DISTRIBUTED APPROXIMATE SPECTRAL CLUSTERING FOR LARGE-SCALE DATASETS

by

Fei Gao
B.Eng., Beijing University of Technology, 2009

Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Fall 2011

APPROVAL

Name:	Fei Gao
Degree:	Master of Science
Title of Thesis:	Distributed Approximate Spectral Clustering for Large-Scale Datasets

Examining Committee: Dr. Tamara Smyth,
Assistant Professor, Computing Science
Simon Fraser University
Chair

Dr. Mohamed Hefeeda,
Associate Professor, Computing Science
Simon Fraser University
Senior Supervisor

Dr. Wael Abd-Almageed,
Adjunct Professor, Computing Science
Simon Fraser University
Supervisor

Dr. Kay C. Wiese,
Associate Professor, Computing Science
Simon Fraser University
Examiner

Date Approved: 5 December 2011

Declaration of Partial Copyright Licence

Abstract

Many kernel-based clustering algorithms do not scale up to high-dimensional large datasets. The similarity matrix, on which these algorithms rely, calls for $O(N^2)$ complexity in both time and space. In this thesis, we present the design of an approximation algorithm to cluster high-dimensional large datasets. The proposed design enables great reduction of the similarity matrix's computing time as well as its space requirements without significantly impacting the accuracy of the clustering. The proposed design is modular and self-contained. Therefore, several kernel-based clustering algorithms could also benefit from the proposed design to improve their performance. We implemented the proposed algorithm in the MapReduce distributed programming framework and experimented with synthetic datasets as well as a real dataset from Wikipedia that has more than three million documents. Our results demonstrate the high accuracy and the significant time and memory savings that can be achieved by our algorithm.

Keywords: distributed clustering; clustering of large datasets; kernel-based clustering; locality sensitive hashing; MapReduce framework;

Acknowledgments

I would like to extend my heartfelt thanks and gratitude to my senior supervisor, Dr. Mohamed Hefeeda, for the patient guidance, encouragement and advice he has provided in the past two years. His passion and insights in research have been an immense help in my graduate study. The completion of this thesis would not have been possible without him.

I would like to thank my supervisor Dr. Wael Abd-Almageed and my thesis examiner Dr. Kay C. Wiese for being on my committee and reviewing this thesis. I would like to thank Dr. Tamara Smyth for taking the time to chair my thesis defense.

I would like to thank my friends for their companionship in the past two years, especially to Yuan Liu, Yu Gao. The time spent together with them will be fantastic memory that I will cherish forever. I would like to thank all the members at the Network Systems Lab for their help, especially to Ahmed Hamza, Cameron Harvey, Yuanbin Shen, Taher Dameh and Hamed Sadeghi Neshat. They created an amiable and stimulating environment in the laboratory, and it is a great pleasure to work with them.

Last but certainly not least, I would like to thank my parents, my grandparents and my elder sister for their unreserved love and endless support. I would not have made it without them. I will always keep in mind what I owe them, and my deepest gratitude to them is beyond words.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we provide a brief introduction to clustering algorithms. We then introduce the problem we address in this thesis and summarize our contributions. The organization of this thesis is given at the end of this chapter.

1.1 Introduction

Clustering, as a form of unsupervised learning, refers to the problem of trying to find hidden structure in unlabeled data [4]. Since the examples given to the learner are unlabeled, there is no error or reward signal to evaluate a potential solution. This distinguishes unsupervised learning from supervised learning and reinforcement learning [4]. Data clustering is seen as an increasingly important tool by modern business to transform unprecedented quantities of digital data into business intelligence giving an informational advantage. It is currently used in a wide range of profiling practices, such as marketing, surveillance, fraud detection, and scientific discovery [12]. In this thesis, we will focus on kernel-based clustering methods and their optimizations.

Kernel-based methods were introduced into the Machine Learning field by Aizerman et al. [1] in 1964 and have been successfully applied to a wide range of research problems such as classification [29], clustering [46], and dimension reduction [56]. Besides intrinsic kernel-based algorithms such as Spectral Clustering [46], many other clustering methods have been modified to incorporate kernels (e.g., K-means [27] and linear Support Vector Machine (SVM) [15]).

Estimation and learning methods utilizing positive definite kernels have become rather popular in Machine Learning. Since these methods have a stronger mathematical slant than earlier Machine Learning methods (e.g., neural networks [23]), there is also significant interest in the statistics and mathematics community for these methods. The key idea behind Kernel-based methods is to implicitly map the data into a high dimensional feature space, where each coordinate corresponds to one feature of the data items. This transforms the data into a set of points in the Euclidean space. Since the mapping can be quite general (not necessarily linear, for example), the relations found in this way are general. This is called the kernel trick, the mapping to the new space is defined by a function called the kernel function [8].

Applying kernel-based clustering algorithms to high-dimensional large-scale datasets is challenging: the similarity matrix [64] takes $O(N^2)$ complexity both in time and in space to compute and store. Moreover, most of the clustering algorithms suffer from the "curse of dimensionality" [23] that comes with high-dimensional data.

1.2 Problem Statement and Thesis Contributions

Our goal is to concurrently optimize the computing time and reduce memory requirement when running kernel-based clustering algorithms on large-scale high-dimensional datasets. At the same time, the proposed method offers great opportunity to parallelize the execution of kernel-based clustering algorithms on Multi-Core machines or cluster computing devices.

1.2.1 Problem Statement

As discussed in [64] [13], the computational resources required by kernel-based clustering, e.g. Spectral Clustering, in processing large-scale data are often prohibitively high. One common known issue is that similarity matrix takes $O(N^2)$ time and space to compute and store. Further, many techniques based on kernel methods require even more time (larger than $O(N^2)$) to compute. If faced with large datasets, these algorithms will simply fail to run, either due to memory insufficiency, or unacceptable long processing time.

It is identified that the intolerable size of similarity matrix is the root of computational incapability for most kernel-based Machine Learning algorithms when faced with large datasets. We propose to apply Locality Sensitive Hashing [43] to the original dataset and divide the dataset into several buckets, each of which contains points that are close to each other. We then run clustering algorithms with the approximated similarity matrix.

Furthermore, we leverage MapReduce framework [18] to parallelize the algorithm on cluster devices.

The proposed method not only works for kernel-based clustering algorithms such as Spectral Clustering [46] and affinity propagation [22], other algorithms running on large-scale datasets and seek for preprocessing data partitioning can also benefit from it.

1.2.2 Contributions

We propose an optimizing algorithm to improve the execution time and memory space demand of kernel-based clustering algorithms when clustering high-dimensional large datasets. Particularly, our contributions can be summarized as follows:

- We present the design of an algorithm which reduces the running time and memory demand of kernel-based clustering algorithms by approximating the similarity matrix. The proposed design is modular and can be used as an improvement preprocessing step in many clustering algorithms.
- We identify that existing kernel-based algorithms can be hardly parallelized due to shared usage of one single similarity matrix. The proposed scheme addresses this problem and enables the easy parallelization of kernel-based clustering algorithms. We show how the parallelization is achieved using MapReduce framework on commodity machines. We conduct experiments on a cluster of machines as well as Amazon Elastic MapReduce cluster to show performance gain.
- The experimental results show that the proposed algorithm can successfully retain the clustering accuracy (within 10% above Spectral Clustering), greatly reduce computing time (with the reduction ratio of 13 and more when comparing against Spectral Clustering, and with the reduction ratio of 10 and more when comparing against Parallel Spectral Clustering), and significantly reduce the memory footprint.

1.3 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, we introduce the MapReduce framework, and also provide a brief review on main classes of clustering algorithms and their respective parallelized implementation. In Chapter 3, we give an overview on Locality

Sensitive Hashing and how they can be used in our distributed environment, then move on to describe the proposed algorithm and conduct an analytical analysis on the effectiveness of the proposed framework. In Chapter 4 and Chapter 5, We describe the setup of our testbed and Amazon Elastic MapReduce cluster, discuss the experimental methodology and present the experimental results using synthetic dataset and Wikipedia dataset. We conclude the thesis in Chapter 6.

Chapter 2

Background and Related Work

In this chapter, we first give a short introduction to the MapReduce framework and one of its implementation Hadoop, then briefly review the main classes of clustering techniques used in Machine Learning fields and their distributed implementations. After that, we summarize previous works that are related to our work.

2.1 MapReduce Framework

2.1.1 Introduction to MapReduce

MapReduce is a programming model introduced by Google [18] in 2004 to support distributed computing on large datasets. The framework is inspired by the map and reduce functions commonly used in functional programming.

- ***Map*** step: A master node takes the input, partitions it into smaller sub-problems, and distributes them to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes that smaller problem, and passes the answer back to its master node.

  The map function is applied in parallel to every item in the input dataset. This produces a list of $(k2, v2)$ pairs for each call. After that, the MapReduce framework collects all pairs with the same key from all lists and groups them together, thus creating a group of values for each different key.

  $$Map(k1, v1) \rightarrow list(k2, v2)$$

- ***Reduce*** step: Each reduce call typically produces either one value $v3$ or an empty return. The returns of all calls are collected as the desired result list. Therefore, the MapReduce framework transforms a list of $(key, value)$ pairs into a list of values. This behavior is different from the typical functional programming map and reduce combination, which accepts a list of arbitrary values and returns one single value that combines all the values returned by map.

$$Reduce(k2, list(v2)) \rightarrow list(v3)$$

The advantage of MapReduce is that it allows for distributed processing of the map and reduce operations. Provided each map operation is independent of the others, all maps can be performed in parallel, though in practice it is limited by the data source and/or the number of CPUs near that data. Similarly, a set of reducers can perform the reduction phase, it only requires that all outputs of the map operation that share the same key are presented to the same reducer. While this process can often appear inefficient compared to algorithms that are more sequential, MapReduce can be applied to significantly larger datasets than commodity servers can handle. The parallelism also offers some possibility of recovering from partial failure of servers or storage during the operation: if one map or reduce fails, the work can be rescheduled, assuming the input data is still available.

2.1.2 Hadoop Architecture

Apache Hadoop is a software framework that supports data-intensive distributed application and was inspired by Google's MapReduce and Google File System (GFS). Hadoop is a client/server computing paradigm, and conceptually, it is made up of two parts. From the perspective of MapReduce, the master is called $JobTracker$, the clients are called $TaskTracker$. The master is responsible for scheduling the jobs' component tasks on the clients, monitoring them and re-executing the failed tasks. The clients execute the tasks as directed by the master. From the perspective of Hadoop Distributed File System (HDFS), a HDFS cluster primarily consists of a NameNode that manages the file system metadata and DataNodes that store the actual data.

There are two performance-critical components in Hadoop, data placement policy and task scheduling policy.

Data Placement Policy

HDFS stores each file as a sequence of blocks; all blocks in a file except the last block are the same size (64MB by default). The blocks of a file are replicated for fault tolerance.

Network locations such as nodes and racks are represented as a tree, which reflects the network distance between locations. The namenode uses the network location when determining where to place block replicas. The jobtracker uses network location to determine where the closest replica is as input for a map task that is scheduled to run on a tasktracker.

Hadoop's strategy is to place the first replica on the same node as the client (for clients running outside the cluster, a node is chosen at random, although the system tries not to pick nodes that are too full or too busy). The second replica is placed on a different rack from the first one at random. The third replica is placed on the same rack as the second, but on a different node chosen at random. Further replicas are placed on random nodes in the cluster, although the system tries to avoid placing too many replicas on the same rack [62].

Task Scheduling Policy

A task assignment is initiated by a *TaskTracker*, then *JobTracker* responses. Tasks are assigned in response to heartbeats (status messages) received from the clients every few seconds. Within the message of a heartbeat, a tasktracker indicates whether it is ready to grab a new task, and if it is, the jobtracker will allocate it a task. Each client has a fixed number of map slots and reduce slots for tasks. The default scheduler fills empty map task slots before reduce task slots, therefore, if the tasktracker has at least one empty map task slot, the jobtracker will select a map task. Otherwise, it will select a reduce task. This is to ensure that maximum effort is given to run map tasks in order to finish them early, since reduce tasks can not run without the intermediate result produced by map tasks.

Hadoop's built-in scheduler runs jobs in FIFO order, with five priority levels. When a task slot becomes free, the scheduler scans through jobs in the order of priority and submission time to find one with a task of the required type.

For map tasks, the scheduler uses a locality optimization technique. After selecting a job, the scheduler picks the map task in the job with data closest to the slave, on the same node if possible, otherwise on the same rack, or finally on a remote rack.

However, for reduce tasks, each reduce task has to copy its portion of the results from all the map tasks, and can only apply the user's reduce function once it has results from all maps. Therefore, little locality can be observed in reduce step. The jobtracker just takes the next in the reduce tasks' list and assign it to the tasktracker.

2.1.3 Other Parallel Distributed Programming Models

OpenMP (Open Multi-Processing) [54] is an API (Application Programming Interface) that supports multi-platform shared memory multiprocessing programming. Therefore, this programming model entails that all threads have access to the same, globally shared memory. Data can be shared or private, shared data is accessible by all thread, while private data can be accessed only by the threads that owns it. However, this model only runs efficiently in shared-memory multiprocessor platforms, and its scalability is limited by memory architecture [54].

Message Passing Interface (MPI) [25] is an Application Programming Interface specification that allows processes to communicate with one another by sending and receiving messages. Besides many other applications, it is a standard for parallel programs running on computer clusters and supercomputers, where the cost of accessing non-local memory is high. Moreover, programmes that create tasks dynamically or place multiple tasks on a processor can require substantial refinement before they can be implemented in MPI, since the programme itself has to deal explicitly with message passing, and the implementation is difficult to debug [25].

Compared against the above two paradigms, MapReduce, being a paradigm-oriented programming model, provides uniform Map and *Reduce* interfaces and most of the parallelization and synchronization work is handled by the framework itself [62]. It allows for distributed processing of the map and reduce operations, each map operation is independent of the others, all maps can be performed in parallel. Similarly, a set of reducers can perform the reduction phase. Various implementations of MapReduce framework in different programming languages are available, for example, Misco is a MapReduce designed for mobile devices and is implemented in Python. Greenplum is a commercial MapReduce implementation, with support for Python, Perl, SQL and other languages. We choose Hadoop - a Java MapReduce implementation to do the experiments.

Distance	Formula
Euclidean distance	$\|\|x-y\|\|_2 = \sqrt{\sum\limits_i (a_i - b_i)^2}$
Squared Euclidean distance	$\|\|x-y\|\|_2^2 = \sum\limits_i (a_i - b_i)^2$
Manhattan distance	$\|\|x-y\|\|_1 = \sum\limits_i \|a_i - b_i\|$
Maximum distance	$\|\|x-y\|\|_\infty = \max_i \|a_i - b_i\|$
Mahalanobis distance	$\sqrt{(a-b)^T S^{-1}(a-b)}$where S is the covariance matrix
Cosine distance	$\frac{a \cdot b}{\|\|a\|\|\|\|b\|\|}$

Table 2.1: Distance measurement.

2.2 Overview of Clustering Algorithms

Clustering is an unsupervised data pattern extraction process that aims at grouping similar patterns into the same category based on certain measures of “distance” [4]. Cluster analysis aims to discover clusters or groups of samples such that samples within the same group are more similar to each other than they are to the samples of other groups. It is a kind of statistical data analysis widely used in many fields, including Machine Learning, Data Mining. It has led to the design and analysis of a wide spectrum of algorithms.

When talking about clustering, a distance measurement is needed to represent the closeness of two points. Commonly used distance of measurements are listed in Table. 2.1. In this thesis, Euclidean distance is used as the default measurement if not otherwise noted.

The similarity matrix (also known as Gram matrix or G-matrix) is a matrix of scores which expresses the similarity between data points. Let $\{X_i\}_{i=1}^N$ be a set of points in $\mathbb{R}^D$ (D is the dimensionality), the similarity matrix is an $N \times N$ matrix, the $(i,j)^{th}$ entry is given by

$$S_{ij} = ||X_i - X_j||_2, \tag{2.1}$$

where $(||expressions||_2$ stands for $L2$ distance or Euclidean distance).

