is taken from [Wil81]

These two corollaries will be useful in the final proof to come, that of the SVD theorem 3.1.

3.2.1.5 The spectral proof of SVD

We are now all set to prove the SVD theorem, and actually, we do not need to prove the full version stated in theorem 3.1, because we have worked out the factorization part with theorems theorem 3.2 and theorem 3.3. Those theorems started from the assumption, that there exist two orthonormal bases such that $A\mathbf{v_i} = \sigma_i \mathbf{u_i}$; what remains to prove then, is the existence of those bases.

Theorem 3.11 (SVD Part 3: existence of the bases)**.** *Let A be a real matrix of $m \times n$ with rank r $\implies$ there exist orthonormal basis $\{\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}\}$ and $\{\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_m}\}$, for $\mathbb{R}^n$ and $\mathbb{R}^m$ respectively; along with positive real values $\sigma_1 \geq \sigma_2 \geq \cdots \sigma_r$, such that:*

$$A\mathbf{v_i} = \sigma_i \mathbf{u_i}$$

Proof. We know from previous sections, that the key is to find first the basis for $\mathbb{R}^n$, such that its orthogonality is preserved through A (this interesting approach, and most if this particular proof, is taken from Kalman [Kal96]).

Per the Fundamental Theorem of Linear Algebra, the symmetric matrix $A^T A$ came to the picture; and here comes the magical step: it turns out, that the eigenvectors of such matrix (whose existence is guaranteed by the Spectral Theorem we just proved), are precisely the orthonormal basis $\{\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}\}$ that we are looking for. Let us verify that is actually the case, that is, that A preserves the orthogonality of the eigenvectors of $A^T A$.

Let $\mathbf{v_i}$ and $\mathbf{v_j}$ be eigenvectors of $A^T A$, and λ_j the eigenvalue of $\mathbf{v_j}$, then:

$$(A\mathbf{v_i}) \cdot (A\mathbf{v_j}) = (A\mathbf{v_i})^T (A\mathbf{v_j}) = \mathbf{v_i}^T (A^T A\mathbf{v_j}) = \mathbf{v_i}^T (\lambda_j \mathbf{v_j}) = \lambda_j \mathbf{v_i} \cdot \mathbf{v_j}$$

The above derivation tells us that the orthogonality of the images of the eigenvectors, named $A\mathbf{v_i}$ and $A\mathbf{v_j}$, totally depends of the orthogonality of

the pre-images $\mathbf{v_i}$ and $\mathbf{v_j}$. Another way of saying that, given that the two eigenvectors were picked arbitrarily, is that the orthogonality of the eigenvectors of $A^T A$ is preserved through A. This is exactly the basis we were looking for!

The real work is to find the the basis $\{\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}\}$ in $\mathbb{R}^n$, as the basis in $\mathbb{R}^m$ is simply calculated to meet the requirement that $A\mathbf{v_i} = \sigma_i \mathbf{u_i}$. When proving that orthogonality of the $\mathbf{v}$'s is preserved, we came up with the following identity:

$$(A\mathbf{v_i}) \cdot (A\mathbf{v_j}) = \lambda_j \mathbf{v_i} \cdot \mathbf{v_j}$$

The particular case of $i = j$, will give us the following relationship between the eigenvalues of $A^T A$ and the images $A\mathbf{v_i}$ (let us recall that the Spectral Theorem guaranteed an orthonormal basis, hence $\|\mathbf{v_i}\|_2 = 1$):

$$(A\mathbf{v_i}) \cdot (A\mathbf{v_i}) = \lambda_i \mathbf{v_i} \cdot \mathbf{v_i} \iff \|A\mathbf{v_i}\|_2^2 = \lambda_i \|\mathbf{v_i}\|_2^2 \iff \|A\mathbf{v_i}\|_2 = \sqrt{\lambda_i}$$

Now we just define the vectors $\mathbf{u}$'s as the unitary version of the images of vectors $\mathbf{v}$'s; and use the above relationship to bring the eigenvalues of $A^T A$ into the picture:

$$\mathbf{u_i} = \frac{A\mathbf{v_i}}{\|A\mathbf{v_i}\|} = \frac{1}{\sqrt{\lambda_i}} A\mathbf{v_i};\ \forall\ i = 1 \ldots r = \text{rank}(A)$$

Do we have enough singular values λ_i in $A^T A$ (we need exactly r), and all of them are positive? (otherwise, $\sqrt{\lambda_i}$ would not be real). We have properly prepared for this moment, and the whole purpose of having mentioned theorem 3.7, was precisely to give a positive answer to these questions. We are safe in this regard then, and can proceed.

In general $\text{rank}(A) = r < m = \dim(\mathbb{R}^m)$, so we must likely need to extend the set $\{\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_r}\}$ to an orthonormal basis of $\mathbb{R}^m$ to complete the SVD factorization. Fortunately, there is a known theorem in Linear Algebra that guarantees that we can do that indeed (see theorem 2.1.1 from

[GVL12], for example).

Finally, by naming $\|A\mathbf{v_i}\|_2 = \sqrt{\lambda_i}$ as σ_i (for $1 \leq i \leq r$), we can finally achieve the long wanted property of the two bases:

$$A\mathbf{v_i} = \|A\mathbf{v_i}\|\,\mathbf{u_i} = \sqrt{\lambda_i}\mathbf{u_i} = \sigma_i\mathbf{u_i}\ ;\ \ \forall\ 1 \leq i \leq r = \text{rank}(A)$$

□

The just proved theorem 3.11 is the precondition that we need to apply theorem 3.2 and theorem 3.3 from previous sections. The eigenvalues of the symmetric matrix $A^T A$ may not necessarily be in descending order, as the SVD theorem requires; but once we have them, we can sort them in such way (which will implicitly sort the $\mathbf{v}'s$ and $\mathbf{u}'s$ vectors in the bases). All together can finally tackle the original SVD theorem 3.1, stated at the beginning of this chapter. This concludes our proof of the SVD factorization, using the Spectral Theorem as the main tool.

It may had seen as an extremely detailed proof, even if not all the auxiliary theorems were proved in this work (but most of them were at least mentioned explicitly, some even formally). This unusual level of detail may appear cumbersome for the professional mathematician, as all the literature we consulted always presented quite compressed proofs which skipped or simplified a lot steps. But we considered that the approach taken here, could be useful for the occasional reader and for people who are introducing themselves to the topic, and want to have an almost self-contained proof of the SVD that requires little previous context (at least much less than regular books and articles). Worth to say also, that this level of detail was needed for the authors' own understanding as well.

3.2.2 Geometric proof (using Compactness)

After the exhaustive proof of the previous section, we wanted to refresh the reader with a totally different type of proof for SVD; one that brings a new perspective for the way in which the special basis for $\mathbb{R}^n$ is chosen.

Thinking about modular theorem proving, that is, factorizing common results for the sake of clarity; one could consider that the theorem 3.2 and theorem 3.3 are some kind of common step for many possible SVD proofs.

They deal with the task of proving that, given certain condition over the bases in $\mathbb{R}^n$ and $\mathbb{R}^m$ ($A\mathbf{v_i} = \sigma_i \mathbf{u_i},\ \forall\ i = 1 \ldots \text{rank}(A)$); the SVD factorization holds. Those auxiliary theorems then, reduce the task of proving the SVD theorem 3.1 to the much more specific (but hard) subproblem of finding the bases. Per discussion in previous section, we know that such problem can be reduced even further, to the one of finding the orthonormal basis for $\mathbb{R}^n$ $\{\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}\}$, such that its orthogonality is preserved through A ([Kal96]). This last remark can be considered the true essence of a whole family of proofs for SVD Theorem, where each one brings a particular way of finding a basis with such an special property.

Intuitively, this property of preserving orthogonality could be thought as a generalization of the eigenvectors behavior, which are the vectors not "moved" by transformation A but just scaled; this in particular implies that if the eigenvectors formed an orthogonal basis of the space prior application, their images under A will still form a basis (eigenvectors are defined when A has signature $\mathbb{R}^n \rightarrow \mathbb{R}^n$, which means A is an square matrix). Something similar occurs for the $\mathbf{v}$'s in SVD, but extended to a couple of spaces instead of just one: we can not expect these vectors are not moved by A, as they migrate of space ($\mathbb{R}^n \rightarrow \mathbb{R}^m$); but we request that whatever landing they do on $\mathbb{R}^m$, they still form an orthogonal basis there.

The proof we are about to present now is thanks to Blank et al [BKS89]; which presents a quite interesting approach: using pure geometric arguments [10] he finds the basis whose orthogonality under A gets preserved.

The whole reasoning occurs on the unit sphere and its image over an square matrix A; and here comes a great connection with our previous comments about the true essence of an arbitrary matrix A of $m \times n$ with rank r: the real information is on the mapping from the row space to the column space ($\text{C}(A^T) \rightarrow \text{C}(A)$), and restricted to those subspaces A is a bijection $\mathbb{R}^r \rightarrow \mathbb{R}^r$; working with a bijective linear transformation means that A^{-1} does exist. Without loosing generality then, we will assume that matrix A is square and non-singular (invertible); because if it was not, we can do a zoom and focus on its embedded bijection $\mathbb{R}^r \rightarrow \mathbb{R}^r$, and calculate the basis there (and later extend to whole basis of host spaces $\mathbb{R}^n$ and $\mathbb{R}^m$).

The unit sphere is picked as the source of the $\mathbf{v}$'s, simply because we

[10] With an implicit use of compactness

want them to be an orthonormal basis (which in particular requires them to be unitary). For starting to form this basis, we could start picking an arbitrary unit vector; picking a second vector in the sphere such that is perpendicular to the former is no issue either; but how pick the second one such that the property the property $\mathbf{v_1} \perp \mathbf{v_2}$ is preserved through A? Every vector $\mathbf{v}$ in $\mathbb{R}^r$ defines a hyperplane P, which actually happens to be a subspace on its own. But such hyperplane is actually the orthogonal complement of the subspace generated by the vector alone; which implies that $P \perp \mathbf{v}$. So what? P merely becomes an infinite source of orthogonal vectors to the second choice $\mathbf{v}$; but the question of how to pick one such that the orthogonality property gets preserved, is still unanswered.

The main point of the proof in [BKS89], is that if we choose properly the first vector $\mathbf{v}$, meaning if we choose the one which gets the maximum expansion through A, then the orthogonality relation of $\mathbf{v}$ with the hyperplane it defines gets preserved. Here comes then a very powerful and beautiful idea at the same time: having found a whole subspace that is orthogonal to the first choice vector, allows one to forget about the original host space $\mathbb{R}^r$ and focus on that subspace only; the new hyperplane would essentially be $\mathbb{R}^{r-1}$ embedded in $\mathbb{R}^r$, and the intersection with the sphere in $\mathbb{R}^r$ would the unit sphere in $\mathbb{R}^{r-1}$. Thus, we can apply recursively the same procedure in that subspace, for finding the next unit vector such that the orthogonality of its hyperplane gets preserved through A!

Therefore, the theoretical recursive algorithm would be to find one vector $\mathbf{v_i}$ at a time, by working only on a subspace of dimension $r - i + 1$ ($\mathbf{v_1}$ is found in whole $\mathbb{R}^r$, $\mathbf{v_2}$ is found in an embedded $\mathbb{R}^{r-1}$, $\mathbf{v_3}$ in the nested embedded $\mathbb{R}^{r-2}$, etc). If we wanted to think in a proof rather than a constructive algorithm, we could use induction and claim that we know how to find the first $r - 1$ vectors $\mathbb{R}^{r-1}$, and proceed to find the remaining vector in $\mathbb{R}^r$.

We have reduced then, the problem of finding the right basis of $\mathbf{v}$'s to the following theorem:

Theorem 3.12. *Let A a non-singular matrix of $r \times r$; let be vectors $\mathbf{v}, \mathbf{w} \in \mathbb{R}^r \ni A\mathbf{v} = \mathbf{w} \wedge \|\mathbf{w}\|_2 = \max\{\|\mathbf{x}\|_2 \ni \|A^{-1}\mathbf{x}\|_2 = 1\}$ and let S; T be hyperplanes in $\mathbb{R}^r \ni S$ is the orthogonal hyperplane of $\mathbf{v}$; and T is image of S under $A \implies T \perp \mathbf{w}$ (T is also the orthogonal hyperplane of $\mathbf{w}$).*

To visualize the artifacts mentioned in the theorem, let us pay attention then to the following picture, which is the unit sphere in $\mathbb{R}^r$, mapped to an ellipsoid in $\mathbb{R}^r$ [11] In order to prove that $T \perp \mathbf{w}$, we will use the auxiliary hyperplanes S_1 and T_1 (where the second is the image under A of the former). Of course the picture aims to represent $\mathbb{R}^3$, and the hyperplanes would be embeddings of $\mathbb{R}^2$; but let us just consider them a visual representation of arbitrary dimension objects (the only representation we can imagine, indeed).

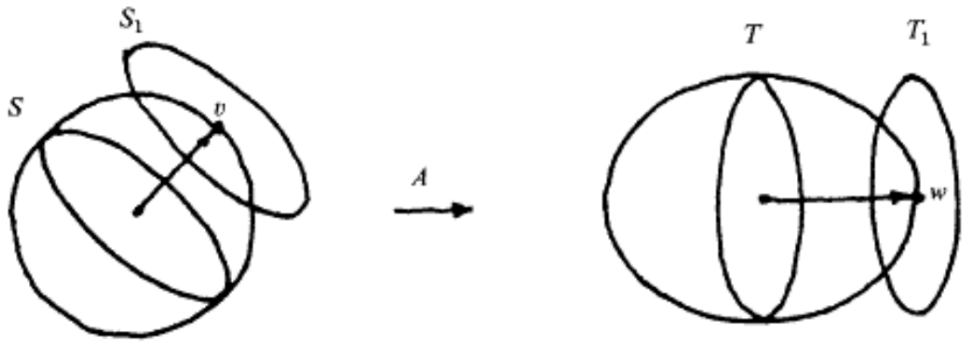

Figure 3.3: Geometrical proof of SVD Theorem: Transformation A preserves the relation $S \perp \mathbf{v}$

Proof. The geometric proof goes like this:

1. The hyperplane S_1 touches the unit sphere only at point $\mathbf{v}$. It is a geometrical result that such hyperplane is unique and that $S_1 \perp \mathbf{v}$.

2. By definition, the hyperplane T_1 is the image of S_1 under A; also, the whole unit sphere is mapped by A into an ellipsoid. Since A is a bijective function, then T_1 must touch the ellipsoid only in point $\mathbf{w}$ (just like S_1 touches the unit sphere only at point $\mathbf{v}$). Actually, T_1 must be the only hyperplane with such property (otherwise, we could

[11] A formal argument is actually required to prove that the image of the unit sphere is an ellipsoid, and even telling that is the surface of a quadratic form requires a little development. But we will omit those details, aiming to keep the spirit of this short proof.

apply A^{-1} to that other hyperplane, and it would produce a different hyperplane that also touches $\mathbf{v}$ in the unit sphere; contradicting previous point about the uniqueness of S_1).

3. Now take another sphere, big enough to cover the deformed image of the unit sphere under A (the ellipsoid). Start to shrink such sphere until it touches the ellipsoid for the first time; per definition, $\mathbf{w}$ must be part of those points of first contact.

4. Now consider the hyperplane T_2 that touches this shrunk sphere, precisely at $\mathbf{w}$. Using the same geometrical theorem of first argument about S_1, we can tell such hyperplane is unique and is orthogonal to $\mathbf{w}$.

5. Since this adjusted sphere covers entirely the ellipsoid (per definition of $\mathbf{w}$), then T_2 also touches the ellipsoid at point $\mathbf{w}$. But we argued that T_1 was the only hyperplane touching the ellipsoid at $\mathbf{w} \implies T_1 = T_2 \ \wedge \ T_1 \perp \mathbf{w}$.

6. Both hyperplanes S and S_1 are orthogonal to $\mathbf{v}$; by geometrical arguments they must be parallel then.

7. Linear transformations, in particular A, preserve parallelism; since, $S \parallel S_1 \implies$ their respective images under A must be parallel as well.

8. The image of S_1 under A is T_1, then, whatever becomes the image of S under A; it must be parallel to T_1. Let us call this image T.

9. $\therefore T \parallel T_1 \ \wedge \ T_1 \perp \mathbf{w} \implies T \perp \mathbf{w}$.

□

The key choice in the proof was $\mathbf{w}$, as being the biggest axis of the ellipsoid makes it coincide with the sphere of radius $\|w\|_2$; and that in turns allows us to transfer the properties of the hyperplane that touches the sphere

at $\mathbf{w}$ to the one that touches the ellipsoid at same point (as they become same hyperplane indeed). It is no coincidence then, that the spectral norm $\|A\|_2$ is actually defined as $\|\mathbf{w}\|_2$; that is, is defined as the maximum expansion $\|A\mathbf{x}\|_2$ that transformation A causes on the vectors belonging to the unit sphere. This norm actually, is used in the algebraic proof of Golub in [GVL12], of the SVD Theorem.

