Summary

This project aims to upscale several key machine learning agorithms by using MapReduce paradigm. To achieve this goal, a set of matrix operatiors is implemented on MapReduce first. With their help, three algorithms (Support Vector Machines, Non-Negative Matrix Factorization and PageRank) are studied and successfully adapted to this new parallel framework with a intuitive and elegent fashion. Tests and experiments adopted shows that all the algorithms are correctly implemented. Moreover they exhibit flexibility and scalbility on large scale of datasets.

We summarise our main contributions in the following list:

- Matrix multiplication is implemented on MapReduce with flexible partitioning strategies, see 3.3 *
- We implement a small linear algerbra library consisting of Matrix multiplciation, matrix transpose, and several elementary-wise operators (addition, substraction, multiplication and division) , see 3.4*
- SVM training algorithm is implemented on MapReduce by using the gradient descent method. It is the first time that SVM has been adapted on MapReduce, see 4.2*
- We implement the NMF by employing the method proposed by Lee and Seung (2000), see 4.3.
- We implement the PageRank algorithm from its foundational mathmatical form using our linear algerbra library, see 4.4.
- A wide range of experiments are proceded on Bluecrystal High Performance Comptuer in this research, see chapter 5.

*: Items marked by * are our original ideas and haven't been explored or studied by other researches before.

Acknowledgements

I would like to express my deep gratitude to Peter Flach and Nello Cristianini for supervising my project and appreciate for his comments and assistance throughout this project.

I am deeply indebted to Tim Kovacs whose suggestions helped me in my study in University of Bristol.

I would like to thank my friends and classmates for their help, support and valuable hints.

I also want to express my sincere thanks to my parents for their encouragement.

This work was carried out using the computational facilities of the Advanced Computing Research Centre, University of Bristol - http://www.bris.ac.uk/acrc/.

TABLE OF CONTENTS

1. INTRODUCTION

Enlarging the scale of machine learning algorithms on parallel High-Performance Computers (HPC) has been studied for a number of years and by many researchers. However, managing and allocating resources among distributed computers may never be a simple task. In recent years, there has been an increasing demand for highly scalable text mining tools in industry to meet the age of "internet booming".

Fortunately, excellent pioneer work has been promoted by many commercial companies such as Google, Amazon, and Microsoft, and they are now enjoying the profit that has been brought by their technologies. It is the so called "Cloud Computing" which provides highly flexible computing power on parallel clusters for a large number of daily tasks. Having achieved remarkable successes, these new technologies have questioned whether they have the ability to express complex machine algorithms across a large scale dataset. We believe the answer is "yes" in most situations.

While some argue that the MapReduce is less flexible and programmable compared with traditional technologies, among all these popular techniques, it actually has attracted most interests from scientific researchers for its ability to process large amounts of data. In fact, positive results have been produced in recent years have proved that the MapReduce is very promising in achieving a comparable performance with its parallel competitors, although most of these achievements only involve simple algorithms which include the Naïve Bayes, K-Means, and other straightforward algorithms.

Selecting Machine Learning algorithms to be adapted is closely related to the aim of this project. In this project, we limited our targeted algorithms within Text/Web mining areas, since we believe tasks in this area are most likely to be limited by the capacity of computation and storage.

Considering the outstanding performance on text categorization problems and the major bottlenecks that have been suffered, Support Vector Machine was chosen from an early stage as the target. Inspired by recent research, Non-Negative Matrix Factorization and PageRank have also been chosen for their potential of being parallelized by using MapReduce. To our knowledge, each algorithm we selected has never been discussed for MapReduce related implementation in previous literatures.

Distinguished from earlier approaches, the implementation in our project is based on a set of foundational matrix operators, which offer a scalable and flexible ability to represent complex algorithms.

1.1 AIMS AND OBJECTIVES

The primary goal of this project can be described by two phases: a) constructing an elementary and flexible linear algebra system on MapReduce, b) composing our target algorithms using this system.

The secondary goal continues with our first goal to examine both scalability and correctness of our implementations. A series of experiments is designed and conducted on BlueCrystal cluster, in order to compare the performance with mainstream toolkits and discover the

potential optimal strategies for our parallel algorithm. In this section, a series of trustable datasets have been chosen as they have been provided by earlier researches have been chosen.

1.2 ORGANIZATION OF THIS DISSERTATION

This dissertation consists of the following 6 chapters.

Chapter 1 introduces the motivation and main aims of the project.
Chapter 2 discusses the works related to MapReduce's parallel paradigm and background knowledge of the algorithms to be implemented in our project.

Chapter 3 describes the flexible underlying Matrix Operator System we have designed on MapReduce, and a particular concentration will be given on Matrix Multiplication Operator.

Chapter 4 presents the implementation details of the adapted algorithms, and explore the potential optimal solutions to maximize the benefit that our implementations can obtained from the power of parallelization.

Chapter 5 illustrate results collected from a wide range of experiments which report the performance of our parallelized algorithm and compare with our expectations.

Chapter 6 concludes the project with both achievements and limitations.

Chapter 7 proposes some possible works that might be conducted on the basis of our project.

2. RELATED WORKS

2.1 INTRODUCTION

In this chapter, the Mapreduce parallel paradigm and several algorithms to be adapted are introduced. As the foundation block of our project, the basic workflow and programming framework of MapReduce will be discussed in detail first. Suffering from a large-dense kernel matrix, SVM and its kernel construction phase will be introduced next. Finally, a brief theoretical introduction will be given for NMF and PageRank algorithm respectively. For each algorithm, potential issues after being parallelization will be also discussed accordingly.

2.2 MAPREDUCE PARALLEL PARADIGM

The Internet has brought new challenges to our traditional view on parallelism. With billions of changing webpages every day, the processing on large amount of data is required to finish within a reasonable time. Limited by the capabilities of modern semiconductor manufacturing, people are also pursuing a more scalable framework that can be simply applied to various heterogeneous systems from a small cluster of several commodity PCs to a high performance sever farm with thousands of super computers. Recently, many efforts have been made for constructing hundreds of special purpose parallelization paradigm; however, there is still a great need for a generic computational framework as the reaction to the increasing complexity for both data and algorithms.

2004, Google has implemented a general purpose parallel model "Mapreduce" for large/web scale data processing (Dean and Ghemawat 2004). With two flexibly defined functions "map" and "reduce", this model is inherently suitable for many text processing tasks. Performance has also been largely improved accordingly with the help of data locality. Compared with most existing parallel computing models, Mapreduce do not have a strict limitation of underlying hardware. Clusters can be consisted by either commodity PCs or high-end servers. Therefore, Mapreduce is also designed to be fault tolerant. By using various strategies to recover from task failures, Mapreduce also prevents crashing for occasional input errors that always happen to online data resource.

With the advantages mentioned above, Mapreduce enjoys the increasing popularity among many recent published parallelization paradigms. In addition, statistics shows, approximately 100,000 Mapreduce applications have been implemented outside Google by a number of organizations (Dean and Ghemawat 2010).

2.2.1 PROGRAMMING MODEL

The programming model of Mapreduce has been well summarised by its name "Mapreduce". With a specified map function that processes input data concurrently, and a reducer function responsible for merging intermediate results, the map-reduce model can be very expressive for many real world problems.

To simplify the design of parallel algorithms and improve the performance, the Mapreduce framework relies on the following four assumptions:

- Initial input data is distributed among computing nodes without any particular pattern
- Moving computation is cheaper than moving data.
- Input data are randomly split and fed to nearest computing node.

- Computation always takes a set of input key/value pairs, and produces a set of output key/value pairs.

With these four assumptions, a user-defined function, mapper, only executes on spot where data is available. It takes only one key/value pair as input, and produces a set of intermediate key/value pairs. Mapreduce framework will group/sort these intermediate result and send them to the next stage, reducer.

The reducer, following to the mapper, combines and merges all the intermediate results from all the mappers by their keys and produces final outputs. Reducer takes the values from the same key as an iterative list, in case that the whole list is too large for the memory. Typically, there are less reducer outputs generated than those from mapper.

The following pseudo code demonstrates how we perform a word count program that counts every word appears in a number of documents.

Pseudo Code 1: A simple MapReduce program for word counting, source: (Dean and Ghemawat 2004)

procedure map(key, value)

$[\text{word}_1, \text{word}_2, \ldots \text{word}_n] \leftarrow \text{TokenizeText(value)}$

for all $\text{word}_i \in [\text{word}_1, \text{word}_2, \ldots \text{word}_n]$ do

emit $(\text{word}_i, 1)$

procedure reducer(key, $[\text{value}_1, \text{value}_2, \text{value}_3, \ldots \text{value}_n]$

$\text{count} := 0$

for all $\text{value}_i \in [\text{value}_1, \text{value}_2, \text{value}_3, \ldots \text{value}_n]$ do

$\text{count} = \text{count} + 1$

emit(key. count)

The mapper function simply emits each word with a count of occurrence (1 in this example), and the reducer function sums up these occurrence, and outputs the final results.

Figure 1 shows the general architecture of Mapreduce from a higher level of perspective. The mapper stage tries to use the data locality as much as possible, while shuffle/sort process broadcast grouped intermediate result to their final nodes where all the key/value pairs will be reduced and outputted.

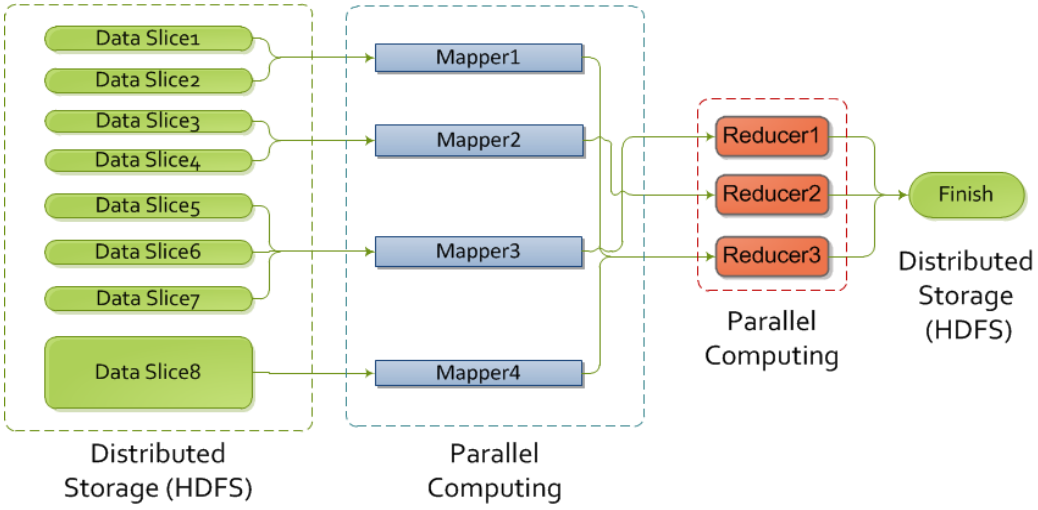

Figure 1: Mapreduce Architecture from Higher Level of View

2.2.2 Data Locality and Fault Tolerance

Since the network bandwidth is usually a scarce resource compared with computing power, the inventors of Mapreduce believe the moving computation is always cheaper than the moving data. This idea of design indicates the Mapreduce grantees not to transfer any data across machines unless absolute necessary (Dean and Ghemawat 2004), (Zaharia, et al. 2008). In this case, a distributed file system, Google File System (GFS) is implemented to ensure the data locality (details section "Data Locality").

Generally, in GFS, each piece of data is stored as its atomic form, block, which is also the smallest transfer unit between computing nodes. Mappers are scheduled in every machine where blocks of data are available. As the first assumption we made in section 2.1, the number of blocks in each node is not fixed, therefore in some extreme cases, auto-balance has to be made before computation. To avoid this situation, GFS will evaluate the disk capability of each node before actual data are stored.

Duplicates are also made when storing data in case of the failure of both software and hardware. Typically, there are three duplicates for each block which are distributed across the machines, and scheduled mapper will attempt to execute the task on one replica. Failing that, a recovery map task will be launched on the other replica of the same input block. When running a large cluster with well-balanced block storage, almost no network traffic is made during the map stage since all data is read locally.

Since Mapreduce library is designed for processing on web scale data where the input resource quality is not guaranteed, the fault tolerant is one of its most important ideas. Mapreduce follows most previous parallel system's design, where a master-slave structure is adopted. Master machine takes the control of each slave machine where actual computation happens, keeps tracking on their eligibility states, and schedules tasks. This design assures

that there is only very small proportion of computation cost on the master machine so that the failure chance is reduced to the minimum level.

Slave Failure

All the failures in Mapreduce fall into two categories: the master failure, and the slave failure, which dominate a large share of the entire failures. Failures may be caused by various reasons, such as data corruption, out of memory, or system error. In the worst situation, slave stops responding the master's schedule requests. Master will mark this slave as disabled and reschedule the incomplete tasks on healthy machines. Similarly, any tasks in progress will be rescheduled on the other machines, as well as the completed tasks. All the rest machines are also notified not to read any data from this machine. However, mostly, a single failure will not crash the machine, where further tasks may be scheduled. In this case, the failed tasks may be simply re-executed on the other machines which satisfy the data locality requirement.

Master Failure

Although given such a single master, failures on that are unlikely, Master periodically builds checkpoint which can help restore schedule information and slaves' conditions. Tasks on slaves will be blocked while they are waiting for the schedule instructions from master. A simple timeout clock can be setup to wait until the master recovers or times out.

2.2.3 PERFORMANCE

- **Terasort Benchmark**

As one of the most challenging tasks in distributed/parallel computing area, sorting of randomly generated numbers remains the benchmark for high performance cluster systems. Jim Gray defined a benchmark (Sort Benchmark Home Page 2010) to compare large sorting programs (Owen O' Malley 2009). Since the distributed sorting is one of the core functions for Mapreduce framework, sort programs can be written within 50 lines Java code. One of the open source implementation of Mapreduce, Hadoop, outperforms many of its competitors and continuously breaks the record of Terasort in the year 2008 and 2009 on Yahoo's Hammer cluster with approximately 3800 nodes. Until Jan 1, 2010, Hadoop still holds the top record of sorting 100 TB in 173 minutes (0.578 TB/min) (Owen O' Malley 2009).

- **Comparison with Relational Database**

As two tools are often used for large scale data process, Mapreduce and parallel relational database are always compared under different system environments and task settings. The latest experiment taken by Andrew Pavlo shows some interesting trade-offs (Pavlo, Paulson and Rasin 2009). Although Mapreduce gains the significant advantage in data I/O and system auto balancing, traditional database system is still observed strikingly better than Mapreduce in many other aspects.

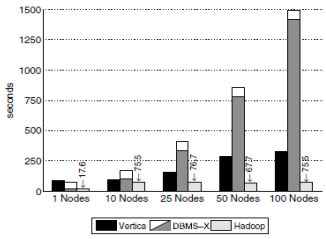

Figure 1: Load Times – Grep Task Data Set (535MB/node)

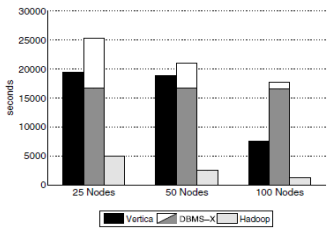

Figure 2: Load Times – Grep Task Data Set (1TB/cluster)

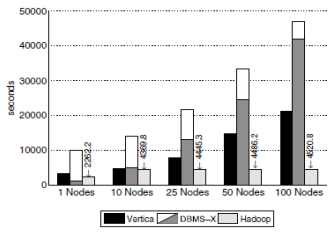

Figure 3: Load Times – UserVisits Data Set (20GB/node)

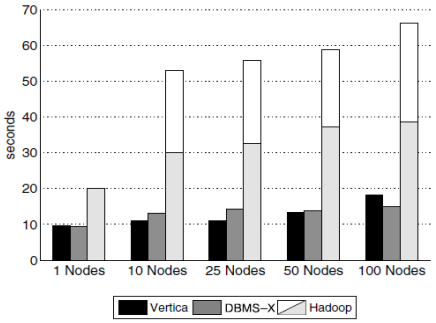

Figure 4: Grep Task Results – 535MB/node Data Set

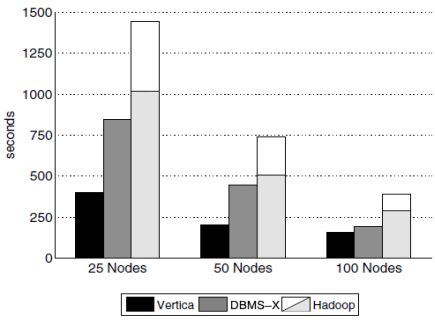

Figure 5: Grep Task Results – 1TB/cluster Data Set

Figure 2: Selected Figure from (Pavlo, Paulson and Rasin 2009) . Experiment has been taken in two stages, Loading Time and Grep Task Results. Source: (Pavlo, Paulson and Rasin 2009)

Interestingly, some recent efforts have been made to integrate the advantages both from Mapreduce and DBMS. By taking the advantage of loading time, many DBMS has implemented the interfaces that make Mapreduce software can easily import them as input data source. Conversely, HBase, a Hadoop based project has also been initiated to provide random access ability and fast queries for Mapreduce programs. Hence, we believe the experience from traditional parallel database can greatly improve the functionality and performance, and there's clear evidence that the APIs of two different systems are fast moving towards each other (Michael Stonebraker 2010).

2.2.4 Distributed File System

Designed for large scale data intensive processing, Mapreduce also requires some essential file system features to provide availability, scalability, and reliability on commodity computing platform. Many cloud computing file storage systems have been supported by Mapreduce, however, the Google File System (GFS) which is also proposed by Google together with Mapreduce, remains to be the best choice in most cases (Ghemawat, Gobioff and Leung 2003).

Driven by the need of large scale storage and massive data processing, a new file system has been modelled to be fault tolerant and highly scalable. In a very large cluster, hardware failures should not be seen as exceptions. GFS is required fast recovery from a series of unpredictable system errors ranging from human mistakes, disk malfunction, network failure, and so on. Moreover, unlike most file systems hosted by operating systems, distributed file systems often have a much larger granularity of storage in order to avoid frequent I/O tasks and network transfer. For a terabytes – level file system (or even petabytes), it is very unwise to maintain billions of small blocks which are only several-kb, instead, larger file (larger than 64MB) blocks are employed in GFS as the atomic unit for the network transfer or auto balancing.

