DISTRIBUTED KERNEL MATRIX APPROXIMATION AND IMPLEMENTATION USING MPI

by

Taher A. Dameh

B.Sc., Jordan University of Science and Technology, Jordan 2008

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

SIMON FRASER UNIVERSITY
Summer 2012

APPROVAL

Name: Taher A. Dameh

Degree: Master of Science

Title of Thesis: Distributed Kernel Matrix Approximation and Implementation Using MPI

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. Jiangchuan Liu
Associate Professor, Computing Science
Simon Fraser University
Examiner

Date Approved: June 11th 2012

Partial Copyright Licence

Abstract

We propose a distributed method to compute similarity (also known as kernel and Gram) matrices used in various kernel-based machine learning algorithms. Current methods for computing similarity matrices have quadratic time and space complexities, which make them not scalable to large-scale data sets. To reduce these quadratic complexities, the proposed method first partitions the data into smaller subsets using various families of locality sensitive hashing, including random project and spectral hashing. Then, the method computes the similarity values among points in the smaller subsets to result in approximated similarity matrices. We analytically show that the time and space complexities of the proposed method are subquadratic. We implemented the proposed method using the Message Passing Interface (MPI) framework and ran it on a cluster. Our results with real large-scale data sets show that the proposed method does not significantly impact the accuracy of the computed similarity matrices and it achieves substantial savings in running time and memory requirements.

Keywords: Large-scale data processing, kernel matrix approximation, big data, distributed clustering, kernel-based algorithms.

To the Memory of My Father.

Simplicity is the ultimate sophistication.

Leonardo da Vinci

Acknowledgments

I am deeply grateful to my senior supervisor, Dr. Mohamed Hefeeda, for his guidance through my research. Mohamed provided valuable insights, a lot of help during my graduate career. 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. Jiangchuan Liu 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 also like to extend my gratitude to the faculty and staff in the school of computing science at SFU, particularly Dr. Joseph G. Peters, Dr. Alexandra Fedorova, Dr. Tamara Smyth, Dr. Arrvindh Shriraman and Dr. Cameron Harvey for what I have learned in their courses.

I would like to thank all the members at the Network Systems Lab at SFU, and my two friends Samer Al-Kiswany and Mutasem AlShare', for their help.

Last but certainly not least, I would like to thank my family for their care, love, and support. This thesis is dedicated to them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The current information explosion has resulted in an increasing number of applications that need to deal with large volumes of data. While traditional algorithm analysis assumes that the data fits in main memory, it is unreasonable to make such assumptions when dealing with massive data sets such as multimedia contents and web page repositories. Kernel methods are one of the techniques that deal with high volume of data. In this chapter, we present an overview of kernel methods and their challenges. In this chapter, we also outline the contributions of the thesis.

1.1 Kernel Methods

Kernel methods are a class of algorithms for pattern analysis. The general task of pattern analysis is to find and study general types of relations; for example clusters, rankings, principal components, correlations, and classifications; in general types of data, such as sequences, text documents, sets of points, vectors, images, etc. Muller et al. [44] give a good introduction to kernel-based learning algorithms.

Kernel methods approach the problem by mapping the data into a high dimensional feature space, where each coordinate corresponds to one feature of the data items, transforming the data into a set of points in an Euclidean space. In that space, a variety of methods are used to find relations in the data. Since the mapping can be quite general (not necessarily linear, for example), the relations found in this way are accordingly very general. This approach is called the kernel trick.

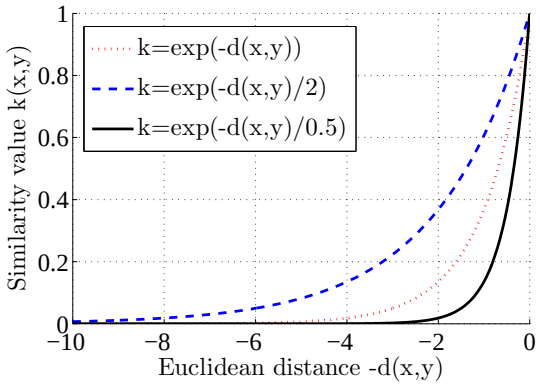


Figure 1.1: Exponential kernel function with different values of parameter c=0.5,1 and 2.

Kernel methods owe their name to the use of kernel functions, which enable them to operate in the feature space without ever computing the coordinates of the data in that space, but rather by simply computing the inner products between the images of all pairs of data in the feature space. The result of such is called the Gram or Kernel matrix, which has a complexity of $O(n^2)$ in terms of space and time.

Gram Matrix: Given a set P of n vectors (points in $\Re^d$), the Gram matrix is the matrix of all possible inner products of P, i.e.,

$$k_{ij} = p_i^T p_j, \tag{1.1}$$

where A^T denotes the transpose.

One of the most common kernel functions is the Gaussian Redial Basis function [44] or the exponential function. For two points x, y in the space, the similarity value k between them is:

$$k(x, y) = exp(\frac{-\|x - y\|^2}{c}), \tag{1.2}$$

where c is a real number and used to control the values of similarities, as shown in Figure 1.1; as c increases similarity value increases.

Algorithms capable of operating with kernels include support vector machines (SVMs) [9], Gaussian processes [53], kernel Fisher discriminant (KFD) [42], kernel principal component analysis KPCA [42] and many others. Kernel-based algorithms are used for applications in many areas and fields, for example: web documents and web images clustering [3] [7] [39], DNA and protein analysis [54], optical pattern and object recognition [13].

1.2 Problem Statement and Thesis Contributions

1.2.1 Kernel Methods Challenges

One of the main challenges encountered by kernel methods is the scalability problem; i.e., the massive amount of information they have to deal with. In particular, the large number of feature dimensions, and the large number of data items.

High-Dimensional Data

In high-dimensional data, we may have thousands or even hundreds of thousands of dimensions. Such high-dimensional data spaces are often encountered in areas such as medicine, where DNA microarray technology [41] can produce a large number of measurements at once. And the clustering of text documents [34], where, if a word-frequency vector is used, the number of dimensions equals the size of the dictionary.

According to Kriegel et al. [37], some of the problems that need to be overcome when dealing with high-dimensional data are:

1. Multiple dimensions are hard to think in, impossible to visualize, and due to the exponential growth of the number of possible values with each dimension, impossible to enumerate. This problem is known as the curse of dimensionality.

2. The concept of distance becomes less precise as the number of dimensions grows, since the distance between any two points in a given dataset converges. The discrimination of the nearest and farthest point in particular becomes meaningless:

$$\lim_{d\to\infty} \frac{dist_{max} - dist_{min}}{dist_{min}} \to 0 \tag{1.3}$$

Large-Scale Data

The other main challenge is the large-scale data. The significant limitations of many such algorithms is that the kernel function $k(x, y)$ must be evaluated for all possible pairs x and y, which is computationally expensive. To overcome such limitation, one should think about distributed and approximation algorithms to process large-scale data sets.

The approximation algorithms are done by observing the eigen-spectrum of the kernel function. It is a Radial Basis function, which is a real-valued function whose value depends on the Euclidean distance. Thus, the kernel function is monotonically decreasing with the Euclidean distance between the input points. Substantial memory requirements reductions can be gained by computing the kernel function only between close points, i.e., preserving the large similarities of the Gram matrix, and filtering out the small similarities. So the problem is to find the close points in the space in a fast way, i.e., in a time complexity that is less than the one of computing the Gram matrix itself. This can be done using spatial indexing or spatial hashing that has locality preserving property. Previous work by Hussein and Abd-Almageed [32] uses space filling curve, specifically, the Z-curve, to order the points in the space, by computing the Z-index of each point, which takes $O(n)$, and then sorting the indeces in $O(n \log n)$. They use a sliding window of a specified length over the sorted points, where they compute the kernel function between points that reside within this sliding window.

In this work, we present a novel approach to scale kernel-based methods, using the locality sensitive hashing (LSH) to hash points in the space so that the probability of collision is high for close points. LSH is widely used in the k-nearest neighbor problem [1] [11] [24]. We focus on low dimensional data (< 100 attributes), Gao [23] has studied high dimensional data (> 1000 attributes), he uses random projection LSH to cluster wikipedia documents. We compute the kernel function between points that reside in the same bucket of the hash table. Kernel-based applications is distributable using our algorithm, we have sub-problems to solve, which are the buckets that generate sub-Gram matrices.

1.2.2 Thesis Contributions

The contributions of the thesis are:

1. We propose an approximation algorithm for computing large scale Gram matrices. Gram matrices are the core component of the kernel based machine learning algorithms. The method achieves substantial memory and computation savings.

2. We present a distributed implementation of the proposed method using MPI framework, our distributed implementation solves the scalability challenge of the kernel methods.

3. We implement a recent kernel-based machine learning algorithm, which is the affinity propagation clustering algorithm, on top of the proposed Gram matrix approximation algorithm.

4. We conduct rigorous empirical evaluation study based on our implementation and deployment on a multi node computer cluster. We use real data from the US census of 1990. Our results show an achievable accuracy of more than 90% comparing to the full Gram matrix.

1.2.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 provides background information and related work. In particular, more information about the kernel-based the affinity propagation clustering algorithm [22], space filling curves, and locality sensitive hashing. Chapter 3 presents our proposed Distributed Kernel Matrix Approximation (DKMA) algorithm and its distributed implementation using MPI. Experimental evaluation is presented in Chapter 4, with details on the performance metrics, experiments setup and data set used. Then we conclude the thesis and describe potential future extensions in Chapter 5. Table 1.1 shows the descriptions of the symbols we use in this thesis for an easy reference.

n	Data set size in points
d	Data set dimension
k	Number of hash value bits
h	Value of 1 bit of the hash value
g	Hash value: k-bits
m	Hash table size $= 2^k$
M	Metric space in d-dimension
S	Metric space in k-dimension
$d(x, y)$	Euclidean distance between two points x and y
p_1	Low probability for two far points to collide
p_2	High probability for two close points to collide
K	Full Gram matrix of size n^2
$\tilde{K}$	Approximated Gram matrix of less size

Table 1.1: Symbols descriptions

Chapter 2

Background and Related Work

This chapter provides background information on the affinity propagation clustering algorithm, which is an example of a kernel-based application. In this chapter also the definition of the spatial indexing or hashing using space filling curves and locality sensitive hashing . We also provide previous works on approximating the Gram matrix.