Similarity matrix plays a crucial role in kernel-based methods. Techniques such as linear Support Vector Machines (SVM) Gaussian Processes [32] can only extract structure from the original dataset by computing linear functions, e.g., functions in the form of dot products. When there are nonlinear structures in the data, the above mentioned linear techniques will lose effectiveness [15]. Kernel-based learning methods have proved to be useful in the above mentioned scenarios [15]. For instance, in kernel-based SVM [15], data items are mapped into high-dimensional space, where information about their mutual positions (in the form of inner products) is used for constructing classification, regression, or clustering rules. Kernel-based algorithms exploit the information encoded in the inner product between all pairs of data items and are successful in part because there is often an efficient method to compute inner products between very complex or even infinite dimensional vectors. Thus, kernel-based algorithms provide a way to deal with nonlinear structure by reducing nonlinear algorithms to algorithms that are linear in some feature space that is nonlinearly related to the original input space. Moreover, Similarity matrices are also widely used in sequence alignment in biomedical studies [24]. Higher scores are given to more-similar characters, and lower or negative scores for dissimilar characters. Nucleotide similarity matrices are used to align nucleic acid sequences [24].

There are several genres of clustering algorithms. In the following sections, we introduce some important classes and explain how they can be implemented in a distributed manner.

2.3 Hierarchical Clustering Algorithms

Hierarchical algorithms [10] find successive clusters using previously established clusters. These algorithms usually are either agglomerative (bottom-up) or divisive (top-down). Agglomerative algorithms begin with each element as a separate cluster and merge them into successively larger clusters. Divisive algorithms begin with the whole set and proceed to divide it into successively smaller clusters.

For agglomerative approach, given a set of N points to be clustered, and an $N \times N$ distance (or similarity) matrix, the process of hierarchical clustering is as follows:

- ***Step 1***: Start by assigning each item to its own cluster, so that if we have N items, we now have N clusters, each containing just one item. Let the distances (similarities) between the clusters equal the distances (similarities) between the items they contain.
- ***Step 2***: Find the closest (most similar) pair of clusters and merge them into a single cluster, so that now we have one less cluster.
- ***Step 3***: Compute distances (similarities) between the new cluster and each of the old clusters.

- ***Step 4***: Repeat steps 2 and 3 until all items are clustered into a single cluster of size N.

Step 3 can be done in different ways, which is what distinguishes single-link from complete-link and average-link clustering. In single-link clustering [10], we consider the distance between one cluster and another cluster to be equal to the shortest distance from any member of one cluster to any member of the other cluster. If the data consist of similarities, we consider the similarity between one cluster and another cluster to be equal to the greatest similarity from any member of one cluster to any member of the other cluster. In complete-link clustering [10], we consider the distance between one cluster and another cluster to be equal to the longest distance from any member of one cluster to any member of the other cluster. In average-link clustering, we consider the distance between one cluster and another cluster to be equal to the average distance from any member of one cluster to any member of the other cluster.

It is inefficient to apply the agglomerative model to MapReduce, because each distributed task needs the entire dataset to make choices about appropriate clusters. It also needs a list of clusters at its current level such that it does not add a data point to more than one cluster at the same level. This is a class of applications that is unsuitable for MapReduce [62].

2.4 Partitional Clustering Algorithms

Partitional algorithms typically determine all clusters at once, and can also be used as divisive algorithms in the hierarchical clustering described in Section 2.3.

2.4.1 K-means

K-means [27] follows a simple and easy way to classify a given dataset through a certain number of clusters (assume k clusters fixed as a priori). The main idea is to define k centroids, one for each cluster. These centroids should be placed in a clever way, because different location causes different result. Therefore, a good choice is to place them as much as possible far away from each other. The next step is to take each point belonging to a given dataset and associate it to the nearest centroid. When no point is pending, the first step is completed and an early clustering is done. At this point we need to re-calculate

k new centroids as centers of the clusters resulting from the previous step. After we have these k new centroids, a new binding has to be done between the same dataset points and the nearest new centroid. A loop has been generated. As a result of this loop we may notice that the k centroids change their location step by step until no more changes are done. In other words centroids do not move any more.

This algorithm aims at minimizing an objective function, in this case a squared error function. The objective function is:

$$J = \sum_{j=1}^{k} \sum_{i=1}^{N} ||x_i^{(j)} - c_j||^2, \tag{2.2}$$

where $||x_i^{(j)} - c_j||^2$ is a chosen distance measure between a data point $x_i^{(j)}$ and the cluster center c_j, the algorithm is composed of the following steps:

- ***Step 1***: Place k points into the space represented by the objects that are being clustered. These points represent initial group centroids.
- ***Step 2***: Assign each point to the group that has the closest centroid.
- ***Step 3***: Recalculate the positions of the k centroids when all points have been assigned.
- ***Step 4***: Repeat Steps 2 and 3 until the centroids no longer move. This produces a separation of the points into groups from which the objective function to be minimized can be calculated.

A MapReduce parallel algorithm for K-means clustering has been proposed by Zhao et al. [49]. Initially k points are chosen as the cluster centers. Each map function gets a partition of the data and accesses this partition in each iteration. The variable data is the current cluster centers calculated during the previous iteration, hence it is used as the input value for the map function. All the map functions get this same input data (current cluster centers) at each iteration and computes partial cluster centers by going through its dataset. Note that each map function can identify the nearest cluster for its data points because it has access to all cluster centers. Thus, each map function is able to update the cluster membership for its data points. A reduce function computes the average of all points for each cluster based on the updated membership and produces the new cluster centers for the next step. Once it gets these new cluster centers, it calculates the difference between the new cluster centers and the previous cluster centers and determines if it needs to execute another cycle of MapReduce computation.

2.4.2 Affinity Propagation

Frey and Dueck present an algorithm called affinity propagation [22] which clusters data points via passing messages between the points. Affinity propagation clusters data by diffusion in the similarity matrix. Input consists of a collection of real-valued similarities between data points. These are represented by $s(i,k)$ which describes how well data point k is suited to be exemplar for data point i. In addition, each data point is supplied with a preference value $s(k,k)$ which specifies a priori on how likely each data point is to be an exemplar.

There are two kinds of messages exchanged between data points, responsibilities and availabilities. Responsibility $r(i,k)$ reflects the accumulated evidence for how well-suited point k is to serve as the exemplar for point i, taking into account other potential exemplars for point i. Responsibility is sent from point i to candidate exemplar k. Availability $a(i,k)$ reflects accumulated evidence for how appropriate it would be for point i to choose k as its exemplar, taking into account the support from other points that point k should be an exemplar. Availability is sent from candidate exemplar k to data point i. In general the algorithm works in three steps:

- Update responsibilities given availabilities. Initially this is a data driven update. Over time it lets candidate exemplars compete for ownership of the data.
- Update availabilities given the responsibilities. This gathers evidence from data points as to whether a candidate exemplar is a good exemplar.
- Monitor exemplar decisions by combining availabilities and responsibilities. Terminate if reaching a stopping point (e.g. insufficient change). The update rules require simple local computations and messages are exchanged between pairs of points with known similarities.

The update is done according to the following formulas:

$$r(i,k) = (1-\lambda) \times r(i,k) + \lambda \times (s(i,k) - \max_{k':k'\neq k}(a(i,k') + s(i,k'))), \tag{2.3}$$

$$a(i,k) = (1-\lambda) \times a(i,k) + \lambda \times \min\{0, r(k,k) + \sum_{i':i'\not\subset i,k} max(0, r(i',k))\}, \tag{2.4}$$

$$a(k,k) = (1-\lambda) \times a(k,k) + \lambda \times \sum_{i':i'\neq k} \max(0, r(i',k)), \tag{2.5}$$

where λ is the damping factor used to avoid numerical oscillations.

We can see from the above formulas, in every iteration, it involves both row scan and column scan, therefore, it is impossible to parallelize the whole process use MapReduce. One possible method is to parallelize r matrix calculation and a calculation separately in a single loop. Wang et al. [60] propose a method to parallelize Affinity propagation. To parallelize $r(i,k)$, all the values of $a(i,k')$ and $s(i,k')$ are taken as map input. In reduce part, all the values taking i as key are passed into the reduce function as a list. To parallelize $a(i,k)$, all the values of $r(i',k)$ are taken as input, since all the key/value pairs are indexed by the column after $r(i,k)$ calculation, all the values in the k-th column are organized as a list.

2.5 Soft Clustering Algorithms - Dirichlet Process Clustering (DPC)

"Soft", in this context, means that a point can fall into two (or more than two) clusters. The algorithms in this genre use probabilistic generative models. Dirichlet Process Clustering (DPC) [7] falls into this genre.

The calculation goes in an iterative manner. There are mainly three steps in each iteration [7]:

- Assign the input data points to a certain cluster.
- Update the model by computing the posterior probabilities and new model parameters by sampling from a Dirichlet distribution.
- Repeat the second step until convergence is reached.

One way of parallelizing DPC using MapReduce framework is described in [49]. In every iteration, there are map and reduce functions. In map function, it assigns the input data points to a certain cluster, then emits the pair $< pointID, clusterID >$. In reduce function, it updates the models by computing the posterior probabilities and new model parameters. Then it emits the pair as the output of this iteration. This loop goes for a predefined iterations.

2.6 Spectral Clustering

Spectral Clustering [46] is a class of methods based on eigen decomposition of affinity, dissimilarity or kernel matrices [46]. Many clustering methods are strongly tied to Euclidean distance, making assumptions that clusters form convex regions in Euclidean space. Spectral methods are more flexible, capturing a wider range of geometries. Spectral Clustering is widely used in Document Clustering [66], Image Segmentation [61]. Moreover, there is a substantial theoretical literature supporting Spectral Clustering [46].

The idea of Spectral Clustering is to form a pairwise similarity matrix S, compute Laplacian matrix L and compute eigenvectors of L. It is shown that the second eigenvector of the normalized graph Laplacian is a relaxation of a binary vector solution that minimizes the normalized cut on a graph [46].

Given a set of points $\{X_i\}_{i=1}^N$ in $\mathbb{R}^d$ (d is the dimensionality), The algorithm proceeds as follows:

- ***Step 1***: Form the similarity matrix $S \in \mathbb{R}^{N\times N}$ defined by $S_{ij} = exp(-||X_i - X_j||^2/2\sigma^2)$ if $i \neq j$, and $S_{ii} = 0$.

- ***Step 2***: Define D to be the diagonal matrix whose (i,i)-element is the sum of S's i-th row, and construct the matrix $L = D^{-1/2}AD^{-1/2}$.

- ***Step 3***: Find $V_1, V_2..., V_k$, the top k eigenvectors of L, and form the matrix $X = [V_1V_2...V_k] \in \mathbb{R}^{N\times k}$ by stacking the eigenvectors in columns.

- ***Step 4***: Form the matrix Y from X by renormalizing each of X's rows to have unit length, $Y_{ij} = X_{ij}/(\sqrt{\sum_j X_{ij}^2})$.

- ***Step 5***: Treat each row of Y as a point in $\mathbb{R}^k$, cluster them into k clusters using K-means [27].

- ***Step 6***: Assign the original point X_i to cluster j if and only if row i of the matrix Y is assigned to cluster j.

The scaling parameter σ^2 controls how rapidly the affinity S_{ij} decreases with the distance between X_i and X_j. The eigenvectors of a square matrix are the non-zero vectors that, after being multiplied by the matrix, remain parallel to the original vector. For each eigenvector, the corresponding eigenvalue is the factor by which the eigenvector is scaled when multiplied by the matrix.

The advantages of Spectral Clustering can be summarized as follows:

- Spectral Clustering performs well with non-Gaussian clusters [46].
- As opposed to K-means clustering, which results in convex sets, Spectral Clustering can solve problems such as intertwined spirals, because it does not make assumptions on the form of the cluster. This property comes from the mapping of the original space to an eigen space.
- Spectral Clustering does not intrinsically suffer from the problem of local optima [46]. Spectral Clustering performs significantly better than the linkage algorithm [41] even when the clusters are not well separated.

Spectral Clustering is parallelized in MapReduce framework [49]. In Laplacian matrix computing part, since it involves matrix multiplication, the parallelization computes each entry in Laplacian matrix separately and reduces the results into one single matrix. In the K-means clustering part, it is parallelized in a way that is introduced in Section 2.4. Spectral Clustering has also been parallelized using MPI by Chen et al. [13].

2.7 Challenges in Clustering Large Datasets

Clustering large-scale datasets using kernel-based methods is challenging. The difficulties are summarized into three folds:

- The similarity matrix, on which many kernel-based clustering algorithms rely, needs $O(N^2)$ complexity in time to compute. When running on large datasets, it can be intolerably long.

- The similarity matrix needs $O(N^2)$ memory space to store. Although the state-of-art high performance computers have very large DRAM, when applying large datasets, the memory will still be insufficient and thrashing is likely to occur in the paging system, which make the running process even slower.
- Most of the clustering algorithms suffer from “curse of dimensionality” [23] that comes with high-dimensionality data. Essentially, the amount of data to sustain a given spatial density increases exponentially with the dimensionality of the input space, or alternatively, the sparsity increases exponentially given a constant amount of data, with points tending to become far apart from one another. In general, this will adversely impact any method based on spatial density, unless the data follows certain simple distributions.

2.8 Related Work

Clustering large datasets is an actively researched area [68] [12] [36]. For example, Kulis and Grauman [36] target large-scale scalable image search. They generalize locality-sensitive hashing to accommodate arbitrary kernel functions, therefore permitting sub-linear time approximate similarity search. Another example is BIRCH [68], which is a method for pre-clustering large datasets. BIRCH incrementally groups the data as tightly as possible based on similarity in their attributes. In the process, it constructs as many representatives of the original data as the available memory can contain. If the amount of space taken by the data runs out during the BIRCH process, the tightness of the groupings is relaxed, the groupings are internally re-assigned, and the processing of the data continues. Therefore, by monitoring the system’s available memory, BIRCH can dynamically adjust the clustering policies. Chen and Liu [12] propose “iVIBRATE”, which is an interactively visualization based three-phase framework for clustering large datasets. Two main components of iVIBRATE are its visual cluster rendering subsystem, which enables human to join the large-scale iterative clustering process through interactive visualization, and its Adaptive ClusterMap Labeling subsystem, which offers visualization-guided disk-labeling solutions that are effective in dealing with outliers, irregular clusters, and cluster boundary extension. However, when presented with high-dimensional large-scale datasets, the visual presentation will not work because of human’s incapability of high-dimensional data perception.

Clustering in high dimensional spaces is problematic as well [5]. The problem of high dimensionality is typically addressed by feature selection and dimension reduction techniques [56]. Radovanovic et al. [53] study the number of times a point appears among the k nearest neighbors of other points in a dataset. They identify that, as dimensionality increases, this distribution becomes considerably skewed and points with very high occurrences emerge. This is a new property that is discovered in high-dimensional datasets. Murtagh et al. also explore hierarchical clustering on high-dimensional datasets using ultrametric embedding [44]. An ultrametric is a metric which satisfies the following triangle inequality: $d(x, z) \leq \max(d(x, y), d(y, z))$ for all points x, y and z. By quantifying the degree of ultrametricity [3] in a dataset, Bartal et al. show that ultrametricity becomes pervasive as dimensionality and/or spatial sparsity increases.

Clustering optimization for specific applications has also been studied. For example, in [40], Manku utilizes simHash [11] to generate a signature for each web page and then separate web-scale dataset apart into different buckets. However, their work is optimized for a specific application and is potentially difficult to generalize, while our work is designed to fit the majority of main stream kernel-based clustering algorithms.

Research on speeding up the execution of Spectral Clustering can be summarized in two directions. One direction is subsampling approach, selecting data points following some form of stratification procedure. One of the best known approximation schemes is sampling using the Nystrom method [63]. This method allows extrapolation of the complete grouping solution using only a small number of typical samples. Fowlkes et al. [21] propose using the Nystrom extension to approximate the eigenvectors of the affinity matrix. They investigate the Nystrom extension solution to the Normalized Cut problem [46] in Spectral Clustering and apply the method in image segmentation. In [57], Schuetter et al. extend the above mentioned method and propose a multiple subsample approach that improves the accuracy of the Nystrom extension.

The other direction is approximating the affinity matrix. Cullum and Willoughby [16] introduce the Lanczos algorithm, which uses a sparse eigensolver to do the decomposition. Matrix sparsity can be created by computing entries for only those elements within an ϵ-neighborhood of each other, or for nearest neighbors. However, this method still runs in $O(N^2)$ time. Drineas et al. [19] develop an algorithm to compute a low-rank approximation to similarity matrix. The low-rank approximation problem is to approximate a matrix by one of the same dimension but smaller rank with respect to some norm, e.g., the Frobenius norm. They use a data-dependent nonuniform probability distribution to randomly sample the columns.