As we mentioned above, Mapreduce can greatly benefit from these features. Under normal conditions, Mapreduce uses GFS as its both input and output storage. Each file divided into fixed-length blocks that are physically stored on local machines which make up our file system. For each block, several replications are produced and spread to different machines. GFS master control machine will take the block information into account when scheduling mapper tasks. When launched, mapper tasks will automatically seek the local replica of corresponding blocks of input files as input. In a large cluster with a well-balanced GFS, most input data are read locally and no extra bandwidth is needed for network transfer.

Like most DBMSs, a logging system and lock system is also built to ensure the data consistency or to recover from a disaster. Compared with most file system, GFS holds a relaxed consistency model that supports many distributed applications. File Namespace Mutations (such as file creation) are guaranteed to be atomic and exclusive. A global order of such operations is defined and maintained in master's log system. Concurrent access to data is depending on the previous mutation type. The defined file operations are based on a single fact that applies to many distributed applications: file appending is more preferred than the overwriting. A typical file writer will generate a file from beginning to the end. Periodically, checkpoints are made to mark the current progress of file operation. Readers are not authorised to read until a file completion signal has been fired or at least one checkpoint has been made. In that case, readers are only accessible to the data written before the last checkpoint was marked. Considering most distributed application paradigm, appending is far more reliable and failure tolerant than random access.

As a core function component of Hadoop, Hadoop Distributed File System (HDFS) has implemented a wide range of file operations described in (Ghemawat, Gobioff and Leung 2003), and is also the default data storage layer for Hadoop Mapreduce which, however, is not only limited for using single underlying file system. Other choices such as Amazon Simple Storage System that provide cloud computing storage are remaining available for Hadoop.

2.2.5 Open-source Implementation: Apache Hadoop [1]

As an apache top-level open source project, supported by Yahoo, AOL, Twitter, and a community with over one thousand companies and individuals, Apache Hadoop provides a complete solution to Mapreduce based data processing problems. Firstly initiated as a parallelism framework for Nutch search engine, it has been widely adopted in a wide range of commercial and academic research projects, such as web indexing, market analysing, and Machine Learning (White 2009).

In a well-known successful business case, New York Times used Hadoop to process ORC programs on terabytes scanned archive newspapers for web browsing. Their entire work finished less than 20 hours on 100 machines on Amazon Elastic Computing Cloud (Amazon EC2) (White 2009) spending only hundreds of dollars. This success will not be archived without the flexible and scalable Mapreduce framework, and user-friendly Hadoop API interface.

- **Apache Mapreduce**

[1] Hadoop: http://hadoop.apache.org/

Mostly inherited from (Dean and Ghemawat 2004), Mapreduce offers A distributed data processing model and execution environment that runs on large clusters of commodity machines using an easy to use API interface. Basically, jobs are supposed to provide the following information before its execution:

- The input source and output destination
- The input and output formant, can be either binary or text
- The definition of mapper and reducer
- A jar file contains all above information using proper written Hadoop program and any supporting library classes

There are also two kinds of processors are defined running on Mapreduce clusters:

TaskTracker is the worker process which actually executes the map or reducer tasks running on each slave machines.

JobsTracker is the controller process which is in charge of the scheduled jobs, and keeps recording on slaves' job processes. It is running only on master machine.

With the help of scalable Mapreduce framework, Hadoop also supports dynamically increase the number of slave machines. Existing jobs will be aware new nodes and spread incomplete tasks onto them.

- **Apache HDFS**[2]

The design of HDFS almost follows the original framework of GFS proposed in (Ghemawat, Gobioff and Leung 2003). Compared with previous parallel file system, HDFS is highly fault tolerant and its target platform is commodity computing equipment. Like Apache Mapreduce, two kinds of processes are established for different purposes.

NameNode process only runs on master machine holding the metadata of the entire file system and providing management and control service.

DataNode process runs on every slave machine providing block storing and retrieval service.

- **Apache HBase**[3]

To provide the ability of random access and schema based storage for Hadoop, a new project called HBase has been launched and is entirely based on Hadoop Mapreduce and HDFS. HBase is basically an open source, column based database for real-time read/write access targeting to build a huge database following Google's Big Table model (Chang, et al. 2006). After years' developing, its random access performance is almost on par with relational database such as MySQL.

2.3 VECTOR SPACE MODEL AND TEXT KERNEL

Last ten years witnessed a golden age of Internet. Boosting Internet users and fast developing web technologies have brought us a flourishing information era. However, the growing data may also become a curse since data always arrives faster than we can process it. Therefore, a number of models and methodologies have been proposed for automatic information mining.

[2] HDFS: http://hadoop.apache.org/hdfs/
[3] HBase: http://hadoop.apache.org/hbase/

2.3.1 VECTOR SPACE MODEL (VSM)

One major obstacle of building text mining problem is that computer almost can not understand human language. One of the earliest attempts, Vector Space Model (VSM) has been presented as SMART system information retrieval system (G. Salton 1971). As a pioneer concept which has been widely used in many text mining systems VSM remains a popular choice for many modern TM tasks. The idea of VSM is to represent a document in a collection as a point in a space (a vector in vector space). Points that are closed to each other are semantically similar, while those points which are distant apart are considered share less similarity. When a query has been made, it will be also translated to a vector, and documents are sorted increasingly as the distance to the given query.

The success that VSM has achieved can be seen from a number of extensive researches that inspired from VSM. Recent efforts have been made to build applications in various areas of Nature Language Processing (NLP). Some of them show that the semantic inference ability of VSM is even comparable with human. Rapp et al. (Rapp 2003) has used a vector based word meaning to achieve 92.5% accuracy in the synonym questions from Test of English as a Foreign Language (TEFOL), whereas the average human score is 64.5%

However, before VSM was proposed, Vector representation has been widely used in Artificial Intelligence. In machine learning, a typical problem is to classify or to cluster a set of items (i.e. examples, individuals, cases, or entities) represented as feature vectors in feature space.

2.3.2 BAG-OF-WORD ASSUMPTIONS: THE DOCUMENT-TERM MATRIX

In this paper, the following notational conventions: Matrices are donated by upper case bold letters **A,** vectors are donated by lower case bold letters **b,** and Scalars are represented by lower case italic *c*.

Like stated above, typical text mining tasks also involves a set of vectors, which are often represented as matrix. A term-document matrix is a matrix whose rows are indexed by documents in corpus, while columns are indexed by terms. The *(i,j)*th entry gives the frequency of term t_j in document d_j.

$$\begin{pmatrix} & T_1 & T_2 & \dots & T_t \\ D_1 & w_{11} & w_{21} & \dots & w_{t1} \\ D_2 & w_{12} & w_{22} & \dots & w_{t2} \\ : & : & : & & : \\ : & : & : & & : \\ D_n & w_{1n} & w_{2n} & \dots & w_{tn} \end{pmatrix}$$

Figure 3: A Document-Term matrix with n rows (articles) and t columns (terms)

Similarly, we can also get term-document matrix **D'** by transposing this document term matrix **D**. Therefore, a matrix **K** that stores the similarity between each pair of documents can be obtained by multiply **D** and **D'. K** is a square where both rows and columns are indexed by documents. The (*i, j*)th entry of matrix K is the similarity between document d_i and d_j.

To construct this matrix, we usually refer to the set of documents as the corpus, and total set of terms as the dictionary. In practical, dictionary is often predefined with a series of frequent words before processing. Thus, we only need to count the occurrence of those words and ignore the others. Hence, we can view a document as bag of words. In mathematics, a bag is like a set, except the duplicates are allowed. For example, a bag $\{a, a, b, b, c\}$ only containing words *a*,*b* and *c* is equivalent to the bag $\{c, b, b, a, a\}$. We can represent this bag with a single vector **X={2,2,1}** by specify the first element of **X** represents the word *a*, the second element of **X** represents the word *b*, and the third element of **X** refers to the word *c*.

The bag-of-word hypothesis first proposed by (Salton, Wong and Yang 1975) is established on a arguably practical experience, which is the frequency (or probability) of words is affected by authors when writing documents of similar topics, which means, the similarity of the frequency of same terms indicates the similar topics of documents. If two documents belong to the same topic, the two row vector of matrix **D** tends to have similar pattern of numbers.

2.3.3 KERNEL: THE DEFINITION AND PROPERTIES

The observation implies that we can apply learning to instances with the indirect evaluation of inner product. The use of kernel means we don't need to pay computational cost on high-dimension similarity comparison.

Considering the inner product in space R^d

$$< \mathbf{x}, \mathbf{y} >= \sum_i x_i, y_i$$

when **x, y** are normalized, the inner product is actually computed the cosine of the angle between **x** and **y.**

Kernel K induces a distance metric d_k as follows:

$$d_k(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{K(\boldsymbol{x}, \boldsymbol{x}) - 2K(\boldsymbol{x}, \boldsymbol{y}) + K(\boldsymbol{y}, \boldsymbol{y})}$$

Where a linear kernel can be got if we replace the K(**x**, **y**) using the <**x**, **y**>. One key difference between kernel measure and distance measure is the kernel measures the similarity while the distance (such as Euclidean distance) measures the dissimilarity.

Linear classifier has a general assumption that our learning instances are linearly separable. However, in many real word applications, most learning problems don't respect this restriction. One solution to this problem is to transform the instances vector **e** in their original feature to some vector $\emptyset(\mathbf{e})$ in a higher dimensional space where they are separable and then apply the inner product.

Then we can use the new kernel in the new space:

$$K(\mathbf{x}, \mathbf{y}) = < \Phi(\mathbf{x}), \Phi(\mathbf{y}) >$$

instead the inner product $< \mathbf{x}, \mathbf{y} >$ in original space. Interestingly, Φ doesn't need to be computed explicitly. Kernel takes the input vectors in original space and output its inner product in transformed space. This trick is called kernel trick which often saves enormous

computational cost as the dimension in the transformed space can be much larger than the original space.

Kernels are basically symmetric positive semi definite matrix, which is also called Mercer Matrix. According to the Mercer's theorem, for every symmetric, semi-positive definite function is a kernel: i.e. there existing a mapping $\emptyset$ such that it is possible to write

$$K(x_1, x_2) =< \emptyset(x1), \emptyset(x2) >.$$

where K is a *kernel function* which returns the inner product of two input vector in some certain feature space, which we don't need to know.

2.3.4 THE KERNEL MATRIX

Given a set of training instances, $S = \{x_1, x_2 \ ... x_l\}$ and a kernel function $K(\cdot\ ,\ \cdot)$, a kernel matrix can be introduced with entries

$$K_{ij} \ = K(x_i, x_j) \ for \ i, j = 1,2, \dots l \ .$$

As we have mentioned in last section, a kernel matrix is a symmetric positive semi definite matrix if kernel function K is a valid kernel function. Obtaining the kernel matrix, learning algorithm can be prevented seeing the actual feature space, as well as the similarity evaluation. Learning from kernel matrix is much more straightforward and time-saving than construct a real high-dimensional feature space, and do the similarity comparison.

The advantage of modularity can be also provided by kernel matrix. Since kernel function and instances evaluation also involves domain knowledge, the construction of them are also apart from typical machine learning domain. Attempting trying to mix these two stages also reduces the universality of leaner. However, kernel matrix offers an elegant solution to this problem by encapsulating background knowledge in kernel matrix. As long as we input a valid kernel matrix, kernel based learning machines can directly infer from these inputs.

In other words, kernel matrix is the interface between the data input and the learning module.

However, the representation of kernels remains an issue. Due to the memory constrains, the kernel matrix may not be stored entirely in memory. In this situation, re-computation may be needed for required matrix entries. Algorithm and implementation may be redesigned for this solution.

2.3.5 KERNEL FOR TEXT

Since learning inference is directly from the kernel matrix, the inputs of our learning system are not necessary to be vectors. Texts, Images, or even any other objects that can be evaluated by a kernel function can be sent to learner as our input, thus, the various kernel functions (Shawe-Taylor and Cristianini 2004) have been proposed for different representation of learning problems recently.

Learning problems with text input are also easy to be *kernelized*. With the help of VSM theory stated above, using the "bag-of-words" hypothesis, a simple text kernel matrix can be constructed as follows:

$$K = D \cdot D'$$

where the matrix D is the document-term matrix, representing a set of vectors. The corresponding vector space kernel function is

$$\mathrm{k}(\mathrm{d}_1, \mathrm{d}_2) = < \emptyset(d_1), \emptyset(d_2) > = \sum_{j=1}^{N} tf(t_j, d_1) tf(t_j, d_2).$$

where the $tf(t_j, d_1)$ means the term frequency of term $\mathrm{t_j}$ in document $\mathrm{d_1}$. More complex kernel can also be built by conducting the Invert Document Frequency (IDF) matrix R, which is a diagonal matrix where stores all term weight as IDF form in its main diagonal.

$$\mathrm{R} = \begin{pmatrix} idf_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & idf_j \end{pmatrix}$$

And the kernel matrix becomes

$$\mathrm{K} = \mathrm{D} \cdot \mathrm{P} \cdot \mathrm{P}' \cdot \mathrm{D}'.$$

This kernel function actually computes the inner products of two vectors multiplying the square of weight of term t (i.e. IDF)

$$\mathrm{k}(\mathrm{d}_1, \mathrm{d}_2) = \emptyset(d_1) RR' \emptyset(d_2)' = \sum_{j=1}^{N} w(t_j)^2 tf(t_j, d_1) tf(t_j, d_2).$$

Furthermore, to include semantic information, word proximity can be included by introducing a matrix S which is an off-diagonal matrix contains non-zero entry at $\mathrm{S_{ij}}$ when term i is similar to term j.

$$\mathrm{P} = \begin{pmatrix} 0 & \cdots & p_{1j} \\ \vdots & \ddots & \vdots \\ p_{1i} & \cdots & 0 \end{pmatrix}$$

We refer to S as the semantic matrix, and define S as

$$\mathrm{S} = \mathrm{R} \cdot \mathrm{P}$$

Thus we have the kernel function with semantic and IDF information

$$\mathrm{k}(\mathrm{d}_1, \mathrm{d}_2) = \emptyset(d_1) SS' \emptyset(d_2)' = \widetilde{\emptyset}(d_1) \widetilde{\emptyset}(d_2)'$$

where we donate the $\widetilde{\emptyset}$ as a new projection function, that carries TF-IDF information rather than the old version which only modelled as bag-of-words assumption.

2.4 SUPPORT VECTOR MACHINES (SVM)

In previous subsection, we have mentioned the application of SVM in text classification.

The idea of building SVM comes from our subjective definition of "good" linear separation hyperplane. A "good hyperplane" should have well generalized boundary and should be computational efficient (be able to learn from 100,000 dimension feature space).

2.4.1 THE MAXIMUM MARGIN AND SOFT MARGIN

As one of the core concepts of SVM, margin hyperplane is easy to understand as a general boundary of instances in linear separable space. A basic rule of choosing such hyperplane is to minimising risk of mistakes. How SVM choose different margin can be viewed as strategies for boundary generalization. As a maximum margin classifier, SVM optimise this bound and by separating the data with the maximal margin hyperplane, and the bound doesn't depend on the dimensionality.

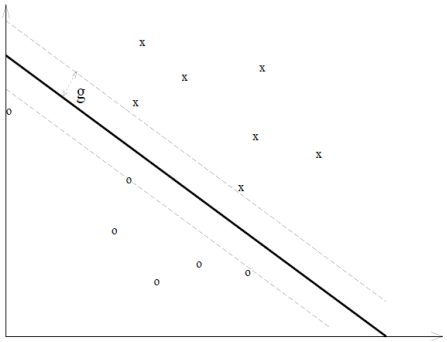

Figure 4 A maximum margin hyperplane in 2-D space

The strategy of finding such hyperplane can be reduced to a convex optimization problem: minimizing a quadratic function under linear inequality constraints. In general, this minimizing problem can be written in the following form:

Given a linearly separable training sample

$$S = ((x_1, y_1), \dots, (x_l, y_l))$$

The hyperplane (w, b) *that solves the optiumization problem*

$$\text{Minimize}_{w,b} \ < \mathbf{w} \cdot \mathbf{w} >$$

$$\text{Subject to} \ \ y_i(< \mathbf{w} \cdot x_i > + b) \geq 1, i = 1,2,3 \dots l$$

Realize the maximal margin hyperplane with the geometric margin $\gamma = 1/||w||_2$

More generally, this problem can also be transformed to a dual problem with a linear decision function by substituting the **w** with the form

$$\mathbf{w} = \alpha_1 y_1 \mathbf{x}_1 + \alpha_2 y_2 \mathbf{x}_2 + \cdots + \alpha_i y_i \mathbf{x}_i$$

where α is the Lagrange multipliers, such that we can write **w** as follows

$$\boldsymbol{w} = \sum_{i=1}^{l} a_i y_i \boldsymbol{x}_i$$

Now, our linear decision function can be written as

$$f(x) = < \boldsymbol{w}, b > + b = \sum a_i \, y_i < \mathbf{x}_i, \boldsymbol{x} > + b$$

In this dual representation form, data only appears inside the products, where the dimensionality is not necessarily important. And this is also where kernel function K can be introduced.

$$f(x) = < \boldsymbol{w}, b > + b = \sum a_i \, y_i K(\mathbf{x_1}, \mathbf{x_2}) + b$$

2.4.2 SVM IMPLEMENTATION

In previous subsection, we have talked that the training of SVM can be reduced to the maximising a convex quadratic from subject to linear constraints. Since such convex quadratic problems have no local maxima, their solution can always be found efficiently. The problem of minimising differentiable functions of many variables has been systematically studied. However, in many cases, the standard methods are not able to perform because of some particular features of SVM. For example, we have discussed how SVM can be effectively adopted even in a very high dimensional feature space. But the size of kernel matrix is also its drawback, since the size of kernel matrix grows quadratically with the sample size. Hundreds of megabytes may just fit for a few thousand examples.

Gradient Accent is simplest numerical solution of a convex optimization problem is obtained by gradient descent, sometimes known as the steepest descent algorithm. This algorithm is an iterative method that starts at initial estimate $\boldsymbol{\alpha}_0$ and then updates the vector **w(α)** follows the steepest direction. The gradient of **w**(α) will be evaluated at each step for the update next iteration. A learning rate η will be specified to control the length of update. In sequential version of this implementation, the approximation will be evaluated for just one pattern at a time, and update the single parameter α_i can be written by the increment

$$\delta\alpha_{\mathrm{i}}^{t} = \eta \frac{\partial \boldsymbol{W}(\alpha^t)}{\partial \alpha^i}$$

where η is a learning rate. Often, this learning rate is chosen as a function of time, or a function of the input pattern has been gained. Too large η will cause the system oscillate without converge, however, too small step length will cost too many iterations before converge.