2.1 Background

2.1.1 Kernel-Based Application (Affinity Propagation Clustering)

One usage of kernel matrix is clustering. Clustering or cluster analysis is the task of assigning a set of objects into groups (called clusters) so that the objects in the same cluster are more similar to each other than to those in other clusters.

Clustering can be achieved by various algorithms that differ significantly in their notion of what constitutes a cluster and how to efficiently find clusters. Popular notions of clusters include groups with 1) low distances among the cluster members, 2) dense areas of the data space/intervals, or 3) particular statistical distributions. So, typical cluster models can be categorized into:

- Connectivity models: for example hierarchical clustering [30] builds models based on distance connectivity.
- Centroid models: for example the k-means algorithm [29] and affinity propagation [22] represents each cluster by a single mean vector.

- Distribution models [16]: clusters are modeled using statistic distributions, such as multivariate normal distributions used by the expectation-maximization algorithm.
- Density models [36]: for example DBSCAN [17] and OPTICS [4] define clusters as connected dense regions in the data space.
- Subspace models [45]: also known as co-clustering or two-mode-clustering, clusters are modeled with both cluster members and relevant attributes, this model is used for high-dimensional data.

In this work, we use centroid-based clustering algorithms, where the notions of clusters include groups with low distance among the cluster members. These types of algorithms can get benefits of the LSH properties. LSH strives to collide close points with higher probability. We choose affinity propagation [22] as an application case study. Affinity propagation is a new algorithm that takes as input measures of similarity between pairs of data points (Gram matrix). Affinity propagation does not need to have the number of exemplars defined in advance as the case in k-means clustering, instead affinity propagation simultaneously considers all data points as potential exemplars. Real-valued messages are exchanged between data points until a high-quality set of exemplars and corresponding clusters gradually emerges.

In details, Affinity propagation takes as input the Gram matrix, where the kernel function is $k(x, y) = -\|x - y\|^2$, $k(x, y)$ indicates how well the data point with index y is suited to be the exemplar for data point x. Rather than requiring that the number of clusters be pre-specified, affinity propagation takes as input a real number $k(i, i)$ for each data point i, so that data points with larger values of $k(i, i)$ are more likely to be chosen as exemplars. These values are referred to as self similarities or preferences. The number of identified exemplars (number of clusters) is influenced by the values of the input preferences, but also emerges from the message-passing procedure. If all data points are equally suitable as exemplars, the preferences should be set to a common value. This common value could be the median of the input similarities (resulting in a moderate number of clusters) or their minimum (resulting in a small number of clusters).

The two kinds of message exchanged between data points are the responsibility and the availability. Messages can be combined at any stage to decide which points are exemplars and, for every other point, which exemplar it belongs to. The responsibility $r(i, j)$, sent from data point i to candidate exemplar point j, reflects the accumulated evidence for how well-suited point j is to serve as the exemplar for point i. The availability $a(i, j)$, sent from candidate exemplar point j to point i, reflects the accumulated evidence for how appropriate it would be for point i to choose point j as its exemplar. To begin with, the availabilities are initialized to zero: $a(i, j) = 0$. Then, the responsibilities are computed using the rule:

$$r(i, j) = k(i, j) - max_{j' \ s.t \ j' \neq j}(a(i, j') + k(i, j')). \tag{2.1}$$

Whereas the above responsibility update lets all candidate exemplars compete for ownership of a data point, the following availability update gathers evidence from data points as to whether each candidate exemplar would make a good exemplar:

$$a(i, j) = min(0, r(j, j) + \sum_{i' \ s.t \ i' \notin i,j} max(0, r(i', j))). \tag{2.2}$$

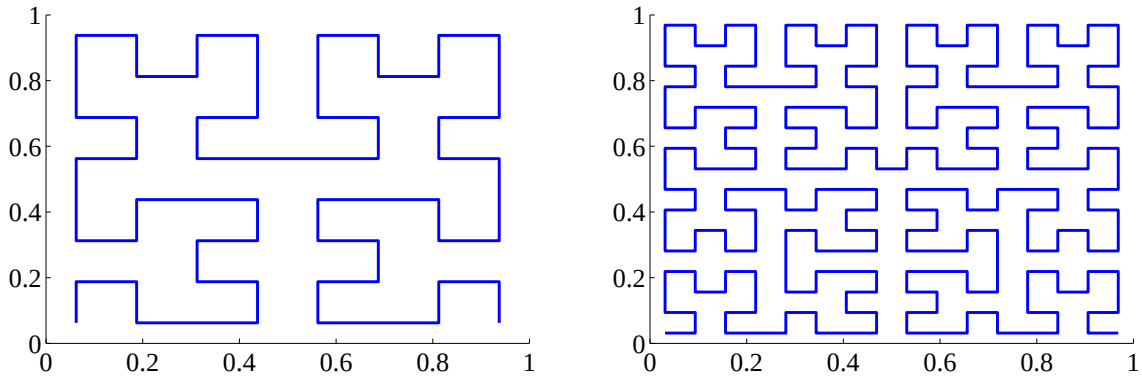
The self-availability a(j,j) is updated differently:

$$a(j, j) = \sum_{i' \ s.t \ i' \neq j} max(0, r(i', j)). \tag{2.3}$$

This message reflects accumulated evidence that point j is an exemplar, based on the positive responsibilities sent to candidate exemplar j from other points. The algorithm proceeds by iterating over the responsibility and availability update steps until convergence or the maximum number of iterations is reached.

2.1.2 Space Filling Curve

An N-dimensional space-filling curve is a continuous function from the unit interval $[0, 1]$ to the N-dimensional unit hypercube $[0, 1]^N$. For example, a 2-dimensional space-filling curve is a continuous curve that passes through every point of the unit square $[0, 1]^2$. One example is the Hilbert Curve, first described by the German mathematician David Hilbert in 1891. We use the H-curve as a ground truth to compare our algorithm using the locality sensitive hashing. Previous work [32] has been done using the Z-curve.



(a) (b)

Figure 2.1: Hilbert space filling curve with third and fourth orders respectively.

Both the true Hilbert curve and its discrete approximations are useful because they give a mapping that preserves locality fairly well. For example in 2D, if (x, y) is the coordinates of a point within the unit square, and d is the distance along the curve when it reaches that point, then points that have nearby d values will also have nearby (x, y) values. The converse cannot always be true. But the Hilbert curve does a good job of keeping those d values close together much of the time. So the mappings in both direction do a fairly good job of maintaining locality. Because of this locality property, the Hilbert curve is widely used in computer science. Figure 2.1 shows the Hilbert curve in 2D, with third and forth orders.

2.1.3 Locality Sensitive Hashing

Locality sensitive hashing [8] is defined with respect to a universe of items U that has a similarity function $sim : U \times U \rightarrow [0, 1]$. An LSH scheme is a family of hash functions H, coupled with a probability distribution D over the functions, such that, a function $h \in H$ chosen according to D satisfies the property that for any $a, b \in U$:

$$Pr_{h \in H}[h(a) = h(b)] = sim(a, b), \tag{2.4}$$

i.e., the probability of collision between two points is proportional to the similarity between them. It is often convenient to have a hash function family that maps objects to $\{0, 1\}$. In

that case, the output of t different hash functions can simply be concatenated to obtain a t-bit hash value for an object.

LSH schemes are known to exist for many distance or similarity measures, for example: l_p norms [24] [11], Jaccard coefficient [6], cosine distance and the earth movers distance (EMD) [8]. Moreover, there are other families that work on the eigen spectrum of the data, such as the spectral hashing [51]. More details on different LSH families are presented in section 3.1.

2.2 Related Work

2.2.1 Low Rank Matrix Approximation Based on Nystrom Theorem

In low rank matrix approximation, we need to solve the problem of approximating a matrix K with another matrix $\tilde{K}$ which has a specific rank r. In the case that the approximation is based on minimizing the Frobenius norm of the difference between K and $\tilde{K}$ under the constraint that $rank(\tilde{K}) = r$. i.e.:

$$\begin{aligned} &minimize \left\| K - \tilde{K} \right\|_F, \\ &subject\ to\ rank(\tilde{K}) = r. \end{aligned} \tag{2.5}$$

$$\tag{2.6}$$

It turns out that the solution is given by the Singular Value Decomposition (SVD) of K:
Let K be a square $n * n$ matrix with n linearly independent eigenvectors, $q_i\,(i = 1, \ldots, n)$. Then K can be factorized as:

$$K = U \Lambda U^T, \tag{2.7}$$

where U is the square $n * n$ matrix whose i^{th} column is the eigenvector q_i of K and Λ is the diagonal matrix whose diagonal elements are the corresponding eigenvalues, i.e., $\Lambda_{ii} = \lambda_i$.

And hence the low rank matrix $\tilde{K}$ is constructed like:

$$\tilde{K} = U \tilde{\Lambda} U^T, \tag{2.8}$$

where $\tilde{\Lambda}$ is the same matrix as Λ except that it contains only the r largest singular values (the other singular values are replaced by zero), so that we say $\tilde{K}$ has the low rank r. This is known as the EckartYoung theorem, as it was proved by those two authors in 1936 [48].

The Eigen decomposition is $O(n^3)$ computations, which is expensive. Nystrom theorem is used to approximate the eigen decomposition, by the following:
For some $m \ll n$ build $U \in R^{n \times m}$ by choosing m rows/columns of U and let $\Lambda_{m \times n} = diag(\lambda_1, \ldots, \lambda_m)$. where in this way we reduce the matrix computation complexity down to $O(mn)$.