We identify two recent works that are close to ours [64] [13]. Yan et al. [64] propose leveraging K-means to pre-process the original datasets and pre-group them. Although we share the same rationale, as shown later in the thesis, our method can be applied to many clustering algorithms beyond the Spectral Clustering algorithm. Also, the method in [64] does not consider the complexity when partitioning the original dataset. On the contrary, our method can achieve $O(N)$ when doing the partitioning step, and the constant factor in $O(N)$ is small. Chen et al. [13] propose parallel Spectral Clustering method that approximates the affinity matrix and compare it against the Nystrom extension method. The method forms the sparse matrix via retaining a group of nearest neighbors. To retain nearest neighbors, for every point in the dataset, the method computes the similarity between this point and others. It keeps a heap to retain the closest points. The method is essentially a variation of k-nearest neighbor search [46]. The advantage of the method is small memory footprint. However, the method still has to compute the similarity for every pair of points, thus its time complexity remains unchanged at $O(N^2)$. On the contrary, our method can achieve $O(N)$ complexity in the preprocessing step with MapReduce framework, and it only computes the similarity values between a subset of data points instead of all points. Therefore, the proposed method can achieve sub-quadratic complexity in both time and space.

We compare the proposed method, which we call distributed Spectral Clustering ($DASC$) against three existing methods: the original Spectral Clustering algorithm (denoted as SC) [46], Parallel Spectral Clustering (denoted as PSC) [13] and Nystrom approximation method (denoted as $NYST$) [21]. SC serves as the baseline comparison, PSC uses nearest-neighbor-search to reduce the size of similarity matrix and is parallelized using MPI, and $NYST$ is a recently proposed scheme which leverages K-means to pre-group neighboring points and produces a set of reduced representative points for Spectral Clustering.

Chapter 3

Proposed Distributed Spectral Clustering Algorithm

3.1 Overview

Kernel-based clustering techniques perform computation based on similarity matrix. In order to reduce the complexity of this expensive operation, which is $O(N)^2$ where N is the dataset size, we propose a preprocessing step using Locality Sensitive Hashing to approximate the similarity matrix. In the similarity matrix, every element reflects the similarity between two points. The larger the element is, the closer those two points are. In the proposed algorithm, those small elements in the original similarity matrix will not be computed and they will be omitted. The proposed method decides whether or not to compute an element in similarity matrix using Locality Sensitive Hashing [43]. Every point in the dataset is labeled with a signature, identical or similar signatures mean that the corresponding points are close to each other, thus their similarity value is large. Big difference between two signatures means that the corresponding points are far away from each other, thus they are unlikely to be in the same cluster and their similarity value is small. Therefore, we do not compute similarity for such two points.

There are several steps in the proposed algorithm. In the first step, the algorithm iterates through the dataset once to generate signatures. In the second step, points whose signatures are similar to each other are projected into one bucket. In the third step, similarity values are computed for the points in each bucket to form the approximated matrix. In the fourth step, Spectral Clustering algorithm will run on the approximated similarity matrix and output the clustering results.

In the following sections, we will first review different families of Locality Sensitive Hashing and then describe the proposed algorithm and implementation details.

3.2 Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) is a probabilistic dimension reduction technique [43]. The idea is to hash points such that the probability of collision is much higher for close points than for those that are far apart. Those points whose hashing values collide with each other fall into the same bucket. Then, one can determine nearby neighbors by hashing the query point and retrieving elements stored in buckets containing that point.

For a domain S of the points set with distance measurement D, an LSH family is defined as follows [43].

Definition 1 (LSH Family) *A family $H = \{h : S \rightarrow U\}$ is called (d_1, d_2, p_1, p_2)-sensitive for D if for any $x, y \in S$, we have:*

- *if $x \in B(y, d_1)$ then $Pr_H[h(x) = h(y)] \geq p_1$,*
- *if $x \notin B(y, d_2)$ then $Pr_H[h(x) = h(y)] \leq p_2$,*

where $p_1 > p_2$, $d_1 < d_2$, and $B(y, d)$ is square area centered at y with radius d. The behavior is shown in Fig. 3.1.

3.2.1 Stable Distributions

According to [47], a distribution is a stable distribution if it satisfies the following property: Let X_1 and X_2 be independent copies of a random variable X. Random variable X is said to be stable if for any constants a and b the random variable $a \cdot X_1 + b \cdot X_2$ has the same distribution as $c \cdot X + d$ with some constants c and d. The distribution is said to be strictly stable if this holds with $d = 0$.

A distribution $\mathcal{D}$ over $\mathbb{R}$ is called $p - stable$, if there exists $p \geq 0$ such that for any n real numbers $v_1, ..., v_n$ and independent identically distributed variables $X_1, ..., X_n$ with distribution $\mathcal{D}$, the random variable $\sum_i v_i \cdot X_i$ has the same distribution as the variable $(\sum_i |v_i|^p)^{1/p} \cdot X$, where X is a random variable with distribution $\mathcal{D}$.

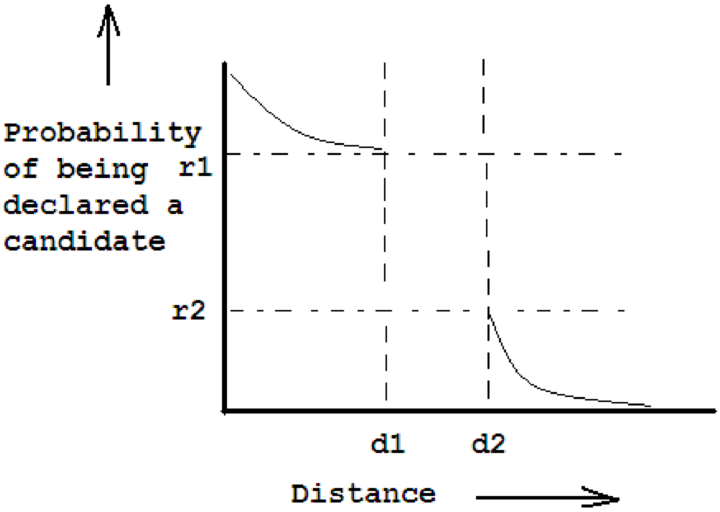


Figure 3.1: Behavior of a $(d1, d2, p1, p2)$-sensitive function.

In the case of $\mathbb{R}^D$ with L_2 distance, p-stable distribution is defined as:

$$H(v) =< h_1(v), h_2(v), ..., h_M(v) >, \tag{3.1}$$

$$h_i(v) = \lfloor \frac{a_i \cdot v_i + b_i}{W} \rfloor, i = 1, 2, ..., M, \tag{3.2}$$

where $a_i \in \mathbb{R}^D$ is a vector with entries chosen independently from the Gaussian distribution $N(0, 1)$ and b_i is drawn from the uniform distribution $U[0, W)$. For different i, a_i and b_i are sampled independently. The parameters M and W control the locality sensitivity of the hash function.

3.2.2 Random Projection

The random projection method [9] is designed to approximate the cosine distance between vectors. The idea of this technique is to choose a random hyperplane (defined by a normal unit vector r) at the outset and use the hyperplane to hash input vectors.

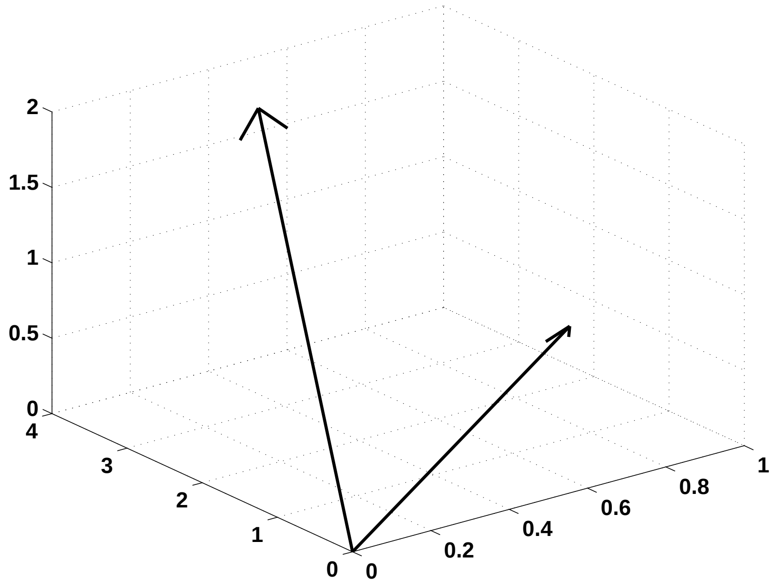


Figure 3.2: Two vectors in three dimensions making an angle.

Given two vectors x and y, the cosine of the angle between them is the dot product $x \cdot y$ divided by the $L_2 - norm$ of x and y, for instance, their Euclidean distances from the origin. In Figure 3.2, it can be observed that two vectors x and y make an angle θ between them. It should be noted that the vectors may be in a many-dimension-space, however, they always define a plane, and the angle between them is measured in this plane. Given an input vector v and a hyperplane defined by r, we let $h(v) = sgn(v \cdot r) = \pm 1$. Each possible choice of r defines a single function. Let H be the set of all such functions and let D be the uniform distribution. For two vectors u,v, we have:

$$Pr[h(u) = h(r)] = 1 - \frac{\theta(u,v)}{\pi}, \tag{3.3}$$

where $\theta(u,v)$ is the angle between u and v.

Given m functions in this family, there will be m bits in the result for every input data point. These m bits are concatenated and serve as a unique signature for the input vector. Points having a large portion of corresponding bits identical are considered to be close to each other.

3.2.3 Summary of Locality Sensitive Hashing

Besides what is covered above, there are other Locality Sensitive Hashing families. For instance, Min-Wise Independent Permutations [14], which uses Jaccard Index [33] to group similar documents together. The documents do not contain explicit numerical features. Locality Sensitive Hashing for Hamming distance operates on binary sequences when the input points live in the Hamming space. For p-stable distribution, Equation (3.2) essentially projects a multi-dimensional vector onto one-dimensional space. We need to tune the parameters b_i, W and M in Equation (3.2) based on the data characteristics. Also, the basic Locality Sensitive Hashing index requires large W, M to satisfy certain recall. This trades space for fast query time. The hash function we use to generate the signatures falls into the family of Random Projection. The advantage of this family is that, after applying hashing function once, we only need one bit to store the result. This characteristic saves a lot of storage space. It has been shown that Random Projection has the best performance in high-dimensional data clustering [20]. Also, with Random Projection, we can compare the generated signatures using hamming distances for which efficient algorithms are available [11].

3.3 Distributed Approximate Spectral Clustering

The proposed algorithm approximates the similarity matrix, speeds up the execution of kernel-based clustering techniques and reduce the memory footprint. Moreover, it can be parallelized easily using MapReduce framework.

For quick reference, we list all symbols used in this chapter in Table 3.1.

The approximation scheme goes in the following steps, ***Step 3*** to ***Step 8*** are the steps in Spectral Clustering:

- ***Step 1***: Create signatures for data points. Every point in the dataset is represented by an index and a group of numerical features. The signature is an M-bit string, each bit is generated as follows: one feature in the dimensionality space is selected. This feature is then compared with a *threshold*. If the feature is larger than the *threshold*, the resulting bit is set to 1, otherwise it is set to 0. This process goes M times, and the resulting bits are concatenated to make the signature. The complexity of this step is $O(M \cdot N)$. This hash function falls into Random Projection family, because every

Symbol	Description
$X_1, ..., X_N \in R^d$	Data point
N	Total number of points in the dataset
d	Dimensionality (feature space)
$Dim[i](i \in [0, d-1))$	Dimension space
K	Number of desired clusters
C	The number of machines in cluster
M	The length of the binary signature
P	Minimum number of identical bits in two binary signatures
T	The number of buckets generated
$S \in R^{N \times N}$	Similarity matrix
$L \in R^{N \times N}$	Laplacian matrix
D	Diagonal matrix, $D_{ii} = \sum\limits_{j=1}^{N} S_{ij}$
A	Symmetric tridiagonal matrix

Table 3.1: List of symbols used in this chapter.

time we apply this hash function, we project all the points in a dataset onto a line. After that, the hashing result is determined by comparing the projection value against a threshold.

- ***Step 2***: All points that have near-duplicate signatures are projected into one bucket. Near-duplicate signatures mean that for two M-bit binary numbers, there are at least P bits in common. Suppose the total number of different signatures generated is T, the complexity of this step is $O(T^2)$.

- ***Step 3***: Compute the sub similarity matrix in every bucket. Suppose we have altogether T different signatures, we then have T buckets, each of which has N_i (where $0 \leq i \leq T-1$ and $\sum\limits_{i=0}^{T-1} N_i = N$) points, the overall complexity of this step is $\sum\limits_{i=0}^{T-1} O(N_i^2)$. The similarity function is:

$$S_{ij} = exp(-\frac{||X_i - X_j||^2}{2\sigma^2}), \tag{3.4}$$

where σ is a scaling parameter to control how rapidly the similarity S_{ij} reduces with the distance between X_i and X_j.

- ***Step 4***: Construct the Laplacian matrix for each sub similarity matrix S_i in ***Step 3***:

  $$L_i = D_i^{-1/2} S_i D_i^{-1/2}, \tag{3.5}$$

  where $D_i^{-1/2}$ is the inverse square root of D_i, D_i is diagonal matrix. For an $N_i \times N_i$ diagonal matrix, the complexity of finding inverse square root is $O(N_i)$. Moreover, the complexity of multiplying an $N_i \times N_i$ diagonal matrix with an $N_i \times N_i$ matrix is $O(N_i^2)$. Therefore, the complexity of this step is $O(\sum_{i=0}^{T-1} N_i^2)$.

  However, if the similarity matrix is sparse, the actual running time can be greatly reduced, as we can skip the computation of zero entries in similarity matrix. For a similarity matrix that contains U non-zero entries, the complexity of multiplying it with a diagonal matrix is $O(U)$.

- ***Step 5***: Find $V_1, V_2 ..., V_{K_i}$, the first K_i eigenvectors of L_i, and form the matrix $X_i = [V_1 V_2 ... V_{K_i}] \in R^{N_i \times K_i}$ by stacking the eigenvectors in columns. To do this, we first transform L_i into a symmetric tridiagonal matrix $K_i \times K_i$ A_i. The transformation routine is shown in Fig. 3.3. The tridiagonal matrix A_i is constructed using the derived α_i and β_i ($i \in [1, K_i]$) in Fig. 3.3.

  The complexity of the transformation is $K_i \cdot N_i$. However, if L_i is a sparse matrix, the actual running time can be greatly reduced, as we can skip the computation of zero entries. This transformation preserves the eigenvectors and the eigenvalues of the original matrix, and can greatly speed up the following QR decomposition [16]. Specifically, QR decomposition on a tridiagonal matrix has complexity of $O(K_i)$, in contrast, QR decomposition on a general matrix is $O(K_i^3)$ [52].

  Following this, we do eigen decomposition on A_i using QR decomposition, as mentioned above, the complexity is $O(K_i)$. Therefore, the total complexity of this step is $O(\sum_{i=0}^{T-1} (K_i \cdot N_i))$.

- ***Step 6***: Form the matrix Y from X by renormalizing each of X's rows to have unit length, $Y_{ij} = X_{ij} / (\sqrt{\sum_j X_{ij}^2})$. The complexity of this step is $O(N)$.

```
**Algorithm 1**: TransformationToTridiagonalMatrix
/* $\boldsymbol{L}$ is the matrix, $\boldsymbol{q_1}$ is a random vector with $|\boldsymbol{q_1}| = 1$ */
$\boldsymbol{q_0} = \bar{\boldsymbol{0}}$ ;
$\beta_1 = 0$ ;
for $i = 1$ to $k$ do
    $\boldsymbol{u} = \boldsymbol{L}\boldsymbol{q_i} - \beta\boldsymbol{q_{i-1}}$;
    $\alpha_i = \boldsymbol{u} \cdot \boldsymbol{q_i}$;
    $\boldsymbol{u} = \boldsymbol{u} - \alpha_i\boldsymbol{q_i}$;
    $\beta_{i+1} = |\boldsymbol{u}|$;
    $\boldsymbol{q_{i+1}} = \boldsymbol{u}/\beta_{i+1}$ ;
end
/* Construct A */
$A = \begin{bmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \beta_3 & \ddots & \ddots & \\ & & \ddots & \alpha_{k-1} & \beta_k \\ & & & \beta_k & \alpha_k \end{bmatrix}$
```

Figure 3.3: Transformation to tridiagonal matrix routine.