As the gradient accent needs the kernel matrix to be stored in to the memory, other strategies have been proposed to avoid such problems. One of the SVM implementation that has the great popularity is **Sequential Minimal Optimization (SMO)** (Platt 1999). SMO takes the advantage of the natures of the support vector problem to reduce the optimization step to its minimum form: updating two α_i weights at one time. Then the computation can be batched to update the KKT optimality conditions for the rest of the weights, and find two maximally violating weights, which are then updated in the next iteration until convergence. The optimality can e tracked through

$$f_i = \sum_{j=1}^{l} a_j y_j \Phi(x_i, x_j) - y_i$$

which is constructed as the learning algorithm processes.

2.5 MATRIX FACTORIZATION

The processing of many text mining systems involves many non-negative high dimensional matrices. As a result, industry urges academy for methodologies of discovering patterns from

massive matrix data. Low-rank matrix factorization is a fundamental component of machine learning, underlying regression, factor analysis, and dimensionality reduction and clustering algorithms. Compared with principle components analysis, matrix factorization enforces a much strong constraint on components selection. Approaches designed for exploring the latent structure of matrix has been studied for years. There are many forms of matrix factorization, and (Singh and Gordon 2008)offers a unified view of several important factorizations including Singular Value Decomposition (SVD) and Non-Negative Matrix Factorization (NMF).

In general, the matrix factorization for a single matrix can be written as X ≈ f$(\mathbf{UV}^{\boldsymbol{T}})$, where the f is a prediction link, **U** and **V** are matrices factorized from the original matrix. The factorization often varies at the choosing of f, the definition of ≈, and the constrains of **U** and **V**.

Recently, Lee and Seung(2000) investigated the properties of algorithm on NMF and proposed a simple and efficient method. It provides us a brand new vision of implementing a paralleled version of NMF algorithm.

In general, the algorithm of NMF can be considered by solving the following problem:

Given a non-negative matrix **A**, find non-negative matrix factors **W** and **H** such that:

$$\mathbf{A} \approx \mathbf{WH}$$

Before the formal definition has been introduced, the cost function is defined by two approaches. One typical cost measure for both **A** and **B** using Euclidean distance can be written in the following form:

$$||\mathbf{A} - \mathbf{B}||^2 = \sum_{ij} (A_{ij} - B_{ij})^2$$

This value is lower bounded by zero, can clearly equals to zero when **A=B**. Another measure is also used for divergence is

$$D(\boldsymbol{A}||\boldsymbol{B}) = \sum_{ij} (A_{ij} \log \frac{A_{ij}}{B_{ij}} - A_{ij} + B_{ij})$$

Similar to the cost function mentioned above, this measure vanishes when **A=B,** however, it cannot be called as a distance measure, since this is not symmetric for **A** and **B**.

Now the NNMF problem can be considered as two forms of problems:

Problem 1 Minimize $||\boldsymbol{A} - \boldsymbol{WH}||^2$ with respect to **W** and **H**, subject to the constraints $\boldsymbol{W}, \boldsymbol{H} >= 0$.

Problem 2 Minimize $D(\boldsymbol{A}||\boldsymbol{WH})$ with respect to **W** and **H**, subject to the constraints

$\boldsymbol{W}, \boldsymbol{H} >= 0$.

Two approaches have been proposed for different problem settings and proved to be non-increasing when updating two factors **W** and **H**. The whole algorithm updates iteratively, after specific precision has been reached.

The rules of updating the first problem can be written as below:

$$H_{\alpha\mu} \leftarrow H_{\alpha\mu} \frac{(W^T V)_{\alpha\mu}}{(W^T W H)_{\alpha\mu}},\ W_{ia} \leftarrow W_{ia} \frac{(V H^T)_{ia}}{(W H H^T)_{ia}}$$

Compared with the additive rules derived from gradient descent:

$$\mathrm{H}_{\alpha\mu} \leftarrow H_{\alpha\mu} + \eta_{\alpha\mu}[\textstyle\sum_i W_{ia} \frac{V_{i\mu}}{(WH)_{i\mu}} - \sum_i W_{ia}],$$

The multiplicative method replaces the learning rate $\eta_{\alpha\mu}$ with $\frac{H_{\alpha\mu}}{\sum_i w_{ia}}$. Although the value of $\frac{H_{\alpha\mu}}{\sum_i w_{ia}}$ is not often small, it can be proved that this algorithm converges and is faster than the additive method. Detailed definition and solution in parallel will be given in 4.3.

2.6 LINK ANALYSIS AND PAGERANK

Links between documents have proved a convenient way of navigating reader to their interested topics. In last decades, an increasing number of document format (e.g. HTML) containing Hyper-Links have been widely published on internet and other electronic media. It wasn't until 1998 that more and more people realised that the connections between documents may potentially carry useful information that may be used to explore the popularity and authority of documents.

During the period of 1997-1998, two influential algorithms are invented, and are still using among various informatics systems and online search engines. PageRank (Brin and Page 1998), invented by Sergey Brin and Larry Page, powers the dominant internet search engine Google while HIT algorithm is derived from Social Network Analysis. Both two algorithms exploit the links system between documents according to their prestige and authority.

These two link analysis approaches are based on two basic concepts of social network analysis: **centrality** or **prestige**.

The degree of centrality (C_d) can be defined by the formula below:

$$C_d(\mathrm{i}) = \frac{d(i)}{n-1}$$

where the $d(i)$ is the degree of node i normalized by n, which is the total number of nodes.

This simplified centrality may have one major disadvantage when used on real-world applications. The degree of a node may be faked when create several webpages containing large collection of links (so called "link farm") point to the node.

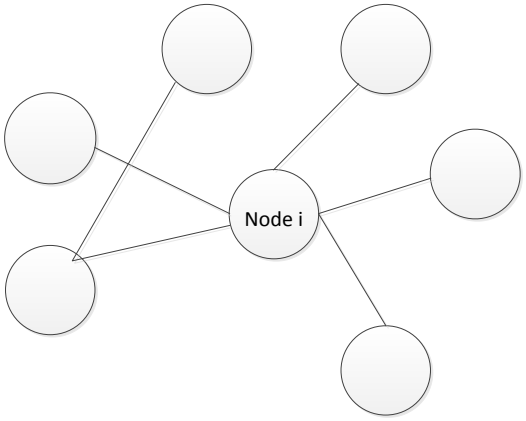

Figure 5: A typical social network with a central node i

However, this disadvantage has been overcome by PageRank which is known as its anti-spam ability. A PageRank of a single node u can be defined by the following formula:

$$\mathrm{PR(u)} = \sum_{v \in B_u} \frac{PR(v)}{l(v)}$$

where B_u is a collection of outlinks of page u , $l(v)$ is the total number of outlinks in page V, and $PR(v)$ represents the PageRank of page v. By defining the PageRank recursively, the problem mentioned earlier can be easily solved. The axiom of PageRank is distinguishing the income links by its authority. An example has been shown below:

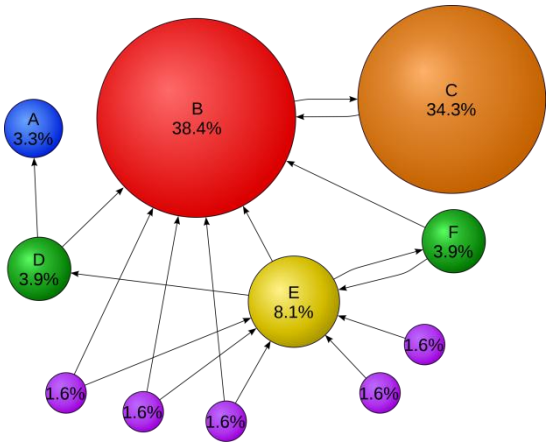

Figure 6 A PageRank share distribution of a small network from Wikipedia

The above graph illustrates the PageRank shared by a very small network. As we can see, although the node C receives only two incoming links, it takes up 34.3% share of total PageRank since both two links are from the most authoritative node B. In contrast, the node E receives 5 links while it only takes up 8.1% of total PageRank as the sources of its in-comining links are isolated and not recognizable by other pages.

It has been studied the PageRank system can be described by stationary probability of a finite Markov Chain defined by a stochastic matrix, which is the adjacent matrix describing the links between documents (B. Liu 2007) (Langville and Meyer 2008).

Parallel PageRank has also been studied on various systems. Gleigch et al. developed a new approach that converges faster than the simple power method and are less sensitive to the changes in teleportation (Gleich, Zhukov and Berkhin 2004). Kohlschutter et al. also illustrate a method that can finish calculation on a web-scale dataset in minutes (Kohlschutter, Chirita and Nejdl 2006). However, to our knowledge, no similar efforts have been done on MapReduce.

There are other algorithms that have been developed for document ranking. HIT algorithm (Kleinberg 1999) is a ranking system that dependent to the user's query. Two rankings of documents will be generated according to the authority ranking and hub ranking while user issues their queries.

2.7 SUMMARY

In this chapter, we mainly introduced the MapReduce framework, and several typical Text/Web mining algorithms and techniques that will be parallelized in our project. We mainly focus on MapReduce paradigm, SVM and Kernel related issues which inspired us most to build a more powerful linear algebra system to solve the major issues which their sequential counterparts suffered from. After addressing the theoretical background, the detailed techniques and implementation approach will be introduced in chapter 3 and 4.

3. CORE MATRIX OPERATORS ON MAPREDUCE

3.1 INTRODUCTION

In this chapter, we present the parallel methodologies for matrix multiplication, matrix transpose, and a general purposed framework for elementary-wise operations. These matrix operators consist of a general foundation of the later work on algorithm implementation.

Linear Algebra operators have been proved efficient in Machine Learning tasks, and have been parallelised using a number of distributed computing frameworks. As a foundational support to our machine learning implementations, a range of matrix operators have been adapted for flexibility and performance considerations.

Considering the unique workflow of the MapReduce and the characteristics of modern text mining algorithms, we redesigned the computing framework to fit the actual demands of learning tasks while taking the full advantage of MapReduce paradigm.

Our efforts illustrate a flexible and standardized computing framework where sophisticated algorithms can be further developed without re-inventing basic components. First in this chapter, we will discuss some principle components of our implementation, after which our various operators corresponding to different linear operations will be introduced.

To our knowledge, the matrix multiplication is the first time to be adapted on MapReduce.

3.2 MATRIX REPRESENTATION

Before any operation can be implemented, we must first design the actual data structure for the matrix operands, which is used to disk storage or network transfer. In our project, the sparse matrix representation is chosen for matrix expression in all cases, since the sparse matrix (e.g. document-term matrix) is widely adopted in many text mining tasks.

Considering the memory consumption and the ability of both iterative and random access, the matrix is represented by a **List of Maps** where the matrix is organized by rows, in which each entry is stored by a (column, value) tuple. This representation improves the performance of iterative reading by row (in natural order), and the random access by column (in hash order).

Although this implementation increases the cost of doing iterative reading by columns, the advantages of choosing this representation are obvious. Concerning the external storage structure on disks, it is much convenient to organizing the matrix by row, rather by columns, because the files on disks can be easily read by lines, and it is natural that data are requested row by row. Even if the column traverse is required, the transpose of a single matrix can be simply constructed (see 3.4.1).

3.2.1 DICTIONARY OF KEYS

For simplicity and readability of text mining tasks, English characters are also allowed in matrix representation as the index of rows and columns. However, this may cause problems when the numeric row/column indices are required.

To assign a positive integer number to rows and columns, we designed a small database which is called **Dictionary of Keys** to contain the indices of rows and columns. One positive integer will be assigned to each row and column according to its stored order in this database.

This dictionary is implemented by a Berkely DB (Olson, Bostic and Seltzer 1999), which is an in-memory database used for a fast access to entries.

Additionally, this dictionary also helps to decrease the cost of iterative-access to non-zero entries by column, since the order of columns has already been specified in dictionary.

For efficiency, each machine of MapReduce cluster maintains a copy of this dictionary, therefore the size of the dictionary is crucial for fast initialization and distribution.

3.2.2 FILE FORMAT ON DISK

In order to maximise the readability and the compatibility with the popular machine learning toolkit Weka (Mark Hall 2009), we simplifies the sparse text representation of Weka's dataset file, Attribute-Relation File Format (ARFF) (Attribute-Relation File Format (ARFF) 2002).

Compared with original representation, we removed the "head" section which is usually used for attribute name declarations, since the declarations may be too verbose to list in some high-dimensional learning problems.

In our representation, each line describes a single row of matrix which is enclosed by two round brackets. Leading by a special identifier "~0" and the row index, the non-zero entries of this line are enumerated and separated by commas. Each entry is written in the form of "column value" pairs, for example, the line below

$$\{\sim 0\ 1, 2\ 1, 3\ 4\}$$

shows the row "1" of matrix contains two non-zero entries in the column "2" and "3" given the value 1 and 4 separately.

For convenience of text mining tasks, non-number indices of rows and columns are also allowed in matrix representation, for example a Vector Space Model of a single web document containing three words may be shown as below:

{~0 abc.com/web1, football 0.5, score 0.2, Bristol 0.9}

3.3 MATRIX MULTIPLICATION

3.3.1 METHODOLOGY

Suffering from the high time complexity ($O(n^3)$ *for squared dense matrix n by n*), matrix multiplication have been widely discussed under many parallel settings. Various designs have been proposed to reduce the running time while many of them are relies on the inter-machine broadcasting availability in some certain distributed paradigms. However, in MapReduce, the "shuffle" stage is the only step that data can be distributed around machines.

Chao Liu, 2010, proposed a matrix multiplication approach on MapReduce for a sparse matrix and narrow matrix. (Liu, Yang and Fan 2010) For a multiplication between a narrow matrix **A** and a sparse matrix **B** which contains only a few columns, they introduced a special partition schema on matrix **A** that split it into a number of row vectors, and multiply these vectors with each column at matrix **B**. This algorithm can be represented by the following formula:

$$C_j = \sum_{i=1}^{m} B_{ij} * A_i = \sum_{i \in \mathbb{O}_j} B_{ij} * A_i$$

where C_j stands for the jthcolumn of result matrix C, and A$_i$represents the i^{th} row of matrix A. They conducted their algorithm as two steps: a) split matrix **A** and combine corresponding row and column vector, b) shuffle vectors to their destination machine and compute the final result. Liu's algorithm can utilize the sparsity of matrix **A** and the narrowness of matrix **B** to perform an efficient computation.

By generalizing the concept "partitioning schema" proposed in their algorithm, we designed a widely capable approach which can be adopted to any kind of matrix multiplication operation.

The very basic idea of our approach is to partition two operand matrixes into small-individual blocks (or slices) that are fit for in-memory calculation and then combine them into computational groups by following partitioned matrix multiplication algorithm, after which, the matrix multiplication for each group are proceeded on different nodes in distributed fashion. In the last step, the generated results are shuffled and summed by reducers to generate our final result. The high-level pseudo code are listed in Pseudo Code 2,Pseudo Code 3 and Pseudo Code 4, since the complexity of our algorithm, this implementation is highly abstracted.

Figure 7 shows a very simple naive matrix multiplication algorithm, and is used to demonstrate our matrix multiplication methodology[4].

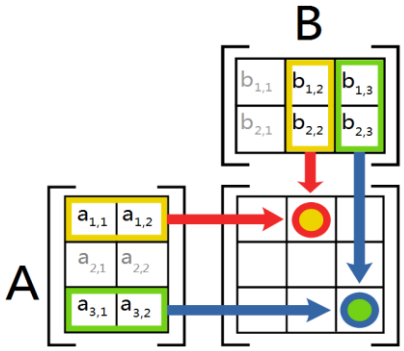

Figure 7 computing a single cell of result matrix requires horizontal blocks of matrix A and vertical blocks of matrix B. (Illustration from Wikipedia)

In order to calculate a single partition of result matrix **C**, horizontal blocks from the first operand matrix **A** and vertical blocks from the second operand matrix **B** must be retrieved, and the block $\boldsymbol{C_{ij}}$ in result matrix **C** can be obtained by summing a series of addends:

Equation 1

$$C_{ij} = A_{i1}B_{ij} + A_{i2}B_{2j} + \cdots + A_{in}B_{nj} = \sum_{k=1}^{n} A_{ik}B_{kj}$$

[4] This graph is originally used to demonstrate the normal matrix multiplication without partitioning. However, it performs the exact same algorithm with the partitioning multiplication, since it can be simply imagined as partitioning multiplication with 1 by 1 blocks.

Marked by different colour, each addend in this equation is regarded as a basic parallel component that can be run in parallel as their computation is independent from each other.

One major challenge is to partition original matrix into smaller blocks and combine the blocks belongs to the same addends as one unity. Since each addend is distinguished from other addends from their horizontal and vertical indices, these indices are used to identify the blocks that belong to the same addend. According to Equation 1, the indices from one addend follows the pattern

$$\boldsymbol{A_{ab}} \times \boldsymbol{B_{bc}}\ contributes\ to\ \boldsymbol{C_{ac}}$$

where three unique indices (6 in total) appears and forms the **key** of matching and sorting all the blocks generated from matrix partitioning.

Pseudo Code 2 illustrates the partitioning procedure on Map stage, where the matrix **A** and **B** are read by row. Row vectors are split and bind with its **key**.

Pseudo Code 2: High-level matrix multiplication partitioning algorithm on MapReduce

Map Stage:

Input: row vector from Matrix A: $\boldsymbol{A_i}$*, and Matrix B:* $\boldsymbol{B_i}$

if input = $\boldsymbol{A_i}$

split $\boldsymbol{A_i}$ *into a set of sub-vectors* $\{\mathbf{A_{i1}}, \mathbf{A_{i2}}, \dots, \mathbf{A_{in}}\}$ *according to matrix partitioning schema*

for each $\boldsymbol{A_{ix}} \in \{\boldsymbol{A_{i1}}, \boldsymbol{A_{i2}}, \dots, \boldsymbol{A_{in}}\}$

output a tuple $< c_a, c_c, a_b, \boldsymbol{A_{ib}} >$

For $\boldsymbol{B_i}$*, do the same*

Pseudo Code 3 Reduce Stage: each tuple will be sorted by c_i, c_j, a_j in parallel.

Input: row slices from Matrix **A** *and* **B** *shares the same* c_a, c_c, a_b

using these slices to construct sub matrices $\boldsymbol{A_{ab}}$ *and* $\boldsymbol{B_{bc}}$

output tuple $< c_a, c_c, \boldsymbol{A_{ab}}, \boldsymbol{B_{bc}} >$

In the reduce stage, sorted by the **key**, input slices of row vectors coming from the same blocks of sub matrices that contribute to the same result sub matrix $\boldsymbol{C_{ac}}$ will be grouped and combined to form two complete sub matrices $\boldsymbol{A_{ab}}$ and $\boldsymbol{B_{bc}}$ together with their targeted block index c_a, c_c in result matrix as a tuple.