The last question the reader may have now is: where was the compactness property used? It may not be explicitly stated, but it lies behind the definition of $\mathbf{w}$: $\|\mathbf{w}\|_2 = \max\left\{\|\mathbf{x}\|_2 \ni \left\|A^{-1}\mathbf{x}\right\|_2 = 1\right\}$. The reason why $\mathbf{w}$ exists on the first place, is because the ellipsoid is a compact set (it inherits that property from its pre-image, the unit sphere, thanks to the continuity of function A[12]). Since the norm function $\|.\|_2$ is also continuous, then by a generalization of the Extreme Value Theorem from Calculus, it must reach its maximum on a point of the ellipsoid (we named that particular point as $\mathbf{w}$ in the proof).

[12]Though we did not find the name for such theorem, it must exists and state that continuous functions preserve compactness.

Chapter 4

The Problem of Latent Semantic Indexing

4.1 Vector Space Model

In information retrieval, it is mandatory to reduce the complexity of the documents to make them easier to manage. A document hence must be transformed from the full version to a document vector that describes its properties. The vector space model is an algebraic mode that allows for such representation. Each document is represented as vectors of identifiers [MRS08].

The ideas of a Vector Space Model first appeared in a paper by Salton and Wong [SWY75]. In that paper, the authors enunciate the properties of such model. One of these properties is that of placing each entities as far aways from the others as possible. It is also mentioned that in a given space, an indexing system may be expressed as a function of the density of the object space.

4.1.1 Formal definition

In the Vector Space Model, both documents and queries are represented as vectors in the following way. Consider a document space consisting of documents d_i. Each document is identified by indexed terms t_j. Each term is weighted according to their importance. In that case, each document is represented as a t-dimensional vector:

$$d_i = (t_{1i}, t_{2i}, t_{3i}, ..., t_{Ti}) \tag{4.1}$$

In other words, each dimension corresponds to a specific term that appears in the corpora. In general, if a term appears in a document, its corresponding dimension is non-zero.

Now, given two documents d_p and d_q, it is possible to obtain a similarity coefficient between them:

$$sim(d_p, d_q) \tag{4.2}$$

which is an indication of how similar both terms are. An example of a similarity functions is the inner product. Normally, document vectors are normalized because only it is enough to know its direction. The main idea in this model is that the distance between two document points in the unitary hyper-sphere is inversely correlated with the similarity between the corresponding document vectors.

4.1.2 Vector computation

The similarity function that was mentioned above is dependent on the type of codification that is chosen to represent each indexed term. Hence, an important problem is how to chose the best assignment of weights for the terms. One measure for "best assignment" could be "the one that allows a maximum possible separation between the individual documents in the space'. The idea is that in the context of information retrieval we want to retrieve a document that is sufficiently close to a user query without also necessarily retrieving its neighbors. This is known as the precision of the IR system.

One possible codification consist on using the raw frequency of a term in a document. In other words, assign as component d_{tj} in the document vector the number of times that a term t occurs in document j.

The discussed paper introduces a measure called TF-IDF (Term frequency - Inverse document frequency) that is shown to keep better precision and recall properties compared to a scheme of raw term frequency. The problem with the simple raw term frequency metric is that all terms in the corpus are considered equally important when computing the retrieved documents. TF-IDF on the other end, allows to attenuate the effect of terms that occur too often in the collection to be meaningful for relevance determination.

In TF-IDF, the definitions of term frequency and inverse document frequency are combined in the following way:

$$tf - idf_{t,d} = tf_{t,d} \times idf_t \tag{4.3}$$

Where:

tf: could be the simple raw frequency of a term in a document.

idf: a measure of how much information a given word provides, that is, whether the term is common or rare across all documents. It can be obtained with:

$$idf(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|} \tag{4.4}$$

Such a metric offers the following advantages:

1. The weight is highest when t occurs many times within a small number of documents
2. The weight is low when the term t occurs fewer times in a documents
3. The weight is low when the term t occurs in many documents
4. The weight is lowest when the term t occurs in all documents.

4.2 Introduction to Information Retrieval (IR)

The problem of Information Retrieval consists on, given a large collection of unstructured nature, find those elements (usually documents) in such collection that satisfy an information need. An example of such problem is, for example, finding plays of Shakespeare where certain characters appear. This is the information need. From there, we need to find the elements that satisfy such requirement from the collection of plays written by Shakespeare. The result must be a subset of this collection [MRS08].

A bit more formally, the setting of the IR problem requires:

1. A finite set of elements called identifiers
2. A finite set of elements called documents
3. A finite set of elements called criteria which allows to compare two documents

And then the retrieval task becomes the process whereby, given a criterion, the documents are associated to a query for which the scores are greatest[MRS08]. Traditionally, encoding the settings and the tasks above mentioned use a vector space representation.

Mathematically, the Vector Space IR is defined as:

Definition 4.2.1. Vector Space Information Retrieval Let D be a set of documents. There is a function $\sigma : D \times D \to [0, 1]$ called *similarity* with the properties of:

1. Normalization: $0 \leq \sigma(a, b) \leq 1$, $\forall a, b \in D$
2. Commutativity: $\sigma(a, b) = \sigma(b, a)$, $\forall a, b \in D$
3. Reflexivity: $a = b \Rightarrow \sigma(a, b) = 1$, $\forall a, b \in D$

Then also let $q \in D$ be a query, and $\tau \in \mathbf{R}$. The set $\Re(q)$ of retrieved documents in response to a query q is defined as: $\Re(q) = \{d \in D | \sigma(d, q) > \tau\}$

4.3 Indexing

One way of resolving an IR problem mentioned above is to create an index just as the one we see in text books. This strategy allows to find a specific subject quickly. Additionally it offers the possibility of managing more flexible matching techniques. In the case of documents, the index could be represented in the form of an *incidence matrix* like the one that follows:

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Figure 4.1: Example of a Term-Document matrix used as index in IR. Taken from [MRS08]

Using such representation, it is possible to, for example, find all documents where the word "Brutus" appear. An strategy for doing so consist on simply go to the row that represents "Brutus" and then traverse all the columns. If the column contains a 1 in it, then it means that the document

represented by that column matches the information query. It is evident that this algorithm is easier that traversing each document in search of the word "Brutus". This is the basic idea of probably the most famous information retrieval tool called *inverted index*.

4.4 Latent Semantic Indexing (LSI)

There are two main problems that appear when dealing with natural languages and that make its processing difficult:

1. the level of ambiguity that exists in natural languages, and
2. the complexity of semantic information contained in even simple sentences

In the concrete case of Information Retrieval, ambiguity plays a central role. As a simple example, if we perform a keyword search for information on a particular subject on the web the result may include many unrelated results. Another intriguing fact is that given a desired search, to people choose the same main key word less than 20% of the time [DDF$^+$90].

As a more concrete example, the Vector Space model that was discussed above has problems when dealing with synonymy and polysemy. The former refers to cases where two different words have the same meaning. We can think of this as a function that maps from the set of words to the set of meanings and two different elements in the domain are mapped to a single element in the codomain. The vector space representation explained in a previous chapter is not capable of recognizing that two words refer to the same concept because each of the terms represent a different coordinate in the space. In the literature, some of the attempts for solving this issue have used thesauri in a way that a query can be expanded to also match other related terms.

On the other hand, polysemy refers to the case where a term has multiple meanings. We can think of it again as a function that maps one single word to two different concepts. The vector space model is not able to handle this case either because two different concepts are represented by the same single dimension. Some proposals to deal with this problem include using a controlled vocabulary or even having a human intermediary to do some sort of translation. Another documented approach consist on applying boolean intersection of terms to disambiguate meaning.

Yet another problem with information retrieval systems is the lack of a technique to deal with *compound-term queries*. This is because some of the

IR systems treat words as independent concepts. The result of that is that such system assign the same importance to two commonly occurring and to two uncommonly occurring terms.

Latent Semantic Indexing is a technique presented by Deerwester et. al.[DDF$^+$90] in 1990 as a new approach to automatic indexing and retrieval that is able to overcome the previously mentioned problems. They wanted to obtain a new representation a set of terms is replaced by some other set of entities that are more reliable for the matter of performing IR. So, the result was a technique that projects the queries and the documents into a space with *latent* semantic dimensions. LSI uses statistical techniques to estimate this obscured semantic structure. One characteristic of this space, as noted by Deerwester, is that a query and a document can have a high similarity even when they do not share terms.

Some of the restrictions that Deerwester and colleges put to his problem were:

1. They wanted to examine from $1K$ to $2K$ documents and $5K$ to $7K$ indexed terms. According to their publication, this was a "reasonable size" problem. For today standards, this would not have been enough. This restriction came from the fact that by that time, existing models required a computation order of $O(n^5)$ where n is the number of documents plus terms.

2. Capture a high-dimensional representation which for them were 100 dimensions. At the same time, they wanted this paramenter to be adjustable. What a high-dimensional representation means will become clearer in a moment.

3. Be able to represent both terms and documents in the same space. This would allow them to do the process of *folding-in* new terms or documents that were not present in the original matrix.

4. Allow to retrieve documents from queries with out need of any need of intermediate clustering.

The mathematical tool that they ended up using was our famous SVD [1]. When SVD is applied to the terms by documents matrix, it reveals the latent semantic structure of the problem that consist on three matrices that

[1] In the seminal paper, they use the term *two-mode factor analysis* to refer to the fact that the working matrix contain information of two types of entity: the words and the documents.

breakdown the original relationships into linearly independent components or factors.

To be more specific, the matrix of terms by documents, that we will call A, will be decomposed into the matrices $U\Sigma V^T$. The matrix U is the matrix of terms and the matrix V is the one of documents. Each of them are composed by columns that represent the information for either terms or documents. Each of the rows represent indexing variables or factors. These factors are "artificial concepts" that represent extracted common meaning components of may different words and documents. Hence, the vectors represent the strength with which a specific term or document are associated to those "artificial concepts". In the cited paper, the authors claim that the three problems previously mentioned can be solved by this SVD decomposition.

The dimension of the obtained document or term vectors can be adjusted by removing elements from the matrix Σ. This reduction is called *Low-Rank approximation* and is what motivates the more precise name of *Truncated SVD*. When this dimensionality reduction is done, the reconstruction of the original matrix will not be exact. However, this reduction is justified by the principle of applying the Eckard-Young-Mirsky theorem.

4.5 Low-Rank approximation and Eckart-Young-Mirsky Theorem

Low-rank approximation consist on finding a matrix C_k with rank k that minimizes the Frobenius norm of the matrix difference $C_r - C_k$, when $k \ll r$. It is interesting to see that the Low-rank approximation can be obtained through the SVD. The way to perform this is following these steps:

1. Given a matrix C, compute its SVD, i.e. decompose in $C = U\Sigma V^T$.

2. Obtain from Σ a matrix Σ_k where the $r - k$ smallest (i.e. right-most values) are zeros.

3. Compute the matrix $C_k = U\Sigma_k V^T$, which is the rank-k approximation to C.

The question is now, why this techniques happens to obtain a Low-rank approximation. First let us respond to why removing zeros from the matrix

Σ give us a lower rank matrix. If we examine the matrix C, we can see that:

$$C_k = U\Sigma V^T = U \begin{pmatrix} \sigma_1 & 0 & 0 & 0 & 0 \\ 0 & ... & 0 & 0 & 0 \\ 0 & 0 & \sigma_k & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & ... \end{pmatrix} V^T = \sum_{i=1}^{k} \sigma_i u_i v_i^T \tag{4.5}$$

From here we can see that the product $u_i v_i^T$ corresponds to a matrix of rank 1. This rank-1 matrix when multiplied by C give us a rank-k matrix obtained by adding k rank-1 matrices affected by a singular value.

The formal answer is given by the following theorem [EY36]:

Theorem 4.1 (Eckart-Young-Mirsky theorem)**.** *Let $C = U\Sigma V^T$, $\Sigma = diag(\sigma_1, ..., \sigma_r, 0, ..., 0)$. For any k with $0 \leq k \leq r$ and $C_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$, then:*

$$\min_{\{Z|rank(Z)=k\}} \|C - Z\|_F^2 = \|C - C_k\|_F^2 = \sigma_{k+1}^2 + ... + \sigma_k^2 \tag{4.6}$$

This means that the best approximation is given by the matrix C_k. Actually, C_k is the best rank-k approximation of C for any unitarily invariant norm. Hence:

$$\min_{\{Z|rank(Z)=k\}} \|C - Z\|_2 = \|C - C_k\|_2 = \sigma_{k+1} \tag{4.7}$$

4.6 Explanation of the theorem

The theorem basically explains that if $rank(Z) = k$ then $\|C - Z\| \geq \|C - C_k\|$. Let us start with the target function:

$$\|C - Z_k\| = \|U\Sigma V^T - Z_k\| \tag{4.8}$$

We know that Frobenious norm is invariant under changes of orthonormal bases. Hence we can do:

$$\|U\Sigma V^T - Z_k\| = \|U^T(U\Sigma V^T - Z_k)V\| = \|\Sigma - U^T Z_k V\| \tag{4.9}$$

Let us call $N = U^T Z V$, so that we can substitute in the equation above. N is a matrix of rank k. Using this substitution, we can partition the sum into three:

$$\|\Sigma - U^T Z_k V\| = \sum_{i,j} |\Sigma_{i,j} - N_{i,j}|^2 = \sum_{i=1}^{k} |\sigma_i - N_{i,i}|^2 + \sum_{i>r} |N_{ii}|^2 + \sum_{i \neq j} |N_{i,k}|^2 \tag{4.10}$$

The minimum of this could be found when all the summands are equal to zero. In other words, N_{ii} for $i > r$ need to be zero. The off-diagonal elements of N must also be zero. Finally, the elements $N_{i,i}$ have to be equal to the singular values of C for $1 \leq i \leq k$. With this, the matrix N has been determined. Its corresponding singular vectors can be determined through $N_k = U^T Z_k V$

The intuitive interpretation of this theorem is that the singular values σ_i with $i > k$ are considered as perturbations or noise that can be eliminated. Studies of how much noise the lower singular values provide have been made in [JM01] and [And01].

This noise elimination is one reason why we do not really care that we can not reconstruct the original terms by document matrix once the dimensionality reduction is performed. It is even beneficial for our purposes because the original space is not reliable due to that noise.

4.7 Comparing objects in the latent space

We have already mentioned that the obtained matrices U and V have a geometric interpretation in the reduced space of k dimensions. In this latent space, it is possible to compare objects. The possible comparisons are:

1. How similar are terms i and j?
2. How similar are documents i and j?
3. How associated are term i and document j?

For the first comparison, what we can do is obtain the dot product of two row vectors of the A_k matrix. This product reflects how much two terms have similar pattern of occurrence. To compare all terms to all terms, we can use:

$$A_k A_k^T = (U_k \Sigma_k V_k^T)(U_k \Sigma_k V_k^T)^T = (U_k \Sigma_k V_k^T)(V_k^T \Sigma_k^T U_k^T) = U \Sigma^2 U^T \quad (4.11)$$

The larger the value at position ij is, the more similar terms i and j are.

To compare two documents we use a similar approach but this time we perform the product between two columns of the A_k matrix. Comparing all documents with all documents is equivalent to:

$$A_k^T A_k = V \Sigma^2 V^T \quad (4.12)$$

Again, the larger the value in cell ij is, the more similar documents i and j are.

Finally, for the third type of comparison the scheme is different because the result of the comparison is already given by the value of positions ij in the reduced matrix A_k.

4.8 How are queries performed?

In order to perform information retrieval, the user's query (q) must be represented as a k-dimensional vector (q_k). To do this, let U_k be the term weights matrix with size $t \times k$, Σ_k is our matrix of singular values.

$$q_k = \Sigma_k^{-1} U_k^{-1} q \tag{4.13}$$

From there, the query vector can be compared to all existing document vectors (d_i) as mentioned in the previous section.

4.9 Characterization of the Term-Document Matrix in Information Retrieval

Before moving forward, we would like to dedicate a couple of lines to present some characterization of the term document matrices that has been found in the literature. Specifically, Berry [Ber92a] describes some corpora that he used in his experiments. The characterization is shown in fig. 4.2.

In the table, μ_c represents average number of zeros per column, and μ_r represents the average number of zeros per row. Also, notice the density which is small, meaning that the matrices are rather sparse. fig. 4.3 shows graphically how the corpus for ADI looks like. These characteristics justify our focus on Algorithms that obtain the SVD for sparse matrices.