- ***Step 7***: Treat each row of Y as a point in R^{K_i}, cluster them into K_i clusters using K-means [27]. The complexity of this step is $O(\sum_{i=0}^{T-1}(K_i \cdot N_i))$.
- ***Step 8***: Assign the original point X_i to cluster j if and only if row i of the matrix Y is assigned to cluster j. The complexity of this step is $O(N)$.

After ***Step 1*** and ***Step 2***, points in every bucket run Spectral Clustering (from ***Step 3*** to ***Step 8***). Therefore, the total complexity of the proposed algorithm is:

$$TimeComplexity = O(M \cdot N) + O(T^2) + \sum_{i=0}^{T-1} \left[2 \cdot O(N_i^2) + 2 \cdot (K_i \cdot N_i)\right] + 2 \cdot N. \tag{3.6}$$

The hyperplane value and threshold value are important factors in the hash function. The threshold value controls at which threshold we separate the original dataset apart, the hyperplane value controls which feature space to compare with the corresponding threshold. We use the principle of k-dimensional tree (k-d tree) [35] to set hyperplane value and threshold value. The k-d tree is a binary tree in which every node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as subspaces. Points to the left of this hyperplane are represented by the left subtree of that node and points right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k-dimensions, with the hyperplane perpendicular to that dimension's axis. For example, if for a particular split the "x" axis is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points with larger "x" value will be in the right subtree. In such a case, the hyperplane would be set by the x-value of the point, and its normal would be the unit x-axis.

To determine the *hyperplane* array, we look at each dimension of the dataset, and calculate the numerical span for all the dimensions (denoted as $span[i], i \in [0, d)$). Numerical span is defined as the difference of the largest value and the smallest value in this dimension. For example, in $Dim[j]$, the largest value is 0.9 and the smallest value is 0.1, then the numerical span associated with this dimension is 0.8. We then rank the dimensions according to their numerical spans. The possibility of one $hyperplane[i]$ being chosen by the hash function is proportional to:

$$prob = span[i]/\sum_{i=0}^{d-1} span[i]. \tag{3.7}$$

This is to ensure dimensions with large span have more chance to be selected. More analysis can be found in Sec. 3.6.

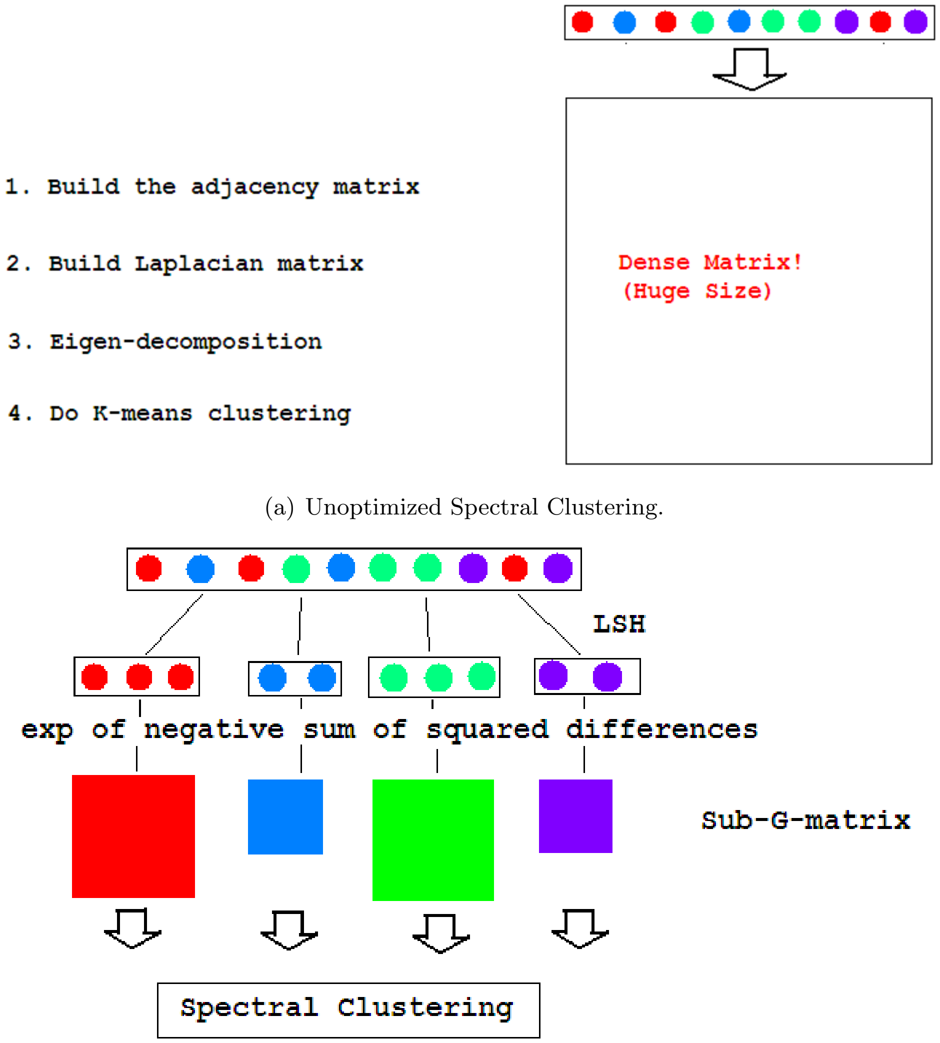
For each dimension space $Dim[i]$, the associated *threshold* is determined as follows: between the minimum (denoted as $min[i]$) and maximum (denoted as $max[i]$) in $Dim[i]$, we create 20 bins (denoted as $bin[j], j \in [0, 19]$), $bin[j]$ will count the number of points whose $i - th$ dimension fall into the range $[min[i] + j \times span[i]/20, min[i] + (j + 1) \times span[i]/20)$. We then find out the minimum in array bin (denoted as $bin[s]$), the threshold associated with $Dim[i]$ is set to $Dim[i] = min[i] + s \times span[i]/20$.

Fig. 3.4(a) illustrates the scenario when we compute the similarity matrix from the original dataset without approximation. Since both space complexity and time complexity of computing the similarity matrix are $O(N^2)$ for a dataset containing N points, the similarity matrix will take too much time to compute. The failure of computing similarity matrix will render Spectral Clustering unfeasible to run. Our proposal is illustrated in Fig. 3.4(b): add a pre-processing step on top of similarity matrix computation to separate the raw dataset apart into several buckets according to their proximity and locality measurements. Then, build the similarity matrix separately for each partition of data.

3.4 MapReduce Implementation of DASC Algorithm

MapReduce [18] is a programming model for writing applications that can rapidly process vast amount of data in parallel on a cluster. Hadoop [62] is an open source implementation of MapReduce. Hadoop makes it possible to run applications on systems with thousands of nodes, these nodes can be of different computing power. We present the design of the proposed DASC algorithm using the MapReduce programming model and we implement it in the opensource Hadoop platform. A MapReduce job is a unit of work to be performed: it consists of the input data, the MapReduce program, and configuration information. Hadoop runs the job by dividing it into tasks. There are two types of tasks: map tasks and reduce tasks.

Table. 3.2 illustrates different parts of a MapReduce job. We parallelize the proposed



(a) Unoptimized Spectral Clustering.

(b) Proposed distributed Spectral Clustering (DASC).

Figure 3.4: Spectral Clustering and DASC.

Component	***Handled By***
Configuration of the job	User
Input splitting and distribution	Hadoop
Start of the individual map tasks with their input split	Hadoop
Map function, called once for each input key/value pair	User
Shuffle, which partitions and sorts the per-map output	Hadoop
Sort, which merge sorts the shuffle output for each partition of all map outputs	Hadoop
Start of the individual reduce tasks, with their input partition	Hadoop
Reduce function, which is called once for each unique input key, with all of the input values that share that key	User
Collection of the output and storage in the configured job output directory	Hadoop

Table 3.2: Components of a MapReduce job.

DASC algorithm using the MapReduce framework. In this section, we first explain the implementation of DASC, then go on to the configuration of MapReduce tasks.

3.4.1 Implementation of DASC

MapReduce works by breaking the processing into two phases: the map phase and the reduce phase. Each phase has key-value pairs as input and output, the types of which are chosen by user. The user also specifies two functions: the map function and the reduce function. We first introduce the parallelization of $Step1$ - $Step3$ described in Section 3.3, then move to the parallelization of $Step4$ - $Step8$.

In map function, the input key-value pair is $(index, inputVector)$, where $index$ is the index of a data point, and $inputVector$ is a numerical array associated with the point. The output key-value pair is $(signature, index)$, where $signature$ is a binary sequence, and $index$ is the same as the input notation. The pseudocode is shown in Fig. 3.5.

In reduce function, the input key-value pair is $(signature, listof(index))$, where $signature$ is a binary sequence, and $index$ is the index of a data point. The reduce function is passed with a list of data points that fall into the same bucket, and it will compute the sub similarity matrix following Equation (3.4), and output the computed result. The pseudocode is

```
Algorithm 2: mapper(index, inputVector)
/*a mapper is fed with one $inputVector$ per time,
which comprises of $d$ dimensions*/
String $Sig = ""$ ;
for $i = 1; i \leq M; i++$ do
    $Threshold = get\_threshold(i)$ ;
    $Hyperplane = get\_hyperplane(i)$ ;
    if $inputVector[Hyperplane] \leq Threshold$ then
        $p = 1$ ;
    else
        $p = 0$ ;
    end
    Convert $p$ to String and add to the tail of $Sig$ ;
end
$emitPair(Sig, index)$ ;
```

Figure 3.5: DASC Map function.

shown in Fig. 3.6.

Approximation error can occur if the following scenario happens. For two points in two separate buckets, they share the majority of corresponding bits in the signature. This means two such points differ from each other in a very small portion, and thus, it is reasonable to say that they still lie close to each other. Suppose we do not have the proposed approximation step and the original dataset goes directly to Spectral Clustering, then two such points still have a high chance to go into one cluster. If we use the proposed approximation step and two such points are projected into different buckets, the underlying Spectral Clustering will cluster the points separately in each bucket, so these two points may go to different clusters. This is how the approximation error is incurred.

Manku et al. [40] have similar problem in large scale web search. They work on finding similar documents when a document is queried. In their work, every document is represented by a binary fingerprint, similar fingerprints stand for similar documents. When a document query is submitted, the system not only returns the documents that have identical fingerprints, but it also returns those documents that have the majority of fingerprint bits in common.

In order to reduce this approximation error, in between map and reduce functions, we use a similar step that merges two buckets whose signatures share no less than P bits into one

```
Algorithm 3: reducer (signature, ArrayList indexList )
/*Compute the sub similarity matrix */
$Length = getLength(indexList)$ ;
for $i = 1; i \leq Length; i++$ do
    for $j = 1; j \leq Length; j++$ do
        if $i \neq j$ then
            $subSimMat[i, j] = simFunc(i, j)$ ;
        else
            $subSimMat[i, j] = 0$ ;
        end
    end
end
$Output\_to\_File(subSimMat)$ ;
```

Figure 3.6: DASC Reduce function.

bucket. For example, if there are two buckets whose signatures are 0110000 and 0110100, then the points in these two buckets will be merged together. Effectively, the two files on disk containing these points will be combined to form a new file.

This step goes in the following way: Every two binary signatures of length M bits generated in the map function go through a pair-wise comparison phase. If they share at least P bits, the files containing the points will be merged together into a new file, effectively, the point indices in the original two files are put into a new file. These two signatures will be removed from the signature pool. This is to prevent the occurrence of the following situation: if there are three signatures, A: 0010, B: 0011 and C: 0001, after we finish merging A and B, if we do not remove A and B from the signature pool, we will later merge B and C together. This will essentially be erroneous because A and C have only two bits in common.

The process of comparing two signatures is also optimized for performance, we use the following bit manipulation to do the comparison between two M-bit-length sequence (denoted as A and B):

$$ANS = (A \oplus B)(A \oplus B - 1), \tag{3.8}$$

If ANS is 0, it means A and B have only 1 bit in difference, thus they will be merged together. Otherwise, A and B should not be merged. The complexity of this operation is $O(1)$.

After the sub similarity matrices have been computed, we run Spectral Clustering on each matrix.

In ***Step 4***, to compute diagonal matrix D, since the computation of one entry D_{ii} in D only needs the information of the ith row in S, this process is parallelized for every row. In the map function, the input pair is $(rowIndex, rowVector)$, the output pair is $(rowIndex, Sum)$, where Sum is the summation of all the entries in vector $rowVector$. The pseudocode of the Map function is shown in Fig. 3.7.

```
Algorithm 4: mapper(rowIndex, rowVector)
/*a mapper is fed with one $rowVector$
comprised of $N$ entries */
Float $result = 0$ ;
for $i = 1; i \leq N; i++$ do
    $result += rowVector[i]$ ;
end
String $resultString = result.toString() +'\_' + rowIndex$ ;
$emitPair(1, resultString)$ ;
```

Figure 3.7: Map function of diagonal matrix computation.

The pseudocode of the Reduce function is shown in Fig. 3.8.

```
Algorithm 5: reducer (key, ArrayList values )
/*Construct diagonal matrix */
$Length = getLength(values)$ ;
for $i = 1; i \leq Length; i++$ do
    $diagMat[i, i] = values[i]$ ;
end
$Output\_to\_File(diagMat)$ ;
```

Figure 3.8: Reduce function of diagonal matrix computation.

After D has been computed, we come to $L = D^{-1/2}SD^{-1/2}$. This involves matrix multiplication. Because D is a diagonal matrix, $D^{-1/2}$ is also diagonal matrix. Matrix premultiplication with a diagonal matrix and matrix postmultiplication with a diagonal matrix are both very efficient. We use matrix IM to denote the intermediate matrix generated, the size of D and S is N.

$$IM_{ij} = \sum_{k=0}^{N-1} (D_{ik}^{-1/2} \cdot S_{kj}) = D_{ii}^{-1/2} \cdot S_{ij}, \tag{3.9}$$

$$L_{ij} = \sum_{k=0}^{N-1} (IM_{ik} \cdot D_{kj}^{-1/2}) = IM_{ij} \cdot D_{jj}^{-1/2}, \tag{3.10}$$

Therefore, we have:

$$L_{ij} = D_{ii}^{-1/2} \cdot S_{ij} \cdot D_{jj}^{-1/2}. \tag{3.11}$$

The process is parallelized on every entry. For every entry L_{ij}, the map function fetches $D_{ii}^{-1/2}$, $D_{jj}^{-1/2}$ and S_{ij}, and performs dot product multiplication. The output pair is $(i \cdot n + j, productValue)$. Therefore, if the size of the matrix is $N \times N$, we use N^2 tasks to parallelize it. The pseudocode of the Map function is shown in Fig. 3.9.

```
Algorithm 6: mapper(Index, tuple)
/*a mapper is fed with one tuple $< D_{ii}, S_{ij}, Djj >$ */
Float $result = 0$ ;
$dii = tuple[0]$ ;
$sij = tuple[1]$ ;
$djj = tuple[2]$ ;
$result = dii * sij * djj$ ;
String $resultString = result.toString() +' \_' + Index$ ;
$emitPair(1, resultString)$ ;
```

Figure 3.9: Map function of Laplacian matrix computation.

The pseudocode of the Reduce function is shown in Fig. 3.10.

In ***Step 5***, we are to compute the eigenvectors and eigenvalues of L. The naive method involves finding the roots of the characteristic polynomial of the matrix. However, for large matrices, this method is simply not feasible [45]. There are several approximation methods to compute eigenvectors and eigenvalues, such as Inverse Iteration [30] and Rayleigh quotient iteration [50]. In this work, Lanczos method [48] is used to compute the eigenvectors. The Lanczos method is good for the eigen decomposition of large sparse matrices [16].