For sampling the m rows or columns in Nystrom method, the most popular sampling scheme is random sampling, which leads to fast versions of kernel machines [52] [38], and spectral clustering [18]. In [46], several variants of multidimensional scaling are shown to be related to Nystrom approximation. Other methods [15] use randomized algorithms by sampling the columns of the Gram matrix based on a pre-computed distribution using the norms of the columns. The reconstruction of the Gram matrix is also normalized by the sampling distribution. The later randomized algorithms is more expensive in terms of computation complexity.

In spite of low-rank approximation algorithms based on Nystrom theorem reduce the computation down to $O(mn)$ where $m \ll n$, they fail to reduce the space complexity, still we need $O(n^2)$ to store the Gram matrix. On the other hand, the following methods reduce both computations and space by using efficient way to compute the Gram matrix.

2.2.2 Efficient Implementation for Computing the Gram Matrix

In these methods, the basic idea is to compute the kernel function only between close points (points with high similarity), based on the fact that the kernel functions are radial basis functions, i.e., their values depend on the Euclidean distance between the points. The question that arises is how to find the close points in a fast way? This can be done either by spatial indexing or spatial hashing where the preprocessing step (hashing or indexing) should be smaller than quadratic, which is the complexity of computing the Gram matrix. Hussein and Abd-Almageed in [32] use the Z-curve to order the points in the space, then by

using a sliding window of size w, the kernel function is computed only within this window, i.e, we compute the kernel function only between each point and the points that are at most $w/2$ far from it on the Z-order in both directions.

The drawbacks of such methods are:

1. their accuracy depends on the size w,
2. they fail for high dimensional space,
3. the sorting step is $O(n \log n)$,
4. and more importantly, they are hard to distribute, as we need to process the approximated Gram matrix as one.

To overcome these drawbacks, we propose a new algorithm using the locality sensitive hashing. We hash the points in space into m buckets. Then the kernel function is computed only between the points that reside in the same bucket. We show that we choose m to be $O(\log n)$ which reduces our computations and space down to sub quadratic. Our proposed algorithm has the following advantages:

1. linear time of preprocessing,
2. and more importantly, it is distributable. We have sub-problems to solve, which are the buckets of the hash table, where each is considered as a sub-Gram matrix. We compute the kernel function on each bucket independently. Moreover, we run clustering algorithm on each bucket independently as well.

2.3 Example of Approximating the Gram Matrix Using H-Curve and LSH

Figure 2.2 shows an example of eight points in 2D, in Figure 2.2(b) an H-curve is used to order the points in the space, and in 2.2(c) an LSH with hash bits $k = 2$ is used to hash the points into four buckets. Table 2.1 shows the full Gram matrix as values and as a gray image (white means 1 and black means 0). From the image, and given that the points are sorted based on H-order, the large similarities are concentrated on and around the diagonal. Table 2.2 left side shows the approximated Gram matrix using H-curve with $winWidth = 2$, and right side shows it using LSH with $k = 2$. Using H-curve, we compute the kernel function between each point and up to $\pm(winWidth/2)^{th}$ point on the curve, using LSH we compute the kernel function between points in the same bucket. In this way, we preserve the large similarities and filter out the small similarities. One of the metrics to measure the method accuracy is the Frobenius norm; see section 4.1. The Frobenius norm for the full Gram matrix in this example is 4.13, where it is 3.82 and 3.44 for the H-curve and LSH respectively.

1	0.9	0.3	0.3	0	0	0.3	0.3
0.9	1	0.4	0.4	0	0	0.2	0.1
0.3	0.4	1	0.8	0.1	0.1	0.1	0
0.3	0.4	0.8	1	0.4	0.3	0.2	0.1
0	0	0.1	0.4	1	0.9	0.3	0.2
0	0	0	0.3	0.9	1	0.3	0.1
0.2	0.1	0	0.2	0.3	0.3	1	0.9
0.2	0.1	0	0.1	0.2	0.1	0.9	1

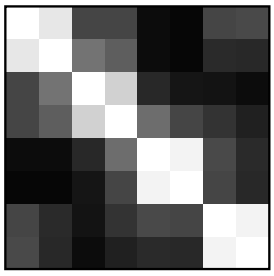


Table 2.1: The full Gram matrix as values and as a gray image for the example in Figure 2.2(a). The points are sorted based on H-order.

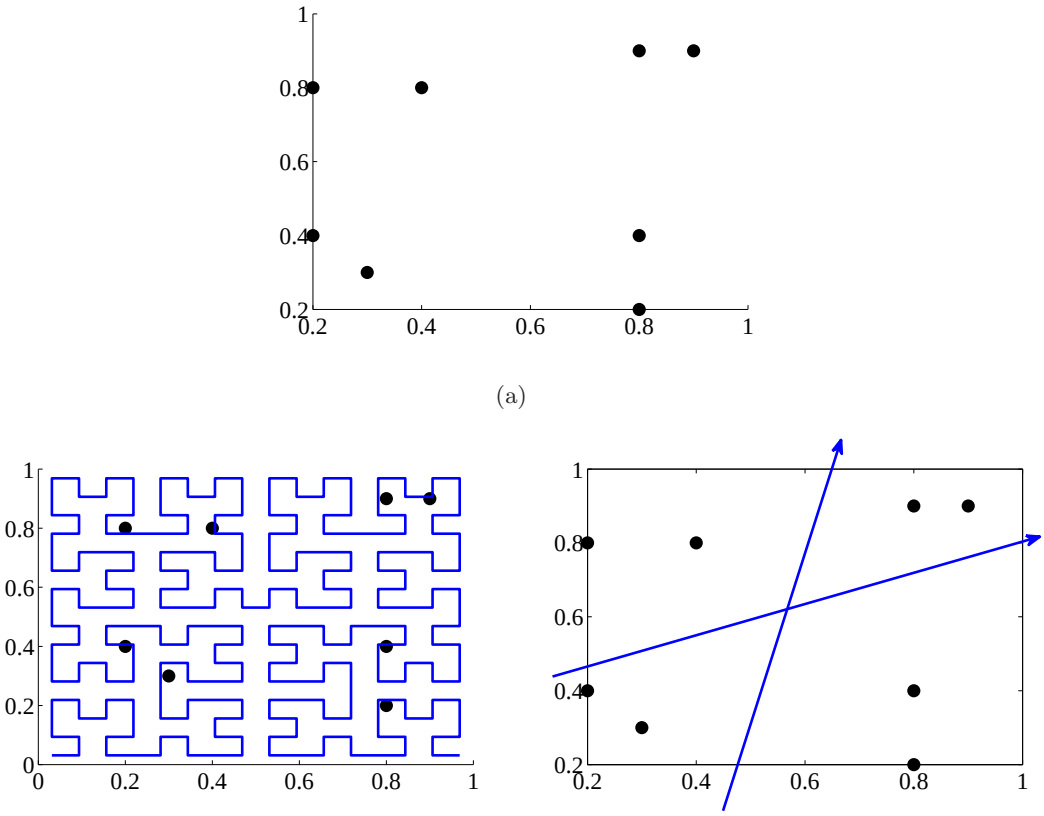


Figure 2.2: An example in 2D of constructing the Gram matrix using the Hilbert space filling curve and using locality sensitive hashing.

1	**0.9**	0	0	0	0	0	**0.3**	**1**	**0.9**	0	0	0	0	0	0
0	**1**	**0.4**	0	0	0	0	0	**0.9**	**1**	0	0	0	0	0	0
0	0	**1**	**0.8**	0	0	0	0	0	0	**1**	**0.8**	0	0	0	0
0	0	0	**1**	**0.4**	0	0	0	0	0	**0.8**	**1**	0	0	0	0
0	0	0	0	**1**	**0.9**	0	0	0	0	0	0	**1**	**0.9**	0	0
0	0	0	0	0	**1**	**0.3**	0	0	0	0	0	**0.9**	**1**	0	0
0	0	0	0	0	0	**1**	**0.9**	0	0	0	0	0	0	**1**	**0.9**
0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	**0.9**	**1**

Table 2.2: The approximated Gram matrix for the points in Figure 2.2(a) using the Hilbert curve (left one) with $w = 2$, and using LSH (right one) with $k = 2$.

Chapter 3

Distributed Kernel Matrix Approximation (DKMA) Algorithm

In this chapter, we present our proposed distributed kernel matrix approximation algorithm, we start with a background information on different locality sensitive hashing families, and a background on the distributed programming. In section 3.3, we outline our proposed algorithm in details, then we present the analysis of the algorithm in section 3.4, and then we conclude the chapter in section 3.5.

3.1 Locality Sensitive Hashing

3.1.1 Definition

Locality sensitive hashing (LSH) is a method of performing probabilistic dimension reduction of high-dimensional data. The basic idea is to hash the input items so that similar items are mapped to the same bucket with high probability.

As been defined in [24] and [33], an LSH family $\mathcal{F}$ is defined for a metric space $\mathcal{M} = (M, d)$, as an $\mathcal{F}$ family of functions $h : \mathcal{M} \to S$ satisfying the following conditions for any two points

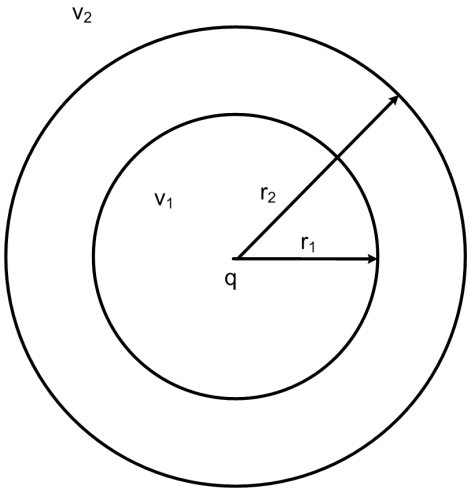


Figure 3.1: Two circles with a radius r_1 and r_2 respectively, q is the center of both, v_1 within r_1 and v_2 outside r_2, then v_1 collides with q with a probability at least p_1, and v_2 collides with q with a probability at most p_2, where $p_2 < p_1$.