The next stage is much easier to be implemented. The generated tuples will be calculated correspondingly among distributed machines, and final result matrix can be obtained after summing up addends targeting at the same sub matrix $\boldsymbol{C_{ac}}$ in result matrix **C** at reduce stage. The pseudo code is given in Pseudo Code 4.

Pseudo Code 4: High-level matrix multiplication calculation algorithm on MapReduce

Map Stage:

Input: tuple $< c_a, c_c, a_b, \boldsymbol{A_{ab}} >$ *and* $< c_a, c_c, a_b, \boldsymbol{B_{bc}} >$

$$\boldsymbol{C_{ac}} = \boldsymbol{A_{ab}} \times \boldsymbol{B_{bc}}$$

output $\boldsymbol{C_{ac}}$

Reduce Stage: generate final result matrix C using chunks outputted by Map stage

Pseudo Code 4 can be well summarised by Figure 8.

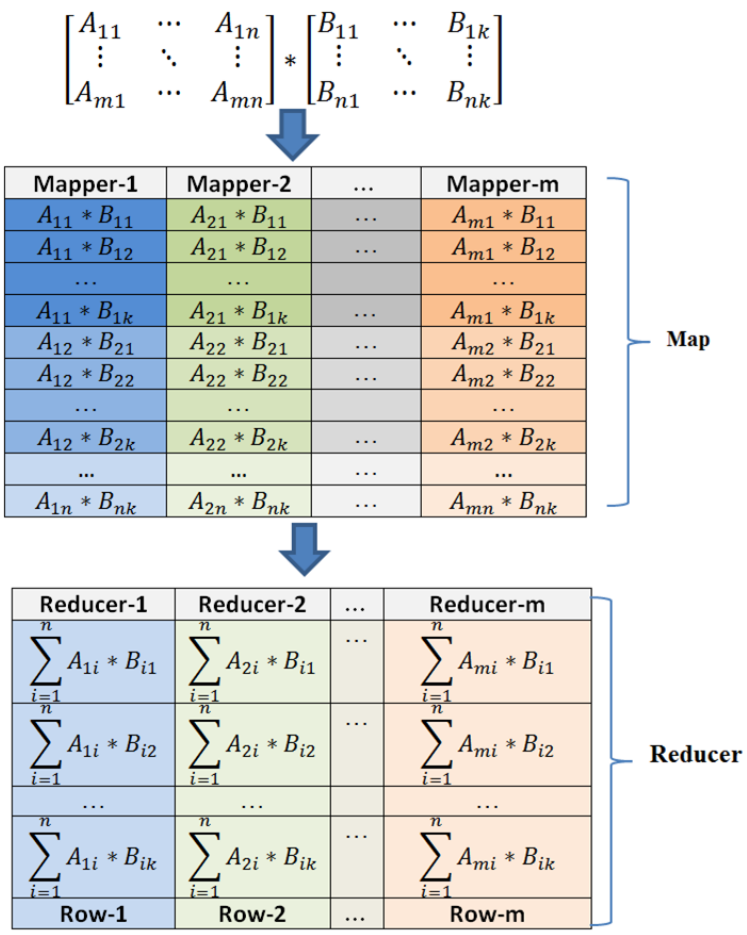

Figure 8: Matrix Multiplication on MapReduce

The whole calculation illustrated above is actually done by two pipe-lined MapReduce sub-jobs. First job is responsible for matrix splitting and partitioning while the second one is set for actual multiplication and summation works.

Compared with the traditional Message-Driven distributed computing, the major advantage of MapReduce is data-locality which can prevent the network traffic overhead during the "shuffle" stage. However, the machine partitioning strategy and shuffling rule should be carefully chosen before the locality is enabled.

3.3.2 MATRIX PARTITIONING AND INTERMEDIATE RESULTS

The major issue for most matrix multiplication is the large amount of intermediate results which are generated by the sub-matrix multiplication. For example, two dense square matrices both partitioned into four matrices may generate four sub-matrices before the final summation, which means the intermediate data is twice as large as the result matrix. The maximum number of sub-matrices generated by the multiplication of two matrices ***A*** and ***B*** can be calculated by the following formula:

Equation 2

$$N_{sub} = \alpha \times \beta \times \gamma$$

where three parameters are the number of partitions on rows of **A**, columns of **A**, and columns of **B**. Clearly, for two dense square matrices that follows the same partitioning strategy, the space complexity is $O(n^3)$.

This formula shows, although multiplication for large matrices can be partitioned into smaller blocks and performs the calculation simultaneously, the space complexity is cubed. Fortunately, in most cases, the matrices are extremely sparse, so the actual number blocks will be much less than the formula given above. In order to find the balanced trade-off between time and space, a series of experiments are designed to reveal the relationship between partitioning parameters and running time.

Experiment settings and results are discussed in Chapter 5.

3.3.3 MACHINE PARTITIONING STRATEGY[5] AND DATA LOCALITY

One solution to matrix multiplication on MapReduce seems complete, but one piece is still missing. We described the partitioning operation in Pseudo Code 2, and state the pieces of row vectors will be "shuffled" to different machines. However, the method of "shuffling" is not specified.

The "shuffling" stage is formally named as "Machine Partitioning" in our project. It provides precise control of distributing intermediate result generated from Map stage, and plays an important role as scheduler in this parallel paradigm. Data locality can be maximized if an optimal machine partitioner has been chosen.

The initial motivation for introducing partitioned matrix multiplication is to reduce the memory consumption on each individual machine, however, side effects accompanying with this approach are also difficult to eliminate by normal distributed paradigms.

According to Equation 2, the size of intermediate result that generated from matrix partitioning is decided by three factors: the number of partitions on row of **A**, the number of partitions on columns of **A**, and the number of partitions on columns of **B**, which are denoted by α, β, γ respectively.

[5] In our project, two partitioning strategies are particularly studied: machine partitioning strategy and matrix partitioning strategy. In this section, only machine partitioning strategy is discussed.

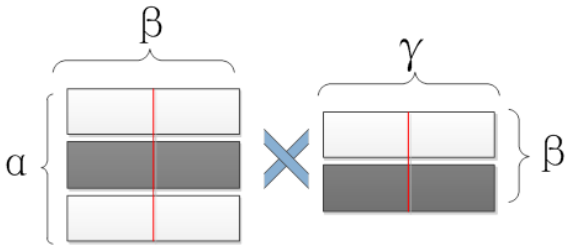

Figure 9: The illustration of α, β, γ. In this example, α=3, β=2, γ=2

Clearly, according to the algorithm, the size of the intermediate result increase as a linear function of $\alpha \times \beta \times \gamma$. For equal α, β, γ, the intermediate result increase cubically. Therefore, a clever strategy needs to be chosen for less inter-machine communication.

Before addends have been distributed, partitioning method (i.e. partitioner) assigns each addend with appropriate computing nodes. For ideal partitioners, these intermediate results will be shuffled only once, and will not be moved around machines again. Obviously, the best strategy is to shuffle the addends to where their calculation result will finally lies on. This idea is the main initiative of using the row based partitioner.

For comparison purpose, two partitioners are designed in our project regarding to the features provided by MapReduce. Concerning various problem settings, both of them have advantages and disadvantages in different circumstances.

One partitioning strategy that can be called "Random Partitioning" behaves exactly as its name said: distribute all the addends to a computing node by random criteria (e.g. Hash function). The partition number ρ for an addend $A_{ik}B_{kj}$ can be represented by the following equation:

$$\rho = |h(i,j,k) \; mod \; r|$$

Where *h* is a hash function and *r* is the number of computing nodes (i.e. reducers). This arrangement ensures that all computing nodes have averaged load if *i, j, k* conform normal distribution.

However, to get the final result C_{ij}, we have to transfer all addends that contains intermediate result to a single node where they will be summed and the C_{ij} will be stored. This will cause a severe network congest at the shuffle stage as all sub matrices in matrix **C** are requiring for their addends, and this is where the three cubed space complexity takes effect.

The ideal solution to this problem is all addends are calculated where they are going to be summed, thus a row based partitioner can be formed:

$$\rho = i \; mod \; r$$

This row partitioner ensures that after the addend is distributed, all the addends targeting on the same sub matrix with horizontal index i will be calculated on one single machine. The space complexity is further reduced form $O(\alpha \times \beta \times \gamma)$ to $O(\alpha)$.

However, this method also suffers from some cases. In most cases, the distribution of non-zero cells is not regular, and some patterns may apply. This partitioner partitions our parallel

components according to their positions in the result matrix, which may cause a computational unbalance. Furthermore, this partitioner can only maximize the parallelisation only when $i \geq r$, this constrain is more strict than the that of random partitioner which is $i \times j \times k \geq r$.

3.3.4 INLINE CALCULATION

The frequent use of light-weighted MapReduce job should be avoided, since the MapReduce framework is designed for large batched jobs, for which the overhead of initialization may be ignored. However, for short jobs with small amount of data, the use of a single MapReduce job is not economical. One solution to this problem is to "embed" small calculations into large operators such as matrix multiplication, so that simple process on input or output matrix may be executed without re-running a new MapReduce job.

The inline calculations of matrix multiplication are designed at three different stages of the multiplication, which are pre-computing stage, post-computing stage and multiplication stage.

- Pre-Computing stage: each row from these two matrices will be processed before being partitioned. Users can choose the specific operation for different multiplier respectively.
- Post-Computing stage: each row generated from the matrix multiplication will pass through the post processor before output.
- Multiplication stage: for any addends appeared in Equation 1, a multiplication function *m* is provided instead of the basic multiplication operator, so the equation () can be written in the following form:

$$C_{ij} = m(A_{i1}, B_{ij}) + m(A_{i2}, B_{2j}) + \cdots + m(A_{in}, B_{nj}) = \sum_{k=1}^{n} m(A_{ik} B_{kj})$$

These inline calculations enable the ability of changing the behaviour of normal matrix multiplication, and these variations of multiplication are widely employed among the algorithms we have adapted (e.g. Kernel Construction and Pagerank). Details will be introduced in the next chapter.

3.4 OTHER MATRIX OPERATORS

3.4.1 MATRIX TRANSPOSE

In linear algebra, the operator transpose gives by the following definition:

Definition 1

If $\boldsymbol{A} = [a_{ij}]$ *is a m ×n matrix, then the transpose of* **A,** *then the* ***transpose*** *of* **A,** $\boldsymbol{A} = [\boldsymbol{a_{ij}}]$ *is an n ×m matrix defined by* $a'_{ij} = a_{ji}$*, Thus the transpose of* **A** *is obtained from* **A** *by interchanging the rows and columns of* **A.**

Therefore the procedure for matrix transpose on MapReduce is simple and elegant:

Pseudo Code 5: The Map stage of matrix transpose operator

Input: *row index i, elements array* $[< c_1, v_1 >, < c_2, v_2 > \cdots < c_n, v_n >]$

Output: row index and new single element

Begin:

foreach vector $< c_i, v_i > \in [< c_1, v_1 >, < c_2, v_2 > \cdots < c_n, v_n >]$

output row index c_i, *single vector* $< i, v_i >$

End

Input: *row index* c_i, *collection of elements* $\{< i_1, v_1 >, < i_2, v_2 > \cdots < i_n, v_n >\}$

Output: row index and new elements array

Pseudo Code 6: The Reduce stage of matrix transpose operator

Begin:

optional*: sort collection of elements*

output row index c_i, *output element array* $[< i_1, v_1 >, < i_2, v_2 > \cdots < i_n, v_n >]$

End

3.4.2 General Purposed Element-wise Operators

Despite the two algorithms mentioned above, there are a number of elementary-wise linear algebra algorithms that are commonly used by Machine Learning tasks, and most of them have a very flexible usage, which may be changed to fit different implementations. Therefore, to increase the elasticity of operators and reduce the work for creating new operators, a simple model is designed for several general purposed operators. This model is described as a single MapReduce job and the elementary operators must follow the general protocol below:

Pseudo Code 7: Protocol for general purposed element-wise operators

Input: a sequence of multiple matrices which must be in the exact same size

$[m_1, m_2, m_3 \ldots m_n]$

Output: a single matrix m_{result}

Begin:

foreach matrix m_i *in matrix sequence*$[m_1, m_2, m_3 \ldots m_n]$

foreach row i and non-zero elements array $[e_1, e_2, e_3 \ldots e_n]$ *in matrix* m_i

map(i, $[e_1, e_2, e_3 \ldots e_n]$*)*

foreach row i in matrix m_{result}

let c be a collection of rows $\{r_1, r_2, r_3 \ldots r_n\}$ *which are emitted by the map stage*

reduce(i, c)

End

In our project, two categories of algorithms have been implemented with this framework.

- Algorithmic Operators
  Most algorithmic operators can be well represented by the above framework. Elementary-wise operations such as addition, subtraction, multiplication and division can be encapsulated in the form of reduce function where a set of rows from the input matrices sequence that shares the same row index will be processed accordingly.
  Sequential operation combination is also feasible to apply to this framework without splitting computation into several MapReduce jobs. The computation will be operated as the order specified in input matrix sequence. In our project, a combination for elementary-wise algorithmic operators is designed. Several more complicated compute procedures have also been developed for some complicated algorithms (e.g. Alpha Updater in SVM implementation).
  The only issue for this protocol is the duplicated operands are not supported since duplicate input source is not allowed in current Hadoop implementation[6]. However, this functionality can be developed as soon as the "Symbolic Link" is supported by the next version of Hadoop 0.21 (What's New in Apache Hadoop 0.21 2010).
- Line-wise Processing
  The line-based data filter is also necessary for some operations, although it may be replaced by inline-calculation so that the MapReduce initialization overhead can be avoided.
  In our project, the 1-norm and 2-norm normalization for line vectors are implemented through this paradigm. Column normalization can be also simply achieved by transposing the original matrix.
  Several specially designed operators (e.g. diagonalization) have also been implemented for supporting of particular algorithms.

3.5 SUMMARY

Supported by MapReduce paradigm, a new matrix operator system is established for higher-level representation of Machine Learning algorithms. Considering the computing complexity and special parallel design, the matrix multiplication and transpose are highlighted in this chapter separately. Various light-weighted elementary-wise operators are also implemented under the general purposed framework.

The naïve way of multiplication is used in this parallel implementation; however, it also takes advantages from the sparsity of matrices because of the "list of maps" matrix representation. To reduce the space complexity, matrix multiplication minimizes the inter-machine communication by using the MapReduce data locality feature and row-based partitioning strategy.

For conveniences and flexibility, the implementations of other operators are designed to fit the "General Purposed Operator Protocol", which describes the general behaviour on input, output and the MapReduce functions.

[6] The duplicated operands can be implemented by assigning the same input data folder two different symbol links, therefore, they can be regarded as two different inputs.

The usage and parameter tunings of these operators are introduced in the next chapter. Performance issues are discussed in Chapter 5 where the linear speedup rate will be given.

4. MACHINE LEARNING ALGORITHMS ADAPTED

4.1 INTRODUCTION

In this chapter, we concentrate on the major issue of our project: enlarging the scale of selected machine learning algorithms by using the matrix operators we have introduced in last chapter. As we have expected, with the help of linear algebra system created in last chapter, the implementation of our algorithms can be very intuitive and effective.

For simplicity of implementation, we choose three straightforward approaches for each algorithm: a) the gradient descent approach for SVM, b) the multiplicative update approach for NMF, c) and the additive iterative approach for PageRank. All the implementations are derived from their original mathematical form.

4.2 PARALLEL SUPPORT VECTOR MACHINES

The Parallelizability of SVM has been discussed by many previous researches (Burges 1998), (Graf, et al. 2005) and (Zeyuan Allen Zhu 2009). In 1998, Burges summarised three promising ways of scaling up the support vector machines:

"These training algorithms may take advantage of parallel processing in several ways. First, all elements of the Hessian itself can be computed simultaneously. Second, each element often requires the computation of dot products of training data, which could also be parallelized. Third, the computation of the objective function, or gradient, which is a speed bottleneck, can be parallelized (it requires a matrix multiplication)."(Burgers, 1998, p147-148)

On the basis of his ideas and the nature of SVM, we separate the whole training process into two independent parts: **kernel construction** and **QP solving** so that the advantage of parallelization can be maximized, and other kernel machines adapted in future can also benefit from the parallelization of the kernel matrix.

4.2.1 TEXT KERNEL MATRIX CONSTRUCTION

In order to obtain the kernel matrix, a document-term matrix is required first. For simplicity considerations, the Bag-Of-Words assumption is adopted first, as described in 2.3.2, a document-term matrix can be described as the following matrix:

$$\boldsymbol{D} = \begin{pmatrix} tf_{11} & \cdots & tf_{1n} \\ \vdots & \ddots & \vdots \\ tf_{m1} & \cdots & tf_{mn} \end{pmatrix}$$

The matrix $\boldsymbol{D}$ may be obtained easily by MapReduce, since the procedure can be well expressed as a Map-Reduce process where the terms are extracted during map stage, while the reduce operation only focuses on organizing the order of rows in this matrix. The whole process is illustrated by Figure 10.

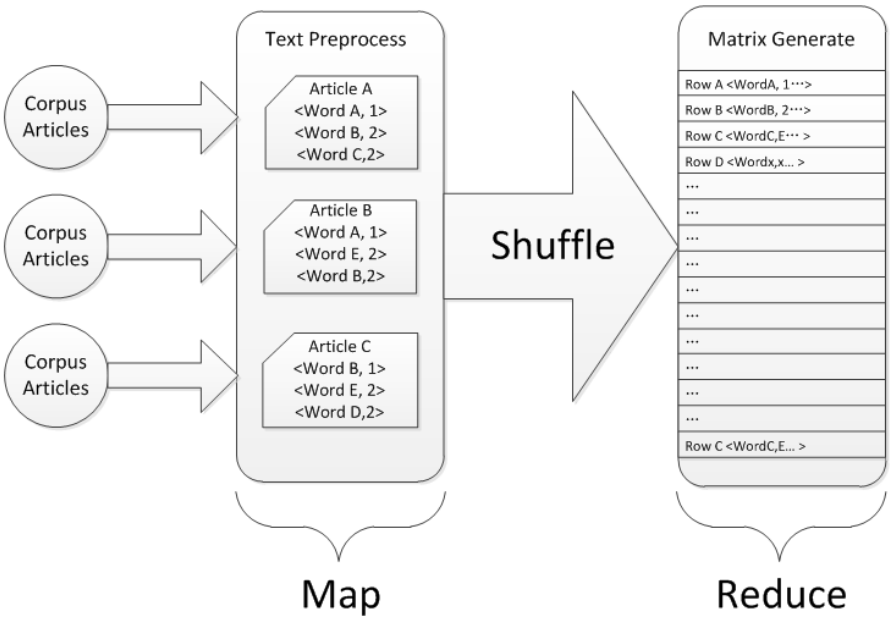

Figure 10: The MapReduce procedure of creating document-term matrix

During the Map stage, the corpus are analysed and extracted from their original representation. The words will be punctuated and stemmed at this step, after which they will be counted within each document. Since these operations for each document have no further connections with other documents in corpus, the Mappers can only focus on their own analysis jobs and output the intermediate results to reducers. After conversion, the whole documents are represented by a single row of vector which indicates the frequencies of words that appear in this document.