Lanczos method transforms the original matrix L into a real symmetric tridiagonal matrix A, whose eigenvalues are more easily computed. This transformation is orthogonal,

```
Algorithm 7: reducer (key, ArrayList values )
/*Construct Laplacian matrix */
$Length = getLength(values)$ ;
for $i = 1; i \leq Length; i++$ do
    for $j = 1; j \leq Length; j++$ do
        $LapMat[i, j] = values[i * N + j]$ ;
    end
end
$Output\_to\_File(LapMat)$ ;
```

Figure 3.10: Reduce function of Laplacian matrix computation.

hence preserves the eigenvalues of L. In each round, matrix multiplication of a matrix by a vector is parallelized on every row. Because L is symmetric matrix, the parallelization on every row does not call for synchronization. The first k eigenvectors of L are found out. The matrix $X = [V_1 V_2 ... V_k] \in R^{N \times k}$ forms by stacking the eigenvectors in columns. The pseudocode of the Map function is shown in Fig. 3.11.

```
Algorithm 8: mapper(Index, tuple)
/*a mapper is fed with one tuple $< Lrow, q >$
$Lrow$ is one row in $L$, $q$ is a vector*/
Vector $Lrow = tuple[0]$ ;
Vector $q = tuple[1]$ ;
Float $result = 0$ ;
for $j = 1; j \leq N; j++$ do
    $result += Lrow[j] \cdot q[j]$ ;
end
String $resultString = result.toString() +' \_' + Index$ ;
$emitPair(1, resultString)$ ;
```

Figure 3.11: Map function of matrix vector multiplication in Lanczos method.

The pseudocode of the Reduce function is shown in Fig. 3.12.

Step 6 is normalization of matrix X by the column. The computation is parallelized by the column since the normalization of every column has nothing to do the other columns. In the map function, the input pair is $(ColumnIndex, ColumnVector)$, the output pair is $(ColumnIndex, NormalizedColumnVector)$, and the reduce function is left empty. The pseudocode of the Map function is shown in Fig. 3.13.

```
Algorithm 9: reducer (key, ArrayList values )
/*Construct resulted $q$ vector */
Vector $q$ ;
$Length = getLength(values)$ ;
for $j = 1; j \leq Length; j++$ do
    $q[j] = values[j]$ ;
end
$Output\_to\_File(q)$ ;
```

Figure 3.12: Reduce function of matrix vector multiplication in Lanczos method.

```
Algorithm 10: mapper(ColumnIndex, ColumnVector)
/*a mapper normalizes a column in $X$ */
Float $magnitude = 0$ ;
for $j = 1; j \leq N; j++$ do
    $magnitude+ = ColumnVector[j] \cdot ColumnVector[j]$ ;
end
$magnitude = sqrt(magnitude)$ ;
for $j = 1; j \leq N; j++$ do
    $ColumnVector[j]+ = ColumnVector[j]/magnitude$ ;
end
$emitPair(ColumnIndex, ColumnVector)$ ;
```

Figure 3.13: Map function of normalization.

In ***Step 7*** and ***Step 8***, we come to the final step, which is K-means clustering. The parallelization strategy is described in Section 2.4.

3.4.2 Configuration of DASC in MapReduce Framework

The configuration parameters of MapReduce are tuned to achieve good performance. Two of the important factors for performance are *input split size* and *replication factor.*

Hadoop divides the input of a MapReduce job into fixed-size pieces called input splits. Hadoop creates one map task for each split, and it runs the user-defined map function for each record in the split. Having many splits means the time taken to process each split is small compared to the time to process the whole input. So if we are processing the splits in parallel, the processing is better load-balanced if the splits are small, since a faster machine will be able to process proportionally more splits over the course of the job than

Parameter	**Value**
heap-size for Maps	1024 MB
heap-size for Reduces	1024 MB
HDFS block size	64 MB
MapReduce task timeout	1800000 ms
Memory-limit while sorting data	200 MB
Size of read/write buffer used in SequenceFiles	128 KB
Size of read/write buffer	4 KB
Maximum number of tasks run simultaneously on one node	2
Number of map tasks	256
Number of reduce tasks	10

Table 3.3: Parameters used in the implementation.

a slower machine. Even if the machines are identical, failed processes or other jobs running concurrently make load balancing desirable, and the quality of the load balancing increases as the splits become more fine-grained. On the other hand, if splits are too small, then the overhead of managing the splits and map task creation begins to dominate the total job execution time. In the experiments, the split size is set to be the size of the HDFS block, which is 64 MB, based on the following reason: It is the largest size of input that can be guaranteed to be stored on a single node. If the split spans two blocks, it is unlikely that any HDFS node stores both blocks, thus some of the split would have to be transferred across the network to the node running the map task, which is less efficient than running the whole map task on local data.

The number of replicas of a file in HDFS is called the replication factor of that file. As described in Section 2.1.2, a map task will consult the namenode to locate the position of the input file. If the input file is not on its local machine, it will bring the input data from another node where the data resides. It will cost extra node-to-node network communication. In the cluster we experiment on, all nodes are connected by a high-speed router, thus forming a single rack (no out-of-rack access). So we set the replication factor to 1.

For the number of map tasks, it has been suggested in [62] that the right number of Maps is around 10-100 maps/node. Therefore, considering there are five nodes in the cluster, we set this parameter to 256. For the number of reduce tasks, it has been suggested in [62] that the right number of reduces is $0.95 \cdot (NumberOfNodes \cdot tasktracker.tasks.max)$. $NumberOfNodes$ is the total number of $TaskTracker$ machines, and $tasktracker.tasks.max$

is the number of maps/reduces spawned simultaneously on a $TaskTracker$. At 0.95 all of the reduces can launch immediately and start transfering map outputs as the maps finish. In our implementation, the parameter *tasktracker.tasks.max* is 2. Therefore, we set the number of reduce tasks to 10.

3.5 Speedup and Memory Consumption Analysis

Given a collection of data (the size is N), the proposed method aims to project nearby points into one bucket and only compute the similarity values between the point pairs that appear in the same bucket. We give a quantitative analysis of the reduction of similarity matrix size using the proposed method.

The space complexity for computing the whole matrix is $O(N^2)$. For the approximated similarity matrix, suppose there are altogether B number of buckets, and there are N_i number of points in bucket i, where $0 \leq i \leq B-1$, and $\sum_{i=0}^{B-1} N_i = N$. The space complexity is: $\sum_{i=0}^{B-1} N_i^2$. Therefore, the space usage reduction ratio is given as:

$$SpaceReductionRatio = \frac{\sum_{i=0}^{B-1} N_i^2}{N^2}. \tag{3.12}$$

If all the buckets contain equal number of points, $N_i = \frac{N}{B}$ where $0 \leq i \leq B-1$. Eq. 3.12 can be rewritten as:

$$SpaceReductionRatio = \frac{\sum_{i=0}^{B-1} N_i^2}{N^2} = \frac{B \cdot (\frac{N}{B})^2}{N^2} = \frac{1}{B}. \tag{3.13}$$

Moreover, the memory is distributed across the cluster machines.

In the aspect of computing time reduction, the time complexity for computing the whole matrix is $O(N^2)$. For the approximated similarity matrix, suppose there are altogether B number of buckets, and there are N_i number of points in bucket i, these points form K_i clusters, where $0 \leq i \leq B-1$, and $\sum_{i=0}^{B-1} N_i = N$. We run Spectral Clustering on each bucket. M is the length of signature bits. Therefore, the time reduction ratio is given as:

$$TimeReductionRatio = \frac{M \cdot N + B^2 + 2 \cdot N + \sum_{i=0}^{B-1} \left[2 \cdot N_i^2 + 2 \cdot (K_i \cdot N_i)\right]}{2 \cdot N^2 + 2 \cdot (K \cdot N) + 2 \cdot N}. \tag{3.14}$$

If all the buckets contain equal number of points, $N_i = \frac{N}{B}$ where $0 \leq i \leq B-1$. Eq. 3.14 can be rewritten as:

$$\begin{aligned}
TimeReductionRatio &= \frac{M \cdot N + B^2 + 2 \cdot N + \sum_{i=0}^{B-1} \left[2 \cdot N_i^2 + 2 \cdot (K_i \cdot N_i)\right]}{2 \cdot N^2 + 2 \cdot K \cdot N + 2 \cdot N} \\
&= \frac{M \cdot N + B^2 + 2 \cdot N + B \cdot \left[2 \cdot (\frac{N}{B})^2 + 2 \cdot (\frac{K}{B} \cdot \frac{N}{B})\right]}{2 \cdot N^2 + 2 \cdot K \cdot N + 2 \cdot N} \\
&= \frac{M + \frac{B^2}{N} + 2 - \frac{2}{B}}{2 \cdot (1 + N + K)} + \frac{1}{B}.
\end{aligned} \tag{3.15}$$

Furthermore, the approximated similarity matrix can be fed into the following-up clustering technique in a parallelized manner. This will bring about further computing time reduction.

3.6 Accuracy Analysis

In this section, we analyze why the proposed method can accurately cluster high-dimensional large datasets. The proposed algorithm can be divided into two parts: the first part is signature generation and similarity matrix computation, the second part is Spectral Clustering.

Given an $N \times d$ dataset, in dimension i (where $i \in [0, d)$), if a group of points have similar values (for example, they all have values around 0.8 in dimension i), it is likely that they belong to the same cluster. Also, if there are two groups A and B, in dimension i, points in group A are all far away from points in group B (for example, points in group A have values around 0.8 and points in group B have values around 0.2), it is likely that these two groups of points belong to two separate clusters. Our hash function is essentially following the above reasoning to do partitioning. In point signature generation part, in every hashing iteration, the hash function selects the values in dimension i (where $i \in [0, d)$) to check. In fact, the hash function is projecting all the points onto a line, as is shown in Fig. 3.14.

After projecting the points onto a line, numerical span is the maximum distance between two projections of all points. The hash function builds upon the fact that, if all projections lie in a small line segment, the cluster formation is hard to differentiate, and the probability of misclassification arises. When the hash function chooses which dimension to examine, the probability of a dimension being chosen will grow as the numerical span increases, as in Eq. (3.7).

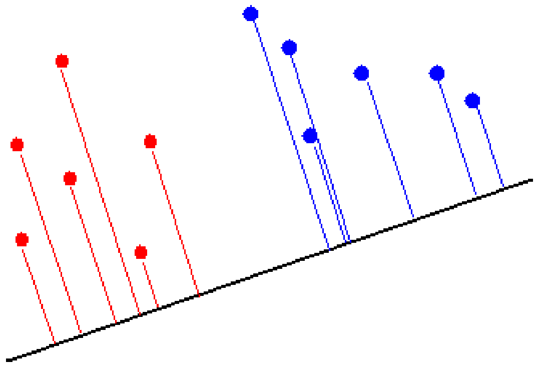


Figure 3.14: The effect of hash function.

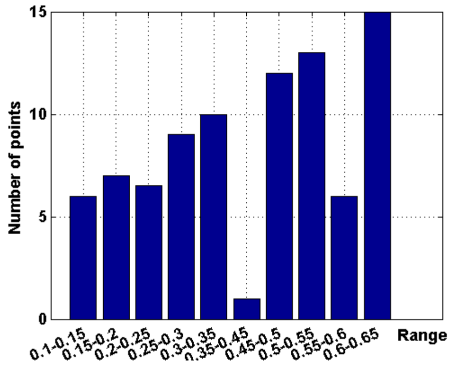


Figure 3.15: An example of counting range histogram.

We also need to decide how to split the dataset into two. It has been discussed in [58] that, by choosing a splitting plane at the place where fewest points appear in dimension i, we can reduce the number of points that are wrongly clustered. Therefore, to determine the splitting point, the hash function finds out a segment in dimension i that has the lowest density.

As shown in Sec. 3.3, for each dimension i (where $i \in [0, d)$), we create 20 equal-length segments and count the number of points falling into each segment. After this step, we form a histogram, as in Fig. 3.15. The starting point of the segment with the lowest count is used as the splitting point. For example, in Fig. 3.15, when performing partitioning, because the Range[0.35, 0.45] has the smallest count of the points compared to other ranges, it is reasonable to say that this range has the biggest possibility to be the borderline of two clusters, therefore, threshold 0.35 is used to partition the dataset.

After the dataset partitioning, Spectral Clustering will run as the second step to cluster the dataset. Every entry in similarity matrix corresponds to a nonlinear transformation of the distance between two points. Also, in order to deal with a dataset that is non-linearly separable in input space, Spectral Clustering works by first computing eigenvectors of the affinity matrix, and then do K-means clustering to get the final result.

Chapter 4

Experimental Evaluation on Synthetic Datasets

In this chapter, we evaluate the proposed algorithm on a cluster. We describe the setup of the experiment, then explain the evaluation metrics. After that, we present evaluation results of the proposed method, and compare them with the results of current optimization techniques that use parallel programming paradigms and approximation methods.

4.1 Experimental Setup

The experiments are conducted on a cluster made up of five machines, each of which is equipped with Intel Core2 Duo Processor E6550 (4M Cache, 2.33 GHz, 1333 MHz FSB) and 1 GB DRAM. One machine in the cluster serves as the master (job tracker) and the others are slaves (task trackers). The dataset used in the experiment is 64-dimension randomly generated data, each dimension takes a real value chosen from the period [0, 1]. We use the range [0,1] because dataset normalization is a standard preprocessing step in Data Mining applications [4]. The size of the dataset ranges from 1024 to 4 million data points.

The software stack that runs on each machine of the cluster and the property of the dataset are illustrated in Table 4.1.

In K-means, once the initial centers are chosen, the distance is calculated from every point in the dataset to each of the centers and each point is assigned to the particular cluster center whose distance is closest. After the assignment of points to cluster centers, each

OS	Ubuntu Linux 2.6.27-14
JDK Version	1.6.0
Hadoop Version	0.20.2
Dataset property	64 dimension, every component of vector X_i is [0,1]

Table 4.1: DASC method experiment setup.

cluster center is recalculated to form the new centroid of its member points. This process is then iterated until convergence is reached. That is, the cluster centers shift by less than a pre-determined convergence delta value. In the experiment, we set the maximum iteration number of K-means (the last step of Spectral Clustering) to 10, and convergence delta is set to 0.5, and cluster number to 3. We monitor the running process of the experiments, and find out when convergence delta is equal to 0.5, the iteration number is always below 10. This phenomenon also shows that the preprocessing steps preceding K-means are effective. By the time the algorithm reaches K-means, the data is already easy to be clustered apart.

We set the parameter of how many buckets to generate from the original dataset as follows: presented with a dataset with size N, the bit length of the signature M generated is set to $M = \lfloor \frac{\log N}{2} \rfloor - 1$. P is the minimum number of identical bits in two binary signatures. We set the P value to $M - 1$. This is based on the reason that if P is set to a smaller value, more initial buckets of points will potentially be merged together, leading to less degree of parallelization. Another reason is that, setting $P = M - 1$ will enable efficient $O(1)$ comparison operation. This bit comparison operation is shown in Eq. (3.8) between two signatures.

We implement the proposed method DASC by modifying Mahout, and compare the proposed method against three other methods. One is the basic distributed Spectral Clustering method (SC), we use the existing implementation of SC in Mahout. The other one is Parallel Spectral Clustering (PSC) in [13]. PSC is implemented by Chen [13] in C++ using PARPACK library [37] as underlying eigenvalue decomposition package and F2C to compile fortran code. The parallelization is based on MPI. The implementation is comprised of five steps:

- ***Step 1***: Compute the distance matrix. In this step, we set the number of nearest neighbors of one point to $\frac{N}{40}$, Arnoldi length [38] is set to the double of the number of clusters.

OS	Ubuntu Linux 2.6.27-14
GCC	4.3.3
MPICH Version	1.3.2p1
Dataset property	64 dimension, every component of vector X_i is [0,1]

Table 4.2: PSC method experiment setup.

- ***Step 2***: Convert the distance matrix to similarity matrix and compute the Laplacian matrix L. The conversion follows:

$$similarity = \exp^{-(distance \cdot distance)/(2 \cdot \gamma_1)}, \tag{4.1}$$

  where γ_1 is the corresponding row average of the distance matrix.

- ***Step 3***: Compute the first k eigenvectors of L using iterative procedure called "implicitly restarted" Arnoldi [13], and construct $X \in R^{N \times k}$, whose columns are the k eigenvectors.

- ***Step 4***: Compute the normalized matrix Y from X.

- ***Step 5***: Use K-means to cluster the points int k groups.