$v, q \in \mathcal{M}$, and a function h is chosen uniformly at random from $\mathcal{F}$:

if $d(v,q) \leq r_1$, then $Pr[h(v) = h(q)] \geq p_1$

if $d(v,q) \geq r_2$, then $Pr[h(v) = h(q)] \leq p_2$

In other words, in a metric space (M, d), the d-sphere of a center q and a radius $r_1 > 0$, if there is a point v resides within this sphere, then v and q collide with high probability at least p_1. If there is a point p resides outside this sphere, then v and p collide with low probability at most p_2, see Figure 3.1. A family $\mathcal{F}$ is interesting when $p_1 > p_2$ and $r_1 < r_2$. Such a family $\mathcal{F}$ is called $(r_1, r_2, p_1, p_2) - sensitive$. For k specified later, define a function family $\mathcal{G} = g : S \rightarrow U^k$ such that $g(v) =< h_1(v), ..., h_k(v) >$, where $h_i \in \mathcal{F}$.

LSH schemes are known to exist for the following distance or similarity measures: Hamming norm [24], cosine distance [8], l_1 distance [40], l_p norms [24] and Jaccard coefficient [6].

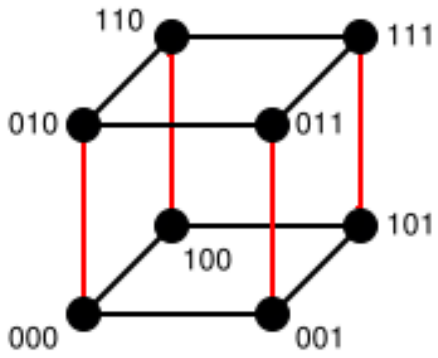


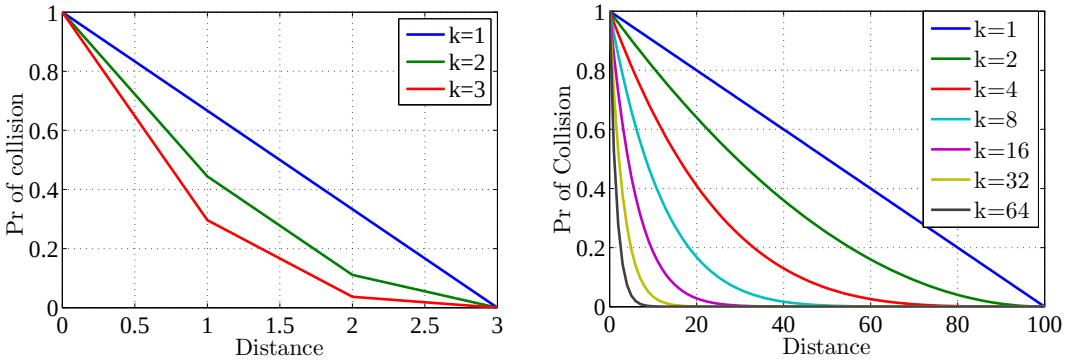
Figure 3.2: The Hamming space for $d = 3$.

3.1.2 Hamming Distance LSH

Hamming Distance LSH [33] is the basic family. Consider points from $\{0,1\}^d$ (Hamming Space) with Hamming distance $D(p,q)$ equals the number of positions by which p and q differ. Define hash function g by choosing a set S of k random coordinates, and setting $g(p)$ = projection of p on S.
In such family, the probability of collision between points will be proportional to their Hamming distance (as distance increases, probability of collision decreases).

$$Pr[g(p) = g(q)] = (1 - \frac{D(p,q)}{d})^k. \tag{3.1}$$

Example: Consider an example of dimension $d = 3$, see the cube in Figure 3.2, let number of bits $k = 2$, and let the set S of randomly chosen coordinates as $\{1,3\}$, this means we have $2^2 = 4$ different buckets, and they are:
00=$\{\underline{0}0\underline{0},\underline{0}1\underline{0}\}$
01=$\{\underline{0}0\underline{1},\underline{0}1\underline{1}\}$
10=$\{\underline{1}0\underline{0},\underline{1}1\underline{0}\}$
11=$\{\underline{1}0\underline{1},\underline{1}1\underline{1}\}$



(a) $d = 3$ (b) $d = 100$

Figure 3.3: Probability of collision versus the Hamming distance for different values of k.

Figure 3.3(a) shows the Hamming distance along with the probability of collision for different values of k, similarly Figure 3.3(b) shows this relation for the case of $d = 100$; as k increases the gap between the high probability p_1 and the low probability p_2 increases.

3.1.3 Cosine Distance LSH

This LSH family [8] is designed to approximate the cosine distance between vectors. The basic idea is to choose a random hyperplane (defined by a normal unit vector r) at the outset and use the hyperplane to hash input vectors.

Given an input vector v and a hyperplane defined by r, we let $h_r(v) = sgn(v \cdot r)$. That is, $h(v) = \pm 1$ depending on which side of the hyperplane v lies.

$$bit = \begin{cases} 1 & \text{if } r.u \geq 0 \\ 0 & \text{if } r.u < 0 \end{cases}$$

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 once again. It is not difficult to prove that, for two

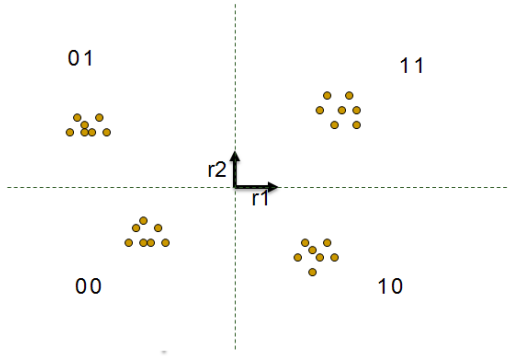


Figure 3.4: Cosine distance based LSH.

vectors u, v,

$$Pr[h(u) = h(v)] = 1 - \frac{\theta(u,v)}{\pi}, \tag{3.2}$$

where $\theta(u,v)$ is the angle between u and v. $1 - \frac{\theta(u,v)}{\pi}$ is closely related to $\cos(\theta(u,v))$, [8].

In Figure 3.4, an example of points in 2D, we use two random vectors r_1 and r_2 to hash. By projecting the points onto those two random vectors, we obtain the four buckets as shown.

3.1.4 l_1 Distance LSH

This LSH family [40] [50] generates k-bit vectors from the d-dimensional vectors, such that the expected Hamming distance between two bit vectors produced is proportional to the l_1 distance between the corresponding vectors. It generates each single bit from each d-dimensional vector, such that, the probability that the bit produced is different for two vectors is proportional to their l_1 distance.

Let the ith coordinate of the d-dimensional vector be in the range $[l_i, h_i]$ and has weight w_i. Let $T = \sum_i w_i \times (h_i - l_i)$, and $p_i = w_i \times (h_i - l_i)/T$. Note that $\sum_i p_i = 1$. To generate a single bit, pick $i \in [0, d-1]$ with probability p_i, and pick a uniform random number $t \in [l_i, h_i]$. For each vector $v = (v_1, ..., v_d)$, we have:

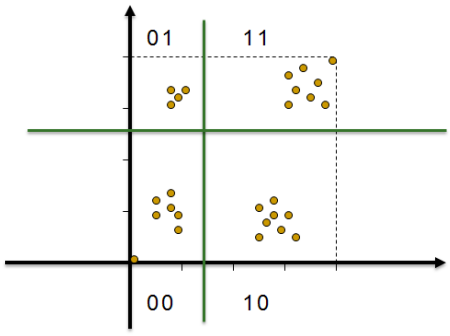


Figure 3.5: Thresholding (l_1 distance) based LSH.

$$bit = \begin{cases} 0 & \text{if } v_i < t \\ 1 & \text{if } v_i \geq t \end{cases}$$

The authors of [40] [50] prove the following lemma regarding this LSH family:

Lemma 1 *If the weighted l_1 distance between two vectors u and v is x, then the probability that the two vectors generate different bits given the same (i,t) pair is $p = x/T$.*

In Figure 3.5, an example of points in 2D, the thresholding (l_1 distance LSH) is used here, with $k = 2$, to hash the points into four buckets. For the x-coordinate ($i_1 = 1$), let $t_1 = 1.5$, and for the y-coordinate ($i_2 = 2$), let $t_2 = 2.5$.

3.1.5 p-Stable Distributions LSH

It is mentioned in [33] that it is possible to extend the algorithm for the Hamming space to the l_2 norm by embedding l_2 space into l_1 space, and then l_1 space into Hamming space. However, it increases the error by a large factor and complicates the algorithm.

In [11] where LSH scheme is proposed based on p-stable distribution, the algorithm works directly on points in Euclidean space without embedding, it works for any l_p norm, as long

as $p \in (0, 2]$. The algorithm inherits a convenient property of LSH schemes, which is it works well on data that have high dimensions.

Stable distributions are defined as limits of normalized sums of independent and identically distributed variables. The most well-known example of a stable distribution is the Gaussian (or normal) distribution.

Stable Distribution: A distribution D over R is called p-stable, if there exists $p \geq 0$ such for any n real numbers $v_1...v_n$ and i.i.d. variables $X_1...X_n$ with distribution D, the random variable $\sum_i v_i X_i$ has the same distribution as the variable $(\sum_i |v_i|^p)^{1/p} X$, where X is random variable with distribution D.
It is known that stable distribution exists for any $p \in (0, 2]$. In particular:

-Cauchy Distribution D_C defined by the density function $c(x) = \frac{1}{\pi}\frac{1}{1+x^2}$, is 1-stable.