The reducer of this operation needs not to be explicitly specified, and the default reducer automatically assigned by Hadoop can organise the order of row vectors to find a suitable machine where they will be stored after the completion of current MapReduce workflow.

However, this naïve Bag-of-Words assumption often fails to express the latent meaning of words. This motivates a more complex version of assumption called "Vector Space Mode (VSM)" described in 2.3.1.The method that used in our project is a basic version of VSM (while the sophisticated implementation is also feasible and easy to adapt), which also considers the weight of each individual term appeared in documents in addition to the frequency-only version.

The weight of term can also been expressed by a diagonal matrix which is given by the Inverted Document Frequency (IDF).

$$D = \begin{pmatrix} tf_{11} & \cdots & tf_{1n} \\ \vdots & \ddots & \vdots \\ tfm1 & \cdots & tfmn \end{pmatrix} \times \begin{pmatrix} w_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & w_n \end{pmatrix}$$

The document-term matrix maps the original corpus into a high-dimensional space which can be further reduced by introducing the **Document-Frequency-Cut (DF-cut)** techniques. Similar techniques have been introduced for reducing the number of similarity pairs within corpus. (Tamer Elsayed 2008)

The aim of DF-Cut is to eliminate the words which appear in documents either too frequently or too rarely. Too frequently used words may appear in a large collection of documents and have no contribution to topic identification, while words which appears only several times can also be ignored since the words may simply be a misspelling or name entities (e.g. people's name, location) that are not well known for public. By setting zeros to the corresponding elements on the diagonal, these terms will be removed from the final document-term matrix ***D***.

The Document Frequency Matrix (DF-Matrix) can be obtained as soon as the document-term matrix has been constructed. By using a mapper that emits each term occurring in corpus with a single value "1", and a reducer that counts the total number of occurrence, MapReduce can be used to do the DF counting conveniently. DF-Cut can also be integrated into this workflow by ignoring the entry which DF is larger / smaller than a certain threshold, and this ignorance is automatically interpreted as zero in sparse representation.

Two MapReduce jobs will be involved in TF/IDF matrix construction. The first job constructs DF-Matrix, and the second is responsible for the multiplication of DF matrix and Document-Term matrix using matrix operator.

At the final step, the multiplication can be preceded by using the two-stage approach described in 3.3.1. However, in this case, the entire matrix B is suitable for memory access on each computing node. Therefore, a simplified version of matrix multiplication can be derived from the previous design.

Since the second operand is very sparse, and normally takes several mega-bytes to store, it can be loaded in memory on each node before the actual MapReduce job is launched. This strategy is equivalent to take the second operand as a unity block without any partitioning strategy on it. As the partitioning has been omitted, the cost of data transfer is reduced, as well as the row and column indexing operation.

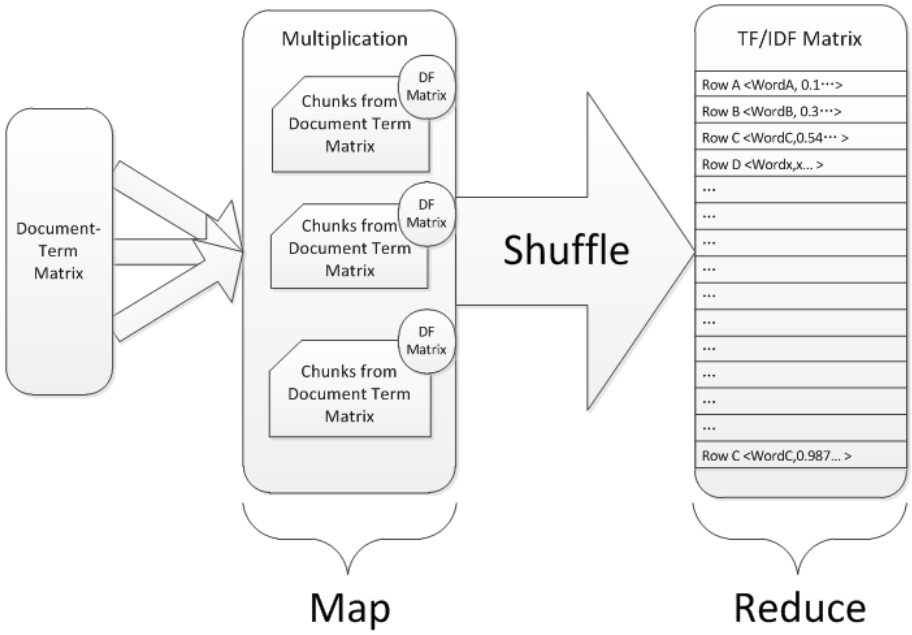

Figure 11: The MapReduce procedure of constructing TF/IDF matrix

Basically, this method can be adopted on any matrix multiplication in which the second operand is small and very sparse.

In our project, we only consider the simplest form of kernel matrix, which is a linear kernel with the entry i, j is inner product of vector instance $\mathbf{x_i}$ and $\mathbf{y_j}$.

$$\boldsymbol{K(x_i, y_i) = < x_i , y_j >}$$

and the matrix can be easily calculated by a simple matrix multiplication

$$\boldsymbol{K = D \times D'}$$

At this stage, the matrices are both sparse and high-dimensional, and neither of them is capable for in-memory storage, so the standard matrix multiplication discussed in 3.3.1 is adopted for scalability reasons. Since this multiplication only involves with the elements in matrix $\boldsymbol{D}$, we were trying to implement an algorithm that can directly make use of the matrix $\boldsymbol{D}$ without explicitly computing the transpose of it. However, the fact that we realised later indicates the matrix partitioning is going to generate the exact same size of intermediate result so that the performance will not be improved significantly.

4.2.2 Gradient Descent for Support Vector Machines

"Divide and Conquer" (Chunking) has been the main approach of parallelizing SVM. It aims to split the whole large dataset of training instances to several smaller and feasible dataset to find the support vectors and naturally, the support vectors within each dataset is only a small proportion of original chunk in most cases. One practical model proposed by H.P Graf (Graf, et al. 2005)suggests that the parallelization can be achieved by using a cascaded approach which is similar to Map and Reduce paradigm by using gradient descent.

Derived from the solution described in 2.4.2, the gradient $\boldsymbol{G = \Delta W(\alpha)}$ of $\mathbf{W}$ with respect to $\boldsymbol{\alpha}$ is:

Equation 3

$$\mathrm{G_i} = \frac{\partial \boldsymbol{W}}{\partial a_i} = -\mathrm{y_i} \sum_{j=1}^{l} y_j a_j \boldsymbol{K}(\boldsymbol{x_i}, \boldsymbol{x_j}) + 1$$

This algorithm updates the vector $\boldsymbol{\alpha}$ gradually until convergence.

Compared with the SMO, gradient descent has several major advantages when implemented with MapReduce.

- **Batched Process on Massive Amount of Data**
  The main issue that obstructs the development of SVM on standalone machines is the size of kernel matrix. However, we solve this problem by adopting an efficient matrix multiplicative approach by using MapReduce which uses a batched process on kernel matrix to compute the gradient. Within each iteration, our method precedes a complete swipe among the entire kernel matrix, which takes the full advantage of MapReduce.
  SMO also gradually updates data. However, only two points are selected to be updated on each iteration. Although it walks around the kernel issue, for MapReduce, the advantages of huge I/O throughput among machines cannot be utilized.
- **Fast Convergence**

Compared with SMO which only updates two $\boldsymbol{\alpha}$ values at each time, the gradient swipes the whole kernel matrix and updates the entire $\boldsymbol{\alpha}$ vector every iteration. Thus we can expect a much faster speedup rate than SMO. Since the initialization cost for MapReduce increases as the number of iteration grows, the fast convergence can lead to a fast solution.

- **Simple Design**
  On one hand, the convergence speed of SMO largely depends on the heuristics of selecting points being updated in the next iteration. Unfortunately, most heuristics are too sophisticated to be expressed using parallel computing framework.
  On the other hand, SMO have a strong dependency across different stages of optimization. The points that selected for optimization have strict order which cannot be shuffled for batched parallel computing.
  By using the gradient descent, the overall design can be greatly simplified for the using of naïve steepest strategy according to its linear algebra representation.

According to the Equation 3, the gradient can be rewritten using linear algebra format as follows:

$$\boldsymbol{G^t = 1 - y.*(Q \times \alpha^T y)}$$

where **Q** is kernel matrix and $\boldsymbol{y, \alpha}$ represents the label vector and Lagrangian multipliers respectively. The operator ".*" means elementary wise multiplication, while the "×" still stands for normal product. The vector of Lagrangian multipliers $\boldsymbol{\alpha}$ can be calculated by gradient descent:

$$\boldsymbol{\alpha^t = \alpha^{t-1} + \eta .* G^t}$$

where $\boldsymbol{\eta}$ is a set of learning rates (steps) for $\boldsymbol{G}$ at current iteration **t**. To ensure each step never leaves feasible region, we set $\eta_i = \frac{\omega}{\boldsymbol{K(x_i,x_i)}}, \omega \in (0,2)$ **,** where ω is a hand-chosen coefficient and is used to implement successive over-relaxation.

To sum up, the described algorithm can be written into pseudo code which listed below:

Pseudo Code 8: Support Vector Machine (1-norm soft margin)

Given*: a kernel matrix* $\boldsymbol{Q}$ *which conform a Mercer's function* $\boldsymbol{K}$**,**

a label vector $\boldsymbol{y} = [\boldsymbol{y_1}, \ldots \boldsymbol{y_n}], \boldsymbol{y} \in \{1, -1\}$

Compute *learning rate vector* $\boldsymbol{\eta}$ *where* $\eta_i = \frac{\omega}{\boldsymbol{K(x_i,x_i)}}$.

$\boldsymbol{\alpha = \eta}$

Repeat

$\boldsymbol{G^t = 1 - y.*(Q \times \alpha^T y)}$

$\boldsymbol{\alpha \leftarrow \alpha + \eta * G^t}$

until meet stop criteria

return $\boldsymbol{\alpha}$

In our project, this algorithm is simply implemented by the matrix operators we created in chapter 3.

One limitation of this design is the linear constraint $\sum_i^l \alpha_i y_i = 0$ which will be violated during the update. This linear constraint derived from optimising the bias *b* in decision function. In this chapter, the algorithm we discussed have already decided a fixed bias in prior (in our experiments, this bias is often set to 0), so that this linear constraint need not to be enforced.

In next chapter the stopping criteria will be discussed for this gradient descent implementation.

4.2.3 Learning Rate and Stop Criteria

Since we are optimising the QP problem, there are suitable strategies of choosing $\boldsymbol{\eta}$ that makes the algorithm converges.

When optimizing the quadratic function of α_i, the update cause of the derivate going to zero is

$$\widehat{\alpha_i} \leftarrow \alpha_i + \frac{1}{\boldsymbol{K}(\boldsymbol{x_i}, \boldsymbol{x_i})}\left(1 - y_i \sum_{j=1}^{l} \alpha_j y_j \boldsymbol{K}(\boldsymbol{x_i}, \boldsymbol{x_j})\right)$$

A feasible value of learning rate can be found:

$$\boldsymbol{\eta} = \frac{1}{\boldsymbol{K}(\boldsymbol{x_i}, \boldsymbol{x_i})}$$

In our project, a variant of this learning rate is used for implementing successive over-relaxation giving the update:

$$\widehat{\alpha_i} \leftarrow \alpha_i + \frac{\omega}{\boldsymbol{K}(\boldsymbol{x_i}, \boldsymbol{x_i})}\left(1 - y_i \sum_{j=1}^{l} \alpha_j y_j \boldsymbol{K}(\boldsymbol{x_i}, \boldsymbol{x_j})\right.$$

In practical, the parameter ω is set to 0.2.

4.3 Non-negative Matrix Factorization

Efforts have also been made for scaling up NMF, and many delicate algorithms have been designed. In our project, we focus on a solution for extremely large dataset that haven't discussed by other researchers.

Considering a large web dataset that is not available to store in memory and dataset is distributed among machines, existing solutions must be adapted to fit the situation.

The precise NMF is described by the following definition:

Definition 2 (Non-negative Matrix Factorization)

Given $A \in R+^{m\times n}$ and a positive integer $k \leq min\{m, n\}$, find $W \in R+^{m\times k}$ and $H \in R+^{k\times n}$ such a divergence function $D(A||\tilde{A})$ is minimized, where $\tilde{A} = W \times H$ is the reconstructed matrix from the factorization.

In this project, for simplicity, we only consider Gaussian Non-negative Matrix Factorization (GNMF) which takes

$$A_{i,j} \sim Gaussian(\widetilde{A_{i,j}}, \sigma^2)$$

Lee and Seung (2000) present a multiplicative approach to find W and H iteratively for GNMF, which can be expressed by the following linear algebra formulas:

$$A_{i,j} \sim Gaussian(\widetilde{A_{i,j}}, \sigma^2)$$

$$H \leftarrow H.* \frac{W^T A}{W^T W H}$$

$$W \leftarrow W.* \frac{A H^T}{W H H^T}$$

where the ".*" is used for element-wise matrix multiplication operators.

This multiplicative algorithm is feasible for our matrix operators system. However, special design is also needed for performance considerations.

In real web scale dataset, matrices **W** and **H** are two dense matrices with small number of rows and columns respectively, while **A** is often an extreme sparse matrix storing TF/IDF information. These characters of matrices can be well utilized to boost the multiplication.

Since the update of **H** and **W** is symmetric, for simplicity, the update of **H** is selected to illustrate our algorithm.

- $\boldsymbol{W^T A}$**:** The computation of $\boldsymbol{W^T A}$ can be optimised when choosing appropriate matrix partitioning strategy. Since the $\mathbf{W^T}$has only a few number of rows, and a large number of columns, the good strategy that can reduce the size of intermediate result is partitioning by columns while no partition is made by rows. In fact, this technique can also be used for matrices that one is sparse and the other is narrow. Moreover, one basic partitioning strategy for common multiplication can also be formed from this example: split the matrix according to the largest side and try to preserve the smaller side as one whole block.

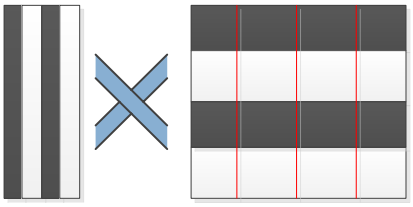

Figure 12: The matrix partition strategy for $\boldsymbol{W^T A}$

- When it comes to the denominator, whether the computation of $\boldsymbol{W^T W}$ or $\boldsymbol{WH}$ takes first has a great influence on performance. The computation of $\boldsymbol{WH}$ multiplies two dense matrices and generates a giant denes matrix which is not acceptable in most circumstances. In contrast, $\boldsymbol{W^T W}$ may be much easier to be calculated which only generates a very small $k \times k$ matrix which is donated as **X**. The next stage can be simply achieved by multiplying one small matrix **X** and one flat and wide matrix **H** which only contains *k* rows. As described above, optimization can also be made

when the matrix partitions following the columns of $\boldsymbol{H}$, while the rows of $\boldsymbol{H}$ can be regarded as a unity. As the first operand is a very small $k \times k$ matrix, it is not necessary to do any further partitioning on it.

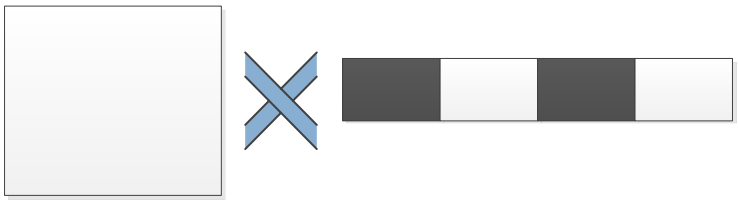

Figure 13: The matrix partition strategy for XH

- At the final stage, three *k*×*n* matrices will be connected by two operator ".*" and "./" to generate the final matrix **H**. This calculation exactly follows the protocol of General Purposed Operator we have discussed in the last chapter, and it can be achieved by this operator efficiently and elegantly.

Major issue for this implementation is the starting overhead that caused by the large number of MapReduce jobs. For each iteration of updating both **H** and **W**, 13 jobs are submitted; while some jobs are very light-weighted (running time is shorter than 1 min). The extra cost of time defeats the purpose of using the parallelization. Further breakdown of the time cost and the speedup rate will be discussed in 5.5.

4.4 PAGERANK (EIGEN VECTOR)

As we have mentioned above, the PageRank problem can be described by a Markov Chain, where **A** is a transition probability matrix which gives the probability of changing from different states. Given the initial probability distribution of each state $\boldsymbol{p_0} = (p_0(1), p_0(2), \dots p_0(n))^T$ a column vector and a n× n transition matrix **A,** we have

$$\sum_{i=1}^{n} p_0(i) = 1$$

$$\sum_{j=1}^{n} A_{\mathrm{ij}} = 1$$

By the Ergodic Theorem of Markov Chain, a finite Markov Chain defined by the stochastic transition matrix **A** has a unique stationary probability distribution if A is irreducible and aperiodic**.** Therefore, an iterative solution to this problem can be described by the following equations.

Definition 3

Given an initial probability $\boldsymbol{p_0}$ *the probability distribution after one iteration can be get from*

$$\boldsymbol{p}_1 = \boldsymbol{A}^T \boldsymbol{p}_0$$

At iteration k+1, the distribution $\boldsymbol{p_{k+1}}$ *can be calculated by*

Equation 4

$$\boldsymbol{p}_{k+1} = \boldsymbol{A^T p_k}$$

and we have

$$\lim_{k\to\infty} \boldsymbol{p_k} = \boldsymbol{\pi}$$

where $\boldsymbol{\pi}$ is the principle eigenvector of matrix $\boldsymbol{A}^T$ with the eigenvalue of 1.

As shown above, this algorithm can also be implemented by our matrix operators.

The general procedure of calculating PageRank can be divided by two stages where the first one generates an initial probability distribution of PageRanks randomly and the second calculates PageRank according to the Equation 4 iteratively until some certain criteria has been satisfied. Two simple rules have been used in our project: a) fixed number of iterations has been preceded; b) certain level of precision has been reached.