The PSC method experiment setup is shown in Table 4.2.

The third method is the Nystrom extension (NYST), we use the existing implementation by Shi [57]. NYST performs the following steps:

- ***Step 1***: Compute the top eigenvectors V_1, V_2,..., V_k of similarity matrix using a Gaussian kernel.

- ***Step 2***: Find the eigenvectors with no sign change up to a small threshold ε. Larger values of ε lead to more eigenvectors having the no sign change property, which results in more groups being detected by the algorithm. Smaller values of ε result in fewer groups. For d-dimensional vector v taking values at locations X_1,..., X_d, $\varepsilon = \frac{1}{d} \max_{i=1}^{d} X_i$.

- ***Step 3***: Cluster the collection of all no sign change eigenvectors.

- ***Step 4***: Combine eigenvectors within each cluster found in ***step 3*** to obtain a single eigenvector for that group.

OS	Ubuntu Linux 2.6.27-14
Matlab	R2009a
Dataset property	64 dimension, every component of vector X_i is [0,1]

Table 4.3: Nystrom method experiment setup.

- ***Step 5***: Extend each group eigenvector to all observations in the original dataset and use these vectors to assign cluster labels.

Nystrom method is implemented in Matlab, the experiment setup is shown in Table 4.3.

4.2 Performance Metrics

We consider the following performance metrics: accuracy, processing time and memory usage.

Accuracy is measured from both low-level matrix property and from high-level clustering algorithm aspect. At the matrix level, we use Frobenius Norm [42]. For the high level, we compare the clustering results of different methods.

4.2.1 Low Level Accuracy Metric

The Frobenius norm (denoted as Fnorm hereinafter) is one of the matrix norms [42], which is also called the Euclidean norm. The Frobenius norm of an $M \times N$ matrix A is defined as:

$$||A|| = Fnorm = \sqrt{\sum_{i=1}^{M}\sum_{j=1}^{N}|a_{ij}|^2}. \tag{4.2}$$

Fnorm is easy to compute and it can be used to assess how close two matrices are. We use it to compare the approximated similarity matrix versus the original one.

For an $M \times N$ matrix A, by singular value decomposition, A can be decomposed into the product of three matrices as follows:

$$\begin{aligned} A &= U\Sigma V^H \\ &= U \begin{bmatrix} \Sigma_k & 0 \\ 0 & 0 \end{bmatrix} V^H, \end{aligned} \tag{4.3}$$

where U and V are two unitary matrices and Σ_k is a $k \times k$ diagonal matrix containing the k ordered positive definite singular values $\sigma_1 \geq \sigma_2 ... \geq \sigma_k \geq 0$. The variable k is the rank of A and represents the number of linearly independent columns in it. The Frobenius norm is invariant under unitary transformations, therefore, we have:

$$||A|| = \sqrt{\sum_{m=1}^{k} \sigma_m^2}. \tag{4.4}$$

According the Eq. (4.4), the larger the $FnormRatio$ is, the closer the sum of singular values in the approximated similarity matrix is to that of the original matrix. If $FnormRatio$ is equal to 0.9, it means 90% of the sum of singular values in the original similarity matrix is preserved in the approximated one. Therefore, the approximated similarity matrix is very close to original one.

Fnorm is used in [67] to develop a new affinity matrix normalization scheme in Spectral Clustering. Anagnostopoulos et al. [2] propose an approximation algorithm for co-clustering problem and use Fnorm for evaluation purpose. Yang and Yang [65] compare the proposed Yang distance against Fnorm in two-dimensional principal component analysis.

4.2.2 High Level Clustering Accuracy Metrics

We compare the clustering results of different algorithms using two metrics. The first one is the Davies-Bouldin index (DBI) [17] is a function of the ratio of the sum of within-cluster scatter to between-cluster separation. It uses both the clusters and their sample means. It is calculated by the following formula:

$$DBI = \frac{1}{C} \sum_{i=1}^{C} \max_{j:i \neq j} \{ \frac{\sigma_i + \sigma_j}{d(c_i, c_j)} \}, \tag{4.5}$$

where C is the number of clusters, c_x is the centroid of cluster x, σ_x is the average distance of all elements in cluster x to centroid c_x, and $d(c_i, c_j)$ is the distance between centroids c_i and c_j. Clustering algorithm that produces the smallest index is considered the best algorithm based on this criterion. DBI has been shown to be a robust strategy for the prediction of optimal clustering partitions [17]. Guerif et al. [26] leverage DBI to determine the best number of clusters in the clustering process of Self-Organizing Maps (SOM). It is also the evaluation index used and discussed in [55] and [58].

The second metric we use to compare clustering results is the average squared error (ASE) measurement [31], which serves as a clustering criterion for many classic clustering techniques, such as k-medoids algorithm [34]. ASE is also used in [22] for validation purposes. The squared distance e_k^2 (where $1 \leq k \leq C$) for each cluster k is the sum of the squared Euclidean distances between each point in k and its centroid, and the average squared error for the entire dataset is expressed by:

$$ASE = \frac{1}{N}\sum_k e_k^2 = \frac{1}{N}\sum_k (\sum_{j=0}^{N_k-1} d(X_j, C_k))^2, \tag{4.6}$$

where N is the size of the dataset, N_k is the number of points in cluster k.

4.2.3 Processing Time and Memory Footprints

Processing time is the running time of the program. Since every entry in the the similarity matrix is of type float (4 bytes), the memory usage is $MemUsage = 4 \times Num\_Entries$, where $Num\_Entries$ is the number of entries in the similarity matrix.

4.3 Results and Analysis

4.3.1 Accuracy

Frobenius Norm

Fig. 4.1 shows Frobenius Norm value of approximated matrix compared to that of the original matrix. The x-axis stands for how many buckets the original dataset is partitioned into. For example, when the value in x-axis is 8, the original dataset is divided into 2^8 partitions. The y-axis stands for the $FnormRatio$, which is defined as the Fnorm value of the approximated matrix to the original full similarity matrix.

We choose M to be $\lfloor \frac{\log N}{2} \rfloor - 1$ when the total number of points is N. Because it is shown in Fig. 4.1 that when we follow this formula, the ratio of Frobenius Norm between

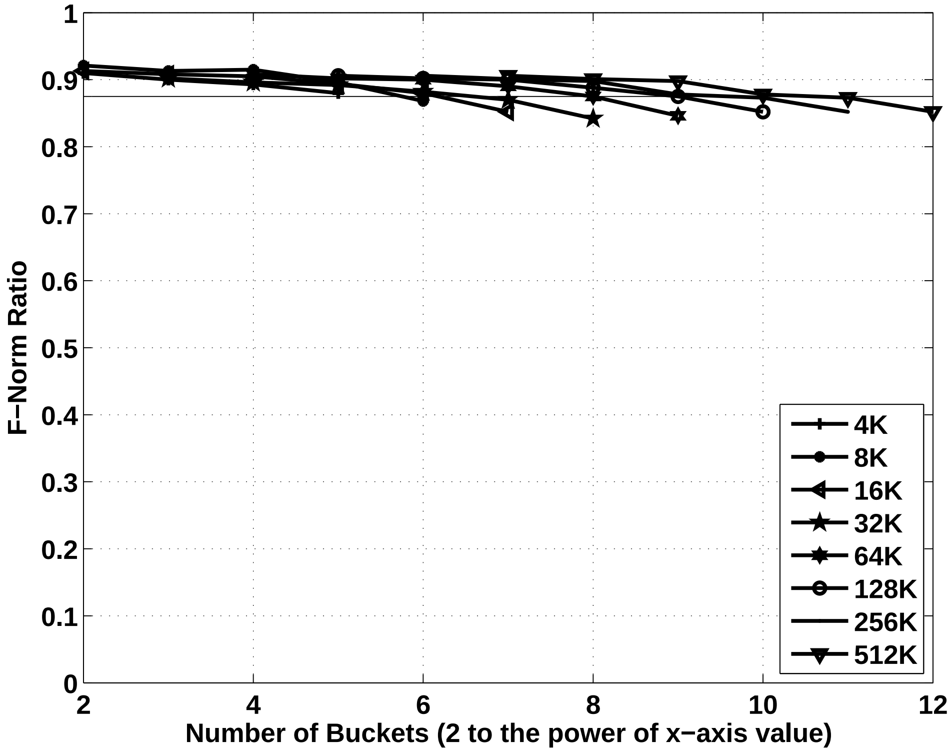


Figure 4.1: Comparison between approximated similarity matrix and original matrix using Frobenius norm.

the approximated similarity matrix and the original similarity matrix will all be above 87.6%, thus it is a good approximation. For a dataset (represented by a line in Fig. 4.1), the more partitions we have, the more information we lose. If we look into this problem in a matrix perspective, having less number of buckets means, in one single bucket, more close-by points have the potential to be clustered in the final stage. On the other hand, if we generate more buckets, the selection bar of entering the same bucket has been raised. Although points that fall in the same bucket are still close to each other, some other points, which are at the boundary of the bucket, will fall into another bucket. Also, if the size of the dataset is increased and keep the bucket number unchanged, on average, more points will fall into the same bucket, the potential clustering quality of underlying Spectral Clustering will be positively influenced.

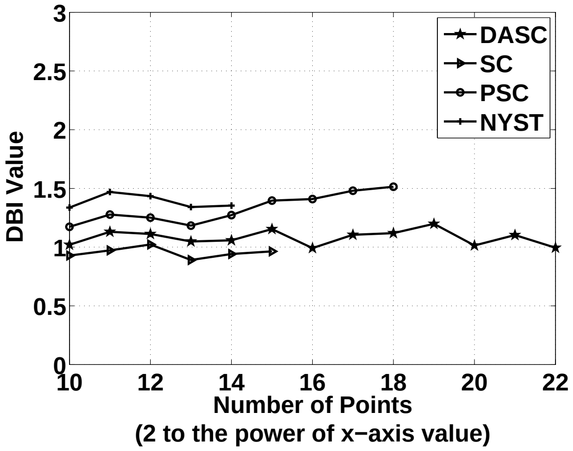


Figure 4.2: DBI comparison.

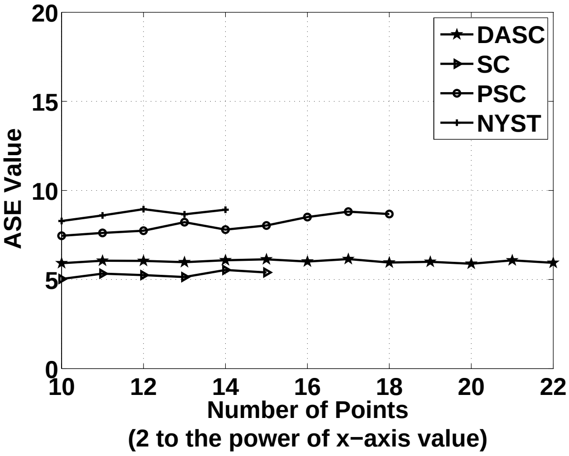


Figure 4.3: ASE comparison.

	SC	DASC	PSC
1024 points	79.2	90.1	12.1
2048 points	256.3	187.5	42.2
4096 points	826.9	346.1	160.5
8192 points	3041.2	703.8	610.2
16384 points	12328.8	1562.6	2318.8

Table 4.4: Processing time comparison from 1024 to 16384 points (in second).

DBI, ASE

Fig. 4.2 and Fig. 4.3, respectively, show DBI value (the distance of two points is the L2 distance) and ASE value comparison. The values in x-axis stand for the size of the dataset, the values in y-axis stand for the corresponding value of DBI and ASE.

In Fig. 4.2 and Fig. 4.3, the curves for the Nystrom method terminates at 2^{14} number of points. The reason is that when applying larger dataset on the Nystrom method, the machine is overwhelmed by large memory consumption. As is shown in Fig. 4.2, the DBI value that DASC achieves is very close to the result achieved by SC. Therefore, DASC can accurately cluster the dataset. When checking the DASC curve alone, the DBI values, although go through some slight up and downs, in general, they stay in the range between 1 to 1.3 and always close to the SC's curve. If we look at the definition of DBI, the numerator is the sum of the average distance of all points in one cluster to its centroid, and the denominator is the distance between the centroids of two clusters. When the DBI value is

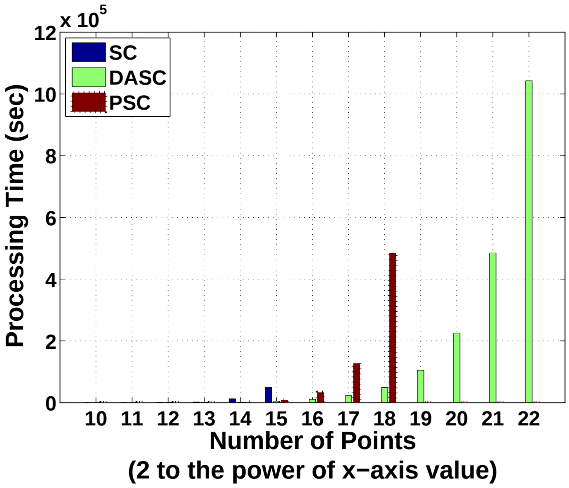


Figure 4.4: Running time comparison.

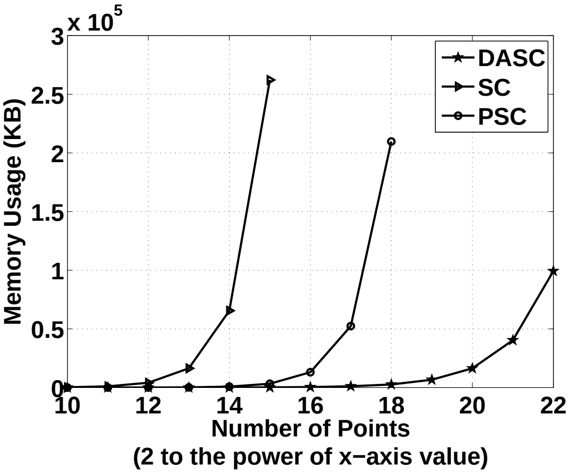


Figure 4.5: Memory consumption comparison.

larger than one, it can be inferred that the shape of certain cluster is long-stretched. But within the cluster, the points tightly reside in a high-dimensional cubic. This is the desired result we want to achieve. On the contrary, PSC and NYST do not perform as good as DASC.

In Fig. 4.3, the average squared distance measures how close the points in the clusters are to their respective centroids. Small ASE value indicates better clustering result. The figure shows that DASC outperforms the PSC and NYST methods. DASC curve stays close to SC, while PSC and NYST are about 30% and 40%, respectively, apart from SC. When the dataset grows larger, the PSC performance tends to deteriorate, as the average distance value grows slightly.

4.3.2 Processing Time

Fig. 4.4 and Table 4.4 show the processing time of DASC, PSC and SC. DASC, PSC and SC all run on the five-node cluster. It can be observed that, starting from 2^{14} dataset, the processing time of the SC method substantially increases. However, the increment ratio of the DASC method is kept to a small value between 1.1 to 1.3. DASC considerably improves the running time, because after the partitioning of the dataset, each subset of points will go separately into their own underlying clustering phase. Therefore, there can be several eigen decomposition subroutines going on in the cluster. It should also be noted that, for the comparison between the results of PSC and DASC, one big drawback of PSC is that, in order to find out a group of nearest neighbors for a data point, it still needs to do pair-wise computation. The time complexity is still not improved. This can be mitigated by using rich computing power (such as multicore machines or a cluster) on small size dataset, however, when the dataset grows even larger, the disadvantage of it is obvious. PSC uses sparse matrix to store the similarity matrix. It should be noted that, the reduction in memory usage does not correlate with the reduction of computing time. The computing time is not improved because it still scans through the dataset N^2 times.

4.3.3 Memory Consumption

Fig. 4.5 illustrates the similarity matrix memory consumption of DASC, SC and PSC. SC is the most memory consuming, because it has to store every entry in the similarity matrix. PSC performs good in the initial stage. However, when the dataset grows, the memory consumption increases rapidly. The reason is that, as the data amount gets bigger, PSC increases the number of the nearest neighbors to retain the accuracy of the approximted matrix. However, the growing slope is smaller than that of SC. DASC outperforms PSC and SC in the aspect of memory footprint, the slope is much flat in comparison to the other two methods. For example, for a dataset of size 2^{18} points, there is a factor of 25 reduction in memory footprint when comparing the proposed DASC algorithm against PSC algorithm. The reduction factor is higher for larger datasets. It should also be mentioned that, in the figures represented above, SC and PSC curves terminate at 2^{15} points and 2^{18}, respectively, because of the large complexity of computing time, and the subsequent memory insufficiency.