-Gaussian Distribution D_G defined by the density function $g(x) = \frac{1}{\sqrt{2\pi}}e^{-x^2/2}$, is 2-stable.

The idea is to generate a random vector a of dimension d whose each entry is chosen independently from a p-stable distribution. Given a vector v of dimension d the dot product $a.v$ is a random variable which is distributed as $\|v\|_p X$, where X is a random variable with p-stable distribution. Such a sketch is linear, i.e., for any $p, q \in R^d$, $a.(p-q) = a.p - a.q$.

Indyk and Motwani [11] use p-stable distribution in a slightly different manner. Instead of using the dot product to estimate the l_p norm, they use them to assign a hash value to each vector v. The hash function family should be locality sensitive, so if the two vectors (v_1, v_2) are close (small $\|v_2 - v_2\|_p$), then they should collide (hash to the same index) with high probability, and if they are far away, they should collide with small probability.

Each hash function $h_{a,b}(\mathbf{v}) : R^d \to N$ maps a d-dimensional vector v onto a set of integers. Each hash function in the family is indexed by a choice of random a and b where a is, as before, a d-dimensional vector with entries chosen independently from a p-stable distribution, and b is a real number chosen uniformly from the range $[0, W]$.

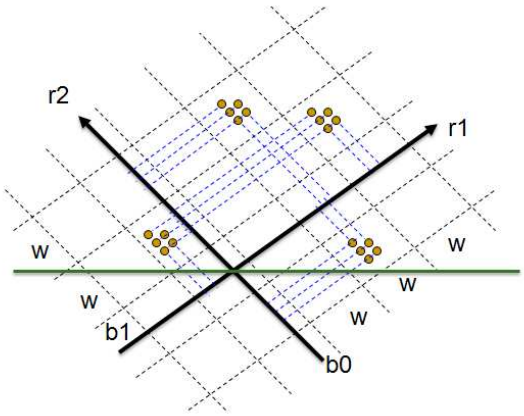


Figure 3.6: p-Stabel based LSH.

For a fixed a and b, the hash function $h_{a,b}$ is given by:

$$h_{a,b}(v) = \left\lfloor \frac{a.v + b}{W} \right\rfloor . \tag{3.3}$$

Figure 3.6 is an example of points in 2D, the points were hashed into buckets using two random vectors r_1 and r_2, a window of width w and two random shifts b_0 and b_1, each square obtained represents a bucket in the hash table.

3.1.6 Spectral Hashing

Random projection techniques mentioned in the previous sections have strong theoretical guarantees. Unfortunately, they do not learn the hash values from the data set. For example, in p-stable LSH family, every bit in the hash value is calculated by a random linear projection followed by a random threshold. Then the Hamming distance between the hash values will asymptotically approach the Euclidean distance between the items. But in practice this method can lead to inefficient hash values. Rather than using random projection to define bits in a hash value, other families have been proposed to learn the hash values from the data. One good example is the spectral hashing proposed by Weiss et al. [51].

Let $\{g_i\}_{i=1}^n$ be the list of hash values or binary vectors of length k for n data points. $W(x,y) = exp(-\|x-y\|^2/\epsilon^2)$ is the affinity matrix which characterizes similarities between data points. We need the average Hamming distance between similar points to be minimal, i.e:

$$minimize \quad : \quad \sum_{ij} W_{ij} \|g_i - g_j\|^2 \tag{3.4}$$

$$subject\ to \quad : \quad g_i \in \{0,1\}^k. \tag{3.5}$$

We need also each bit to be 0 or 1 with equal probability of 50%, and the bits to be uncorrelated. i.e:

$$\sum_i g_i = 0 \tag{3.6}$$

$$\frac{1}{n} g_i g_i^T = I \tag{3.7}$$

For a single bit, solving the previous problem is equivalent to the balanced graph partitioning problem, which is NP hard [2]. But by introducing an $n \times k$ matrix G whose jth raw is g_j^T and a diagonal $n \times n$ matrix $D(i,i) = \sum_j W(i,j)$, we can rewrite the problem as:

$$minimize \quad : \quad trace(G^T(D-W)G) \tag{3.8}$$

$$subject\ to \quad : \quad G(i,j) \in \{-1,1\} \tag{3.9}$$

$$G^T 1 = 0 \tag{3.10}$$

$$G^G Y = I \tag{3.11}$$

$$\tag{3.12}$$

This is still a hard problem, but by removing the constraint that $G(i,j) \in \{-1,1\}$, we obtain an easy problem whose solutions are the k eigenvectors of $D-W$ with minimal eigenvalue, which is similar to spectral graph partition [47] that could be solved by computing generalized eigenvalue problem.

So, the problem is to compute the eigenvector and eigenvalue of the graph $D-W$, but the eigenproblem is computationally expensive, because it has a time complexity of

$O(n^3)$. Thus it can not handle large data sets. The solution is by using eigenfunctions of the weighted Laplace-Beltrami L_p operators defined on manifolds [5].

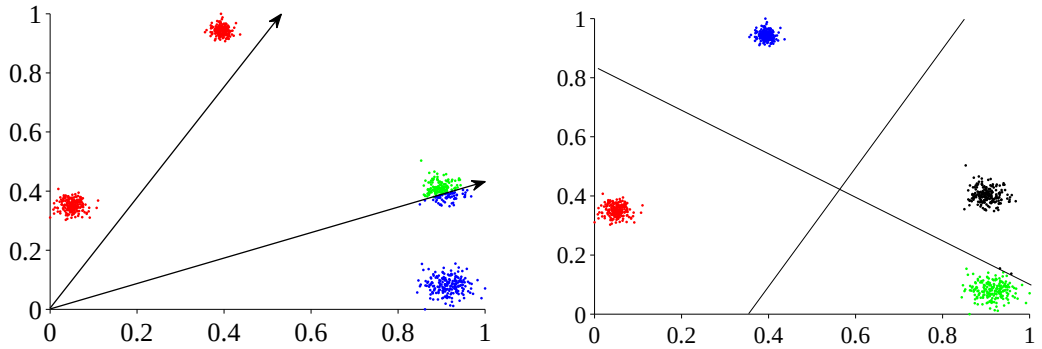
In summary, given a training set of points x_i and a desired number of bits k, the spectral hashing algorithm works by [51]:

- Finding the principal components of the data using Principle Component Analysis (PCA).
- Calculating the k smallest single-dimension analytical eigenfunctions of L_p using a rectangular approximation along every PCA direction. This is done by evaluating the k smallest eigenvalues for each direction, thus creating a list of dk eigenvalues, and then sorting this list to find the k smallest eigenvalues.
- Thresholding the analytical eigenfunctions at zero, to obtain binary codes.

Spectral hashing is linear time, because the Principle Component Analysis step is done in linear time, by first finding the covariance matrix of the data set, and then finding the eigen solution of that matrix in linear time using the Lanczas algorithm [25]. Golub and van Loan give very good description of the various forms of Lanczos algorithms in their book Matrix Computations [25].

Spectral hashing has been shown to be effective in hashing large-scale, low-dimensional data since the important PCA directions are selected multiple times to create hash values [49]. However, for high dimensional data sets ($d \gg n$) where many directions contain enough variance, usually each PCA direction is picked only once. This is because the top few projections have similar range and thus a low spatial frequency ($k = 1$) is preferred. In this case, spectral hashing approximately replicates a PCA projection followed by a mean partition [49]. Moreover, in spectral hashing, the similarity matrix W is fixed as $exp(-\|x-y\|^2/\epsilon^2)$, and thus it does not apply for other kernel/similarity functions. The authors in [31] propose a new hashing algorithm for large-scale data that is applicable for any kernel function.

In Figure 3.7, we show an example of data points in 2D, where we use random projection to hash the points in 3.7(a) , and we use spectral hashing in 3.7(b), $k = 2$ for both.



(a) Random Projection LSH

(b) Spectral Hashing

Figure 3.7: Random projection versus spectral hashing for $d = 2$ using $k = 2$.

Each color represents a bucket. The figures show how the spectral hashing is more powerful, because it takes in account the eigen spectrum of the data, and not just randomly projecting.

3.1.7 Summary and Discussion of LSH Families

In the previous sections, we reviewed different locality sensitive hashing families. In general they can be categorized into two groups, random projection and spectral hashing.

In random projection technique, the basic idea behind these families is dimension reduction, where the data points are projected on smaller dimension space. These families are built on top of Johnson-Lindenstrauss lemma [35], which shows that any data in high dimension space can be mapped into smaller dimension space of size related logarithmically to the data size, such that distances between points are preserved within a small factor. These families are simple, work well on high-dimensional data, and have strong theoretical guarantees.

On the other hand, the spectral hashing [51] has been proposed to learn the hash values from the data. In spectral hashing we compute PCA, calculate the smallest single-dimension analytical eigen functions, and then threshold the analytical eigen function of zero to obtain values' bits. Spectral hashing has been shown to be effective for low-dimensional data, and for the exponential kernel function [49].

3.2 Distributed Programming

One popular class of parallel architectures is clusters, which often use standard components and often standard network technology, so as to leverage as much commodity technology as possible. Cluster is called distributed-memory computing or shared-nothing computing, which is in contrast to the shared-memory multiprocessors. Cluster programming can be done using the message passing interface MPI [27], or a high level framework; the MapReduce [12].

3.2.1 MPI Programming

MPI is a standardized and portable message-passing system designed by a group of researchers from academia and industry [27] [28] to be used for parallel programs running on distributed-memory systems. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs. It is a language-independent communications protocol.

In MPI programming, user writes a single program, that has multiple instances execute on (potentially) different data sets. The program is parameterized so that each process works on its own subset of data. If the total number of processes is less than the number of available nodes, each single node will handle one process. Number of processes is fixed at start of program (load time), each process has a unique ID, and performs one of two things: computation on local data or communication with other processes. In MPI programming, we have a distributed file system or network file system (NFS), which is a file system that allows access to files from the multiple nodes sharing the network. This makes it possible for multiple nodes to share code and data.