The algorithm discussed above can be summarised by the following pseudo code:

Pseudo Code 9: Parallel PageRank Implementation

Given a Markov Chain Probability Transition matrix A^T, number of webpages N,

initialise $\boldsymbol{p} = [\frac{1}{N}, \frac{1}{N} \dots, \frac{1}{n}]$

repeat

$$\boldsymbol{p_{new}} = \boldsymbol{A^T p}$$

$$\boldsymbol{p} = \boldsymbol{p_{new}}$$

until stopping criteria has been met

However, beside the implementation addressed above, another algorithm is also feasible in mathematics. This method regards the PageRank $\boldsymbol{p}$ and $\boldsymbol{p_{new}}$ as row vectors rather than column vectors, thus the calculation inside the loop can be written as $\boldsymbol{p_{new}} = \boldsymbol{pA}$, and the transpose $\mathbf{A}^{\boldsymbol{T}}$ is not needed at the beginning. Although these two approaches are mathematically equal, only the first one we mentioned in Pseudo Code 9 is practical in parallel computing.

As our matrix is organized by row on disk, each row is shuffled to one single reducers as a unity. In the latter situation, the entire PageRank vector $\boldsymbol{p_{new}}$ will be stored on single computer while the solution in Pseudo Code 9 distributes vector elements to different machines averagely. This disparity may cause huge performance difference for the parallel computing adopted at the reduce stage.

Another major advantage of the earlier solution is the suitability of row based partitioner. As we have mentioned in 3.3.3, row based partitioner requires a larger *i* to maximize the parallelization (for $i \geq r$ constraint). This prerequisite is satisfied in the earlier algorithm where the matrix A^T is huge while in the later one, the left operand is a row vector which serializes our parallel implementation on single machine and defeat the purpose of using the row based partitioner.

The "dumping factor" β is also applied in our implementation by using the post computing stage of inline-calculation. For each iteration of updating vector $\boldsymbol{p}$, an inline function is adopted as Pseudo Code 10:

Pseudo Code 10: The inline calculation of β correction

Input: PageRank for webpage i: p_i*; total number of webpages: N,*

Output: PageRank after β correction: p_i'

$$p_i' = p_i \times d + \frac{1-d}{N}$$

output p_i'

4.5 SUMMARY

In this chapter, we have presented three typical machine learning algorithms that have been implemented in our project. Represented by our matrix operators, most implementation can be derived directly from their classic mathematical solution with an intuitional and elegant expression.

However, the maximization of parallelism needs a particular configuration specific to each algorithm. In order to examine the performance of our implementations and discover the optimal settings to each algorithm, a series of experiments and analyses are designed and conducted in the next chapter.

5. EXPERIMENTS AND RESULTS

5.1 INTRODUCTION

This chapter consists of several performance experiments that have been conducted in our project.

In the first phase of our experiments, we focus our experiments on matrix multiplication in order to explore the potential optimal setting among a wide range of parameters and reveal the relationship between time performance and the size, sparsity, and partitioning strategy of input matrices. The final objective is to form a practical guide for choosing optimal parameters.

The next stage of the experiment is designed for the adapted algorithms. These experiments are conducted for two purposes: a) verifying the correctness of our implementations, b) illustrating performance impact brought by parallelization. Evaluations and analyses will be given after each group of experiments respectively.

5.2 THE BLUE CRYSTAL CLUSTER

All the experiments mentioned in this Chapter are preceded on the BlueCrystal High Performance Computing Cluster[7], phase 2.

Having achieved a performance of 28.4 TFlop/s, BlueCrystal was placed 66th in the Top500 in June, 2008.[8]

BlueCrystal phase2 follows the hardware specification below:

- 416 nodes each with 2.8 GHz Intel Harpertown E5462 processors, memory 8 GB RAM per node (1 GB per core)
- QLogic Infinipath high-speed network
- IBM's General Parallel File System (GPFS) providing data access from all the nodes
- Data storage - 73TB SATA storage.

In order to cope with the queue based resource allocation system, our Hadoop jobs are wrapped by a standard tasks script where the command of allocate and de-allocate operation is hard coded. Although more convenient methods can be adopted, limited by the software and environment used in Bluecrystal, it cannot be used before the next upgrade of Hadoop. The major defect that it brings to our experiment is the speedup experiments may apply and de-allocate the cluster frequently, which may cost several days before our job is actual running.

5.3 EXPERIMENTS ON MATRIX MULTIPLICATION

In this set of experiments, the matrix multiplication operator is examined for scalability and effectiveness. All the experiments are tested against a randomly generated matrix.

Three factors are mainly concerned in our experiments for their implementation features and the huge influence it may bring on performance.

[7] http://www.bris.ac.uk/acrc/
[8] http://www.top500.org/list/2008/06/100

- Size of Input Matrix
- Sparsity of Matrix
- Strategy of Matrix Partitioning

5.3.1 DATASET CONSTRUCTION

With the help of MapReduce, a huge matrix $\boldsymbol{A} = \mathrm{R}^{m \times n}$ that required in the experiment can be generated in parallel by using the matrix generator we implemented on MapReduce.

This dataset generator requires four parameters which include sparsity δ, distribution of elements in each row, and the height m and width n of the matrix. By default, $\delta = 2^{-7}$, $m = 2^{13}$ and $n = 2^{13}$.

In this group of experiments, all the elements in generated matrices are uniformly distributed and are between 0 and 1. The height and width are equal, and both are in a factor of 2 in order to demonstrate the potential non-linear relationship between each experiment variables. The square matrix is also straightforward in the space and time complexity test.

The major advantage of generating a matrix by using MapReduce is the output matrix from a MapReduce job which will be well balanced among all the computing nodes, which means the work load for each Mapper will not be significantly distinguished. Therefore, the data locality can be enabled at the beginning.

This generator is also employed for other purposes to generate particular matrix (e.g. random initial values), which may have different requirements compared with the experiments listed here. The verbose of the implementation details are skipped for simplicity.

5.3.2 THE SPACE AND TIME COMPLEXITY W.R.T **m**

In this chapter, the time and space complexity of matrix multiplication will be explored and understood. The discussion is limited on matrix multiplication operator, for its leading complexity ($O(n^3)$ for squared matrix) that outstands among all operators.

Implemented from the naïve algorithm, the complexity of our operator remains the same. However, in most cases (particularly in Text Mining applications), the matrices are often extreme sparse although they usually have very high dimensions. Considering this fact, the actual computational complexity should be much smaller than the theoretical complexity.

In this experiment, the performance of a matrix multiplication is investigated through 6 tests, each of which proceeds a multiplication between two randomly generated squared matrices $\boldsymbol{A}, \boldsymbol{B} = \boldsymbol{R}^{\boldsymbol{m} \times \boldsymbol{m}}$. The parameters remained at their default settings.

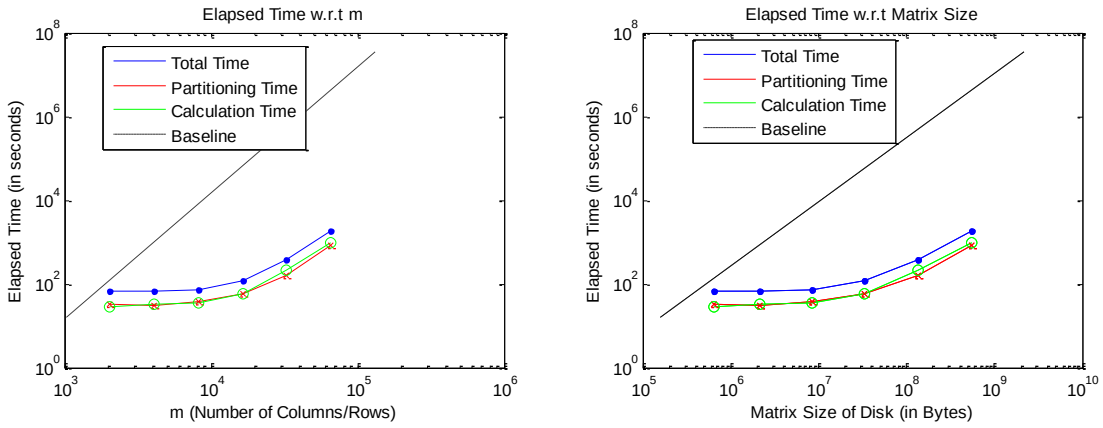

Figure 14 illustrates the elapsed times w.r.t the number of rows/columns and the input matrix size. Both are plotted in Log-Log space. In two pictures, baselines are lines through the first blue points with slopes = 3, 2 respectively. Partitioning time and calculation time are also plotted along with the total elapsed time.

Two graphs illustrated generally agree with the assumptions we have made earlier. As can be seen from the two charts, all lines grow with increasing slopes as the *m* (size of matrix) grows. Unfortunately, due to the capacity of Bluecrystal, the experiments fail to continue for two 131072×131072 matrices, which generate a huge amount of intermediate results that exceed the storage limit of cluster.

The time complexity for different input matrix scale can be obtained by calculating the slopes of the curve in different sections. Clearly, in both graphs, the complexities are almost constant at the very beginning of curves, which shows that, for small matrices, the scale of input matrix have only slight influence on performance and, the capacity of clusters has not been saturated.

The "Baseline" is plotted as a theoretical benchmark for each graph. On the first graph, according to the computational complexity of naïve matrix multiplication, the elapsed time should increase cubically as the number of rows/columns grows. The baseline shown on the first graph goes through the first blue point and illustrates the theoretical prediction of elapsed times for the next experiments. According to the cubic complexity, the slope of this line is set to 3. A similar story can be found on the second graph whereas the slope is set to 2 for the quadratic correlation between file size and elapsed time.

From these charts we can see that the gap between the theoretical prediction and actual results are huge. In the first graph, for all sections of the curves, slopes are lower than the predict. Even at the final stage of the curves, the slope is approximately 2 compared with 3 of the prediction baseline. This gap can be explained by the sparsity of input matrices for which a large number of zero cells are not stored and actually computed.

These two graphs also illustrate that the total running time is averagely shared by the partitioning and calculation stage.

In order to study the storage requirements and space complexity of our implementation, the size of intermediate results and result matrix is plotted below.

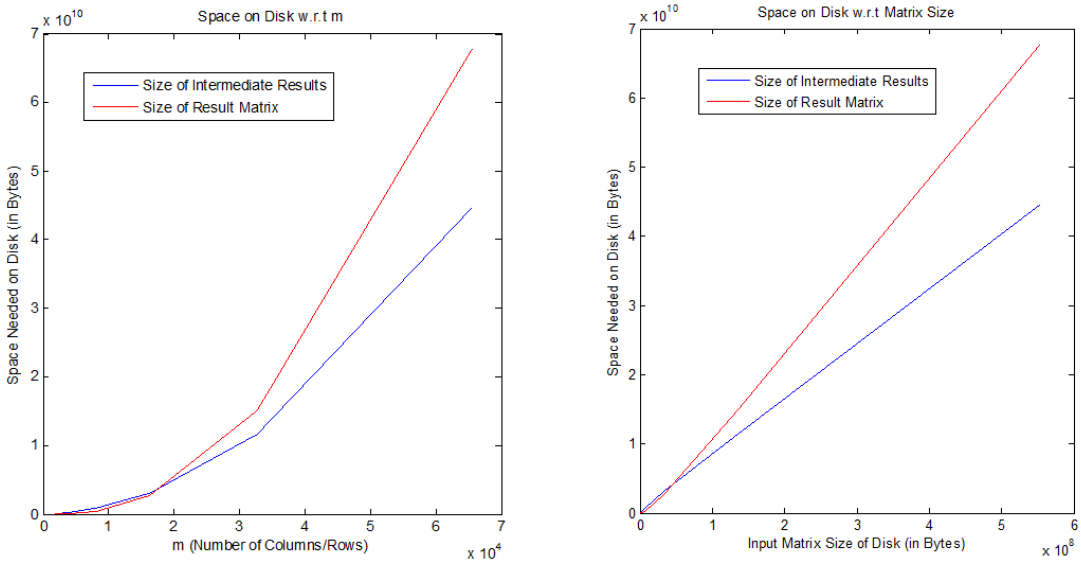

Figure 15 reveals the change of intermediate results size and the size of result matrix as *m* and the size of input matrix grows. Both charts come from the same experiment.

The pattern of the intermediate results size is slightly different. The first picture displays the size of both intermediate results and result matrix as being quadratically correlated with *m*, while the second picture illustrates that they are linearly correlated with the size of the input matrix. Considering the size of the input matrix also increases quadratically with *m*, the second picture generally states the same fact derived from the first graph.

5.3.3 Performance w.r.t Sparsity

In order to discover the dependency between the elapsed time and the number of non-zero cells in operand matrices, we plot the T_{elapse} vs sparsity in the following graph. The whole experiment is preceded on two randomly generated square matrices ($m = 2^{12}$).

From this picture it can seen that the elapsed time is a linearly of the number of non-zero cells grows.

The linear pattern can also be explained when looking back at Figure 14. Since the elapsed time and the number of non-zero cells are both quadratically correlated with *m*, a linear relationship can be expected between these two variables.

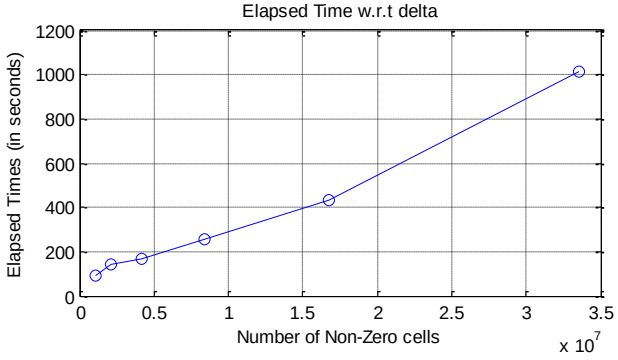

Figure 16 plots the elapse time vs. the number of non-zero cells in operands matrices.

This figure also suggests that better performance can be achieved by reducing the sparsity of operand matrices.

5.3.4 PERFORMANCE W.R.T PARTITIONING STRATEGY

As has mentioned earlier, choosing different partitioning strategies have significant impacts on performance. The row partitioner is developed for maximizing the usage of data locality. This experiment demonstrates how much benefit can be brought when utilizing row based partitioning strategy.

This experiment uses the same settings that have been employed in 5.3.2. However, the experiments in this section are repeated three times with different partitioners: a) random partitioner (α=20, β=6, λ=20), b) random partitioner (α=20, β=6, λ=20), c) row partitioner.

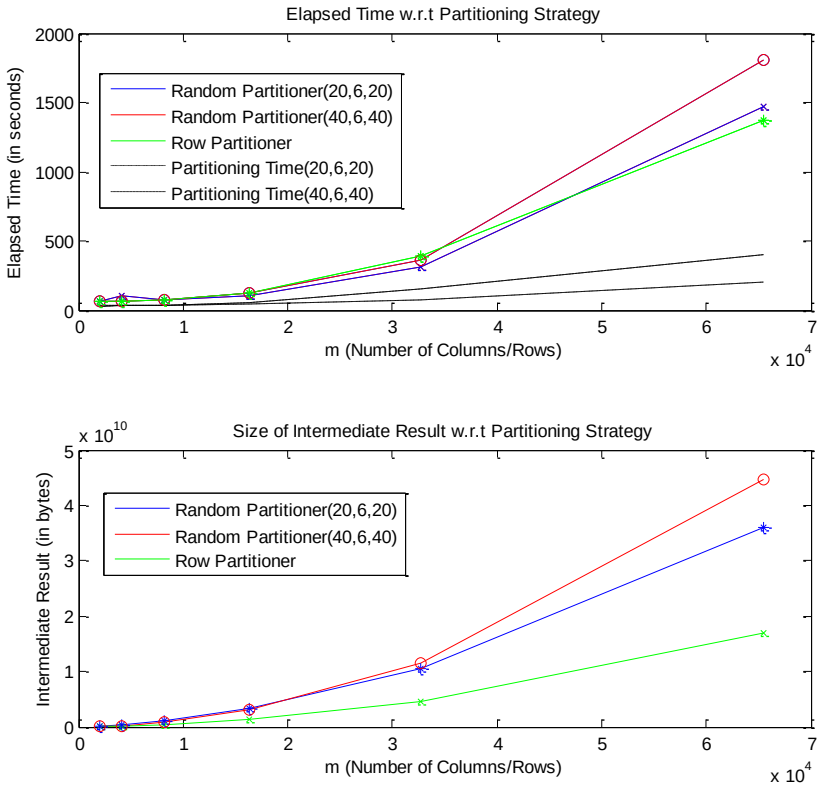

Figure 17 shows the performance of three different partitioners used in matrix multiplication. Partitioning time for two different random partitioners also illustrates on the first graph.

From the two charts above, we can see clearly that compared with the two random partitioning strategies, the row based partitioner enjoyed less intermediate results and smaller T_{elapse} on both pictures. As we have predicted in the 3.3.2, the row based partitioner has a great potential in reducing the size of intermediate results for the usage of data locality. It can be predicted that, the size of intermediate results is related with the settings of α, β, λ, therefore the random partitioner with smaller partition values outweighs that with larger values on performance on both charts. Our guess can be examined directly from the graph. A Larger number of α, β, λ produces more intermediate results. As a result, the elapsed time also increases as the size of intermediate results grows. However, surprisingly, the elapsed time of the row based partitioner is just slightly lower than the random partitioner with α=20, β=6, λ=20. We believe this fact can be explained by the mechanism of row partitioner.

In order to maximize the data locality, the row partitioner is set at α equals the number of computing nodes to guarantee that the intermediate addends are transferred to the computer where the corresponding summation is going to happen. However, as the number of

computers in the cluster is large (30 machines each with 4 cores available), the number of α is also a large number (which is 120 in this case). As we have concluded before, larger α increases the time cost. As a result, the time that saved by adopting the data locality is counteracted by the time consumed on partitioning.

5.3.5 SPEEDUP RATE

One most important measure for the performance of the parallel algorithm is speedup which is designed to identify how much a parallel algorithm is faster than a corresponding sequential algorithm (JaJa 1992).

Speedup is defined as the following formula:

$$S_p = \frac{T_1}{T_p}$$

where T_1 is the execution time for sequential execution time, and the T_p is the execution time for paralleled implementation. p represents the number of processors (or machines).

Good scalability is obtained when $S_p = p$. This speedup is often called "linear speedup" or "ideal speedup".