4.3.4 Summary

In this section, we evaluated the proposed DASC algorithm using synthetic datasets and compared it against three closest algorithms in literature: SC [46], PSC [13] and NYST methods [39] which approximate and parallelize Spectral Clustering. The experimental results show that DASC can successfully retain the clustering accuracy (within 10% above SC in DBI value, and within 16% above SC in ASE value), greatly reduce computing time (with the reduction ratio of 13 and more when comparing against SC, and with the reduction ratio of 10 and more when comparing against PSC), and significantly reduce the memory footprint.

Chapter 5

Experimental Evaluation on Large-Scale Wikipedia Dataset

In this chapter, we evaluate the proposed algorithm on the five-node cluster testbed described in Chapter 4 as well as in Amazon Elastic MapReduce cluster. The dataset we use is real-life Wikipedia semantic dataset. We describe how the Wikipedia dataset is crawled and how the raw dataset is processed. We also introduce the setup of the experiments, then explain the evaluation metrics. After that, we present evaluation results of the proposed method, and compare it with the results of PSC and SC algorithms.

5.1 Dataset Crawling and Processing

Wikipedia is a rich corpus source for text clustering. It has millions of articles along with good category hierarchy. Hereinafter, the whole dataset is denoted as the ***corpus***, a web page is denoted as a ***document***, and a word that appears in the corpus is denoted as a ***term***.

We developed a crawler in Python. The crawler utilizes urllib and urllib2 libraries. It starts crawling from an indexing page:

http://en.wikipedia.org/wiki/Portal:Contents/Categories

The page lists the categories and their links. The organization of one category is as follows: One category has several sub-categories and some pages associated with it. It can be modeled as a tree structure, and the web pages are the leaf nodes, and other nodes have sub-categories.

For these sub-category links, Wikipedia also differentiates them into two genres, both are encoded in HTML files: One genre is identified with ***CategoryTreeBullet***, meaning the sub-category this link points to also contains its own sub-categories, another genre is identified with ***CategoryTreeEmptyBullet***, meaning this sub-category only contains the leaf nodes.

The crawling process goes in several steps:

- Get the tree structure.
- Get the links of the leaf nodes and associate them with the corresponding category.
- Get the content of the leaf nodes (the HTML files).

The web crawler downloaded 3,550,567 documents, forming 579,144 categories. HTML files are further processed as follows:

- Remove tags in the files and leave the raw text.
- Keep the summary only.

The reason why we trim a document to leave the summary only is that, in most cases, one document is very large in size and it contains too many terms. We need to compute the tf-idf (term frequency-inverse document frequency) value for every term in the whole dataset. This is unrealistic given the huge amount of terms. On the contrary, the summary of a document contains the digest of the article, it contains the key terms associated with this document. These key terms play an important role in document clustering. Note that clustering based only on the document summary is more challenging than based on whole datasets.

5.2 Converting Semantic Corpus into Numerical Dataset

The corpus goes through the following steps, and tf-idf is calculated for every term in the corpus.

- Convert characters to lowercases.

- Remove punctuation.
- Remove stop words.
- Stem all terms.

We use Apache Lucene [28] to process the corpus. Apache Lucene is a text search engine library. The stop-word list we use is concatenated from several lists to capture the majority of the stop words. Stemming is the process for reducing inflected (or sometimes derived) words to their stem, base or root form. The stem does not need to be identical to the morphological root of the word, it is usually sufficient that related words map to the same stem, even if this stem is not in itself a valid root. In our experiments, we use Porter stemming algorithm [51] in Lucene library to do this step.

We derive tf-idf values from the corpus and form the numerical dataset. The term count in a specific document is the number of times a given term appears in that document. This count is usually normalized to prevent a bias towards longer documents. It gives a measure of the importance of the term t within the particular document d. We have the term frequency $tf(t,d)$ defined as:

$$tf(t,d) = \frac{occurrence\_of\_t}{total\_number\_of\_terms\_in\_d}. \tag{5.1}$$

The inverse document frequency is a measure of the general importance of the term, obtained by dividing the total number of documents by the number of documents containing the term, therefore, $idf(t)$ is defined as:

$$idf(t) = log\frac{|D|}{1 + |d : t \in d|}, \tag{5.2}$$

where $|D|$ is the total number of documents in the corpus, and $|d : t \in d|$ is the number of documents where the term t appears.

Thus, *tf-idf(t,d)* $= tf(t,d) \cdot idf(t)$.

Since the number of terms contained in one document is much smaller than the total number of terms in the corpus, the numerical dataset contains a lot of zeros. In order to save disk space and memory usage, we use sparse matrix to store the dataset.

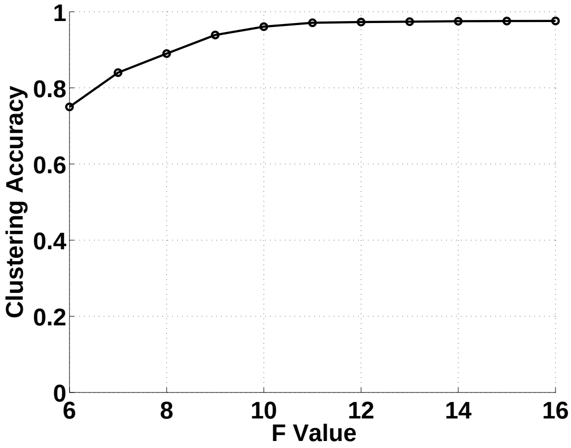


Figure 5.1: Clustering accuracy.

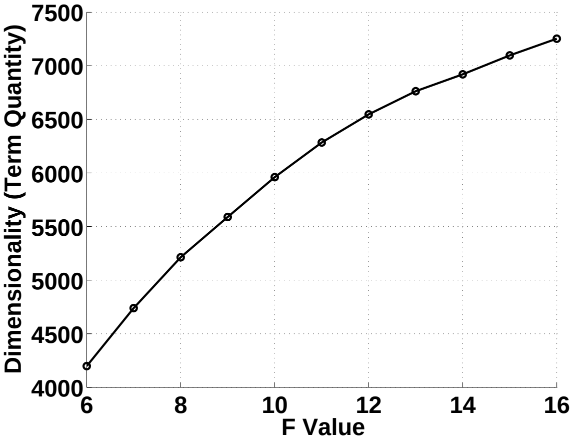


Figure 5.2: Dimensionality increase of Wikipedia dataset.

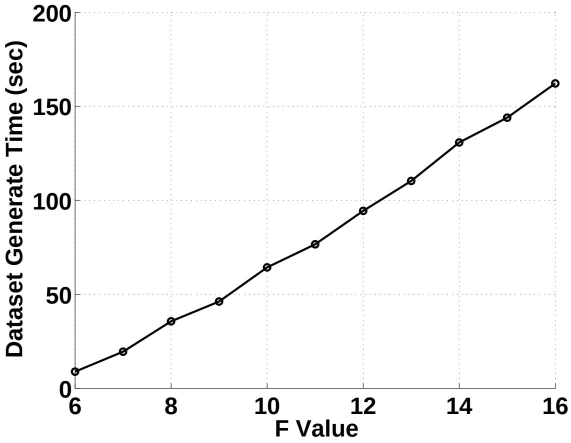


Figure 5.3: Time of dataset generation.

5.2.1 Reducing Feature Set

One problem encountered during the process is that, if we include every term that appears in the corpus, the dimensionality of this three million document dataset will be huge. We tackle this problem in the following manner: after getting the tf-idf values for one document, we rank these tf-idf values in descending order and keep only the first F terms. The reason is that when computing the cosine distance in later steps, smaller values will contribute less to the result. Omitting the trailing tf-idf values brings a small amount of inaccuracy to the result, but speeds up the computation dramatically.

We conduct an empirical experiment to tune the parameter F. We take 1,084 docs from the corpus, these 1,084 docs contains 11,102 different terms. We run Spectral Clustering on the datasets generated using F from 6 to 16 and compare the clustering accuracy. The clustering accuracy is the ratio of correctly clustered docs to the total number of docs. The clustering accuracy comparison is shown in Figure 5.1. The increase of term dimensionality is shown in Fig. 5.2. The increase in dataset generation time is shown in Fig. 5.2. It can be seen that, after F reaches 11, the accuracy improvement is trivial. On the contrary, the increases of term dimensionality and computation time are significant. Therefore, we use $F = 11$ in the experiments.

5.2.2 Similarity Matrix Computation

For the similarity matrix computation, we use cosine distance to compute the similarity between two documents. In document clustering, cosine distance has been widely used [59] [6]. The cosine distance between two vectors (two documents) is defined as:

$$similarity(A, B) = \cos(\theta) = \frac{A \cdot B}{||A|| \cdot ||B||}, \tag{5.3}$$

where $A \cdot B$ represents the inner product of A and B, $||A||$ represents the magnitude of vector A.

5.3 Experimental Setup

The experiments are conducted in two separate clusters. The first cluster is the five-machine cluster testbed introduced in Sec. 4.1. We run PSC, DASC and SC on this five-machine cluster. We compare the results of DASC against those of PSC and SC. The second cluster is Amazon Elastic MapReduce cluster (denoted as EMR). We run DASC on EMR and compare the performance against that of DASC running in the five-machine cluster.

5.3.1 Experimental Setup of Five-Machine Cluster

The first cluster is the five-machine cluster testbed introduced in Sec. 4.1. Configurations can be found in Sec. 4.1. The datasets are taken from the crawled Wikipedia dataset, the size of the datasets range from one thousand to two million points. The clustering information is listed in Table. 5.1.

Dataset size	Number of categories
1024	17
2048	31
4096	61
8192	96
16384	201
32768	330
65536	587
131072	1225
262144	2825
524288	5535
1048576	14237
2097152	42493

Table 5.1: Clustering information of Wikipedia dataset.

5.3.2 Experimental Setup of Amazon Elastic MapReduce Cluster

The second cluster is Amazon Elastic MapReduce cluster. The setup is as follows: there are 64 machines in the cluster, each machine has 1.7 GB memory, 350 GB of disk storage and runs Hadoop 0.20.2 on top of Debian Linux. We list the setting of important parameters in Table. 5.2. The dataset is the whole Wikipedia dataset comprised of 3,550,567 docs. We use the method in Sec. 5.2 to do data processing.

EMR works in conjunction with Amazon EC2 to create a Hadoop cluster, and with Amazon S3 to store scripts, input data, log files, and output results. Our experiments on EMR go in the following steps:

- Upload to Amazon S3 the input data, as well as the mapper and reducer executables that process the data, and then send a request to EMR to start a job flow.
- Start a Hadoop cluster and run Hadoop on each node. Hadoop executes a job flow by downloading data from Amazon S3 to core and task nodes.
- Process the data and upload the results from the cluster to Amazon S3.

A job flow is a collection of processing steps that EMR runs on a specified dataset using a set of Amazon EC2 instances. A job flow consists of one or more steps. Our experiment is a job flow, which is comprised of several steps. The first step is dataset partitioning step. In this step, we partition the dataset into buckets. Every bucket is a file stored in S3 containing

Parameter	Value
Hadoop jobtracker heapsize	768 MB
Hadoop namenode heapsize	256 MB
Hadoop tasktracker heapsize	512 MB
Hadoop datanode heapsize	256 MB
Maximum map tasks in tasktracker	4
Maximum reduce tasks in tasktracker	2
Data replication ratio in DFS	3

Table 5.2: Elastic MapReduce cluster setup.

points in the corresponding bucket. After the first step, a group of Spectral Clustering steps are added into the job flow. Each Spectral Clustering step runs on a bucket. The job flow terminates after all the buckets are processed.

5.4 Performance Metrics

We consider the following performance metrics: clustering accuracy, processing time and memory usage. Clustering accuracy is measured by the ratio of correctly clustered number of points to the total number of points. Processing time is the running time of the program. Since every entry in the the similarity matrix is of type float (4 bytes), the memory usage is $MemUsage = 4 \times Num\_Entries$, where $Num\_Entries$ is the number of entries in the similarity matrix.

5.5 Experimental Results on Wikipedia Dataset

5.5.1 Accuracy

Fig. 5.4 shows the clustering accuracy comparison. The values in x-axis stand for the size of the dataset, the values in y-axis stand for the clustering accuracy ratio (in percentage). It can be observed that, SC has the highest accuracy, it uses the similarity matrix without approximation to do clustering. For DASC and PSC, they use approximated matrices for Spectral Clustering, but the accuracies of DASC and PSC are both close to that of SC, and DASC performs better than PSC. This result reveals two things: firstly, the method of converting the raw Wikipedia corpus into numerical dataset can successfully preserve the semantic meaning of each document, and help the clustering algorithms yield good results.

Secondly, the preprocessing step in DASC can successfully partition the original dataset into buckets according to the documents' similarity.

5.5.2 Memory Consumption

Fig. 5.5 shows the memory consumption comparison of DASC, SC and PSC. It can be seen that, memory footprint nearly quadruples for SC and PSC when the dataset size doubles, the memory footprint is very large when using large dataset. PSC is superior to SC in the aspect of memory usage. The reason is that, for every point in the dataset, it only keeps a portion of large similarity entries in the matrix, thus it takes smaller amount of memory space. The figure also shows that the memory usage of the proposed DASC is smaller than those of PSC and SC. The reason is that, instead of computing pair-wise similarity across all the data points, it computes sub-similarity matrix on every bucket's basis. Therefore, it can handle much larger datasets. Moreover, DASC enables the parallel computing of several Spectral Clustering tasks on a cluster. The memory pressure on a single machine in the cluster is alleviated by its peers.

5.5.3 Processing Time

Fig. 5.6 and Table 5.3 show the processing time comparison of DASC, SC and PSC. SC works on a fully filled similarity matrix. PSC works on an approximated similarity matrix, this matrix only retains the entries that are close to a point using a threshold. However, the common feature SC and PSC share is that, in the whole process, there is one single Spectral Clustering instance on the original dataset. As is shown in Sec. 3.3, similarity matrix computation and Laplacian matrix computation both require quadratic time to compute, the computing time of SC and PSC grow very rapidly with the growth of dataset. In contrast, DASC initiates a group of Spectral Clustering instances, each of which is based on a subset of the original dataset. Each instance deals with a small portion of the whole dataset. What's more, there is no dependency over the instances, DASC runs the instances simultaneously in the cluster machines. It achieves sub-quadratic computing time. Therefore, DASC achieves significant speedup. This reduction of computing time also makes it possible to process larger datasets.

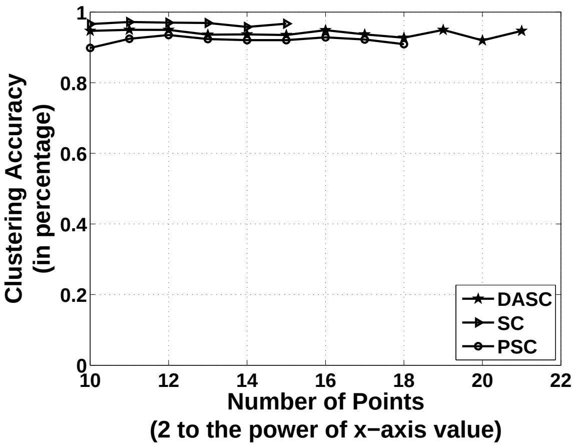


Figure 5.4: Clustering accuracy comparison on Wikipedia dataset.

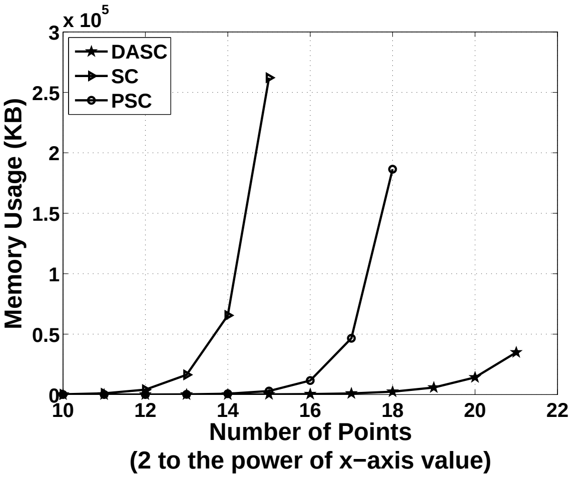


Figure 5.5: Memory usage comparison on Wikipedia dataset.