The basic MPI library functions are:

1. MPI_Init: to initialize MPI,
2. MPI_Finalize: to shut down MPI,
3. MPI_Get_Rank: to get the current process ID,

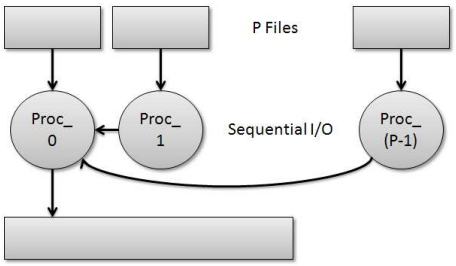


Figure 3.8: Sequential I/O.

4. MPI_Get_Size: to get the total number of processes, and

5. MPI_Barrier: to synchronize with everyone.

For example the following for loop:

```
for (i=0; i<MAX; i++)
  do_somthing(i);
```

can be distributed on cluster using a total number of processes equals to p, and assuming each loop is independent from the other, like:

```
id = MPI_get_rank();
p = MPI_get_size();
for (i=id; i<MAX; i+=p)
  do_somthing(i);
```

Moreover, MPI programming supports parallel I/O [28]. In sequential I/O, all processes send data to rank 0 process (master process), then master process writes the data to the file, see Figure 3.8. Such lack of parallelism limits scalability and performance. One solution is to make each process write/read to/from a separate file, but in this case, lots of small files to manage, and difficult to read back data from different number of processes. On the other hand, using parallel I/O, multiple processes of a parallel program accessing data, reading or writing, from or to a common file at the same time, see Figure 3.9.

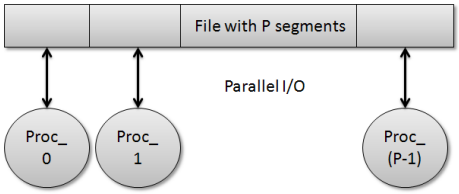


Figure 3.9: Parallel I/O.

3.2.2 MapReduce Programming

MapReduce is a high level framework introduced by Google in 2004 [12] to support distributed computing on clusters of computers. In the Map step: the 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 the smaller problem, and passes the answer back to its master node. On the other hand, in the Reduce step: The master node collects the answers to all sub-problems and combines them to form the output, i.e., the answer to the problem it was originally trying to solve.

In MapReduce, communication management is effectively gone, i.e., I/O scheduling is done for us. Fault tolerance and monitoring are implicitly handled. I.e., MapReduce is more reliable. But on the other hand, MapReduce restricts solvable problems, that it might be hard or more complicated to express some problems using it. Where in MPI, it is more flexible, and can be used to solve most distributed problems. Our distributed problem can be implemented using both MapReduce and MPI. In our thesis, we choose MPI to implement our distributed algorithm. Comparing between the two implementation is a future work.

3.3 Proposed DKMA Algorithm using MPI

3.3.1 Overview

Our approximation algorithm depends on the observation that the kernel function is Radial Basis, whose value depends only on the distance. We use the locality sensitive hashing to hash the points that reside in the space. LSH preserves the locality by making the close points in the space collide with high probability. We compute the kernel function between points that collide in the same bucket. We cluster each bucket independently. Clustering is distributable using our method.

In summery our distributed algorithm works by, see Figure 3.10:

- Initialize all processes with the same k vectors or eigenfunctions.
- Load each process with a segment from the data set.
- Each process hashes the segment to bucket files.
- Combine buckets that have the same index number.
- Load each process with a bucket.
- Each process clusters each bucket and stores results in a shared clusters file.

3.3.2 Distributed Implementation of LSH

Random Projection LSH

We initialize all processes with the same k **vectors**. Each vector has the same dimensionality of the data set and is used to generate one bit of the hash value for each data item. The data set is divided into small segments. Each single segment s will be handled by a process. All processes use the same k vectors mentioned to hash the points into local buckets. Each local bucket has a unique index number, then all local buckets from the different processes that have same index are combined into one bucket.

Spectral Hashing

In spectral hashing instead of using random vectors, it needs to learn eigenfunctions from the data set using principle component analysis (PCA). To do so, we need to scale the PCA function. The step of PCA that has a problem when running on large-scale data set is computing the covariance matrix of it.

We scale the covariance matrix computation by observing that: for a matrix K of size n and dimension d, the covariance matrix C is of size $d \times d$, where c_{ii} is the variance of the column i in matrix K, and c_{ij} is the covariance between the two columns i and j in matrix K.

Different processes work independently on matrix K to generate each element of matrix C, master process - slave processes paradigm is used. For example, to compute the variance of column i in matrix K, that is the element c_{ii} in matrix C, the master process pings all slave processes to start, each slave process reads a chunk from column i, aggregates it and returns the sum to the master process. The master process receives all sums to compute the mean of the column i. The master process will ping the slave processes again by passing the mean to each one, each slave will subtract the mean from each element of its chunk, aggregate and then return the result back to the master. The master process receives all results to compute the variance and then update the element c_{ii} in C.

3.3.3 Distributed Implementation of Clustering

We mentioned that kernel-based applications are distributable using locality sensitive hashing. We do not need to store and process the whole approximated Gram matrix, instead we store and process sub-Gram matrices independently, where each sub-Gram matrix represents a different bucket from the hash table. Each bucket will be handled by an independent process.

Algorithm 1 shows more details of the implementation using MPI.

The Algorithm has four steps:

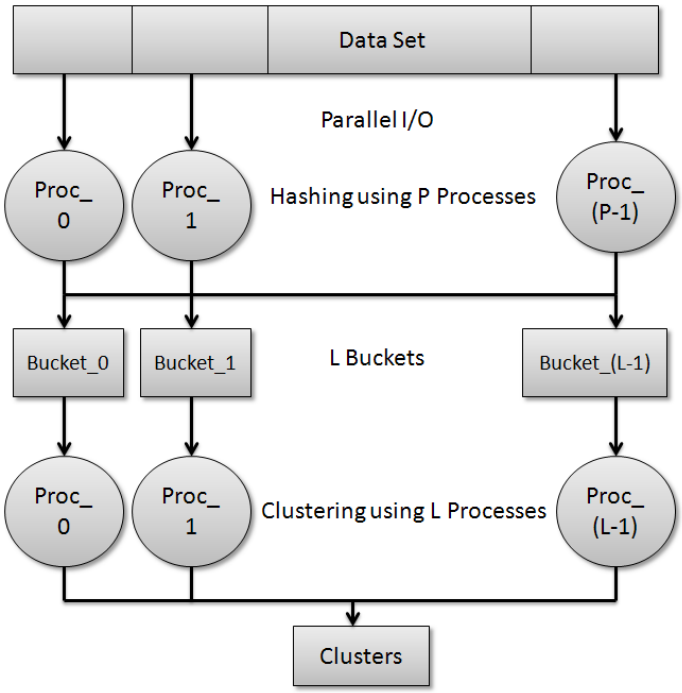


Figure 3.10: The flow chart of the DKMA algorithm.

1. **Pre-processing:** The data set is divided into small segments, where each segment can be handled by one process.

2. **MPI_PartitionData:** We initialize the MPI program with number of processes equals to the number of the segments. We initialize each process with same k hash functions. Each process reads single segment and hashes it to local bucket files.

3. **Mid-processing:** We combine the local buckets from the different processes that have the same index number into one bucket.

4. **MPI_Cluster:** We initialize number of processes equals to the total number of buckets generated after hashing. Each process reads one bucket and uses the affinity propagation to cluster it. All processes write to a shared clusters file.

Algorithm 1 DKMA algorithm using MPI

```
1: Pre-processing divide the data set into numofSegments segments
2:
3: function MPI_PartitionData( numofSegments, k )
4:     Input dataSegments[numofSegments], hashFunctions[k]
5:     output bucketFiles[numofSegments][totalBuckets]
6:     MPI_Initialize_numofProcesses(numofSegments)
7:     myRank ← MPI_Get_Rank()
8:     mySegment ← read(dataSegments[myRank])
9:     LSH_Initialize(hashFunctions[k])
10:    for i = 0 → segmentSize − 1 do
11:        index ← LSH_Index(mySegment[i])
12:        bucketFiles[myRank][index].add(mySegment[i])
13:    end for
14:    return totalBuckets
15: end function
16:
17: Mid-processing combine buckets that have same index
18:
19: function MPI_Cluster( totalBuckets )
20:    Input bucketFiles[totalBuckets]
21:    output clustersFile
22:    MPI_Initialize_numofProcesses(totalBuckets)
23:    myRank ← MPI_Get_Rank()
24:    myBucket ← read(bucketFiles[myRank])
25:    myGramMatrix ← computeGramMatrix(myBucket)
26:    myClusters ← cluster(myGramMatrix)
27:    clustersFile.write(myClusters)
28: end function
```

3.4 Analysis and Complexity

3.4.1 Time Complexity of Computing the Gram Matrix

We show that using the random projection locality sensitive hashing, the quadratic complexity of the brute force approach for computing the gram matrix is reduced down to sub quadratic.

Johnson-Lindenstrauss lemma [35] states that any n points subset of Euclidean space can be embedded in $k = O(\log n/\epsilon^2)$ dimensions without distorting the distances between any pair of points by more than a factor of $(1 \pm \epsilon)$, for any $0 < \epsilon < 1$. In other words, any set of n points in d-dimensional Euclidean space can be embedded into k-dimensional Euclidean space - where k is logarithmic in n and independent of d - so that all pairwise distances are maintained within an arbitrary small factor. All known constructions of such embedding involve projecting the n points onto a random k-dimensional hyperplane. The proof of Johnson-Lindenstrauss lemma was subsequently simplified by Frankl and Maehara [20]. The proof given by Dasgupta and Gupta [10] uses elementary probabilistic techniques to obtain the result. Indyk and Motwani [33] have also given similar proofs of the theorem using simple randomized algorithm for their p-stable LSH, they show that the value of hash bits k is $\log_{1/p_2} n$, where p_2 is the low probability for two far points to collide in the same bucket.