Speedup divided into two categories: absolute speedup or relative speedup. The only difference is that the T_1 in "absolute speedup" is the time for the best sequential algorithm while in the relative speedup, parallel version of algorithm is used on a single processor (machine).

Considering the capacity of a single machine and the software available, only the relative speedup is adopted for the test in this experiment.

From the graph below, a clear trend of speedup rate can be seen. The linear speedups with three different δ are all lower than the linear speedup which upper-bounds all the practical speedup according to the **Amdahl's law**.

On the matrix with $\delta = 2^{-7}$, the linear speedup is almost 7 when 8 workers are enabled. The other curves with $\delta = 2^{-10}$ and $\delta = 2^{-13}$ also have similar speedup rates when the number of working machines less than 8.

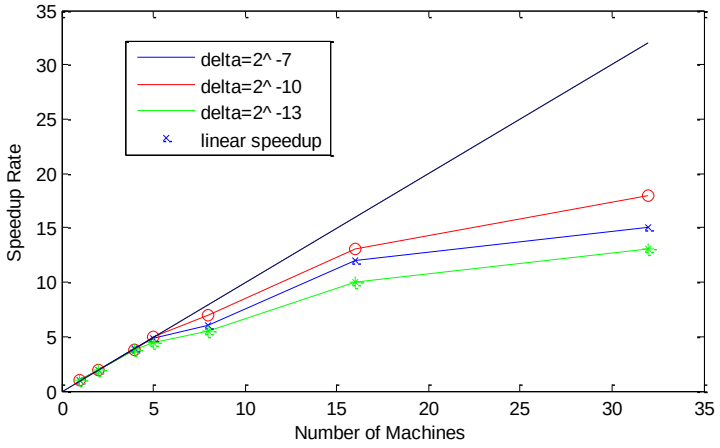

Figure 18 displays the speedup rates of three different δ choices. The ideal speedup is illustrated at the diagonal of the space by dash line.

The reasons for the gap between the practical speedup and the theoretical upper bound may be various. Although the code we written in Map-Reduce fashion can be all parallelized, several maintaining operations may be conducted during the experiment, such as check-pointing and auto-balancing.

Interestingly, we can see speedup for sparser matrices are lower than those with higher sparsity. This fact suggests that for small and sparse matrices, a small number of clusters can do the best job, while for larger clusters, the whole system is not saturated and the rest of the computing resources (e.g. memory and idle CPUs) are wasted.

5.4 Experiments on Support Vector Machines

In this section, the correctness and parallel performance of our SVM implementation is tested against a range of datasets.

5.4.1 Data Sets

For the test purpose, we select a set of testing problems is provided for our implementation. In considering two different experiment objectives, two groups of problem sets are chosen for different purposes.

For correctness experiments, we test our implementation against a series of small problem sets which are fit for both our parallel implementation and sequential algorithms.

IRIS: IRIS may be the best known dataset in pattern recognition literature and is first provided by R.A. Fisher (Fisher 1936). It contains 150 instances represented by 4-dimensional vectors in three classes. One class is linearly separable from the other two classes. In this experiment, it is only used as binary classification.

ADULT: ADULT dataset is used to predict whether this person's salary may exceed 50k/year. It was originally a dataset mixed with categorical and integer values. During the pre-processing stage, we convert all values into float number representation.

WINE: The dataset WINE is used to determine the origin of wines by a series of chemical components. It is also a linearly separable problem containing 178 instances with 13 attributes and is often used to test new classifiers.

HEART: The heart problem set is a medical prediction problem addressed for demonstrating the performance of LibSVM. We convert this dataset into a binary classification problem based on the scaled version provided on the LibSVM website.

Reuters-21578: A well selected and scaled dataset comes from the Reuters Corpus in 1988. Stemming and filtering of stop words are pre-processed on articles in the original corpus.

For parallel performance test, the scale of our testing set is enlarged and ranging from 6K to 64K instances, which is automatically generated from Reuters Corpus 1998[9].

In order to produce the large (over 3.7GB when decompressed) Reuters Corpus, a document-term matrix is obtained by using the MapReduce fashion as mentioned above, after which a TF/IDF matrix will be generated by using the matrix multiplication. A small Perl-written script program will automatically extract an equal number of positive and negative instances randomly as training subset from this matrix.

5.4.2 Correctness Experiments on Single Machine

The SVM implementation on above mentioned datasets is tested for correctness evaluation. The aim of this set of tests is to prove that our SVM classifier can achieve a comparable accuracy on popular datasets with main-stream SVM software kits.

The LibSVM is used for comparison because it is arguably the best classifier of both speed and accuracy, and has been tested against most existing datasets. In this experiment, it is set to the default model without using any optimization techniques. For the implementation issues mentioned earlier, our implementation can only have a fixed bias, and zero bias is used in this case. Parameter C is set to a large integer for 1-norm hard margin classification[10]. For simplicity of implementation, the number of iterations is fixed at 100. All results are obtained by ten-fold cross validation.

Data Set	Accuracy	
	LibSVM	Gradient Descent (Hard Margin)
IRIS*	97.33%	98.7%
ADULT	82.1%	69.3%
WINE*	98.5%	95.2%
HEART (Binary)	97.6%	95.2%
Reuters-21578	92.2%	87.2%

Table 1 shows accuracy comparison of LibSVM and the Gradient Descent implementation. Linear separable datasets are marked by *

As can be seen from the table shown above, for two linear separable problem sets, our implementation offers very close accuracies compared with those achieved by LibSVM. The best performance is reached on the IRIS dataset with 98.7% compared with 97.33% from LibSVM. Besides, for non-linear separable problems (Reuters-21578 and ADULT), our implementation still achieves a promising accuracy (69.3% and 87.2% respectively), although LibSVM enjoys a much higher accuracy for its soft-margin classification implementation.

[9] http://about.reuters.com/researchandstandards/corpus/

[10] Although it is possible to turn on the soft margin classification in our SVM implementation, the parameter C needs to be carefully chosen for each dataset before the experiments. For simplicity reasons, hard-margin algorithm is adopted.

5.4.3 PARALLEL PERFORMANCE EXPERIMENTS

In this section, our SVM classifier is tested against with the Reuter's Corpus in parallel.

Our SVM trainer will be trained on a series of subsets of Reuters Corpus with increasing the number of instances. The experiments start with 12K training examples, and end up with 192K training examples. Experiments are divided into 2 steps:

a) Kernel Construction
b) Iterative SVM learning for vector **α**
c) Extraction of support vectors from original dataset

After that, accuracy will be computed.

The most straightforward termination criterion is implemented by monitoring the Karush–Kuhn–Tucker conditions. This condition is examined at the end of each iteration. Similar stopping criterion has been developed in (Hsu and Lin 2002), and adopted in LibSVM.

Compared with typical machine learning performance measures, the accuracy is not calculated by 10-cross fold validation, since for large corpus, the split of test and training set is extremely time consuming. The alternative method in our experiment is randomly selecting testing examples with equal number of positive and negative instances from original corpus. The results are reported in the following table.

Data Set	Kernel Sparsity	Kernel Matrix(GB)	T_{elapse} one Iteration	Accuracy
Reuters(12K)	0.82	1.47	33s	82.5%
Reuters(24K)	0.83	5.70	45s	80.3%
Reuters(48K)	0.86	22.3	66s	74.2%
Reuters(96K)	0.87	74.0	180s	72.4%
Reuters(192K)	0.86	230	420s	70.0%

Table 2 shows the size and sparsity of kernel matrices on different scale of training set and the corresponding T_{elapse} and accuracy performed.

The first two columns listed above illustrate the major issue that SVM suffers from. Dense and huge kernel matrix makes the SVM less attractive for large scale problem sets. All the kernel matrices that listed above have a very high sparsity level (above 0.8). Moreover, the size of the kernel matrix also grows with the number of training examples quadratically. For single computers, these kernel matrices are not acceptable for in-memory access.

The last two columns report the elapsed time for each iteration and accuracy. Generally, our algorithm can perform an acceptable accuracy when adopted on large scale of dataset within a reasonable time. Interesting patterns can be found at the last column where the accuracy is dropping as the number of instances increase. We believe this is may be caused by our stopping criterion that terminate our algorithm at an immature stage, however, we can't find a very strong proof for our guess, and we believe this remains a tasks for future study.

5.5 EXPERIMENTS ON NON-NEGATIVE MATRIX FACTORIZATION

The aim of this set of experiments is to show our implementation has gained the full capability of solving the large scaled matrix-factorization problem on cluster.

5.5.1 DATA SETS

The datasets preparation for Non-Negative Matrix Factorization is very similar to the approach that has been employed in 5.3.The random matrix generator is again used for generating a random matrix that is to be factorized with different sparsity. The height and weight is fixed at 2^{18}.

5.5.2 PARALLEL PERFORMANCE EXPERIMENTS

In this set of experiments, our NMF implementation is tested under parallel configuration. Since the update of W and H are symmetric, in this section, only performance of updating H will be discussed.

The performance is measured by three different computational components in our algorithm. For each component, four conditions under two categories will be considered. Similar to the evaluation of matrix multiplication, the effect of matrix sparsity is of interests. Moreover, the parameter k will also be considered under each sparsity setting.

Computational Components	Sparsity $\delta = 2^{-7}$		Sparsity $\delta = 2^{-10}$	
	K=8	K=32	K=8	K=32
	T_{elapse}(s)	T_{elapse}(s)	T_{elapse}(s)	T_{elapse}(s)
$X = W^T A$	46	129	20	30
$Y = W^T WH$	23	60	13	16
$H = H.* X./Y$	12	14	11	12

Table 3: The computational breakdown of NMF implementation on random generated matrix with different sparsity and parameter k

The first interesting pattern that can be found among the data listed in the above breakdown is the Elapsed time of computing $X = W^T A$ dominates both computational costs in terms of both sparsity settings. The reason for this is that the matrix ***A*** is commonly larger than both **W** and **H**, and the partitioning may have a higher cost compared with others. For this reason, we can expect a drop of elapsed time when ***A*** becomes sparser. This expectation has been verified by the other half of our table, which shows, T_{elapse} reduces dramatically when the sparsity decreases to 2^{-10}. The performance of other components drops accordingly.

However, it can also be observed that the computation of H=H.*X./Y is almost a constant, although the computational complexity of this component should be $O(n)$. This result can be explained by the size of each operand in this operator all being small matrices (k by n) that require very small storage in both memory and disk compared with mass storage obtained within the clusters. It also suggests that the capacity of the current cluster is not saturated. In this case, it may be more efficient to run on local machines than in a distributed environment.

5.6 EXPERIMENTS ON PAGERANK

Pagerank has been introduced into many web-scaled applications. This set of experiments aims to examine the performance of PageRank on a range of various sized datasets.

5.6.1 DATA SETS

We first test our algorithm on two small datasets listed below

Harvard500 is a directed graph adjacent matrix generated from 500 webpages crawled from Harvard University website. To our knowledge, it is the smallest PageRank dataset that can be found.

For comparison purposes, we also examine our approach on Hollins.edu dataset which is also a web matrix organised from crawled web-content on Hollins.edu.

Their miniaturised size allows us to explore the internal structure of these adjacent matrices as well as the patterns of PageRank distribution.

In order to test the performance of our implementation on real-world web-scale dataset, we carry on our experiments on the Wikipedia 2008 corpus[11] which contains 5,716,808 compressed wiki-pages from Wikipedia English Website. 1.14 GB adjacent matrix has been generated after hyperlinks have been extracted.

5.6.2 EXPERIMENTS ON HOLLINS.EDU AND HARVARD500

At the first step, results on two small sized dataset Hollins.edu and Harvard500 are compared.

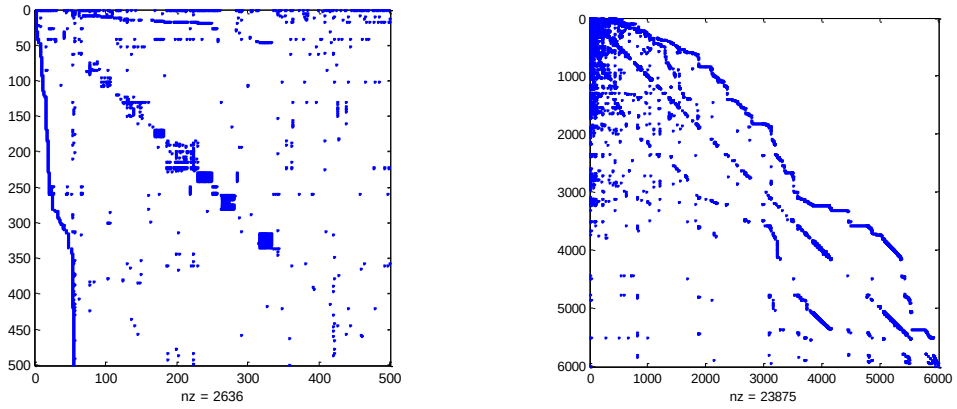

Figure 19: The sparsity pattern of adjacent matrices of Harvard500 and Hollins.edu respectively

The sparsity pattern of two web matrices is illustrated above. As can be see, clusters emerge in both very sparse matrices. In the first matrix, clusters are distributed along with the main diagonal, while in the second matrix, large clusters are located at the top left corner and most clusters appear beneath the main diagonal.

[11] http://about.reuters.com/researchandstandards/corpus/

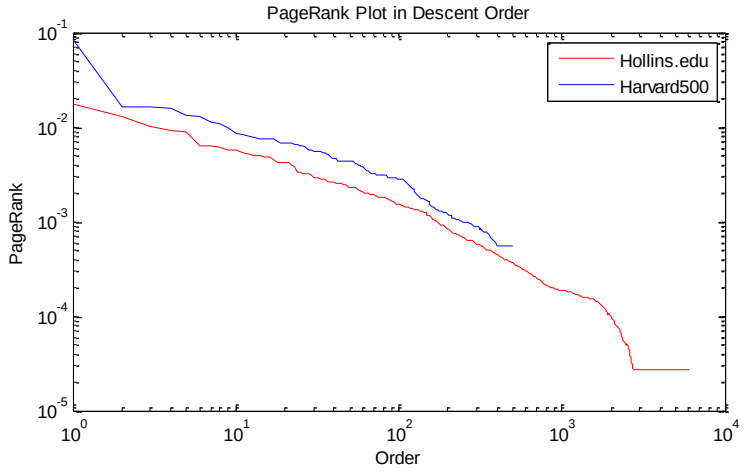

Figure 20: The PageRank plot of Harvard500 and Hollins.edu respectively in descent order

After adopting our PageRank calculation on both datasets, the obtained PageRank array is sorted in descent order, and plotted into Log-Log space.

It can be noticed that the PageRank of Hollins.edu is smaller than Harvard500, since the number of webpages (i.e. row/column in web-matrix) in Harvard500 is much smaller than that in Hollins.edu. (According to the algorithm, the sum of all PageRank should be exactly 1.)

The PageRank for both datasets reduces linearly in LogLog space. It suggests that the PageRank of these two datasets conforms to certain logarithm distribution, and this assumption can be verified by the following two probability plots.

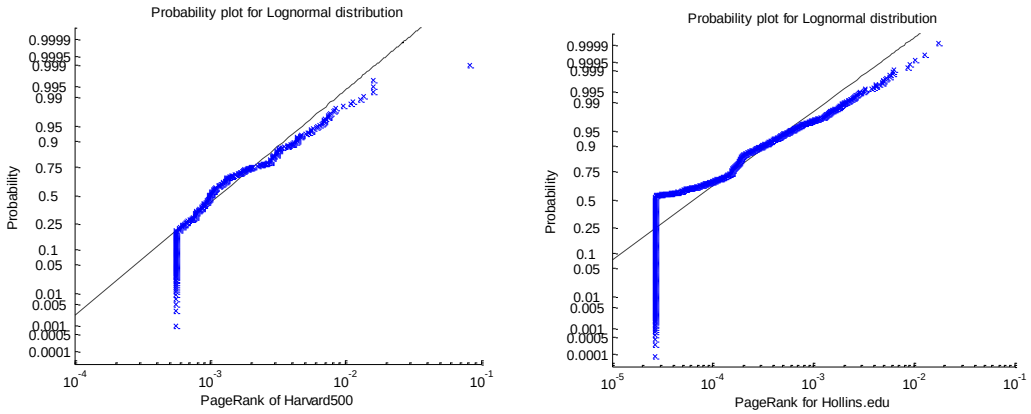

Figure 21: The Probability plot of PageRank for lognormal distribution

From the two charts above, it can be seen that PageRank on both datasets fits the lognormal distribution well except for samples on the left hand side, where the PageRanks is lower than a certain value and remains equal to each other. This sign may suggest that the PageRank on both datasets hasn't fully converged for small values. Another implementation, which is a new approach using power method, can achieve a faster convergence within fewer number of iterations. (Kamvar, Haveliwala and Golub 2003)

5.6.3 PERFORMANCE ON WIKIPEDIA CORPUS

In order to evaluate performance on real-world dataset, our algorithm is executed on the Wikipedia dataset in parallel. By using 30 nodes, our implementation finishes 100 iterations within 1hour, and provides us with the following results.

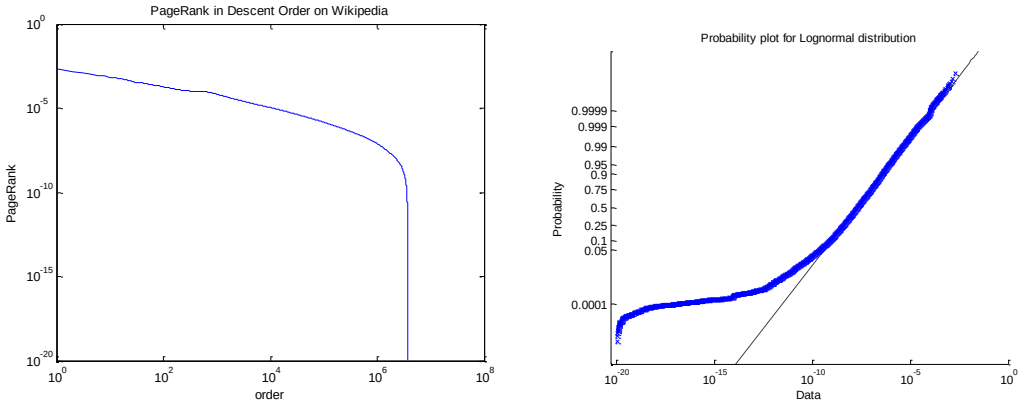

Figure 22: The PageRank plot and Probability plot on Wikipedia Corpus

The two graphs plotted above show different patterns compared with those generated on small datasets. In Log-Log space, the PageRank falls suddenly after the $10^{6\text{th}}$, which implies that there is a small collection (approx.$2\sim3\times10^{6}$) of Wikipedia entries not connected with the most regular entries below a certain level of PageRank(approx. 10^{-7}). The probability distribution shows that top ranks also conform to Lognormal Distribution, however, it can be noticed that there is a large PageRank shift for the smallest ranked group of entries. It can be regarded as another interpretation of the pattern on the left chart in Log-Log space.