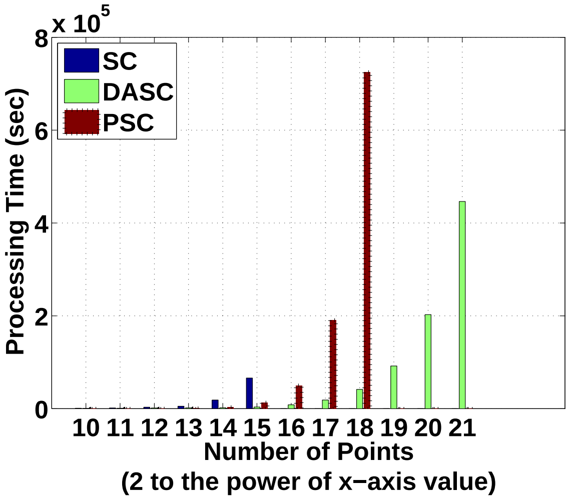


Figure 5.6: Processing time comparison on Wikipedia dataset.

	SC	DASC	PSC
1024 points	99.8	109.9	16.6
2048 points	419.0	252.1	63.3
4096 points	1499.9	523.7	240.8
8192 points	5248.3	1009.7	915.3
16384 points	18612.8	2057.1	3478.2

Table 5.3: Processing time comparison from 1024 to 16384 points using Wikipedia datasets (in second).

Criterion	EMR	5-node cluster
Accuracy	96.6%	95.8%
Memory usage	72492 KB	71488 KB
Processing time	49 hours and 51 minutes	295 hours and 34 minutes

Table 5.4: Wikipedia dataset clustering results on EMR and 5-node cluster.

5.5.4 Scalability and Experimental Results Using Amazon Elastic MapReduce

In this section, we compare the performance of DASC running on two different clusters. These two clusters are five-machine cluster testbed introduced in Sec. 4.1 and Amazon Elastic MapReduce cluster (EMR) introduced in Sec. 5.3.2. Through comparison, we show that the proposed DASC algorithm is highly scalable, the computing time can be greatly reduced when we increase the underlying computing power. The experimental result is listed in Table 5.4.

We can see from Table 5.4 that when using more machines to run DASC on the large-scale Wikipedia dataset, the running time is reduced significantly. The reduction ratio is about 83.4%. At the same time, memory usage and clustering accuracy are not affected. It should be noted that, on the 5-node cluster, we run DASC alone without the interference from other computing intensive applications. On the contrary, the EMR cluster is shared by Amazon clients, we do not have exclusive access to the cluster. Therefore, in a certain period of time, there might be a surge in simultaneously running applications, our share of CPU time and possible IO delay will then churn.

5.5.5 Summary

In this section, we evaluated the proposed DASC algorithm using real-life Wikipedia datasets and compared it against two closest algorithms in literature: SC [46], PSC [13]. The experimental results show that DASC can successfully group Wikipedia documents on similar topics into separate clusters (with around 95% accuracy), greatly reduce computing time (with the reduction ratio of 16 when comparing against SC, and with the reduction ratio of 10 when comparing against PSC), and significantly reduce the memory usage. By comparing the clustering performances of running Wikipedia dataset on five node cluster and 64 node Amazon Elastic MapReduce cluster, we show that the proposed algorithm DASC has great scalability. If the underlying computing power is increased, the running time of DASC reduces significantly.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Large-scale datasets are prevalent in data mining applications nowadays. Many kernel-based clustering techniques in the Machine Learning field are not scalable to large datasets, either because of unacceptable running time or significant memory usage. In this thesis, we studied the problem of approximating similarity matrices to facilitate the computation of kernel-based techniques on large datasets. The proposed method can significantly reduce the running time and the memory consumption of kernel-based techniques, while at the same time retain clustering accuracy. We showed that the proposed approximation method outperforms other similar solutions that approximate the similarity matrices and parallelized the computation process. We implemented the proposed idea using the MapReduce framework, conducted extensive experiments on a cluster testbed as well as in Amazon Elastic MapReduce cloud system. The evaluation study confirms the significant potential performance gain. It shows that the proposed method can handle large-scale datasets with good clustering accuracy. Specifically, comparing against the original Spectral Clustering, the proposed approximate method can achieve (i) up to 76% memory footprint reduction, (ii) good clustering accuracy (the clustering error is within 8%). Moreover, the proposed method is highly scalable, the degree of parallelization is adaptive to the size of the nodes in the cluster. Therefore, if the cluster size is increased, the processing time is greatly reduced.

6.2 Future Work

The work in this thesis can be extended in multiple directions. For example, in order to reduce the number of points to be processed, we can first select some representative points using a coarse-grained low-complexity clustering technique, such as Canopy Clustering. After that, use the resulted centroids to do Spectral Clustering. We can also implement the proposed method as a pluggable library that runs preceding the kernel-based clustering techniques to enable them to process large-scale datasets. Another possible extension is to investigate the interaction between the proposed method and the cloud computing hardware. Finally, it is interesting to study how raw datasets (semantic corpus, graphs, social network links, etc) can be processed to better preserve and reveal the innate clustering information such that clustering process can yield better performance.

Bibliography

[1] M. Aizerman, E. Braverman, and L. Rozonoer. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control, Vol. 25*, pages 821–837, 1964.

[2] Aris Anagnostopoulos, Anirban Dasgupta, and Ravi Kumar. Approximation algorithms for co-clustering. In *Symposium on Principles of Database Systems*, pages 201–210, 2008.

[3] Yair Bartal, Nathan Linial, Manor Mendel, and Assaf Naor. Low dimensional embeddings of ultrametrics. *European Journal of Combinatorics*, 25:87–92, 2004.

[4] P. Berkhin. A survey of clustering data mining techniques. In Jacob Kogan, Charles Nicholas, and Marc Teboulle, editors, *Grouping Multidimensional Data*, chapter 2, pages 25–71. 2006.

[5] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *Int. Conf. on Database Theory*, pages 217–235, 1999.

[6] Mikhail Bilenko and Raymond J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *In Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2003)*, pages 39–48, 2003.

[7] David M. Blei and Michael I. Jordan. Variational inference for dirichlet process mixtures. *Bayesian Analysis*, 1:121–144, 2005.

[8] James Breneman. Kernel methods for pattern analysis kernel methods for pattern analysis. *Technometrics*, 47:237–237, 2005.

[9] Jeremy Buhler and Martin Tompa. Finding motifs using random projections. In *Proceedings of the fifth annual international conference on Computational biology*, RECOMB '01, pages 69–76, 2001.

[10] Tony Chan. Hierarchical algorithms and architectures for parallel scientific computing. In *In Proc. ACM Int'l Conf. Supercomputing*, pages 318–329, 1990.

[11] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. of 34th STOC*, pages 380–388, 2002.

[12] Keke Chen and Ling Liu. ivibrate: Interactive visualization based framework for clustering large datasets. *ACM Transaction on Information Systems*, pages 245–294, 2006.

[13] Wen-Yen Chen, Yangqiu Song, Hongjie Bai, Chih-Jen Lin, and Edward Y. Chang. Parallel spectral clustering in distributed systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 568 – 586, 2010.

[14] Ondrej Chum, James Philbin, and Andrew Zisserman. Near duplicate image detection: min-hash and tf-idf weighting. In *British Machine Vision Conference*, pages 25–31, 2008.

[15] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods.* Cambridge University Press, 1 edition, 2000.

[16] J. Cullum and R. Willoughby. Lanczos algorithms for large symmetric eigenvalue computations. *IEEE Transactions on Information Theory*, pages 43–49, 1985.

[17] David L. Davies and Donald W. Bouldin. A cluster separation measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1:224–227, 1979.

[18] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, 2004.

[19] Petros Drineas and Michael W. Mahoney. Approximating a gram matrix for improved kernel-based learning. In *Computational Learning Theory*, pages 323–337, 2005.

[20] Xiaoli Zhang Fern and Carla E. Brodley. Random projection for high dimensional data clustering: a cluster ensemble approach. In *In Proc. 20th International Conference on Machine Learning (ICML'03)*, pages 186–193, 2003.

[21] Charless Fowlkes, Serge Belongie, Fan Chung, and Jitendra Malik. Spectral grouping using the nystrom method. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:214 – 225, 2004.

[22] Brendan J. Frey and Delbert Dueck. Clustering by passing messages between data points. *Science*, 315:972–976, 2007.

[23] J. H. Friedman. An overview of predictive learning and function approximation. In *Statistics to Neural Networks, Theory and Pattern Recognition Applications*, pages 217–235, 1994.

[24] Warren Gish and Stephen F. Altschul. Improved sensitivity of nucleic acid database searches using application-specific scoring matrices. *Methods*, 3:66–70, 1991.

[25] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface.* MIT Press, Cambridge, MA, 1999.

[26] Sébastien Guérif and Younès Bennani. Selection of clusters number and features subset during a two-levels clustering task. In *Artificial Intelligence and Soft Computing*, pages 28–33, 2006.

[27] J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. *Journal of the Royal Statistical Society, Series C (Applied Statistics)*, pages 22–29, 1979.

[28] Erik Hatcher and Otis Gospodnetic. *Lucene in Action.* Manning Publications Co., Greenwich, CT, USA, 2004.

[29] Bernd Heisele, Purdy Ho, Tomaso Poggio, Purdy Ho, and Tomaso Poggio. Face recognition with support vector machines: Global versus component-based approach. In *In Proc. 8th International Conference on Computer Vision*, pages 688–694, 2001.

[30] Ilse C. F. Ipsen. Computing an eigenvector with inverse iteration. *SIAM Rev.*, 39:254–291, June 1997.

[31] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data.* Prentice-Hall, Inc., 1988.

[32] Ashish Kapoor, Kristen Grauman, Raquel Urtasun, and Trevor Darrell. Active learning with gaussian processes for object categorization. *Computer Vision, IEEE International Conference on*, 0:1–8, 2007.

[33] Michihiro Kobayakawa, Shigenao Kinjo, Mamoru Hoshi, Tadashi Ohmori, and Atsushi Yamamoto. Fast computation of similarity based on jaccard coefficient for composition-based image retrieval. In *IEEE Pacific Rim Conference on Multimedia*, pages 949–955, 2009.

[34] Raghu Krishnapuram, Anupam Joshi, and Liyu Yi. A fuzzy relative of the k-medoids algorithm with application to web document and snippet clustering. In *Snippet Clustering, in Proc. IEEE Intl. Conf. Fuzzy Systems - FUZZIEEE99, Korea*, 1999.

[35] Jeremy Kubica, Joseph Masiero, Andrew Moore, Robert Jedicke, and Andrew Connolly. Variable kd-tree algorithms for efficient spatial pattern search. Technical Report CMU-RI-TR-05-43, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, September 2005.

[36] Brian Kulis and Kristen Grauman. Kernelized locality-sensitive hashing for scalable image search. In *IEEE International Conference on Computer Vision (ICCV)*, pages 2130 – 2137, 2009.

[37] R. B. Lehoucq, D. C. Sorensen, and C. Yang. Arpack users guide: Solution of large scale eigenvalue problems by implicitly restarted arnoldi methods, 1997.

[38] R.B. Lehoucq and J. A. Scott. Implicitly restarted arnoldi methods and eigenvalues of the discretized navier stokes equations. *SIAM J. Matrix Anal. Appl*, 23:551–562, 1997.

[39] Xiuli Ma, Wanggen Wan, and Licheng Jiao. Spectral clustering ensemble for image segmentation. In *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, GEC '09, pages 415–420, 2009.

[40] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 141–150, 2007.

[41] O. Maqbool and H. A. Babri. The weighted combined algorithm: A linkage algorithm for software clustering. *European Conference on Software Maintenance and Reengineering*, 0:15–19, 2004.

[42] T. Moon and W. Stirling. Mathematical methods and algorithms for signal processing. pages 145 – 154, 2000.

[43] Rajeev Motwani, Assaf Naor, and Rina Panigrahi. Lower bounds on locality sensitive hashing. In *Proceedings of the twenty-second annual symposium on Computational geometry*, SCG '06, pages 154–157, 2006.

[44] Fionn Murtagh, Geoff Downs, and Pedro Contreras. Hierarchical clustering of massive, high dimensional data sets by exploiting ultrametric embedding. *Siam Journal on Scientific Computing*, 30:707–730, 2008.

[45] Ramesh Natarajan and David Vanderbilt. A new iterative scheme for obtaining eigenvectors of large, real-symmetric matrices. *J. Comput. Phys.*, 82:218–228, 1989.

[46] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Advances in Neural Information Processing Systems*, pages 849–856. MIT Press, 2001.

[47] J. P. Nolan. *Stable Distributions - Models for Heavy Tailed Data.* Birkhauser, Boston, 2011.

[48] Bahram Nour-Omid, Beresford N. Parlett, and Robert L. Taylor. Lanczos versus subspace iteration for solution of eigenvalue problems. *International Journal for Numerical Methods in Engineering*, 19:859–871, 1983.

[49] Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. *Mahout in Action.* Manning Publications, 2011.

[50] Victor Y. Pan and Ai-Long Zheng. Real and complex polynomial root-finding with eigen-solving and preprocessing. In *International Symposium on Symbolic and Algebraic Computation*, pages 219–226, 2010.

[51] Martin F. Porter. An Algorithm for Suffix Stripping. *Program*, 14(3):130–137, 1980.

[52] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007.

[53] Miloš Radovanović, Alexandros Nanopoulos, and Mirjana Ivanović. Nearest neighbors in high-dimensional data: the emergence and influence of hubs. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 865–872, 2009.

[54] Seung-Jai Min Rudolf Eigenmann Ayon Basumallik. Programming distributed memory systems using openmp. In *Workshop on High-level Interfaces for Parallel Systems, Int'l Parallel and Distributed Processing Symposium*, pages 1 – 8, 2007.

[55] S. Saha and S. Bandyopadhyay. A new cluster validity index based on fuzzy granulation-degranulation criteria. *Advanced Computing and Communications*, pages 353 – 358, 2007.

[56] Bernhard Schlkopf, Alex J. Smola, and Klaus robert Mller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10:1299–1319, 1996.

[57] J. Schuetter and Tao Shi. Multi-sample data spectroscopic clustering of large datasets using nystrom extension. *Journal of Computational and Graphical Statistics*, pages 531–542, 2011.

[58] S.Thilagamani and N.Shanthi. Literature survey on enhancing cluster quality. *International Journal On Computer Science And Engineering*, pages 115 – 119, 2010.

[59] Sandeep Tata and Jignesh M. Patel. Estimating the selectivity of tf-idf based cosine similarity predicates. *SIGMOD Record*, 36(4):75–80, 2007.

[60] Wenhua Wang, Hanwang Zhang, Fei Wu, and Yueting Zhuang. Large scale of e-learning resources clustering with parallel affinity propagation. *Proc. of International Conference on Hybrid Learning (ICHL)*, pages 30– 35, 2008.

[61] Yair Weiss. Segmentation using eigenvectors: A unifying view. In *In International Conference on Computer Vision*, pages 975–982, 1999.

[62] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 1st edition, 2009.

[63] Christopher Williams and Matthias Seeger. Using the nystrom method to speed up kernel machines. In *Advances in Neural Information Processing Systems 13*, pages 682–688. MIT Press, 2001.

[64] Donghui Yan, Ling Huang, and Michael I. Jordan. Fast approximate spectral clustering. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '09, pages 907–916, 2009.

[65] Jian Yang and Jing yu Yang. From image vector to matrix: a straightforward image projection technique - impca vs. pca. *Pattern Recognition*, 35:1997–1999, 2002.

[66] Dani Yogatama and Kumiko Tanaka-Ishii. Multilingual spectral clustering using document similarity propagation. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, EMNLP '09, pages 871–879, 2009.

[67] Ron Zass and Amnon Shashua. Doubly stochastic normalization for spectral clustering. In *Neural Information Processing Systems*, pages 1569–1576, 2006.

[68] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: A new data clustering algorithm and its applications. *Data Mining Knowledge Discovery*, 1:141–182, 1997.