Another interesting comment about the matrices has to do with whether its size $m \times n$, generates a horizontal or vertical matrix. Remember that in the context of Information Retrieval the columns represent documents and the rows represent indexed terms [MRS08] just as shown in fig. 4.1. In the beginning when the theory of LSI started, the number of handled

Data	Columns	Rows	Nonzeros	Density	μ_c	μ_r
ADI	82	374	1343	4.38	16.0	4.0
CISI	1460	5143	66340	0.88	45.4	12.9
CRAN	1400	4997	78942	1.10	56.4	15.8
MED	1033	5831	52012	0.86	50.4	8.9
TIME	425	10337	80888	1.80	190.3	7.8
TECH	6535	16637	327244	0.30	50.0	20.0

Figure 4.2: Characteristics of the Term-Document matrices used by Berry. Taken from [Ber92a]

documents were smaller than the number of terms. Specifically, Deerwester et. al. [DDF$^+$90] uses matrices of $1,033 \times 5,823$ and $1,460 \times 5,135$. Two years later, Berry's paper [Ber92a] shows his biggest corpus as a matrix of $6,535$ documents and $16,637$ terms. During this time, computation of LSI used to take approximately a day on one machine. Even by the time [MRS08] was written, there was no successful experiments that handled more than one million documents. The book mentions that this was a big obstacle to widespread adoption to LSI.

However, modern studies, like the one presented by Řehuřek [RR11] use larger matrices with $m \ll n$. A concrete example is the *Wikipedia* corpus used by him whose size is $100,000 \times 3,199,665$. It is precisely here where another justification of our work lies. We are dealing with massive amount of documents that need to be processed through the Truncated SVD.

4.10 A brief example of LSI

Let us now briefly give a general notion of how all these ideas are put together to execute queries in the sense of Information Retrieval. After that, we will present a numerical example that we hope will further clarify any persisting doubts.

Suppose that we are given a corpus of documents and we would like to have a tool that allow us to pick a subset of documents that satisfy an

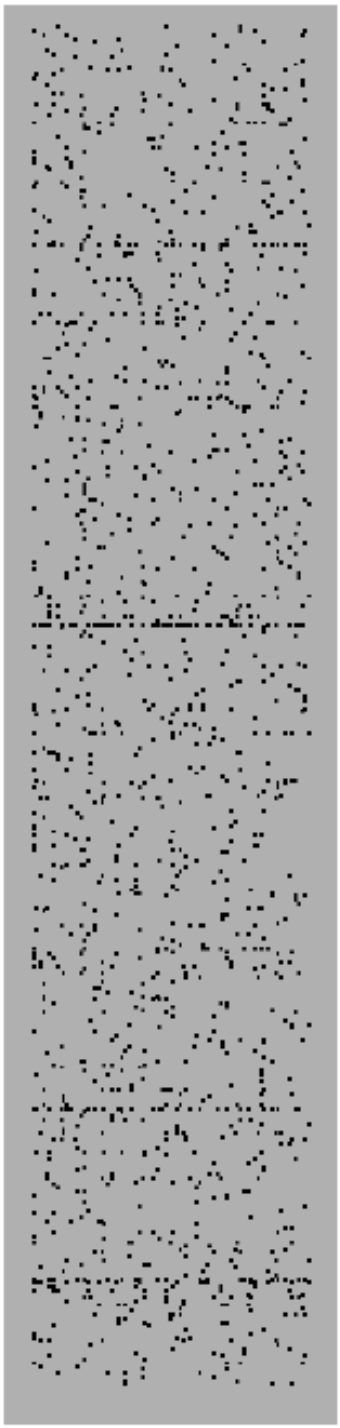

Figure 4.3: Pattern of non-zero entries in matrix ADI. Taken from [Ber92a]

	d1	**d2**	**d3**	**d4**	**d5**	**d6**
ship	1	0	1	0	0	0
boat	0	1	0	0	0	0
ocean	1	1	0	0	0	0
voyage	1	0	0	1	1	0
trip	0	0	0	1	0	1

Table 4.1: Terms-Documents matrix taken from [MRS08]

information need given by a user in the form of a query. A more concrete example would be to pick *Twitter* posts related to an specific event, for example "Presidential Elections". Other examples could be *Google* searches, or even looking for a product with an specific description in *Amazon*.

The first thing that we need to do is to codify our documents. To do so, we can use the *Vector Space Model* that we presented in section 4.1.1. Each document will be represented as a vector of size m which is the number of considered terms. Also, each entry in this vector describes in some way how much a given term is important in that document. For this, we can use the $TF - IDF$ mentioned in section 4.1.2 or simply the term frequency.

Next we need to do the same treatment to the user query and express it as if it was a document vector. Again, this vector, which we call q, will contain m entries each one representing a term.

Once we have our matrix that represents the corpus and our vector that represents the user's query, we can proceed to manipulate and simplify the matrix through the use of SVD and Latent Semantic Indexing. For this, we first need to calculate the SVD of the $m \times n$ matrix. Let it be A. The result will be $A = U\Sigma V^T$. After that, we convert the matrix Σ of rank r to a matrix of rank k by zero-ing out the smallest singular values. By doing this, we can also zero-out $m - r$ rows of U and $n - r$ rows of V (which are columns of V^T).

After that we can convert the query q to a query q_k by using the transformation eq. (4.13). Finally, we can compare this query with each column in A_k and determine the closest ones. The result that the user will get is the list of documents that correspond to those column vectors.

Now, let us elaborate a simple example to help further clarify our understanding. The example is based on the one presented in [MRS08].We will suppose that we already have our encoded matrix table 4.1 of term-documents and also the user's query.

table 4.1 corresponds to our matrix A:

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \tag{4.14}$$

When we obtain the SVD, we obtain:

$$U = \begin{pmatrix} -0.45 & -0.30 & 0.57 & 0.58 & 0.25 \\ -0.13 & -0.33 & -0.59 & 0 & 0.73 \\ -0.48 & -0.51 & -0.37 & 0 & -0.61 \\ -0.70 & 0.35 & 0.15 & -0.58 & 0.16 \\ -0.26 & 0.65 & -0.41 & 0.58 & -0.09 \end{pmatrix} \tag{4.15}$$

$$\Sigma = \begin{pmatrix} 2.16 & 0 & 0 & 0 & 0 \\ 0 & 1.59 & 0 & 0 & 0 \\ 0 & 0 & 1.28 & 0 & 0 \\ 0 & 0 & 0 & 1.00 & 0 \\ 0 & 0 & 0 & 0 & 0.39 \end{pmatrix} \tag{4.16}$$

$$V^T = \begin{pmatrix} -0.75 & -0.28 & -0.20 & -0.45 & -0.33 & -0.12 \\ -0.29 & -0.53 & -0.19 & 0.63 & 0.22 & 0.41 \\ 0.28 & -0.75 & 0.45 & -0.20 & 0.12 & -0.33 \\ 0 & 0 & 0.58 & 0 & -0.58 & 0.58 \\ -0.53 & 0.29 & 0.63 & 0.19 & 0.41 & -0.22 \end{pmatrix} \tag{4.17}$$

Now, based on theorem 4.1, we truncate the matrix Σ to get a new matrix of rank 2:

$$\Sigma_2 = \begin{pmatrix} 2.16 & 0.00 \\ 0.00 & 1.59 \end{pmatrix} \tag{4.18}$$

And finally, we get A_2, which is the matrix A but expressed in rank $k = 2$.

$$A_2 = \begin{pmatrix} 1.0199 & 0.0060 & 1.0112 & 0.0095 & 0.0069 & -0.0047 \\ 0.0004 & 1.0057 & -0.0046 & 0.0009 & 0.0033 & 0.0052 \\ 1.0062 & 1.0063 & -0.0016 & 0.0052 & 0.0094 & 0.0006 \\ 0.9933 & 0.0025 & -0.0140 & 1.0045 & 1.0064 & -0.0039 \\ -0.0069 & -0.0071 & -0.0059 & 1.0021 & -0.0011 & 1.0084 \end{pmatrix} \tag{4.19}$$

Let us verify that this reduced matrix honors the Eckart-Young-Mirsky theorem:

$$\|A - A_k\|_2 = 1.28 \tag{4.20}$$

Which is the value of σ_3. Now, in order to find the representation of the documents in two dimensions, we do:

$$\Sigma_2 V_2^T = \begin{pmatrix} -1.62 & -0.60 & -0.44 & -0.97 & -0.70 & -0.26 \\ -0.46 & -0.84 & -0.30 & 1.00 & 0.35 & 0.65 \end{pmatrix} \tag{4.21}$$

The document vectors document vectors can be represented as showed in fig. 4.4.

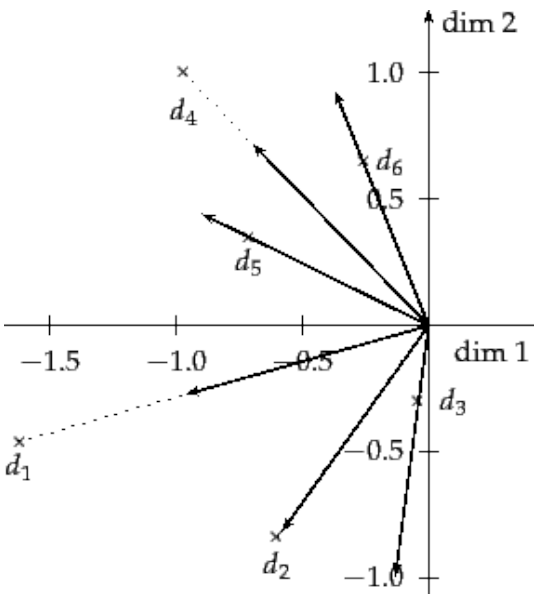

Figure 4.4: Column vectors of matrix A_2

If we suppose that our query is:

$$q = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{4.22}$$

then the query q_k would be:

$$q_k = \begin{pmatrix} -0.06 \\ -0.21 \end{pmatrix} \tag{4.23}$$

And it could be folded-in into the matrix V. We could then compare how close this vector is to the other vectors in fig. 4.4 through the product $V_2\Sigma_2^2V_2^T$ and figure out that document d_2 and document d_1 would be presented to the user as matches of the input query, which is the final objective of this Information Retrieval task.

Chapter 5

Numerical Properties of the SVD Problem

In this chapter we are concerned about the numerical properties of the SVD algorithms in general. What we want to do is to do a quick study of the characteristics that a good algorithm to compute the Singular Value Decomposition should have. We divide this chapter in two sections. The first one provides an overview of the types of errors and algorithms. Next we characterize the SVD problem from the Numerical Analysis point of view.

5.1 Overview of Numerical Properties

Algorithms that solve problems numerically perform calculations that generate local computational errors. The local computational errors are carried out through the steps of the algorithm and generate what is called the accumulated computational error. A corollary of this is that if the number of local operations is very high, then it is possible that the final result is completely false [EMU96]. So it is quite important to recognize and mitigate these errors.

5.1.1 Generation of errors

Local computational errors come from different sources. Examples of sources are:

1. Round-off error. Arises because it is not possible to represent all real numbers exactly with a finite memory. An example is the use of π.

2. Truncation error. occurs when an iterative method is terminated or a mathematical procedure is approximated and the obtained solution differs from the exact solution.

3. Discretization error. The solution of a discrete problem does not correspond to the solution of the continuos problem.

5.1.2 Well-conditioned problem vs Ill-conditioned problem

Depending on the characteristics of a problem, they can be classified in two groups:

1. Well-conditioned problem. In this group are the problems whose solution changes by only a small amount if the problem data are changed by a small amount.

2. Ill-conditioned problem. This is the opposite case, were any small error in the data will grow to be a large error in the final result.

5.1.3 Numerical Stability

The classification in Well or Ill-conditioned problems depends on the characteristics of the problem itself. However, a given problem can be solved by several algorithms. These algorithms can evidently vary in precision or accuracy. So, a further classification used in numerical algorithms is given by what is called *numerical stability*.

Informally, algorithms that avoid numerical errors presented before to a large degree are called numerically stable[EMU96]. Another way of saying this is that an algorithm is called numerically stable if an error, whatever its cause, does not grow to be much larger during the calculation. Numerical stability is a desirable property of numerical algorithms and its precise definition depends on the context, e.g. numerical linear algebra, differential equations, etc.

One of the arts of numerical analysis is to find a stable algorithm for solving a well-conditioned mathematical problem.

Let us mention two possible definitions of what a numerically stable algorithm is. This will allow us to understand that numerical stability can be understood in several senses. For these definitions let us distinguish between a theoretical algorithm f which maps each input x, to a output y. That is, $f(x) = y$. For the theoretical algorithm f, there is also a numerical realization called f^*. Also, we need to define another input x^*, which is close to x.

The following definition is given by Stewart [Ste73], who is based on Kahan:

Definition 5.1.1. Numerical Stability. A numerical algorithm f^* is numerically stable if for each input x, there is a nearby input x^* such that $f(x^*)$ is close to $f^*(x)$.

The effect of these type of algorithms on well-conditioned problems is that the output of the numerical algorithm approximates the exact solution of the problem. On the other hand, for ill-conditioned problems the effect is that both $f(x^*)$ and $f^*(x^*)$ differ from the actual solution $f(x)$.

Another definition given by Wilkinson [Wil60], is the following:

Definition 5.1.2. Numerical Stability. A numerical algorithm is stable if there is a x^* near x, such that $f^*(x^*) = f(x)$.

As we will see in a moment, this second definition originates the concept of backward error.

For yet another definition of numerical stability see [BHSS65].

5.1.4 Types of Algorithms

We can distinguish yet two types of errors found in numerical algorithms:

1. Forward error. It is the difference between the result given by the numerical algorithm and the actual solution. If we call $y = f(x)$ and $y^* = f^*(x)$, then the forward error is $\Delta y = y^* - y$.

2. Backward error. It is the smallest Δx such that $f(x + \Delta x) = f^*(x)$.

Based on these errors, algorithms can be further classified in three groups:

1. Backward stable. An algorithm is said to be backward stable if the backward error Δx is small for all possible inputs x. Usually we want this error to be the same as the unit round-off.

2. Mixed stable. An algorithm is said to be mixed stable if it solves a nearby problem approximately, i.e. if there exist a Δx such that both Δx is small and $f(x + \Delta x) - f^*(x)$ is small. Note that Stewarts definition corresponds to this type of algorithms.

3. Forward stable. An algorithm is said to be forward stable if its forward error Δy divided by the condition number κ of the problem is small. In other words, if it has a forward error of magnitude similar to some backward stable algorithm.

5.2 Singular Value Decomposition

Errors concerning SVD come from two sources [Ste90]:

1. Rounding-errors during the calculation. These are of concern from the algorithmic perspective.

2. Errors initially present in the matrix. These are considered as perturbation or noise.

Note that the effect of (1) is as if the original matrix had been perturbed, hence becoming a particular case of (2). So we need to analyze how perturbations in the original matrix affect the calculation of the Singular Value Decomposition. Particular effects of (1) will be briefly mentioned in a later chapter, and its analysis will be complemented in later stages of this project. In this section we are concerned on the actual effects of (2) which can be large compared to the rounding errors.

We start our study with the case $\tilde{A} = A + E$, which is the matrix A plus a perturbation E. Let also be $U^T A V = \Sigma$ the exact decomposition and $\tilde{U}^T \tilde{A} \tilde{V} = \tilde{\Sigma}$ be the singular decomposition of $\tilde{A}$, which is an approximation of A. The question is how far Σ and $\tilde{\Sigma}$ are.

There are two ways of analyzing this:

1. Perturbation bound analysis, which gives a upper bound on the difference between the perturbed quantity and the actual quantity, that is, between $\tilde{\Sigma}$ and Σ

2. Perturbation expansion analysis, which tries to approximate the perturbed quantity as a function of the perturbation, i.e. find a function f, such that $f(E) = \tilde{\Sigma}$.

5.2.1 Perturbation bounds for Singular Values

The perturbation bounds are due to [Wey12] and [Mir60]

Theorem 5.1. *Weyl's Theorem*

$$|\tilde{\sigma}_i - \sigma_i| \leq \|E\|_2, i = 1, \ldots, n \tag{5.1}$$

Theorem 5.2. *Mirsky's Theorem*

$$\sqrt{\sum_i (\tilde{\sigma}_i - \sigma_i)^2} \leq \|E\|_F \tag{5.2}$$

The important thing about these two theorems are that the theorems are true for any E, there is no restriction.

In particular, Weyl's theorem means that the singular values of a matrix are perfectly conditioned because no singular value can move more than the norm of the perturbations.