Given $k = \log_{1/p} n$ and since the hash value is a k-bit value, our hash table size m is $2^k = 2^{\log_{1/p} n}$. Assuming all buckets have the same size. Also assume that we have enough nodes such that each node will handle one process at most. Thus, computing the Gram matrix using random projection LSH will need:

$$\frac{n^2}{m^2} = \frac{n^2}{2^{2\log_{1/p} n}}, \tag{3.13}$$

$$= \frac{n^2}{2^{\frac{2\log_2 n}{\log_2 1/p}}}, \tag{3.14}$$

Thus,

$$\frac{n^2}{m^2} = \frac{n^2}{n^{\frac{2}{\log_2 1/p}}} = O(n^{2-c}). \tag{3.15}$$

where $c = 2/\log_2(1/p)$, and assuming the low probability $0 < p < 0.25$, then $0 < c < 1$. Hence, the complexity is sub-quadratic.

In our implementation using MPI, we divide the data set into p segments to be processed by p processes as a pre-processing step, which takes $O(n)$. Also in a mid-processing step before the clustering and after hashing, we combine all buckets of the same index number into one bucket, this takes $O(n)$ as well. Therefore, the total time complexity is $O(n^{2-c})$.

3.4.2 Clustering Complexity

When using the full Gram matrix, the complexity of the affinity propagation clustering algorithm is $O(n^2 \log n)$ [22], where n is the data size. Using our algorithm, and given that the bucket size is n/m, where $m = 2^k$ is the hash table size, and assuming each node will handle one process at most, the complexity will be reduced down to:

$$
\begin{aligned}
(\frac{n}{m})^2 \log \frac{n}{m} = \frac{n^2}{2^{2k}} \log \frac{n}{2^k} &= \frac{n^2}{2^{2 \log_{1/p} n}} \log \frac{n}{2^{log_{1/p} n}} && (3.16)\\
&= \frac{n^2}{2^{\frac{2 \log_2 n}{\log_2 (1/p)}}} \log \frac{n}{2^{\frac{\log_2 n}{\log_2 (1/p)}}} && (3.17)\\
&= \frac{n^2}{n^{2/log_2 (1/p)}} \log \frac{n}{n^{2/2log_2 (1/p)}} && (3.18)\\
&= O(n^{2-c} \log n^{1-c/2}), && (3.19)
\end{aligned}
$$

where $c = 2/\log_2(1/p)$, and given the low probability $0 < p < 0.25$, then $0 < c < 1$.

3.5 Summary

In this chapter, we presented the DKMA algorithm. It partitions large data sets using locality sensitive hashing (LSH). We analyzed different families of LSH. We found that spectral hashing out performs other families, as it learns the hash values from the data set itself. Then we presented the distributed implementation of DKMA using the MPI framework. Our implementation is hardware independent, it means that it can run on any size of cluster. If we have enough resources such that each node will handle one process at most, we will achieve a maximum speedup on a cluster up to the total number of buckets which is 2^k, where k is the number of hash functions. As we will see in the results shortly, our method accuracy is inversely proportional to k, but by using JohnsonLindenstrauss lemma, we can achieve reasonable accuracy using $k = \log_{1/p} n$, where n is the data set size in points, and p is the low probability for two far points to collide.

Chapter 4

Experimental Evaluation

In this chapter, we present the experimental evaluation of the DKAM algorithm. We start with the performance metrics we use to evaluate our algorithm comparing to the full Gram matrix and comparing to the method using Hilbert curve. Then we describe the data set we use which is the US census of 1990, and the setup of our experiments. We show the results in section 4.4 for the accuracy and the memory consumption, as well as the results of using large-scale data set of size one million items.

4.1 Performance Metrics

4.1.1 Low Level - Gram Matrix Level Metrics

Frobenoius Norm

The Frobenius norm, sometimes also called the Euclidean norm, is a matrix norm of an $m \times n$ matrix K defined as the square root of the sum of the absolute squares of its elements, i.e., for the matrix K, the Frobenoius norm is:

$$\|K\|_F = \sqrt{\Sigma_{i=1}^{m} \Sigma_{j=1}^{n} |k_{ij}|^2} \tag{4.1}$$

In the Gram matrix, the Frobenius norm reflects the large similarities; as a larger value of an element has more effect on the Frobenoius norm value. If we have two approximated matrices of the same size, in terms of number of elements, then the one with larger Frobenoius norm value is the closer to the full Gram matrix.

The Frobenoius norm depends on the size of the matrix; as the size increases the Frobenoius norm increases. Moreover, we need to compute the Frobenoius norm for the full Gram matrix, to evaluate the Frobenoius norm ratio. This is infeasible for large-scale matrices.

To overcome those issues, we propose a new metric: the FrobNorm.MemReduction product, which is defined in the following section. This metric is inspired from the Energy.Delay product used in Energy-Efficient computing proposed first by Gonzalez and Horowitz [26]. Energy.Delay product considers both performance and energy consumption of processors. Similarly, our metric considers both accuracy and memory usage simultaneously, which makes better comparisons between different methods and different configurations in terms of efficiency.

FrobNorm.MemReduction Product

FrobNorm.MemReduction product takes in account both the amount of information the matrix has as well as its size. This metric is size independent and can be used to pick up the optimal value of number of hash functions k. It is the result of multiplying the Frobenius norm with the memory reduction achieved at some value of k. On the other hand, FrobNorm^2.MemReduction or even FrobNorm^3.MemReduction can be used to assign more weight to the Frobenius norm.

$$FrobNorm.MemReduction = FrobNorm \times (1 - \frac{Approximated\ Gram\ matrix\ size}{Full\ Gram\ matrix\ size}). \tag{4.2}$$

Figure 4.1 illustrates the different performance metrics for a synthesized data set of size 1000 points, and each point has 64 dimensions. The data set is hashed using random projection. Figure 4.1(a) is the Frobenius norm for the approximated Gram matrix obtained at different values of k. In Figure 4.1(b), as k increases, the Gram matrix size decreases, and thus the Frobenoius norm decreases. Figure 4.1(c) is the memory reduction achieved. From the first raw of figures, there is no indication on the optimal value of k. On the other hand, FrobNorm.MemReduction, FrobNorm^2.MemReduction and FrobNorm^3.MemReduction are

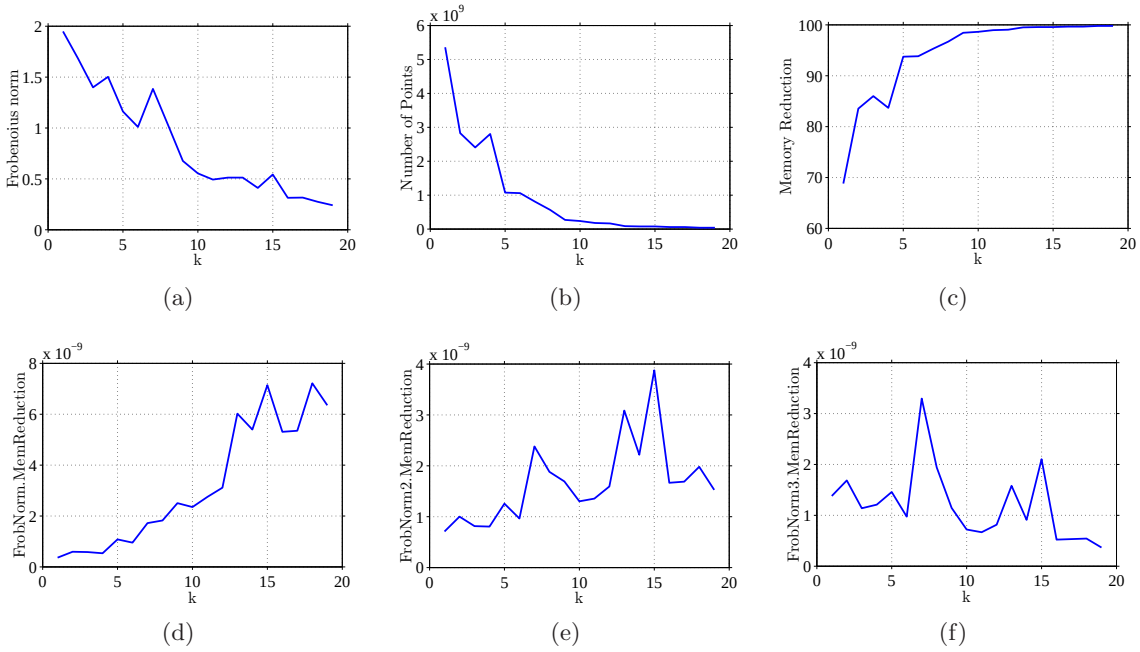


Figure 4.1: Performance Metrics

Random projection hashing for 1000 points each has 64 dimensions. First raw shows the Frobenoius norm, number of points and the memory reduction achieved for the approximated Gram matrix using different values of k (hashing bits). The second raw shows FrobNorm.MemReduction, FrobNorm^2.MemReduction and FrobNorm^3.MemReduction.

shown in the second raw of Figures. We use this metric to choose the optimal value of k; which is the one with the maximum value of FrobNorm.MemReduction. Moreover, assigning more wight to Frobenoius norm, by using FrobNorm^c.MemReduction, shifts the optimal value of k to the left, i.e., smaller value for which Frobenoius norm or accuracy is higher.

4.1.2 High Level - Application Level Metrics

Clustering Error

The previous two metrics are low level, as they work directly on the Gram matrix. We consider another metric that is high level or application level, which is the clustering error. It results after applying the clustering algorithm on the approximated Gram matrix.