Another interesting result is reported in the table below which lists the top 10 results sorted by PageRank.

Wikipedia Entry	PageRank	NO. Inlinks
United_States	0.0021	374934
2007	0.0014	266614
2008	0.0014	286409
United_Kingdom	0.0011	139325
Wiktionary	0.0009	< 29076
Geographic_coordinate_system	0.0009	294604
2006	0.0009	146336
Wikimedia_Commons	0.0008	70096
English_language	0.0006	69408
Germany	0.0006	95366

Table 4 The Top-10 ranked Wikipedia Entry with their PageRank and NO. inlinks

Although Wikipedia does not release its official popularity ranking, measured by our daily life experience, a reasonable popularity list seems to be produced by our PageRank implementation. However, it also gives an arguable rank of entry "Geographic_coordinate_system".

Generally, as the PageRank of webpage decreases, its number of inlinks also drops. The only exception is the entry "Wiktionary", which is ranked 5^{th}, while its inlinks number fails to list

in the Top 100 entries sorted by No. inlinks, thus it is marked by "< 29076" (where 29076 is the smallest inlink number among Top 100).

5.7 SUMMARY

In this chapter, all experiment results and analysis have been listed in order to verify the correctness and performance of our implemented operators and algorithms.

From the results of the experiments, it can be concluded that the matrix operators is scalable and efficient for large collection of input data, and achieves the maximal 19 speedup when 30 computing nodes are enabled, although the speedup rates may vary according to sparsity.

Our SVM implementation is proved that it can achieve comparable (or even higher) accuracy compared with LibSVM, arguably the best SVM classifier so far. The accuracy of the classic classification problem set IRIS reached 98.7% which is 1.37% better than LibSVM. Our implementation also performed acceptable accuracy (avg. 75.4%) within reasonable training time (less than 420s per iteration) during parallel test.

The NMF algorithm is implemented by adopting the existing multiplication operator. We also test them among a series dataset with different sparsity where the pattern of T_{elapse} is found to be similar to the pattern that reported in 5.3.3.

The PageRank algorithm is firstly tested on a series of small datasets where a logarithm distribution has been found on top ranking results. After adopting the same algorithm in parallel on Wikipedia dataset, we found a similar distribution which suggests that the implementation is both feasible and functional for the web scaled dataset .

According to our analysis and reports, it is believed that all algorithms are correctly implemented and show full capability of solving large scaled datasets.

6. CONCLUSION

In this dissertation, researches and literature reviews on MapReduce, Support Vector Machines, Non-Negative Matrix Factorization and PageRank algorithms were conducted, after which a design of matrix operator system and implementation of these algorithms on MapReduce was put forward in the following two chapters. In chapter 5, experiments and results are discussed.

As we can see from chapter 5, the goals we set in 1.1 have generally achieved, although there are some limitations remain for future discussion. Therefore, in this chapter, it is concluded that the project can be divided into two parts: achievements and limitations.

6.1 ACHIEVEMENTS

Inspired by recent work on MapReduce, we tried to investigate some possible solutions to processing a large scale dataset in Machine Learning algorithms on this brand new parallel framework. In this project, several machine learning algorithms have been successfully adapted to the MapReduce paradigm, and some achieves comparable accuracy compared with mainstream sequential implementations.

The major achievements are listed below:

1. A set of matrix operators has been implemented. In order to provide a flexible and programmable underlying foundational framework to support high-level algorithms, a set of matrix operators is first adapted. A variety of tests that have preceded showed that it achieves good speedup and is robust in different matrix sizes and sparsity settings.
2. The effect of the partitioning strategy has been studied. To reduce the size of intermediate results, a new concept “Partitioning Strategy” is introduced into the implementation. Results illustrates that it can significantly reduce the size of output generated by the partitioning stage while the improvements on running time are also observable.
3. The construction of a large kernel matrix has been studied and tested. Several suggestions on reducing the size of the kernel matrix has been given.
4. Support Vector Machines has been implemented on the matrix operators system by using the Gradient Descent fashion. With the help of large throughput provided by MapReduce, the implementation is able to deal with large dense kernel matrix with a size of up to 230 GB.
5. Non-Negative Matrix Factorization is adapted with our matrix operator system using the multiplicative method proposed by Lee and Seung (2000) .
6. PageRank implementation is successfully calculated on Wikipedia corpus with 5,716,808 entries for 100 iterations within 1 hour, and shows encouraging results.

As far as can be known, although 1 and 2 have been discovered in several online materials, it is the first time that this has been studied in academia. 3 and 4 are made up of completely original ideas, and it is the first time that 5 and 6 have been implemented from their primitive mathematical definition.

However, the limitations cannot be ignored when drawing such conclusions.

6.2 LIMITATIONS

In this project, the limitations fall into two categories which are distinguished by different sources: a) limitations brought by the MapReduce framework, b) limitations brought by the implementations. The major drawbacks caused by the MapReduce framework are listed below:

- Instant inter-machine communicate mechanism is not provided. Compared with some other popular message-driven parallel solutions, MapReduce may be less flexible about its inter-machine communication. The only stage that messages and data can be distributed among machines is the "shuffling" stage. Although it is more transport efficient and more band-width saving, some algorithms may face difficulties being adapted when instant messages need to be broadcasted. This feature can be very efficient for some iterative solutions (e.g. SVM chunking algorithm) as it speeds up the convergence speed using early results. Fortunately, several attempts have already beeninitiated to improve the MapReduce framework on this issue (Tyson Condie 2010).
- The purpose of introducing MapReduce is to utilize commodity PCs as a cheap computing power. In order to increase the reliability and robustness, a number of maintaince demons are launched together with MapReduce jobs, which may slightly slows down the progress of jobs. However, the testing platform for this project is HPC. It has a very strong hardware ensured stability, so that the protection provided by MapReduce is almost redundant in this implementation. The performance has also been limited by these extra settings.
- The Hadoop Implementation of MapReduce does not have a full compatibility on BlueCrystal. The only way to use Hadoop on BlueCrystal is through the "Hadoop on Demand (HOD)" toolkit. However, the computational capacity of BlueCrystal has exceeded the maximal settings in HOD toolkit, while the alternative method we use is much less convenient than the original support in HOD.

Limitations deriving from our implementations are listed below:

1. The matrix multiplication algorithm can be simplified in some special cases. Although the matrix multiplication operator implemented provided a flexible way of expressing algorithms. In some cases, some specially designed approaches can achieve much better performance compared with our implementation. A typical situation is one normal matrix multiplies one diagonal matrix. In this case, a specially designed algorithm can store the whole diagonal as a list in memory. Each row on the result matrix can be calculated by multiplying the original row in left operand matrix with this list. This operation is much faster than our implementation which involves a complicated partition-calculation procedure.
2. Not all calculations in our SVM implementation suits parallel implementation. For example, the update of vector $\boldsymbol{\alpha}$ can be conducted within seconds when implemented by a sequential program on a single machine. In contrast, this operation takes almost 30 seconds on the cluster because of extra MapReduce job overhead. This suggests that not every computational task can gain advantages after being distributed to parallel clusters.
3. The update step of matrix H in NMF implementation is overly verbose. For this reason, the initiate overhead cannot be ignored. Recently, it has been argued that the

update of both **H** and **W** can be executed in a parallel fashion (Liu, Yang and Fan 2010), which saves half the time by maximizing the parallel usage.

4. The NMF algorithm has not been tested for correctness on some classic datasets since we fail to find a dataset that is specially designed for NMF.

6.3 SUMMARY

In conclusion, this project achieves the aims and objectives we set out in 1.1, although some limitations are found in 6.2.

It is believed that this project is meaningful to the real world applications, and points out some approaches to implementing complex algorithms by fully utilizing the features provided by MapReduce.

Some discussion on future improvements can be found in the next chapter.

7. FUTURE WORK

Although the goals and objectives have been accomplished, there are still a large number of improvements that could be made based on our project. In this chapter, some possible plans and strategies for our future works will be discussed.

7.1 NEW FEATURES IN HADOOP 0.21.0

One possible improvement that can be conducted in the future is to adapt our project for the Hadoop 0.21.0 series. Hadoop 0.21.0 releases at the time when we are preceding our project, however, due to stability concerns and incompatible APIs, this project has not been translated into new version of APIs. The latest version of Hadoop 0.21.0 largely improves its stability and performance. Several management toolkits have also been introduced in this version. For example, the new cluster administration interface allows us to test speedup rates without restarting the whole cluster, and the new FileContext object provide the capacities for handling large shared HDFS files fromwhich our multiplication operator can be benefit. It is believed that these new features enable a number of new testing and implementation techniques to be more efficient and productive.

7.2 PROGRAMMABLE MATRIX OPERATORS

The reason that we bring in matrix operators are used is to construct a programmable linear algebra computation system that can simplify the job of creating new machine learning algorithms. It is believed, in future, a new simple script language can be invented to describe the matrix operators, so that a sophisticated algorithm can be written into a small script program (like Matlab language).

For example, to create a text kernel from document term matrix, simply type the following command

$$\mathbf{K = DPP'D'},$$

and then the calculation will be submitted to cluster for execution. However, the problem also comes with the simplicity it brings. The optimal strategies of executing the sequence should be considered in priori.

7.3 SPECIAL IMPLEMENTATIONS

As it has been revealed before, special implementations exist for many of the algorithms implemented and can achieve a better performance, although for the sake of simplicity, only the naïve approaches have been adopted in our project.

However, for practical applications, special implementations may only be concerned about the performance.

7.4 KERNEL/SIMILARITY BASED ALGORITHMS

As the construction of the kernel is very modular, the kernel building stage can be ditinguished from the actual SVM algorithm. Moreover, there are a many algorithm that utilize the “kernel trick” which can be implemented on MapReduce without recreating the kernel matrix. Several possible algorithms are listed below:

- **Kernel based Principle Components Analysis** is a principle components analysis algorithm that utilizes the kernel trick to reduce a Kernel-Hilbert space using non-linear mapping.
  It can be proved that, the eigenvectors and eigenvalues can be computed through a dual representation computed from the eigenvectors and eigenvalues from the Kernel Matrix. Here we donate the U_k as the subspace spanned by the first k eigenvectors in the feature space. Using the following equation

$$P_{U_k}(\emptyset(\boldsymbol{x})) = (u'_j \emptyset(\boldsymbol{x}))_{j=1}^{k} = (\sum_{i=1}^{l} \alpha_i^j \, k(\boldsymbol{x_i}, \boldsymbol{x}))_{j=1}^{k},$$

  we can project new vectors into the subspace where

$$\boldsymbol{\alpha}^j = \lambda_j^{1/2} \boldsymbol{v_j}$$

- **Similarity based learning (such as KNN, K-Means)** can also benefit from kernel construction methods, since the kernel matrix can easily transformed into similarity representations.

7.5 SUMMARY

This chapter has mainly introduced several works remaining for future improvements based the project. Most of ideas coming from the limitations have been concluded in previous chapter. Possible work plans include utilizing the latest features in Hadoop 0.21.0, constructing a script language for algorithm description, adopting special implementations for performance considerations, and implementing other kernel based algorithms by using the existing kernel matrix construction module.

BIBLIOGRAPHY

Attribute-Relation File Format (ARFF). 1 August 2002. http://www.cs.waikato.ac.nz/~ml/weka/arff.html (accessed September 29, 2010).

Brin, Sergey, and Lawrence Page. "The Anatomy of a Large-Scale Hypertextual Web Search Engine." *COMPUTER NETWORKS AND ISDN SYSTEMS.* Elsevier Science Publishers B. V, 1998. 107-117.

Burges, Christopher J.C. "A Tutorial on Support Vector Machines for Pattern Recognition." *Data Mining and Knowledge Discovery*, 1998: 121-167.

Chang, Fay, et al. "Bigtable: A Distributed Storage System for Structured Data." *OSDI'06: Seventh Symposium on Operating System Design and Implementation.* Seattle, WA,, 2006.

Dean, Jeff, and Sunjay Ghemawat. "Mapreduce, A Flexible Data Processing Tool." *Communications of ACM*, 2010: VOL. 53.

Dean, Jeffrey, and Sabhay Ghemawat. "MapReduce : Simplified Data Processing on Large Clusters." *OSDI*, 2004.

Fisher, Ronald A. "The use of multiple measurements in taxonomic problems." *Annual Eugenics*, 1936: 179-188.

Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google File System." *19th ACM Symposium on Operating Systems Principles.* New York, USA, 2003.

Gleich, David, Leonid Zhukov, and Pavel Berkhin. *Fast Parallel PageRank: A Linear System Approach.* Yahoo! Technical Report, 2004.

Graf, H.P, Eric Cosatto, Leon Bottou, Igor Durdanovic, and Vladimir Vapnik. "Parallel Support Vector Machines: The Cascade SVM." *Advances in Neural Information Processing Systems 17*, 2005: 521–528.

Hildreth, Clifford. "A quadratic programming procedure." *Naval Research Logistics Quarterly*, 1957: 79–85.

Hsu, Chih-Wei, and Chih-Jen Lin. "A Formal Analysis of Stopping Criteria of Decomposition Methods for Support Vector Machines." *IEEE TRANSACTIONS ON NEURAL NETWORKS*, 2002: VOL. 13, NO. 5.

JaJa, Joseph. *An introduction to parallel algorithms.* Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1992.

Kamvar, Sepandar, Taher Haveliwala, and Gene Golub. *Adaptive Methods for the Computation of PageRank.* Stanford, 2003.

Kleinberg, Jon M. "Authoritative sources in a hyperlinked environment." *ACM46(5)*, 1999: 604-632.

Kohlschutter, Christian, Paul-Alexandru Chirita, and Wolfgang Nejdl. "Efficient Parallel Computation of PageRank." *Advances in Information Retrieval*, 2006.

Langville, Amy N., and Carl D. Meyer. "Google's PageRank and Beyond: The Science Behind Search Engine Rankings." *The Mathmatical Intelligencer*, 2008: 68-69.

Lee, Daniel D., and H. Sebastian Seung. "Algorithms for non-negative matrix factorization." *NIPS*, 2000.

Lee, Daniel D., and H. Sebastian Seung. "Learning the parts of objects by non-negative matrix factorization." *Nature*, 1999: 788–791.

Liu, Bing. *Web Data Mining: exploring hyper links, contents and usage data.* Springer, 2007.

Liu, Chao, Hung-chih Yang, and Jinliang Fan. "Distributed Nonnegative Matrix Factorization for Web-Scale Dyadic Data Analysis on MapReduce." *WWW2010.* Raleigh: ACM, 2010.

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, Ian H. Witten. "The WEKA Data Mining Software: An Update." *SIGKDD Explorations*, 2009: Volume 11, Issue 1.

Michael Stonebraker, Daniel Abadi, David. J. DeWitt, Samuel Madden, Erik Paulson, Andrew Pavlo, Alexander Rasin. "MapReduce and Parallel DBMSs: Friends or Foes?" *Communications of the ACM*, 2010: 64-71.

Olson, Michael A., Keith Bostic, and Margo Seltzer. "Berkeley DB." *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference.* Monterey, California, USA: 1999, 1999.

Owen O'Malley, Arun C. Murthy. *Winning a 60 Second Dash with a Yellow.* Technical report, Yahoo!, 2009.

Page, Lawrence, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank Citation Ranking: Bringing Order to the Web.* Stanford InfoLab, 1999.

Pavlo, Andrew, Erik Paulson, and Alexander Rasin. "A Comparison of Approaches to Large-Scale Data Analysis." *SIGMOD.* 2009.

Platt, John. "Using Analytic QP and Sparseness to Speed Training of Support Vector Machines." *Advances in Neural Information Processing Systems*, 1999.

Rapp, Reinhard. "Word sense discovery based on sense descriptor dissimilarity." *Ninth Machine Translation Summit.* 2003. 315-322.

Salton, G., A. Wong, and C. S. Yang. "A vector space model for automatic indexing." *Communications of the*, 1975.

Salton, Gerrard. "The SMART retrieval system: Experiments in automatic document pro-." New York, USA, 1971.

Shawe-Taylor, John, and Nello Cristianini. *Kernel Methods for Pattern Analysis.* Cambridge Press, 2004.

Singh, Ajit P., and Geoffrey J. Gordon. "A unified view of matrix factorization models." *In PKDD.* 2008. 358–373.

Sort Benchmark Home Page. 1 Jan 2010. http://sortbenchmark.org/ (accessed May 9, 2010).

Tamer Elsayed, Jimmy Lin, Douglas W. Oard. "Pairwise document similarity in large collections with MapReduce." *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers.* Morristown, NJ, USA: Association for Computational Linguistics, 2008. 265-268 .

Thilo-Thomas Frieb, Nello Cristianini, Colin Campbell. "The kernel-adatron algorithm: a fast and simple learning procedure for support vector machines." *Proceedings of ICML.* San Mateo, 1998. 188-196.

Thorsten Joachims, Fachbereich Informatik , Fachbereich Informatik , Fachbereich Informatik , Fachbereich Informatik , Lehrstuhl Viii. "Text categorization with support vector machines: Learning with many relevant feature." 1997.

Tommi Jaakkola, David Haussler. "Exploiting Generative Models in Discriminative Classifiers." *In Advances in Neural Information Processing Systems 11*, 1998: 487--493.

Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein. "MaprReduce Online." *NSDI2010.* San Jose, CA, 2010.

What's New in Apache Hadoop 0.21. 26 August 2010. http://www.cloudera.com/blog/2010/08/what%E2%80%99s-new-in-apache-hadoop-0-21/ (accessed September 28, 2010).

White, Tom. *Hadoop: The Definitive Guide.* O'Reilly Media, 2009.

WikipediaUsers. *Matrix Multiplication Illustration.* n.d. http://upload.wikimedia.org/wikipedia/en/e/eb/Matrix_multiplication_diagram_2.svg (accessed 9 2, 2010).

Zaharia, Matei, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. "Improving MapReduce Performance in Heterogeneous Environments." *USENIX Symposium on Operating Systems Design and Implementation*, 2008.

Zeyuan Allen Zhu, Weizhu Chen, Gang Wang, Chenguang Zhu, Zheng Chen. "P-packSVM: Parallel Primal grAdient desCent Kernel SVM." *2009 Ninth IEEE International Conference on Data Mining.* 2009. 677-686.