5.2.2 Perturbation Expansion

Stewart establishes the perturbation problem and the solution in the following way:

Let $\sigma \neq 0$ be a simple singular[1] value of A with u and v as left and right singular vectors, respectively. Then, as E approaches zero, there is a unique singular value $\tilde{\sigma}$ of $\tilde{A}$ such that:

$$\tilde{\sigma} = \sigma + u^T E v + O(\|E\|^2) \tag{5.3}$$

He doesn't provide a proof though.

5.2.3 Perturbation of the Singular Subspaces

Some matrices are very sensible in the sense that small perturbations completely change the singular vectors [Ste90]. As pointed out by Stewart, although this fact might seem problematic, there are two reasons that lessen this disruption:

1. In most applications, singular vectors are used only as a change of basis where matrix A assumes a diagonal form.

2. If we use an stable algorithm and G is a perturbation in the order of rounding the matrix A, then the computed SVD satisfy:

$$A + G = U\Sigma V^T \tag{5.4}$$

Now let us talk about the Wedin's theorem which provides a perturbation bound for the singular subspaces and which is based on an measure of the angle between the real subspaces and the computed subspaces.

The analysis starts with the following decomposition[2]:

[1] Simple singular in this context means that there are no repeated singular values

[2] Note that though this analysis is done for matrices that have more rows than columns, it also applies to the case where there are more rows than columns. This is an important distinction since modern IR applications deal with a huge amount of documents as in [RR11]

$$(U_1U_2U_3)^T A(V_1V2) = \begin{pmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \\ 0 & 0 \end{pmatrix} \tag{5.5}$$

and we want to bound U_1 and V_1 [3], and the perturbed subspaces are:

$$(\tilde{U_1}\tilde{U_2}\tilde{U_3})^T \tilde{A}(\tilde{V_1}\tilde{V}2) = \begin{pmatrix} \tilde{\Sigma}_1 & 0 \\ 0 & \tilde{\Sigma}_2 \\ 0 & 0 \end{pmatrix} \tag{5.6}$$

Let's assume that Φ and Θ are the matrix of canonical angles between U_1 and $\tilde{U_1}$, and V_1 and $\tilde{V_1}$, respectively. The problem is to find a bound in terms of these two matrices.

Wedin's theorem says that:

Theorem 5.3. *Wedin's theorem. If there is a $\delta > 0$ such that*

$$\min |\sigma(\tilde{\Sigma}_1) - \sigma(\tilde{\Sigma_2})| \geq \delta \tag{5.7}$$

and

$$\min \sigma(\tilde{\Sigma}_1) \geq \delta \tag{5.8}$$

then

$$\sqrt{\|\sin\Phi\|_F^2 + \|\sin\Theta\|_F^2} \leq \frac{\sqrt{\|R\|_F^2 + \|S\|_F^2}}{\delta} \tag{5.9}$$

where:

$$R = A\tilde{V_1} - \tilde{U_1}\tilde{\Sigma}_1 \tag{5.10}$$

$$S = A^T\tilde{U_1} - \tilde{V_1}\tilde{\Sigma}_1 \tag{5.11}$$

are called the residuals.

We can notice that this bound combines both matrices of canonical angles with both (left and right) residuals.

An important fact is that the bound of this theorem is better than if we had used E instead of the residuals R and S.

[3] Wedin's theorem provides a single bound for both subspaces

Chapter 6

Lanczos SVD Algorithm

Although quite useful for understanding the SVD factorization, the implicit algorithms mentioned in the chapter 3 can not be used directly to calculate SVD in practice. This is in part due the intrinsic errors associated with using a finite-precision device like a computer (while the theorems we used to prove the existence of SVD, assumed infinite precision). There is a whole area of Mathematics, called Numerical Analysis, which targets the proper translation of theoretical algorithms into numerical ones; which can produce accurate results on a computer. A quite notable subarea, Numerical Linear Algebra, has received special attention over decades of research. The chapter 5 mentions some of the main aspects to consider, with this regard.

Therefore, in order to calculate SVD in practice, more specialized theorems that consider the limitations of computers are required; such theorems allow one to create particular algorithms, which possess the desired qualities: not only accuracy but high performance as well. This is specially true in today's world, where the scale of the matrices to analyze by far exceeds those used in the past. The particular case of Latent Semantic Indexing (LSI), the application we chose to restrict our study of SVD, illustrates very well this trend in the change of scale: original papers about LSI used a few thousands of documents, while today applications can easily reach millions, hundreds of millions or even more (like the Oracle product mentioned on chapter 1, which considers as documents "posts" in social networks).

In this chapter we will describe the serial algorithm that is most commonly used, for solving the SVD factorization that comes from the LSI problem. A particular characteristic of this algorithm, at least in the form

presented here, is that it is implicitly assumed that the matrix and auxiliary data fit in the computer memory; therefore, it will have certain limitations in terms the size of the matrix to factorize (at least in commodity hardware, which is our focus). The chapter 7 describes another distributed algorithm, which does allow to spread the matrix in a cluster of commodity computers.

Although the algorithm presented in this chapter is essentially serial, some of the routines it uses can accept parallelism and take advantage of either multi-core computers with shared memory; or vectorial processors. We mention those details in the last section as well.

6.1 SVD as an eigenproblem

Aiming to calculate numerically the SVD factorizations, made researchers reformulate that problem as the quite related eigendecomposition (or eigenproblem). Such problem consists in finding, for an square matrix A, the eigenvalues and eigenvectors. If we arrange the eigenvectors in an orthogonal matrix Q and the eigenvalues in a diagonal matrix Λ, the eigenproblem can be restated as the following factorization:

$$A = Q\Lambda\, Q^T$$

In order to see the connection between the SVD and the eigenproblem, we need to recall the gramian matrix $A^T\, A$ from the theory chapter. It was the gramian, which provided the matrix V on the first place; because the vectors $\mathbf{v}$ were its eigenvectors (see chapter 3). Finding the matrix V then, can be thought as the eigenproblem for matrix $A^T\, A$; which can be stated as finding its diagonal factorization $\ni$:

$$A^T\, A = V\Sigma^2\, V^T \tag{6.1}$$

But the same is true for matrix U, if we now consider the matrix $A\, A^T$, which can be diagonalized if one finds its eigenvectors and place them into the matrix U (the eigenvalues are the same as the gramian):

$$A\, A^T\ = U\Sigma^2\, U^T \tag{6.2}$$

It is the second eigenvalue problem equivalence, that is used for the distributed algorithm of chapter 7. Per the SVD factorization $A = U\Sigma V^T$, if we have the original matrix A, plus the diagonal Σ and the matrix U; we can reconstruct the matrix V (if required):

$$V^T = S^{-1} U^T A = PA$$

The matrix $P = S^{-1} U^T$ is called the projection matrix, and is used in LSI for "folding-in" new document vectors $\mathbf{x}$, by calculating $P\mathbf{x}$; that is, the matrix P is used as a predictive (rather than descriptive) model, to predict where the position of document $\mathbf{x}$ will be in the latent space.

For this chapter though, we could use either eigenproblem from eq. (6.1) or eq. (6.2). Actually, the literature originally reported the former, perhaps due the early shape of the matrices used for LSI (more terms than documents). Today's LSI applications have much more documents than terms, but still these early algorithms are useful, as we will see in chapter 7 (where the original matrix is split into several submatrices, which do have the shape expected by the Lanczos algorithm that we document here).

6.2 Derivation of the serial algorithm

The algorithm to numerically solve the SVD problem, that we chose for this report, is essentially the one published by Berry on his PhD thesis ([Ber91a]). Despite of being more than 20 years old, we can tell that it is still widely used, in particular by LSI software. A big part of the opensource LSI implementations that we found (see chapter 1), refer to either the Fortran77 SVDPACK ([Ber92c]), to its C incarnation SVDPACKC ([BDO+93]), or to its even newer skin SVDLIBC ([Roh07]). All of them are essentially the same algorithm that Berry published in his PhD thesis.

We will proceed to derive the algorithm in the next sub-sections. A cautionary warning about the level of detail presented is appropriate: although we would like to offer the same level of detail and technicality than the theory chapter 3, time constraints for the delivery of this report forced us to omit the theorems and its proofs. A pending task, for this project to evolve into a full Msc. thesis, would be to achieve the same level of formality than chapter 3, indeed. Such exercise is actually required, if one pretends to offer an innovation to the problem of efficiently compute the SVD for LSI

problem. Let us consider it a pending task then, for the time being.

Since we established the equivalence of the SVD problem, to that of the eigenproblem for gramian matrix $A^T A$, is important that we keep in mind such alternate formulation the next sections to come. From now on, to the end of this section, our goal will be to find the eigenvalues and eigenvectors of a symmetric matrix; assuming that the calculations are to be performed on a computer with finite precision. We will start with simple methods, and evolve them until we reach the level of sophistication that we require for a useful SVD algorithm.

6.2.1 The Power Method

According to Golub [GVdV00], quoting Householder, the power method has its origin in the work of Müntz in 1913 [Mun13]. The method is the simplest algorithm for solving the eigenproblem; it basically consists in picking carefully a vector, and then apply the matrix A iteratively until it converges to an eigenvector. And not to any eigenvector, but precisely to the one with largest eigenvalue (in absolute value). The following pseudocode is taken from Golub [GVL12]:

Algorithm 1: The Power Method

Input : A unit vector $\mathbf{q_0} \in \mathbb{R}^n$ and a symmetric matrix $A^{n \times n}$

Output: The tuple $(\lambda_k, \mathbf{q_k})$ which is expected to approximate an eigenpair $(\lambda, \mathbf{q})$ of A

for $k = 1, 2, \ldots$ **do**
- $\mathbf{z_k} \leftarrow A\mathbf{q_{k-1}}$
- $\mathbf{q_k} \leftarrow \dfrac{\mathbf{z_k}}{\|\mathbf{z_k}\|_2}$
- $\lambda_k \leftarrow \mathbf{q_k}^T A\mathbf{q_k}$

return $(\lambda_k, \mathbf{q_k})$

One immediate trick that is detected, is that we are not giving a precise value for the number of iterations; this is because we do not really know

how many in advance, though we know how "fast" we can reach convergence, more about this in a minute. Though the algorithm looks trivial, a powerful theorem justifies why it works. Golub mentions the conditions which are required for its convergence: the maximum eigenvalue of A must be unique (no repetition), and the initial vector q_0 must not "deficient" (its component on the direction of the eigenvector with maximum value must not be zero). A proof of convergence/correctness can be consulted in [GVL12].

Golub also mentions the computable error bounds of this method. The real eigenvalue and eigenvectors of A will satisfy the equation below:

$$A\mathbf{q} = \lambda\mathbf{q}$$

But accepting the fact that the algorithm 1 will not produce exactly the eigenvalue nor the eigenvector, we can at least see how close we are in meeting above condition. That is, we can calculate the residual δ:

$$\|A\mathbf{q_k} - \lambda_k\mathbf{q_k}\|_2 = \delta$$

Golub shows that there is an eigenvalue λ that satisfies $|\lambda_k - \lambda| \leq \sqrt{2}\delta$; which is a way to tell that we can really approximate an actual eigenvalue, as long as we are capable of reproducing its defining property with good accuracy (which in turn, will depend on how many iterations we make).

Alright, so we know how to calculate one eigenpair; why not calculating them all? We may be tempted now to recall the geometric proof of SVD (see chapter 3), and consider the following procedure for finding all the eigenpairs (assuming preconditions met):

1. Pick carefully initial vector.
2. Apply algorithm 1 to find the first eigenpair.
3. Obtain the hyperplane that is orthogonal to the first eigenvector found. From this hyperplan, take a new vector and repeat recursively the procedure until we have all the desired eigenpairs[1].

[1]The third step is usually called "deflation" (see [GVL12]), when mentioned in the context of the matrix, as it is reduced to dimensions $(n-1) \times (n-1)$.

The problem with this procedure, also exposed by Golub in his proof of correctness, is that the rate of convergence depends on $\left|\frac{\lambda_2}{\lambda_1}\right|^k$; where λ_2 is the second largest eigenvalue in absolute value. Thus, unless there is a considerable gap between first and second largest eigenvalues of A, the Power Method will converge quite slowly. That makes it unsuitable for practical purposes, at least in the standalone version we just presented. Further sections will show how it can evolve to overcome this limitation.

6.2.2 The Rayleigh-Ritz Method

The method presented in this subsection is not really an step forward from the Power Method, but rather a parallel development (both will be merged in the Lanczos Algorithm of further subsections). It is actually an auxiliary tool that many eigenproblem solvers need; not necessarily for symmetric matrices (although we still assume that, in order to maintain our desired scope).

Suppose that, in order to find the eigenpairs [2] of a given matrix A, we generate a sequence of orthogonal matrices W_k which contain progressively better approximations of the eigenspace [3]. A common problem for any procedure that goes that way, is how to "extract" the actual eigenvectors from such subspace (the eigenvalues are the same, so those do not require further calculations). The Rayleigh-Ritz [4] method addresses precisely this common need.

Before providing the pseudocode, let us explain a bit better what we mean by having "calculated subspaces" W_k; as that is a rather vague expression (though is quite common in the literature). What we really mean, is that we have a *characterization* of the subspace; which is nothing more than a basis for it. The vectors of such basis are arranged as columns of the matrix W_k, and then, we are basically asking for the eigen decomposition of that matrix.

Does not the above sound a bit circular? We start with the generic problem of finding eigenvalues and eigenvectors of symmetric matrix A; then we calculate through an iterative process another matrix W_k, which contains

[2] An eigenpair is the tuple $(\lambda_i, \mathbf{v})$ of an eigenvalue and its corresponding eigenvector

[3] Subspace generated by the eigenvectors.

[4] Leissa argues that the method should not really be attributed to Rayleigh but only to Ritz, (see [Lei05]).

the basis of a subspace that we know has good approximations to the eigenvectors of our original matrix A. Then, we proceed to solve the eigenproblem for that new matrix W_k ... looks like we finish right where we began! Of course that, though not mentioned always in literature, the intuitive idea is that the new matrix W_k is less generic than A. It is expected to have certain qualities that make the solution of its eigenproblem an easier task (compared to solving that of the original matrix A).

Having clarified a bit the main idea of subspace eigenproblem solvers, let us continue to list the pseudocode for the Rayleigh-Ritz Method; which offers a way to "extract" the eigenvectors of original matrix A, out of the approximation matrix W_k. We based our procedure in [JS01]:

Algorithm 2: The Rayleigh-Ritz Method

Input : Approximation subspace orthogonal matrix W_k, symmetric matrix A

Output: Set of desired (approximated) eigenpairs

1 $B \leftarrow W^T AW$

2 **for** *each desired eigenpair* $(\lambda_i, \mathbf{v_i})$ *of* A **do**

3 Solve eigen equation $B\mathbf{x_i} = \tilde{\lambda_i}\mathbf{x_i}$ (where $\tilde{\lambda_i} \simeq \lambda_i$)

4 $(\tilde{\lambda_i}, \tilde{\mathbf{v_i}}) \leftarrow (\lambda_i, W\mathbf{x_i})$, where $\tilde{\mathbf{v_i}} \approx \mathbf{v_i}$

5 return $\{(\tilde{\lambda_1}, \tilde{\mathbf{v_1}}), (\tilde{\lambda_2}, \tilde{\mathbf{v_2}}), \cdots, \}$

If W_k was the orthogonal matrix with the eigenvectors of A, then matrix B would be diagonal (containing the eigenvalues). As W_k is rather an approximation to such matrix, is usually the case that is something close to a diagonal (like a tridiagonal or bidiagonal); from there comes the fact that calculating its eigenpairs, is much easier than for original matrix A.

Probably the less intuitive step from the algorithm is the assignment $\tilde{\mathbf{v_i}} = W\mathbf{v_i}$; but is not hard to prove its validity:

$$B\mathbf{x_i} = \tilde{\lambda}_i \mathbf{x_i}$$

$$\iff \quad (W^T AW)\mathbf{x_i} = \tilde{\lambda}_i \mathbf{x_i}$$

$$\iff \quad W^T A(W\mathbf{x_i}) = \tilde{\lambda}_i \mathbf{x_i}$$

$$\iff \quad A(W\mathbf{x_i}) = \tilde{\lambda}_i (W\mathbf{x_i})$$

$$\therefore \quad W\mathbf{x_i} \text{ is an (approx.) eigenvector of } A \quad \triangle$$

The vector $\mathbf{v_i}$ is called a Ritz vector, and we will refer to it in such a way when we review the complete Lanczos algorithm.

For further details of convergence or error analysis, please refer to Jian [JS01].

6.2.3 The Lanczos Tridiagonalization Step

Golub explains in [GVL12] that one of the problems with the Power Method, is that it does not take advantage of the previously calculated information. During the iterations of the Power Method, say until step k, we have calculated already the set of vectors $K(A, q_0, k) = \{A\mathbf{q_0}, A\mathbf{q_1}, \ldots, A\mathbf{q_k}\}$; still, they are not used at all when looking for an estimate of the eigenvector. Such limitation is addressed by the Lanczos Algorithm, named after its creator in 1950 ([Lan50b]). The subspace spanned by the set $K(A, q_0, k)$ is called Krylov Subspace of order k [5], which is why the Lanczos Algorithm is usually cataloged as a Krylov Subspace method.

Going back to Lanczos, this subsection will only explain the iterative step (called Lanczos Tridiagonalization Step). It works as follows; let us suppose that we have an square symmetric matrix $A^{n\times n}$, and that we want a few of its biggest eigenvalues (as it is the case in LSI applications) [6]. Each

[5] The concept itself of Krylov Spaces is thanks to Krylov and dates back to 1931 (see [Kry31])

[6] The Lanczos Process can also calculate a few of the smallest eigenvalues, but we are not interested in such case for LSI applications.

execution of the step algorithm generates a tridiagonal matrix $T_k \in \mathbb{R}^{k\times k}$ [7], and the whole sequence $\{T_k\}$ is progressively approximating the biggest eigenvalues of the original matrix A.

There are several ways of stating the algorithm for the Lanczos Tridiagonalization Step, the following is taken from Golub [GVL12]; though it is not the most numerically stable. That honor corresponds to the ones created by Paige ([Pai71],[Pai76]); we preferred Golub's one for our exposition, aiming to have an easier introduction to the procedure:

Algorithm 3: The Lanczos Tridiagonalization Step

Input : A unit vector $\mathbf{q_1} \in \mathbb{R}^n$ and a symmetric matrix $A^{n\times n}$

Output: The sequences $\{\alpha_i\}$, $\{\beta_i\}$ and matrix $Q = [\mathbf{q_1}|\mathbf{q_2}|\cdots]$, where $1 \leq k \leq n$

$k \leftarrow 0, \beta_0 \leftarrow 1, \mathbf{q_0} \leftarrow 0, r_0 \leftarrow \mathbf{q_1}$

while $k = 0 \vee \beta_k \neq 0$ **do**

  $\mathbf{q_{k+1}} \leftarrow \frac{\mathbf{r_k}}{\beta_k}$

  $k \leftarrow k + 1$

  $\alpha_k \leftarrow \mathbf{q_k}^T A\mathbf{q_k}$

  $\mathbf{r_k} \leftarrow A\mathbf{q_k} - \alpha_k\mathbf{q_k} - \beta_{k-1}\mathbf{q_{k-1}}$

  $\beta_k \leftarrow \|\mathbf{r_k}\|_2$

return $(\{\alpha_i\}, \{\beta_i\}, Q = [\mathbf{q_1}|\mathbf{q_2}|\cdots])$

The algorithm 3 is essentially applying Gram-Schmidt process, but only against the last two vectors. Golub derives the algorithm from a relation between tridiagonalization, and the QR factorization of the matrix formed by vectors $K(A, q_0, k)$; see [GVL12] for further details.

Golub goes even further in the cited book, and proves the following properties about algorithm 3. We will omit the theorem statement, and just comment directly its results:

[7] Matrices with the middle, upper and lower diagonals.

- The algorithm runs until $k = m = rank(K(A, q_0, k))$. This contrasts with the unknown number of steps of the Power Method (algorithm 1).

- For $k = 1 : m$ we have $AQ_k = Q_k T_k + \mathbf{r_k}\, {e_k}^T$, where $Q = [\mathbf{q_1}|\cdots|\mathbf{q_k}]$ has orthonormal columns that span the Krylov subspace $K(A, \mathbf{q_1}, k)$, and $e_k = I_n(:, k)$ (the k column of the identity matrix). This justifies the orthogonalization step of the algorithm (line 6), which only considers the last two vectors; whether that is enough to guarantee that all the $\mathbf{q}$'s will be orthogonal is certainly not evident, and gets proved on the same theorem.

- The matrix T_k has tridiagonal shape, that is:

$$\begin{bmatrix} \alpha_1 & \beta_1 & \cdots & 0 \\ \beta_1 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \beta_{k-1} \\ 0 & \cdots & \beta_{k-1} & \alpha_k \end{bmatrix}$$

  This shape allows us to calculate its eigenvalues with much less effort than for original matrix A (which was the whole motivation on the beginning). There are several options for such calculation, but we will consider only the (implicit) QL Algorithm (see [DMW71a]), as that is the one used by Berry for his famous routines in the context of LSI (see further subsections).

In addition to the above properties, which kind of guarantee the "correctness" of the algorithm 3 (to some extent); Golub also cites in [GVL12] another theorem that establishes the approximation quality of matrix T_k as a function of k. This is the result that justifies our original claim that the sequence of matrices $\{T_k\}$, approximates better the eigenvalues of A.

Finally, Golub adds in [GVL12] that not everything is flakes and honey with this algorithm; the orthogonality that we expect on vectors $\mathbf{q}$'s is at jeopardy as $\tilde{\beta_k}$, the numerical approximation of β_k, becomes really small; this is because that implies the cancellation of $\mathbf{r_k}$. Main credit of this result

goes again to Paige ([Pai71],[Pai76]), and we will come back to it on next subsection, when we show the full Lanczos Algorithm.

6.2.4 The Single-Vector Lanczos Algorithm

We are armed now with all the required tools to present the main algorithm that is used for SVD, in the context of LSI (at least in the serial form). As mentioned on the introductory section of this chapter, the algorithm exists thanks to Berry ([Ber92a],[BDO95]); we should probably present our respects to Berry in this moment, as he worked for more than a decade around the particular problem of solving efficiently the SVD/LSI problem (and in essence, today's applications still use his contributions).

The algorithm 4 we present below is not the final one, as there are more practical considerations to cover at the end of this subsection; but it is easier to present this simplified version, as it has all the main ingredients. We can see in particular, how it combines Lanczos Tridiagonalization Step (algorithm 3) (which implicitly uses the Power Method algorithm 1), with the Rayleigh-Ritz Method (algorithm 2).

We also take the opportunity to come back to our original context, where A is a large and sparse matrix coming from an LSI problem. The pseudocode is based on [Ber92a], filling additional details from the C code of the implemented routine (LAS2) [8] [9] .

Although complete in appearance, algorithm 4 still has a serious numerical issue: the potential loss of orthogonality in the vectors of matrix Q_c (mentioned at the end of last subsection). To solve that problem, we could reorthogonalize all vectors at every execution of algorithm 3; but that would be kind of brute force, and eliminate the advantages of the whole proposal. A clever approach, selective reorthogonalization, was selected by Berry in order to complete his master-piece: the LASVD/LAS2 routine [10].

[8] Originally the routine was coded in Fortran77, but we found more comfortable to check the C port instead; both made by Berry, by the way.

[9] The eigenvectors of matrix $A^T A$ are called right singular vectors of A, while those of $A A^T$ are the left singular vectors.

[10] Berry actually proposed four different methods of calculating SVD, for the LSI problem; LAS2 (descendant of LASVD) routine is just one of them. But it seems the fastest, and it was the only one ported to the new skin of Berry's SVDPACKC, which is SVDLIBC [Roh07]. Interestingly though, Berry mentions in [Ber91a] that LASVD is suitable only for low to medium precision in the singular values. A pending task then, is to confirm

Algorithm 4: The Single-Vector Lanczos Algorithm

Input : A matrix $A^{m \times n}$ and a truncation factor k

Output: The k singular values and its associated right singular vectors of A (which are the first k eigenpairs of symmetric matrix $A^T A$). Both are numeric approximations.

1 Use Lanczos Tridiagonalization step algorithm 3 to generate a family of symmetric tridiagonal matrices, $\{T_j\}(j = 1, 2, \ldots, c) \ni c > k$. Note that these matrices approximate the eigenvalues of symmetric matrix $A^T A$ (which happen to be the singular values of A).

2 Compute the eigenvalues and eigenvectors of T_k using the Implicit QL Method.

3 For each computed eigenvalue λ_i of T_k (hence of gramian matrix $A^T A$), calculate the associated unit eigenvector $\mathbf{z_i}$ such that $T_k \mathbf{z_i} = \lambda_i \mathbf{z_i}$.

4 For each calculated eigenvector $\mathbf{z_i}$ of T_k, compute the Ritz vector $\mathbf{v_i} = Q_c \mathbf{z_i}$ as an approximation to the i-th eigenvector of $A^T A$ (hence, to a right singular vector of A). Note that the matrix Q_c is a side product of the first step.

5 return $(\{\lambda_1, \lambda_2, \cdots, \lambda_k\}, \{\mathbf{v_1}, \mathbf{v_2}, \cdots, \mathbf{v_k}\})$

The selective reorthogonalization approach, as explained by Golub in [GVL12], is inspired on the error analysis made by Paige [Pai71]. Paige shows that the most recently computed vector $\mathbf{q_{k+1}}$, tends to have a non trivial and unwanted component in the direction of the already converged Ritz vectors [11]; this will make us lose the orthogonality. Therefore, we do not need to re-orthogonalize against all the previously calculated vectors, rather use only the already converged ones.

Such adjustment is done during the Lanczos step (algorithm 3), using a criteria devised by Parlett et al [PS79], which allows one to know when a Ritz vector has converged.

Berry does not include a final pseudocode of his LASVD routine (inspired on the LANSOS routine from Parlett, Simon et al). The routine eventually got renamed as LAS2 and made its way into the famous SVDPACK (Fortran77) and SVDPACKC libraries; and more recently in the modern version called SVDLIBC. It is the latest, which is currently used by several LSI applications.

6.3 Profiling and Parallelization

Berry does some interesting profiling about the algorithm 4, in his PhD thesis [Ber91a]. He was specially interested in parallelizing such algorithm, along with other three methods he proposed. The numbers he reported used a term-document matrix of 5831×1033; he tested in the medium size Alliant FX/80 computer (with 8 processors), as well as the supercomputer Cray-2S/4-128 (with 4 processors). The Cray computer was able to deliver, in theory, 1.9 Gigaflops; as opposed to the 200 megaflops of the Alliant computer. The wall times he reports may no longer be relevant for today's LSI applications, as the data and the computers have changed much in the last 3 decades. But the profiling he did is still relevant, and actually we could not find a more up to date experiment (it would be an interesting exercise to do one).

---

that modern incarnations still have such limitation.

[11] Recall that the Ritz vectors approximate the eigenvectors of the gramian matrix of A, hence the singular vectors of A

6.3.1 Linear Algebra Kernels: BLAS and LAPACK

The algorithm 4 was implemented in the tradition of Linear Numerical Algebra; one never reinvents the wheel but reuses existing standard libraries (called *kernel* routines). This is specially important to avoid introducing numerical errors; as it would be quite impossible that all the people knew the specialized details which are required to produce high-quality routines. A bonus that scientific programmers get by using these standards, is the potentially parallel implementation of the kernels (routines) being used. Today's standard are:

- BLAS (Basic Linear Algebra Subroutines) [LHKK79]: which originated in the Fortran77 world, but now have bindings to many modern languages. They are classified in three levels: level 1 for vector-vector operations, level 2 for matrix-vector operations and level 3 for matrix-matrix operations. These routines are highly specialized for particular processors/architectures, taking advantage of the memory hierarchy, multi-cores, vectorial capabilities of processors[12], custom assembler instructions, etc.

- LAPACK (Linear Algebra Package): is the modern incarnation of the old libraries Linpack [DBMS79] and Eispack [SBG$^+$13], which implemented several numerical algorithms of Linear Algebra in general, and in particular for solving the eigenproblem. LAPACK's original goal was to make efficient implementation of those libraries, by having specialized and highly optimized code for specific architectures. It is built on the lower level BLAS library, but it also has its own optimizations for many hardware vendors.

By the time Berry wrote his PhD thesis, LAPACK was not yet the standard, so he used Eispack instead; BLAS was already available. He used the optimized implementations of these libraries for the two computers described above. The original implementation of Berry used the routines mentioned in table 6.1 (the list is not exhaustive, but includes the most relevant ones, performance-wise).

[12]The ability to execute the same basic operation against several data; known examples out of the super-computers world are the Intel SSE features (see [Kli01]).

Table 6.1: Original BLAS and EISPACK routines used by algorithm 4

Routine	Library	Description
SPMXV	BLAS level 2	Sparse matrix-vector multiplication
IMTQL2 / TRED2	EISPACK	Implement the (implicit) QL Algorithm.
DAXPY	BLAS level 1	$\mathbf{x} \leftarrow \gamma\mathbf{x} + \mathbf{y}$
DAXPY	BLAS level 1	$\mathbf{x} \leftarrow \mathbf{y}$
DDOT	BLAS level 1	$\mathbf{x} \cdot \mathbf{y}$

6.3.2 The two hot spots: SPMXV and IMTQL2

The table 6.2 shows the results obtained by Berry; he measured the speedup of the subroutines when incrementing the number of processors from 1 to 8 on the Alliant FX/80 computer (unfortunately he did not include speedups details for the Cray-2S/4-128). In addition, he includes the results of his profiling, by showing the percentage of the total time that each routine consumed.

Table 6.2: Original profiling and speedups for algorithm 4

	Alliant FX/80		Cray-2S/4-128	
Routine	Speedup	%CPU Time	Speedup	%CPU Time
SPMXV	3	27%	-	72%
IMTQL2	4.3	14%	-	12%
DAXPY	5	17%	-	-
DCOPY	3.6	20%	-	-
DDOT	7.7	2%	-	-

The above numbers quickly tell us that the routine SPMXV is the main bottleneck, and the one which would benefit more from the optimizations in BLAS libraries. This multiplication comes from the Lanczos Tridiagonalization Step (algorithm 3), while calculating the product of the input matrix $A^T A$ by the vectors $\mathbf{q_k}$. The fact that such matrix is never referred in a matrix-matrix operation, but only matrix-vector ones, is the main reason for claiming that Berry's algorithm is suitable for sparse matrices. Internally,

the routine SPMXV may exploit the format of the sparse gramian matrix in order to perform wise optimizations.

The BLAS level 1 routines seem to have a quite different performance across the two tested computers, and Berry mentions in [Ber91a] that it was due to a synchronization required on the Alliant computer. Still, all these routines have great speedups with several processors; which is not surprising given their SIMD [13] nature. Together with the SPMXV routine, the BLAS level 1 and level 2 kernels are likely to represent beyond 50% of the total time. Simply installing an optimized BLAS library should suffice to give our algorithm a good parallel boost.

The second candidate for enjoying the parallelization is the higher level routine IMTQL2, which calculates the eigenvalues and eigenvectors of the tridiagonal matrix produced by algorithm 3; it uses the Implicit QL Algorithm [DMW71a] for such purpose. Berry claims in [Ber91a] and [Ber92a], that such routine could clearly use parallel techniques; and the speedups reported in table 6.2 seem to confirm such claim indeed. However, newer tests need to be performed with new hardware, new data and new libraries; in order to see if this routine is still worth to be parallel. Expectation is that the matrix-vector and vector-vector operations, still dominate whole performance of the algorithm.

6.3.3 SVDLIBC: a history of lost parallelism

When one reads from Berry's papers about parallel SVD for large sparse matrices, that the algorithm 4 accepts parallelism indeed; one takes for granted that modern incarnations inherited this feature. We will proceed to show that such assumption is incorrect.

The original Fortran77 implementation of Berry was in SVDPACK [Ber92c], which used directly BLAS routine SPMXV, as well as Eispack IMTQL2. This allowed transparent parallelism, as long as the environment had installed the optimized vendor libraries.

But, perhaps motivated by the profiling results we showed in table 6.2, Berry changed the implementation in the C incarnation SVDPACKC [BDO+93]; he stopped using directly the BLAS routine SPMXV, and instead accepted

[13] Single Instruction Multiple Data, a type of parallelism.

it as a user parameter (aiming to achieve higher flexibility, we presume). The other change he did, was to include directly a serial implementation of IMTQL2; this decision was crucial, as it prevented his algorithm 4 from enjoying parallelization in step 2.

Old "Fortranish" conventions are difficult to grasp by new generations of programmers, and this motivated the rewriting of SVDPACKC into a modern skin called SVDLIBC [Roh07]. Although is mostly a change of style, it made another implicit serialization: the matrix-vector routines that were previously accepted as parameters, are now included with a serial implementation. In essence then, users of SVDLIBC are using a serial implementation of Berry's parallel algorithm 4. This seems like an unfortunate accident, as todays computers (even personal ones), usually have multi-core and vectorial capabilities.

Above finding is specially relevant for us, as the SVDLIBC implementation plays the role of the Basecase-SVD function in the distributed SVD algorithm 5 (is used inside SVD-Node function, see also algorithm 6). Its author does not offer profiling reports to see how much time is spent in Basecase-SVD function; but it is suspected to be a significant part. Such function can definitely benefit from using an optimized version of SVDLIBC (which internally takes advantage of BLAS/LAPACK installations). Currently, its author reports that only algorithm 7 takes advantage of vendor BLAS/LAPACK implementations [14]. We added this task to our TODO list, in case this project evolves into our Msc. thesis.

[14] Řehuřek codes the algorithm 7 himself, using NumPy routines [Oli06]; which definitely takes advantage of the optimized BLAS/LAPACK kernels.

Chapter 7

Distributed SVD algorithm

In the previous chapter we discussed the state of the art, regarding the serial version of the SVD algorithm, on the context of the LSI problem; such algorithm was discussed under the assumption that the matrix and auxiliary artifacts fit into the RAM of such computer. We also mentioned that such algorithm, could benefit from parallel or vectorized linear algebra kernels; speciall for its most expensive operations (like the sparse matrix-vector multiplication). In this chapter, we will discuss a chosen distributed version of SVD algorithm; where the calculation is spread across computing nodes in a cluster, aiming mainly to scale in time (due the inherent parallelization).

The most scalable and documented algorithm for SVD-LSI, that we found in literature, was that of Radim Řehuřek ; who published his results into a series of articles ([Ř+10b], [Ř+10a] and [Řeh11]), and culminated the effort with his PhD thesis ([RR11]). All the articles are pretty much contained in Řehuřek 's Phd thesis, then unless stated explicitly, all the references to his work in this chapter will be from that publication. Is is fair to emphasize though, that Řehuřek thesis covers other topics besides those we care about in this project; thus, we focused on his chapter of SVD/LSI only.

Another pertinent clarification about Řehuřek 's work, is that he offers the distributed algorithm mostly as a way of speeding up the SVD calculation (scale in time); and not precisely for tackling bigger problems that simply do not fit in the memory of a single computer (scale in space). On the large scale experiments that he reports, the resulting matrix can pretty much fit into the RAM of a modern personal computer; actually, he uses that

to compare the reduction in time on the serial execution (single machine) vs the distributed execution (cluster).

7.1 The one-pass distributed algorithm

The essence of the distributed strategy is to achieve almost perfect parallelism, by splitting the input matrix into several smaller matrices called *jobs*.

$$A^{m\times n} = \left[A_1^{m\times c_1} \mid A_2^{m\times c_2} \mid \cdots \mid A_k^{m\times c_k}\right] \ \ni \ \sum_{i=1}^{k} c_i = n$$

A subset of these smaller matrices or *jobs* is assigned to each node in the cluster, depending on their capabilities; the objective is to assign matrices that fit into the node's RAM memory. Each node will calculate the SVD factorization of the submatrices assigned, but merging those results into a single SVD approximation that covers all the input data it received. At the end, a global merge step across all the nodes is performed, giving the global SVD approximation for original matrix A. The algorithm 5 describes the overall distributed algorithm:

Algorithm 5: Distributed-SVD: Distributed SVD for LSI (global)

Input : Truncation factor k, queue of jobs $A = [A_1, A_2, \dots]$

Output: Matrices $U^{m\times k}$ and $\Sigma^{k\times k}$, from the SVD decomp. of A

1 **for** ***all*** *(node i in cluster)* **do**

2 $\quad B_i \leftarrow$ subset of the queue of jobs $[A_1, A_2, \dots]$

3 $\quad P_i = (U_i, \Sigma_i) \leftarrow$ SVD-Node(k, B_i)

4 $(U, \Sigma) \leftarrow$ Reduce(Merge-SVD, $[P_1, P_2, \dots]$)

5 return (U, Σ)

The first important detail from the algorithm just shown, is that we are not calculating the matrix V from the SVD factorization, how come! Such detail is explained at the end of the last section. For the moment, let us just

say that such matrix is not required for our purposes.

We can also observe the map-reduce pattern in this algorithm, with the map part being the iteration done over p nodes (in parallel); and the reduce part being the final merge of those partial results. The algorithm 6 describes the part done inside each node.

Algorithm 6: SVD-Node: Distributed SVD for LSI (node)

Input : Truncation factor k, queue of jobs $A_1, A_2, \ldots$

Output: Matrices $U^{m \times k}$ and $\Sigma^{k \times k}$, from the SVD of $[A_1, A_2, \ldots]$

1 $P = (U, \Sigma) \leftarrow 0^{m \times k} 0^{k \times k}$

2 **for** *each job* A_i **do**

3 $\quad P' = (U', \Sigma') \leftarrow$ Basecase-SVD(k, A_i)

4 $\quad P = (U^{m \times k}, \Sigma^{k \times k}) \leftarrow$ Merge-SVD(k, P, P')

5 return (U, Σ)

It is important to realize that the iteration in this algorithm 6 is done serially, but that the procedure Basecase-SVD that resolves the SVD of a matrix that fits in memory (base case), internally may exploit the multicore or vectorial capabilities of the node computer. This procedure serves as a black box SVD calculator, and Řehuřek mentions at least two algorithms which can be plugged on its place:

1. The Lanczos algorithm as implemented by SVDLIBC ([Roh07]), which in turn is based on SVDPACKC written by Berry et al ([BDO+93]), which in turn is based on its Fortran77 predecessor SVDPACK ([Ber92c]). All of them ultimately based on seminal paper by Berry [Ber92a] (which in turn comes from his PhD thesis [Ber91a]).

2. A custom stochastic algorithm based on the work of Halko et al (see [HMT11]).

For the scope of this project, we considered appropriate to focus only on the Lanczos based algorithm; as that is essentially what we described

in the previous chapter. In that sense, the work of Řehuřek is interesting because by using the divide and conquer strategy for the SVD problem, he is leveraging on the decades of research and numerical accuracy of the work done by Berry et al. At the same time, his key contribution becomes the procedure Merge-SVD, which we will describe in further sections.

7.2 Subspace tracking

Řehuřek does a very comprehensive survey of the state of the art regarding SVD algorithms, in order to position the variant that he proposes in a wider context. This is because there are a lot of variants of SVD algorithms over there, each one emphasizing a different subset of aspects. Actually, let us remember that we have already restricted ourselves in a couple of aspects, when we focused our attention to the particular application of LSI:

- The LSI term-document matrices are highly sparse, which allows one to prune a big branch of the SVD tree of algorithms.
- The LSI applications require a truncated SVD factorization, usually of a few hundred entries; therefore, algorithms that take advantage of such truncation are preferred.

Řehuřek goes even further on this specialization approach, and imposes himself additional restrictions:

1. Distributable: he seems specially interested in achieving a high-level parallelization of the problem, that can be split across the nodes of a commodity cluster.

2. Online: contrary to a batch SVD algorithm, he is interested in an algorithm that is capable of reusing the already computed SVD factorization, in such a way that one can update previous solution when new data comes available. This can be useful in today's applications for LSI, which may get the documents from social networks or similar environments that can be thought as a permanent and basically unlimited source of data. Recalculating from scratch the SVD may be unfeasible under those circumstances, hence updating an existing

solution is desired.

3. One pass: as discussed in previous chapter, among the most advanced parallel algorithms for SVD use the so called approach of Krylov subspaces (Lanczos,Arnoldi); they do require though, several passes to the data. But Řehuřek is interested on streamed environments, where saving all the data may be just unfeasible; hence, he proposes instead an algorithm that consumes the data in a single pass and discards it. This applies again to the documents of the LSI problem; let us recall that the term-matrix of $m \times n$ is very wide in the horizontal sense ($n >> m$); this situation comes from the fact that we have much more documents (columns) than terms (rows). Putting again the sample application that extracts the documents from social networks, the terms used for English language is typically around $100,000$ and is assumed to be static, while documents can be generated constantly in volumes of millions. Wanting to accumulate them all, for the sake of having the SVD factorization that covers everything, does not seem practical either; hence, on a given time we update existing solution with new data and immediately discard it (keeping only results the factorization).

4. Constant memory: strong emphasis is placed on the memory complexity of the algorithm, aiming to avoid dependency on the input data. The memory complexity of an algorithm that saved all documents, historically, can be seen just as $\mathrm{O}(n)$, where n is the number of columns of the term-document matrix. But the distributed algorithm ensures that memory requirements are controlled, and depend mostly on the size of truncated matrix (which is usually a few hundreds for LSI).

An algorithm that posses all these attributes: online, one pass and using constant memory, can be considered an instance of the so called "subspace tracking" approach. The term may not intuitively reflect all the properties, but we tried to come up a justification of the name. It may come from the fact that the SVD factorization, among several other factorizations, essentially gives us subspaces that characterize our input matrix (recall that the matrices V and U contain basis for the four subspaces $\mathrm{C}(A^T), \mathrm{N}(A), \mathrm{C}(A), \mathrm{N}(A^T)$). As we update our factorization due new data, such subspaces may change; then, by continuously updating the basis (SVD

matrices) we could say that the subspace they generate is being "tracked" across time [1].

7.2.1 SVD as an eigenvalue problem

It may not be evident but the characteristics imposed on the algorithm for being one-pass and online, actually imply that we can not store the matrix V from the SVD factorization. As suggested above, such storage is prohibitive because its dimensions $n \times n$, which come directly from the number of documents to handle throughout time (which taken historically, can be a huge amount). An essential variant of the algorithm proposed by Řehuřek then, is that it just deals with the calculation of the matrices U and Σ; leaving V behind. How is that possible? Please refer to section section 6.1 for an explanation of the equivalence between the SVD problem, and the Eigenproblem of either symmetric matrix $A^T A$ or $A A^T$.

Therefore, if we are just interested in matrices U and Σ of the SVD factorization; we can restate our goal as solving the eigenproblem for symmetric matrix $A A^T$. That is, finding its eigenvalues (Σ^2) and eigenvectors (U).

Before proceeding to review the details of procedure Merge-SVD, which serves to merge two SVD factorizations, is important to clarify that Řehuřek uses the matrix P in his pseudocode as a tuple (U, Σ) rather than as the product $\Sigma^{-1} U^T$. This is due practical reasons, as we need to individually access the original matrices; but still the name P is kept, to remind us that they can form the projection matrix.

7.3 Merging Two SVD factorizations

The core logic of the algorithms presented in last section (algorithm 5 and algorithm 6), relies on the procedure Merge-SVD. Is may not be evident at all, but the essence of this merge is to use SVD factorization again! The The PhD thesis of Řehuřek presents a series of refinements, until he reaches the optimized version presented below:

[1] This attempt to explain the origin of the term is of our own, and is just to help one understanding the term of "subspace tracking"; but texts and books about it, usually in the area of Signal Processing, do not seem to explain the concept in this way.

Algorithm 7: Merge-SVD: Merge of two SVD factorizations

Input : Truncation factor k, decay factor γ, $P_1 = (U_1^{m\times k_1}, \Sigma_1^{k_1\times k_1})$, $P_2 = (U_2^{m\times k_2}, \Sigma_1^{k_2\times k_2})$

Output: $(U^{m\times k}, \Sigma^{k\times k})$

1 $Z^{k_1\times k_2} \leftarrow {U_1}^T U_2$

2 $U^{'}R \xleftarrow{QR} U_2 - U_1 Z$

3 $U_R \Sigma {V_R}^T \xleftarrow{SVD_k} \begin{bmatrix} \gamma\Sigma_1 & Z\Sigma_2 \\ 0 & R\Sigma_2 \end{bmatrix}^{(k_1+k_2)\times(k_1+k_2)}$

4 $\begin{bmatrix} R_1^{k_1\times k} \\ R_2^{k2\times k} \end{bmatrix} = U_R$

5 $U \leftarrow U_1 R_1 + U^{'} R_2$

6 return (U, Σ)

The algorithm 7 is a quite compressed piece of work, and none of its steps are intuitive. We proceed to explain them in more detail in the following subsections.

7.3.1 Input and Output Parameters

Is worth to remark a couple of new features that appear as input parameters of the merge procedure: we are introducing a new decay factor $\gamma \in (0.0, 1.0)$ that helps to give less relevance to old documents. Let us recall that these algorithms are designed to update an existing SVD calculation, where each update processes a new set documents (encoded as columns of the term-document matrix A).

There are three truncation parameters (k, k_1 and k_2), instead of just one; this is to give further flexibility to the algorithm, as it supports that the truncation factor varies with time. Each of the previous factorizations then could have been done with different truncation factors; but we homogenize the final result with the new truncation factor k. This feature may not be heavily used, as usually k is fixed in a few hundreds and not changed during the entire life of the LSI applications; there is no need though, to loose generality and impose the artificial restriction that the truncation fac-

tor shall remain static.

The output parameter, or result of the merge algorithm, is a new factorization U, Σ which covers the two partial SVD factorizations received.

7.3.2 Construction of a new basis

Most of the algorithm is about building a new basis (columns of matrix U), that spans the subspaces generated by basis in U_1 and U_2, respectively. This is done by taking advantage that U_1 and U_2 have orthonormal basis already as columns; hence, one of them is picked (U_1), and we only build the delta $U^{'}$ required to extend basis U_1 into required basis U.

The first two lines of algorithm 7 are basically to build the delta basis $U^{'}$, and thought not evident (nor explained in the articles by Řehuřek), we can find an intuitive interpretation of this procedure. Let us think in two vectors in $\mathbb{R}^3$ named $\mathbf{u_1}$ and $\mathbf{u_2}$, which are linearly independent. Let us suppose that we are given the task of building a basis of the two-dimensional subspace that those vectors span, with the additional requirement of making such basis orthonormal. Let us suppose that we pick $\mathbf{u_1}$ to be part of the basis, and now we just need to find an orthogonal vector to u_1, in order to complete our task. It can be proven that if we subtract from $\mathbf{u_2}$ the projection of $\mathbf{u_2}$ into $\mathbf{u_1}$, we get a vector that is orthogonal to $\mathbf{u_1}$ (let us name it $\mathbf{u_3}$):

$$\mathbf{u_3} = \mathbf{u_1} - (\mathbf{u_2} \cdot \mathbf{u_1})\mathbf{u_1} \ \ni \ \mathbf{v_3} \ \perp \ \mathbf{u_1}$$

If we consider the resulting set from the above recipe, that is $\{\mathbf{u_1}, \mathbf{u_3}\}$, we can not tell yet that is an orthonormal basis. However, they are at least linearly independent, hence we can apply standard procedures like Gram-Schmidt (see [Str88]) to produce the desired orthonormal basis.

Of course the above recipe works for any dimension, and that is essentially the calculation done in the first two lines of algorithm 7; though it states all the vector equations at once, by using matrix notation (the columns of matrices U_1 and U_2 play the role of vectors $\mathbf{u_1}$ and $\mathbf{u_2}$ from our example; and the right side of assignment of line two corresponds to vector $\mathbf{u_3}$). On the first line we calculate matrix Z which is the projection matrix of the columns of U_2 into columns from U_1; this give us the component of the projections only (the dot products), but multiplying that by U_1 is equivalent

to the expression $(\mathbf{u_2} \cdot \mathbf{u_1})\mathbf{u_1}$ from our example. The matrix substraction is a compressed way of introducing the vector equations from our example; and the QR factorization used to produce the orthonormal basis, is basically the application of the Gram-Schmidt process that we mentioned as well. It is not mentioned by [RR11] but the way of calculating $U^{'}$ is quite similar (if not the same), to the one reported by Hall et all in [HMM00] and [HMM02] (where it is done in the context of merging eigen models, which in particular contain eigen decompositions).

The usage of factorization QR deserves more comments, as we have not mentioned much about it until now. Given a rectangular matrix $B^{m\times n}$, it produces a factorization which consists of an orthogonal matrix $Q^{m\times m}$ (which is essentially a basis for the subspace spanned by the columns of A); followed by an upper triangular matrix $R^{m\times n}$. The triangular form of R comes from the application of the Gram-Schmidt algorithm: the column R_1 contains the coordinates of original column A_1 respect to the basis Q (it only depends on Q_1), the column R_2 indicates that original column A_2 depends only on the first two columns of Q, and so on. The QR algorithm is chosen by Řehuřek , not only due its ability to produce the missing vectors we needed for our basis (matrix $U^{'}$); but also due its side product, the triangular matrix R which is used in further steps.

7.3.3 Producing the diagonal matrix Σ

The probably most obscure step appears in line 3, where another SVD factorization is being applied, in order to produce the first part of the final result (the diagonal matrix Σ); along with an auxiliary rotation that we need to produce the other half of the final result (matrix U). But let us connect this with previously used QR algorithm (line 2), in order to clarify further.

In the SVD literature, there is a variant called R-SVD which uses the QR factorization as an intermediate step for SVD calculation. The name seems to come from Golub's book [GVL12], where is introduced as a previous step to the so called R-Bidiagonalization (the method proposed by Golub brings the original matrix A to a bidiagonal form, from where calculating SVD is easier). Putting aside this bidiagonalization context, the main idea of using QR factorization as an intermediate step in SVD calculation, is summarized in equation below:

$$A = QR = Q(U^{'}\Sigma V^{T}) = (QU^{'})\Sigma V^{T} \tag{7.1}$$

We can appreciate from equation above that the final matrix U is obtained, by composing the $U^{'}$ matrix (from the SVD factorization of triangular matrix R), with the orthogonal matrix Q (obtained from the QR factorization of A). Interestingly, the matrices Σ and V from the SVD of R, become the same as if one would have done SVD directly on matrix A. This is essentially the idea of line 3 from algorithm 7, which produces the diagonal matrix Σ that we need as final result; but it also produces a couple of additional matrices:

- The orthogonal matrix V_R^T, which is discarded (let us recall we just care about U and Σ).
- The matrix U_R, which like in the example with QR factorization, is just an auxiliary item for producing the final matrix U that we need (more about this on next section).

But the side products of the SVD_k calculation on step 3 is perhaps the less problematic to understand, the real trouble may come from the matrix we are using as input for such calculation. Let us name such matrix on the right hand side as X, it can be deduced from the following requirement that we impose on the final matrix $U = \left[U_1 \mid U^{'}\right]$ [2]:

$$\left[U_1\Sigma_1 \mid U_2\Sigma_2\right] = \left[U_1 \mid U^{'}\right] X$$

If we clear the matrix variable X by multiplying each side (on the left) by $\left[U_1 \mid U^{'}\right]^T$, we get the following (please note that we are using the matrix block operations, which nicely behave like scalars):

$$X = \left[U_1 \mid U^{'}\right]^T \left[U_1\Sigma_1 \mid U_2\Sigma_2\right] = \begin{bmatrix} {U_1}^T U_1\Sigma_1 & {U_1}^T U_2\Sigma_2 \\ {U^{'}}^T U_1\Sigma_1 & {U^{'}}^T U_2\Sigma_2 \end{bmatrix} \tag{7.2}$$

[2] The equality claimed on this equation is not totally clear, as after the SVD_k calculation of X we drop its V matrix; and the left side does not involve any matrix V. We contacted a couple of times the author (Radim Řehuřek) for kindly asking for a clarification about a related equation in his thesis, but unfortunately we did not got a final answer.

We need now a few additional equalities that can be inferred from the algorithm 7:

1. U_1 is orthogonal $\implies {U_1}^T U_1 = I$

2. By construction, the set of columns from where $U^{'}$ is calculated (that is, $U_2 - U_1 Z$), is orthogonal to $U_1 \implies$ the subspace spanned by such set is also orthogonal to U_1. In particular, any basis of that subspace is also orthogonal to U_1. Therefore $U^{'}$ is orthogonal to U_1, that is, ${U^{'}}^T U_1 = 0$.

3. Using the above, and the QR calculation from line 2 of algorithm 7, Řehuřek claims that $R = {U^{'}}^T U_2$. Such equality is not totally clear, as it seems as if we would be isolating R from that step; however, such step represents an assignment, not an equation. The claim may be due may a property of QR calculation itself (seen as a function of matrices, rather than a procedure). We take it for granted [3].

Using the three equalities just mentioned, the matrix X from eq. (7.2) can be further simplified as follows:

$$X = \begin{bmatrix} \Sigma_1 & {U_1}^T U_2 \Sigma_2 \\ 0 & {U^{'}}^T U_2 \Sigma_2 \end{bmatrix} = \begin{bmatrix} \Sigma_1 & Z\Sigma_2 \\ 0 & R\Sigma_2 \end{bmatrix} \tag{7.3}$$

It is equation eq. (7.3) that justifies the right hand side of step 3 in algorithm 7.

A final note about this step, is that the SVD routine being called is not the same as Basecase-SVD from algorithm 5; while the former is a full SVD for shorter "dense" matrices, the second is a truncated SVD calculation for large sparse ones. The dense SVD calculation is done with the standard algorithm called Golub-Kahan-Reinsch ([GK65], [GR70]), available as a LAPACK routine ([ABB+99]); while the truncated sparse SVD is done with

[3] If this work is used for a thesis, we will seek to clarify this part though.

the also fameous LASVD routine (later incarnated as SVDPACKC LAS2), that Berry did from the Lanzos version of Parlett and Simon ([PS79],[Sim84]).

7.3.4 Calculating the final matrix U

All these auxiliary results may take us apart from our final goal, so let us remember what it is: to produce a couple of matrices, U and Σ, which represent the merged eigen decomposition of the two pair of matrices we received as input (U_1,Σ_1 and U_2,Σ_2). So far, we have calculated already the diagonal Σ; hence the remaining task is to calculate U. We have all the auxiliary devices at our disposal, from previous steps of the algorithm.

We began by picking orthonormal basis U_1, and extending it with $U^{'}$ in order to get a new orthonormal basis (in matrix form) $\left[U_1 \mid U^{'}\right]$; such basis covers the spanning subspaces of both U_1 and U_2. We may be tempted to think that such matrix is the desired U, but the problem is that we took the diagonal Σ from an SVD calculation; that means we got already one orthogonal matrix for the term-space U_R. We need to compose such U_R with our orthonormal basis $\left[U_1 \mid U^{'}\right]$, in order to get the final basis U (due same reasons exposed in eq. (7.1)):

$$U = \left[U_1 \mid U^{'}\right] U_R$$

But now we exploit the shape of matrix U_R; if we were doing full SVD calculation in the line 3 of algorithm 7, we would have a matrix U_R of dimensions $(k_1 + k_2) \times (k_1 + k_2)$. But since we are calculating the truncated SVD instead, it gets dimensions $(k_1 + k_2) \times k$. Furthermore, it can be split in two blocks R_1 and R_2 as follows:

$$U_R = \begin{bmatrix} R_1^{k_1 \times k} \\ R_2^{k_2 \times k} \end{bmatrix}$$

Using block multiplication in the submatrices, we can get the final assignment from line 5 of algorithm 7:

$$U \leftarrow \left[U_1 \mid U^{'}\right] U_R = \left[U_1 \mid U^{'}\right] \begin{bmatrix} R_1^{k_1 \times k} \\ R_2^{k_2 \times k} \end{bmatrix} = U_1 R_1 + U^{'} R_2$$

7.4 Complexity and performance

7.4.1 Time complexity of the Merge-SVD algorithm

The overall complexity of the algorithm 5 can be expressed in terms of functions Basecase-SVD and Merge-SVD; but given the former is seen as a black box over which we have little control, and that the main contribution of Řehuřek (from SVD perspective), is the algorithm 7, we focus on the complexity of that Merge-SVD alone.

Let us review the cost of its main steps of algorithm 7 individually, as a way to arriving to the the overall complexity (we will not consider the possible parallelization or vectorization of the basic kernel operations [4], which is usually achieved by using standard libraries like BLAS or LAPACK):

- The matrix multiplication that produces Z in line 1, is done against matrices ${U_1}^T$ (of dimensions $k_1 \times m$) and matrix U_2 (of dimensions $m \times k_2$); hence it has a complexity $\mathrm{O}(mk_1k_2)$.

- The second step is dominated by the QR calculation; according to Golub [GVL12], the complexity of a QR factorization based on the Gram-Schmidt process for a matrix $A^{m \times n}$, is $\mathrm{O}\left(mn^2\right)$. Applying that result to the particular case of line 2, give us a complexity of $\mathrm{O}\left(mk_2^2\right)$.

- It seems hard to find reported complexities for the SVD algorithms, in the available literature; Řehuřek mentions that the complexity of the full SVD calculation from line 3 is $\mathrm{O}\left((k_1 + k_2)^3\right)$[5]. Hence, given that the truncation factors k_1 and k_2 are usually a few hundreds in the context of LSI; the cost of this step can be neglected.

- Finally, the complexity of the matrix operations in the last step (focusing on the products only), is $\mathrm{O}(mkk_1 + mkk_2)$.

[4] A "kernel" in the context of Numerical Linear Algebra, is a basic routine which is heavily used by higher level algorithms; hence, its performance is crucial and they are heavily optimized.

[5] We could find at least one reference that also mentions this complexity, see [Pla05a].

In practice the truncation factors do not vary much in LSI applications, thus, we can simplify further. Let us assume that $k \approx k_1 \approx k_2$, then the reported complexities in the list above become: O $\left(mk^2\right)$, O $\left(mk^2\right)$, O $\left(k^3\right)$, O $\left(mk^2\right)$. Given that the number of terms m will be much bigger than the truncation factor k (hundred of thousands, vs a few hundreds); we can conclude that the overall time complexity is O $\left(mk^2\right)$.

Due time constraints we did not enter into detailed memory complexity analysis of the algorithm, but is part of our todo list (for the case that this project evolves into a full thesis).

7.4.2 Performance with a large scale corpus

Řehuřek used 3 different corpus to test his distributed SVD algorithm, in the context of LSI. We focused only on the large corpus, which was the English Wikipedia. By that time, it contained 3.2 million documents; where 100,000 terms were chosen after removing the stop words. That resulted in an sparse matrix of dimensions $100,000 \times 3,199,665$, with 0.5Gb of non zero entries. Such matrix can fit in memory of a modern personal computer, but as explained earlier, the main objective of using the distributed algorithm is to scale in time. The truncation factor k was set to 400 during this experiment.

On his Phd thesis, Řehuřek reports the following wall times of the distributed algorithm 5, running on a single computer and on a cluster:

- 8.5 hours on a dual-core 2.53GHz MacBook Pro with 4GB RAM and vecLib, a fast BLAS ([LHKK79])/ LAPACK ([ABB+99]) library provided by the vendor.

- 2 hours 23 minutes on a cluster of four dual-core 2GHz Intel Xeons, each with 4GB of RAM, which share the same Ethernet segment and communicate via TCP/IP. These machines did not have any optimized BLAS library installed.

The above numbers suggest what we expected: given that the parallelization achieved by the distributed algorithm is almost perfect (only communication needed is on the final merge), the scaling in time is basically linear

with respect to the number of computing nodes.

The gensim page (see [ŘS11]) has a more up to date experiment, which reports 5 hours 25 minutes for a single machine; and 1 hour with 41 minutes for a cluster with 4 nodes (this time, the cluster nodes got ATLAS installed, an open source BLAS/LAPACK implementation (see [Wha11]).

Řehuřek does an additional comparison for the execution on a single machine, by contrasting with a custom implementation of the SVD algorithm published in [ZMS98], named as ZMS in his Phd thesis. The ZMS algorithm took 109 hours, which brutally contrast with the 2 hours 23 minutes mentioned above for the algorithm 5. A probably more fair comparison, would be against the SVD algorithm implemented by SLEPc ([HRTV07c]); thought this opensource implementation does not target specifically the LSI problem, it claims to be distributed and highly scalable. Other comparisons with more opensource implementations are possible: together, along with reproducing the results published by Řehuřek with more nodes in the cluster, are planned to be performed in a further stage of this project.

7.5 Accuracy of the merge algorithm

The overall numerical accuracy of any algorithm, is of crucial relevance in the area of Numerical Analysis; in particular in the subarea we care about in this project (Numerical Linear Algebra). Řehuřek offers detailed and promising accuracy comparisons between his proposal and several other available implementations, though he does that mainly for the serial executions (single node) of small/medium corpus sizes (not the Wikipedia experiment described in previous section). Another pending task to verify the accuracy against a golden standard, could be to perform experiments on a supercomputer with enough RAM to hold the large corpus matrix; using an standard SVD software like [Roh07]. The authors of these lines have added such pending task, to the TODO list for further stages of this project.

Despite of the above, an interesting analysis about the effect of nested truncation that algorithm 5 introduces, is exposed in Řehuřek Phd thesis [RR11]. Citing the work of Zha et al ([ZZ00]), he remarks that his distributed SVD algorithm meets the conditions to be an stable algorithm (on the numerical sense), though no longer exact. This should not surprise us, as the almost perfect parallelism achieved can not come without a price: every

merge of two SVD factorizations, as produced by algorithm 7, introduces some error, in the sense that the following equality does not hold:

$$\mathrm{SVD_k}(\left[A_1 \mid A_2\right]) = \mathrm{SVD_k}(\left[\mathrm{SVD_k}(A_1) \mid \mathrm{SVD_k}(A_2)\right]) \tag{7.4}$$

In other words, calculating truncated SVD against the original input matrices A_1 and A_2 (concatenated by columns), is not the same as calculating the same over their truncated SVD_k approximations. Let us recall that the matrix produced by $SVD_k(A)$ is just an approximation of original matrix A.

The precision lost by accepting as inputs rank-k approximations, instead of the original matrices, is not that bad though; it is shown in [ZZ00] and reused by Rehurek in [RR11], that the typical matrix A that emerges from Natural Language Applications like LSI, "do indeed possess the necessary structure and that in this case, a rank-k approximation of A can be expressed as a combination of rank-k approximations of its submatrices without a serious loss of precision".

The above quote means, that the equality eq. (7.4) can be considered to hold in practice. In strict theory we shall replace the equality sign by an approximation sign though, as the equality sign can be stated only on the idealistic case of exact arithmetic. Hence, we can claim that:

$$\mathrm{SVD_k}(\left[A_1 \mid A_2\right]) \approx \mathrm{SVD_k}(\left[\mathrm{SVD_k}(A_1) \mid \mathrm{SVD_k}(A_2)\right]) \tag{7.5}$$

This is in part, because the SVD_k factorization of a matrix A is not actually just an approximation (as we claimed paragraphs above), it is "the best" approximation by a matrix of rank k, per the Eckart-Young Theorem [EY36]. The eq. (7.5), derived from the work of [ZZ00], can be considered the angular stone for the divide-and-conquer strategy of the distributed algorithm 5. Without it, we would not know if it is valid to use the SVD_k calculation itself, as a way of combining two already calculated SVD_k factorizations. A quite interesting research path for this project, could be to seek for other alternatives for doing the merge; or, to confirm that the scheme proposed by Řehuřek is the optimal way of merging two truncated SVD factorizations.

Chapter 8

Conclusions and Future Work

The Singular Value Decomposition has a very long history that began in the 19th Century with the publication of Beltrami. His publication, which can be considered one of the seminal articles on the subject, had the purpose of aiming the Italian students to focus on the study of the bilinear forms. The consequences of that paper remain to our days. However, at that time it did not receive so much attention that even other mathematicians published similar results during the first part of the 20th Century.

Golub was probably another father of the SVD from the numerical point of view. We owe him and Kahan the first fast algorithms. Also, due to Golub the SVD was brought to prominence because he explained how it could be applied. He also pointed out pitfalls of other SVD methods and how to overcome them to achieve numerical stable SVD algorithms.

However, we should not forget about Cornelius Lanczos to whom we owe the most cited algorithm for getting the factorization (with some modifications, though). On this, we ought to recognize Paige and Ojalvo and Newman because they complemented Lanczos' work. Ojalvo and Newman made the method numerically stable and Paige provided the error analysis.

From the side of Information Retrieval, it was in [SWY75] where the Vector Space Model for representing documents was introduced (1975). The contribution he made is the possibility of converting text to a mathematical tractable representation in the form of vectors. This model have been explored in the literature since then. However, when directly applied it suffers due to the phenomena of synonymity and polysemy.

As mentioned in this paper, it was Deerwester et al. the ones that

proposed a novel method to overcome those difficulties giving birth to what today is called Latent Semantic Analysis. This happened in 1990. For this formulation, they used a known fact established in 1936 by Eckart and others regarding matrix approximations. Here is where the relation of the two branches is found and then started to walk side by side.

The work of Berry can be considered fundamental in the history of LSI since it was him the one that paid attention to the nature of the matrices generated by the Vector Space Model. His work has inspired many of the real world implementations that we have today.

The table 8.1 shows what we consider the seminal papers in this area. It is important because it greatly discriminates the big steps that science has walked in the development of the SVD algorithms and also because it separates outstanding advances from other types of contributions. In that sense, the work by Řehuřek deserves an special mention since it is currently (as far as the authors of this work know) the most scalable algorithm for SVD-LSI.

For that reason, in this work we have paid special attention to how this particular implementations works. Řehuřek 's algorithm is, in general terms, a divide and conquer algorithm. In modern terms, we can think of it as a Map-Reduce program. The map part begins by splitting the original matrix A into chunks that are called *jobs*. A set of these jobs are sent to different clusters for processing.

In each node of the cluster, the SVD of the jobs are calculated using what Řehuřek calls the SVD base case. This is basically a black box for the algorithm and so, any other implementation for smaller matrices can be used. His particular implementation uses a Lanczos solver from SVDLIBC. Once the SVDs are obtained, they are merged locally using Řehuřek 's merge algorithm, which can be considered one of his main contributions to the problem's solution.

In the end, Řehuřek obtains a close approximation of the SVD of the original matrix A. This is so because the merge algorithm introduces some errors. Specifically, he tries to obtain the truncated SVD of the matrix A based on the truncated SVDs of two sub-matrices which does not exactly match. However, it has been proved that this approximation is good enough. A deeper understanding of such result is still in our *TODO* list.

According to Řehuřek , the complexity of his merge algorithm is $O(mk^2)$. In a real experiment, he was able to process the *Wikipedia* containing $100K$ indexed terms and $3.2M$ documents in nearly 2.5 hours in a 4 nodes cluster. A subsequent experiment performed by him as well improved the running time to just 1 hour and 41 minutes by using highly optimized BLAS libraries.

Year	Author	Publication	Contribution
1873	Beltrami	[Bel73]	First time SVD appeared in the literature
1912	Weyl	[Wey12]	Proved the singular value problem is well conditioned
1913	Müntz	[Mun13]	Power Method
1936	Eckart & Young	[EY36]	Eckart-Young-Mirsky Truncated SVD Theorem (used later for LSI)
1950	Lanczos	[Lan50a]	Most cited algorithm for sparse SVD problem
1970	Golub	[GR70]	Made SVD popular. Also most popular algorithm for dense SVD problem.
1971	Dubrulle et al.	[DMW71a]	The implicit QL algorithm
1971	Paige	[Pai71]	Stable algorithm for Lanczos step and error analysis.
1975	Salton et al.	[SWY75]	Introduced the Vector Space Model
1979	Parlett et al.	[PS79]	Selective Re-orthogonalization for Lanczos alg.
1990	Deerwester et. al.	[DDF$^+$90]	Introduced LSI
1991	Berry	[Ber91a]	Parallel SVD algorithms for LSI, widely used SVDPACK
2000	Zha and Zhang	[ZZ00]	Theoretical basis for Řehuřek SVD-Merge
2000	Hall et al.	[HMM00]	Algorithm for merging eigenmodels
2011	Řehuřek	[RR11]	Most scalable distributed algorithm to the date

Table 8.1: Seminal papers for LSI

Another comparison that he did was against a custom implementation of another SVD algorithm published in the literature. In that case, this custom algorithm took 109 hours(!) to finish.

The study that we did was not as complete as we would like due to time constraints. Some specific items remains in our *TODO* list. For example, in Řehuřek 's thesis, in the merge algorithm and specifically during the QR calculation, he claims that $R = 'U^T U_2$. We did not have enough time to verify or to find a reference for such assignment.

Regarding the space complexity, we could not dig into it either. At the moment, we are not sure if there has been an study on that matter or if it is something that researchers still need to investigate.

The research so far seems to be, at the eyes of the authors of this report, a work in progress. For example, we need a modern sensible experiment that compares the best parallel/distributed algorithms taking into account the architecture of modern computers. In particular, the comparison that Řehuřek did against another SVD algorithm seems unfair because he might have spent several months or a few years constructing his software. In contrast, we do not know how much effort was put into his opponent. For us, it would have been more fair if he had taken a professional software with all reasonable optimizations. Actually, we have already a list of open source candidates for such comparison.

Another bit that seems to be missing is to experiment with different implementations of the base case SVD solver. From Řehuřek 's work, it seems pretty much clear that he was more concerned about the merge step. So, we think that a more detailed study of this part is required. Profiling experiments similar to those made by Berry, could help to know if the base case solver requires parallelization (we already have a proposal for this).

Also, another set of interesting experiments would be to try other merge alternatives. Řehuřek did not compare his merge algorithm to other divide and conquer strategies that we found in the literature. Related to that, we need to research whether Řehuřek 's merge algorithm is optimal or not.

We expect to go through these details if this work is accepted as a thesis.

Bibliography

[ABB+99] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, S Hammerling, Alan McKenney, et al. *LAPACK Users' guide*, volume 9. Siam, 1999.

[And01] Rie Ando. *The document representation problem: An analysis of LSI and Iterative Residual Rescaling*. PhD thesis, Faculty of the Graduate School of Cornell University, may 2001.

[BDO+93] Michael Berry, Theresa Do, Gavin O'Brien, Vijay Krishna, and Sowmini Varadhan. Svdpackc (version 1.0) user's guide1. 1993.

[BDO95] Michael W Berry, Susan T Dumais, and Gavin W O'Brien. Using linear algebra for intelligent information retrieval. *SIAM review*, 37(4):573–595, 1995.

[Bel73] E. Beltrami. Sulle funzioni bilineary. *Giornale Di Matematiche Ad Uso Degli Studenti Delle Universita Italiane*, 1873.

[Ber91a] Michael Waitsel Berry. *Multiprocessor sparse SVD algorithms and applications*. PhD thesis, Citeseer, 1991.

[Ber91b] Michael Waitsel Berry. *Multiprocessor sparse SVD algorithms and applications*. PhD thesis, Citeseer, 1991.

[Ber92a] Michael W Berry. Large-scale sparse singular value computations. *International Journal of Supercomputer Applications*, 6(1):13–49, 1992.

[Ber92b] Michael W Berry. Large-scale sparse singular value computations. *International Journal of Supercomputer Applications*, 6(1):13–49, 1992.

[Ber92c] Michael W Berry. Svdpack: A fortran-77 software library for the sparse singular value decomposition. 1992.

[BHSS65] F. L. Bauer, J. Heinhold, K. Samelson, and R. Sauer. *Moderne Rechenanlagen.* Teubner, Stuttgart, 1965.

[BKS89] SJ Blank, Nishan Krikorian, and David Spring. A geometrically inspired proof of the singular value decomposition. *American Mathematical Monthly*, 96(3):238–239, 1989.

[BMPS06] Michael W Berry, Dani Mezher, Bernard Philippe, and Ahmed Sameh. Parallel algorithms for the singular value decomposition. *Statistics Textbooks and Monographs*, 184:117, 2006.

[CB11] Bob Carpenter and Breck Baldwin. Natural language processing with lingpipe 4, 2011.

[DBMS79] Jack J Dongarra, James R Bunch, Cleve B Moler, and Gilbert W Stewart. *LINPACK users' guide*, volume 8. Siam, 1979.

[DDF+90] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, 41(6):391–407, 1990.

[DMW71a] A Dubrulle, RS Martin, and JH Wilkinson. The implicit ql algorithm. In *Linear algebra*, pages 241–248. Springer, 1971.

[DMW71b] A Dubrulle, RS Martin, and JH Wilkinson. The implicit ql algorithm. In *Linear algebra*, pages 241–248. Springer, 1971.

[DS85] Jack J Dongarra and DC Sorcnsen. A fast algorithm for the symmetric eigenvalue problem. In *Computer Arithmetic (ARITH), 1985 IEEE 7th Symposium on*, pages 337–342. IEEE, 1985.

[EMU96] Gisela Engeln-Müllges and Frank Uhlig. *Numerical Algorithms with C.* Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[EY36] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[GCGO07] Gene Howard Golub, Raymond H. Chan, Chen Greif, and Dianne Prost O'Leary. *Milestones in matrix computation : selected works of Gene H. Golub with commentaries*. Oxford science publications. Oxford University Press, Oxford, 2007.

[GK65] Gene Golub and William Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial & Applied Mathematics, Series B: Numerical Analysis*, 2(2):205–224, 1965.

[Gor06] Genevieve Gorrell. Generalized hebbian algorithm for incremental singular value decomposition in natural language processing. In *EACL*. Citeseer, 2006.

[GR70] Gene H Golub and Christian Reinsch. Singular value decomposition and least squares solutions. *Numerische mathematik*, 14(5):403–420, 1970.

[gra] Graphlab. `https://dato.com/products/create`. Accessed: 2015-08-08.

[GVdV00] Gene H Golub and Henk A Van der Vorst. Eigenvalue computation in the 20th century. *Journal of Computational and Applied Mathematics*, 123(1):35–65, 2000.

[GVL12] Gene H Golub and Charles F Van Loan. *Matrix computations*. Johns Hopkins University Press, 4th edition, 2012.

[GW05] Genevieve Gorrell and Brandyn Webb. Generalized hebbian algorithm for incremental latent semantic analysis. In *INTERSPEECH*, pages 1325–1328. Citeseer, 2005.

[Hal12] Nathan P Halko. *Randomized methods for computing low-rank approximations of matrices*. PhD thesis, University of Colorado, 2012.

[HMM00] Peter Hall, David Marshall, and Ralph Martin. Merging and splitting eigenspace models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(9):1042–1049, 2000.

[HMM02] Peter Hall, David Marshall, and Ralph Martin. Adding and subtracting eigenspaces with eigenvalue decomposition and singular value decomposition. *Image and Vision Computing*, 20(13):1009–1016, 2002.

[HMT11] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[HRTV07a] V Hernandez, JE Roman, A Tomas, and V Vidal. Lanczos methods in slepc, 2007.

[HRTV07b] V Hernandez, JE Roman, A Tomas, and V Vidal. Restarted lanczos bidiagonalization for the svd in slepc, 2007.

[HRTV07c] V Hernandez, JE Roman, A Tomas, and V Vidal. Restarted lanczos bidiagonalization for the svd in slepc, 2007.

[HRV05] Vicente Hernandez, Jose E Roman, and Vicente Vidal. Slepc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):351–362, 2005.

[JM01] E. R. Jessup and J. H. Martin. Computational information retrieval. chapter Taking a New Look at the Latent Semantic Analysis Approach to Information Retrieval, pages 121–144. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[JS01] Zhongxiao Jia and GW Stewart. An analysis of the rayleigh–ritz method for approximating eigenspaces. *Mathematics of Computation*, 70(234):637–647, 2001.

[Kal96] Dan Kalman. A singularly valuable decomposition: the svd of a matrix. *The college mathematics journal*, 27(1):2–23, 1996.

[Kli01] Alex Klimovitski. Using sse and sse2: Misconceptions and reality. *Intel developer update magazine*, pages 3–8, 2001.

[Kry31] AN Krylov. On the numerical solution of the equation by which in technical questions frequencies of small oscillations of material systems are determined. *Izvestija AN SSSR (News of Academy of Sciences of the USSR), Otdel. mat. i estest. nauk*, 7(4):491–539, 1931.

[Lan50a] Cornelius Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 1950.

[Lan50b] Cornelius Lanczos. *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office, 1950.

[Lan04] Serge Lang. *Linear Algebra*. Springer-Verlag, New York, 3rd edition, 2004.

[Lay12] David C Lay. *Linear algebra and its applications*. Addison-Wesley, 4th edition, 2012.

[LBG+12] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[Lei05] Arthur W Leissa. The historical bases of the rayleigh and ritz methods. *Journal of Sound and Vibration*, 287(4):961–978, 2005.

[LGK+14] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[LHKK79] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[lin] Lingpipe. `http://alias-i.com/lingpipe/index.html`. Accessed: 2015-08-08.

[LPS87] Sy-Shin Lo, Bernard Philippe, and Ahmed Sameh. A multiprocessor algorithm for the symmetric tridiagonal eigenvalue problem. *SIAM Journal on Scientific and Statistical Computing*, 8(2):s155–s165, 1987.

[mah] Mahout. `https://mahout.apache.org/`. Accessed: 2015-08-08.

[Mir60] L. Mirsky. Symmetric gauge functions and unitarily invariant norms. *QJ Math., Oxf. II. Ser.*, 11:50–59, 1960.

[MP12] Carla D. Martin and Mason A. Porter. The extraordinary svd. *The American Mathematical Monthly*, 119(10):pp. 838–851, 2012.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK, 2008.

[Mun13] CL Muntz. Solution direct de l'equation seculaire et de quelques problems analogues. *Comptus Rendus de l'Academie des Sciences*, 156:43–46, 1913.

[Oli06] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[oraa] Oracle and collective intellect. `http://www.oracle.com/us/corporate/acquisitions/collectiveintellect/index.html`. Accessed: 2015-08-08.

[orab] Oracle social cloud platform text analytics. `http://www.oracle.com/us/solutions/social/social-engagement-monitoring-cloud-service/social-cloud-text-analytics-1870526.pdf`. Accessed: 2015-08-08.

[Pai71] Christopher Conway Paige. *The computation of eigenvalues and eigenvectors of very large sparse matrices.* PhD thesis, University of London, 1971.

[Pai76] Christopher C Paige. Error analysis of the lanczos algorithm for tridiagonalizing a symmetric matrix. *IMA Journal of Applied Mathematics*, 18(3):341–349, 1976.

[Pla05a] Gerald E Plassman. A survey of singular value decomposition methods and performance comparison of some available serial codes. *NASA Technical Report CR-2005-213500*, 2005.

[Pla05b] Gerald E Plassman. A survey of singular value decomposition methods and performance comparison of some available serial codes. *NASA Technical Report CR-2005-213500*, 2005.

[pro] Welcome to the propack homepage. `http://sun.stanford.edu/~rmunk/PROPACK/`. Accessed: 2015-08-08.

[PS79] Beresford N Parlett and David S Scott. The lanczos algorithm with selective orthogonalization. *Mathematics of computation*, 33(145):217–238, 1979.

[Ř+10a] Radim Řehuřek et al. Fast and faster: A comparison of two streamed matrix decomposition algorithms. 2010.

[Ř+10b] Radim Řehuřek et al. Speeding up latent semantic analysis: A streamed distributed algorithm for svd updates. 2010.

[Řeh11] Radim Řehuřek. Subspace tracking for latent semantic analysis. In *Advances in Information Retrieval*, pages 289–300. Springer, 2011.

[Roh07] Doug Rohde. Svdlibc, 2007.

[RR11] RNDr Radim Rehurek. *SCALABILITY OF SEMANTIC ANALYSIS IN NATURAL LANGUAGE PROCESSING*. PhD thesis, Masaryk University, 2011.

[ŘS11] R Řehuřek and P Sojka. Gensim–python framework for vector space modelling. *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic*, 2011.

[SBG+13] Brian T Smith, James M. Boyle, BS Garbow, Y Ikebe, VC Klema, and CB Moler. *Matrix eigensystem routines-EISPACK guide*, volume 6. Springer, 2013.

[SBST11] Mike Symonds, Peter Bruza, Laurianne Sitbon, and Ian Turner. Modelling word meaning using efficient tensor representations. In *Proceedings of the 25th Pacific Asia Conference on Language, Information and Computation, PACLIC 25, Singapore, December 16-18, 2011*, pages 313–322, 2011.

[Sim84] Horst D Simon. Analysis of the symmetric lanczos algorithm with reorthogonalization methods. *Linear algebra and its applications*, 61:101–131, 1984.

[sle] Slepc. `http://slepc.upv.es/`. Accessed: 2015-08-08.

[Ste73] G. W. Stewart. *Introduction to Matrix Computations*. Academic Press, 1973.

[Ste90] G. W. Stewart. Perturbation theory for the singular value decomposition. In *IN SVD AND SIGNAL PROCESSING, II: ALGORITHMS, ANALYSIS AND APPLICATIONS*, pages 99–109. Elsevier, 1990.

[Ste93] Gilbert W Stewart. On the early history of the singular value decomposition. *SIAM review*, 35(4):551–566, 1993.

[Str88] Gilbert Strang. *Linear algebra and its applications*. Saunders, 3rd edition, 1988.

[Str93] Gilbert Strang. The fundamental theorem of linear algebra. *American Mathematical Monthly*, pages 848–855, 1993.

[SWY75] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, November 1975.

[Wey12] H. Weyl. Das asymptotische verteilungsgesetz der eigenwerte linearer partieller differentialgleichungen (mit einer anwendung auf die theorie der hohlraumstrahlung). *Mathematische Annalen*, 71:441–479, 1912.

[Wha11] R Clint Whaley. Atlas (automatically tuned linear algebra software). In *Encyclopedia of Parallel Computing*, pages 95–101. Springer, 2011.

[Wil60] J. H. Wilkinson. Rounding errors in algebraic processes. In *Proceedings of UNESCO Conference on Information Processing, 1959*, pages 44–53, pub-But:adr, 1960. pub-But.

[Wil81] Herbert S Wilf. An algorithm-inspired proof of the spectral theorem in E^n. *American Mathematical Monthly*, pages 49–50, 1981.

[ZMS98] Hongyuan Zha, Osni Marques, and Horst D Simon. Large-scale svd and subspace-based methods for information retrieval. In *Solving Irregularly Structured Problems in Parallel*, pages 29–42. Springer, 1998.

[ZZ00] Hongyuan Zha and Zhenyue Zhang. Matrices with low-rank-plus-shift structure: Partial svd and latent semantic indexing. *SIAM Journal on Matrix Analysis and Applications*, 21(2):522–536, 2000.