This metric can be represented in many ways. One common way is by using the average distance between each point to its assigned exemplar. To get benefit of that, we fix the number of exemplars, since a smaller average distance for the same number of exemplars indicates higher clustering accuracy. For the affinity propagation, the number of exemplars is controlled by changing a parameter; the self similarity value $s(i, i)$; where small values of self similarities reduce small number of exemplars and vice versa. For k-means clustering, number of exemplars is a pre-defined parameter to the algorithm. We use the clustering error in one of our experiments. It gives us the same information we can get using the low level metrics.

4.2 Data set

We use The USCensus1990 data set [19], it is a discretized version of the USCensus1990raw data set. The USCensus1990raw data set contains a one percent sample of the Public Use Microdata Samples person records (PUMS) drawn from the full 1990 census sample.

There are 68 categorical attributes. Some of the attributes are listed below. Many of the less useful attributes in the original data set have been dropped, the few continuous variables have been discretized and the few discrete variables that have a large number of possible values have been collapsed to have fewer possible values.

More specifically the USCensus1990 data set was obtained from the USCensus1990raw data set by the following sequence of operations:

- Randomization: The order of the cases in the original USCensus1990raw data set were randomly permuted.

- Selection of attributes: Some of the 68 attributes included in the data set are given in Table 4.1. In the USCensus1990 data set a single letter prefix have been added to the original name. The letter 'i' is added to indicate that the original attribute values are used and 'd' to indicate that original attribute values for each case have been mapped to new values. The mapping is described in Table 4.2.

Old Variable	New Variable
Age	dAge
Ancstry1	dAncstry1
Ancstry2	dAncstry2
Avail	iAvail
Citizen	iCitizen
Class	iClass
Depart	dDepart
Disabl1	iDisabl1
Disabl2	iDisabl2
English	iEnglish
Feb55	iFeb55
Fertil	iFertil
Hispanic	dHispanic
Hour89	dHour89
Hours	dHours
Immigr	iImmigr
Income1	dIncome1
Income2	dIncome2
Income3	dIncome3
Industry	dIndustry
Lang1	iLang1
Marital	iMarital
Occup	dOccup
POB	dPOB
Sex	iSex

Table 4.1: Some of the selected 68 attributes in the USCensus1990 data set.

- Mapping: In this step all of the old values for variables with prefix 'd' are mapped to new values. The mappings for the variables dAncstry1, dAncstry2, dHispanic, dIndustry, dOccup, dPOB were designed to correspond to a natural coarsening of the original values based on the information of the coding. The remaining variables are continuous valued variables and the mapping for these variables was chosen to make variables that were fairly uniformly distributed across the states (quantiles). The precise mappings are specified in T-SQL procedures. These procedures can be used directly in SQLServer to map the original values or translated to some other language.

Variable	Procedure
dAge	discAge
dAncstry1	discAncstry1
dAncstry2	discAncstry2
dHispanic	discHispanic
dHour89	discHour89
dHours	discHours
dIncome1	discIncome1
dIncome2	discIncome2to8
dIncome3	discIncome2to8
dIncome4	discIncome2to8
dIncome5	discIncome2to8
dIncome6	discIncome2to8
dIncome7	discIncome2to8
dIncome8	discIncome2to8
dIndustry	discIndustry
dOccup	discOccup
dPOB	discPOB
dPoverty	discPoverty
dPwgt1	discPwgt1
dRearning	discRearning
dRpincome	discRpincome
dTravtime	discTravtime
dWeek89	discWeek89
dYrsserv	discYrsserv

Table 4.2: Mapping some attributes of USCensus1990 data set into new discrete variables.

4.3 Setup

We setup an MPICH cluster, where five machines are interconnected through a gigabit Ethernet switch. Each machine has Intel(R) Core(TM)2 Duo CPU E6550A processor and a 2 GB of RAM. An Ubuntu Linux operating system is installed on each. A master folder is shared among all machines using the Network File System (NFS). NFS allows us to create a folder on one machine and have it synced on all other machines. This folder is used to store programs and data to be used by all machines.

Our implementation uses LSHKIT which is a C++ Locality Sensitive Hashing Library written by Dong [14], fast Hilbert curves without recursion written in C by Moore [43], and affinity propagation clustering algorithm code written by their authors Frey and Dueck [21].

We write one file code and store it along with the data set file on the shared folder. By parameterizing our code, each process will run on a separate data that is assigned to it by the code. If the number of processes is larger than the number of available nodes, each node will run more than one process. The number of processes is fixed in advance before the run time.

4.4 Results

In our first set of experiments, we use 4000 items from US census 1990 data set. We compare three methods: random projection LSH denoted by DKMA-RP in the figures, spectral hashing (DKMA-SH) and Hilbert curve. To unify the comparison, we compare the results using the same size of the approximated Gram matrix. To do so, we set the Hilbert curve window width as $n/2^k$, where n is data set size and k is the number of hash function bits in LSH. The approximated Gram matrix size using k for LSH is $n^2/2^k$, where 2^k is the number of buckets, assuming all buckets have the same size. At the same time, the approximated Gram matrix size using Hilbert curve is $n \times winWidth = n \times (n/2^k) = n^2/2^k$.

4.4.1 Results for Accuracy

Figure 4.2 shows the results using the 4000 data points. In Figures 4.2(a) and 4.2(b) the Frobenius norm is obtained using different values of k. As k increases, the accuracy decreases. Using Hilbert curve or DKMA-SH, and for $k < 6$, we maintain a reasonable accuracy that is more than 90% of the Frobenius norm of the full Gram matrix. If we use Johnson-Lindenstrauss lemma, we find that $\log_{1/p} 4000 < 6$, and hence $p < 0.25$ the low probability for two far points to collide. Moreover, from the figures, Hilbert curve and DKMA-SH out perform DKMA-RP.

In Figures 4.2(c) and 4.2(d), we show the clustering error using the affinity propagation on the approximated Gram matrix for $k = 5$. The clustering error is represented by the average distance between each point to its exemplar. We compare this value against the number of exemplars, small value at the same number of exemplars means higher accuracy. The Figures show close performance for both Hilbert curve and DKMA-SH, and both out perform DKMA-RP.

4.4.2 Results for Memory Consumption

We have shown previously that our proposed DKMA algorithm reduces the quadratic complexity for computing the Gram matrix, down to sub-quadratic. In Figure 4.3, the approximated Gram matrix size is drawn in log scale, it is the same for all methods at the same value of k, given that $W_{H-curve} = n/2^{k_{LSH}}$. At $k = 5$, we use a space for the approximated Gram matrix that is 3.57% of the full Gram matrix size, and at $k = 6$ we use only 1.78%, both $k = 5$ and $k = 6$ give a Frobenius norm more than 90% of the full Gram matrix using spectral hashing or Hilbert curve.

The advantage of using LSH over Hilbert curve is the distributed property of LSH. Each process handles a bucket independently, and in this case, our method using LSH is scalable. If we have enough nodes, such that, each node will handle one bucket at most, and if we ignore the communication and the synchronization overheads, we will get a speedup up to 2^k, where k is the number of hash bits, and 2^k is the total number of buckets.

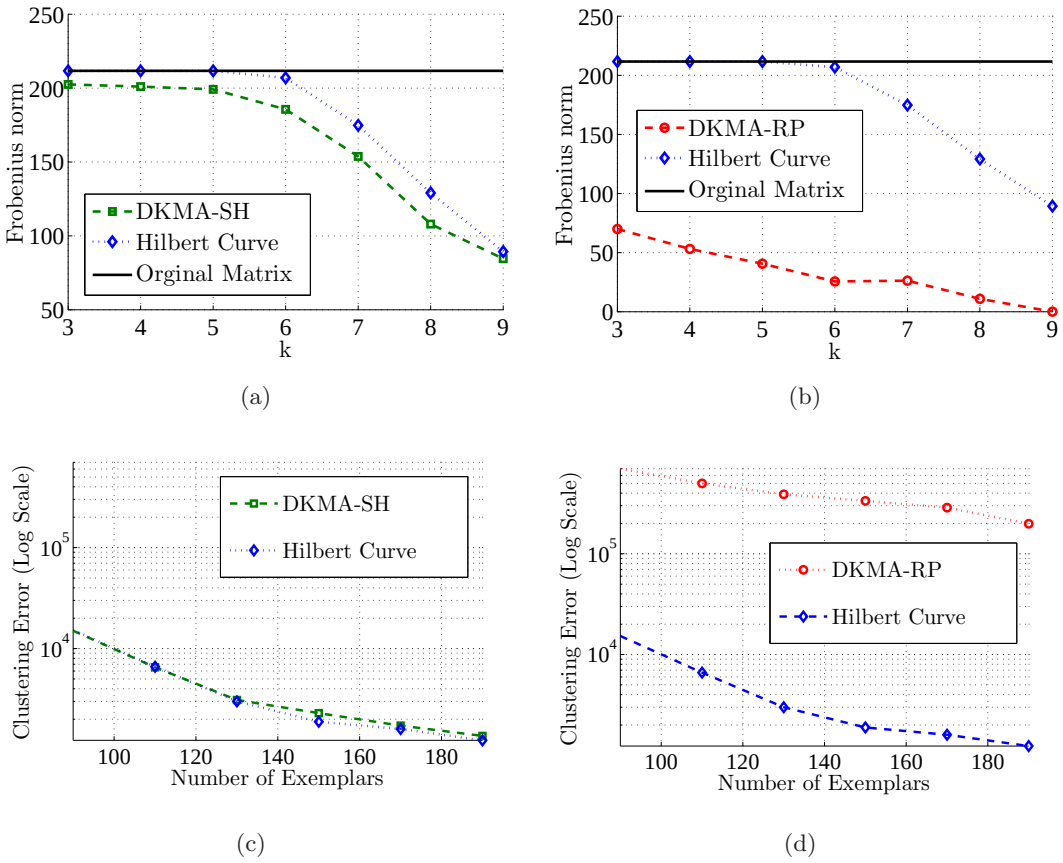
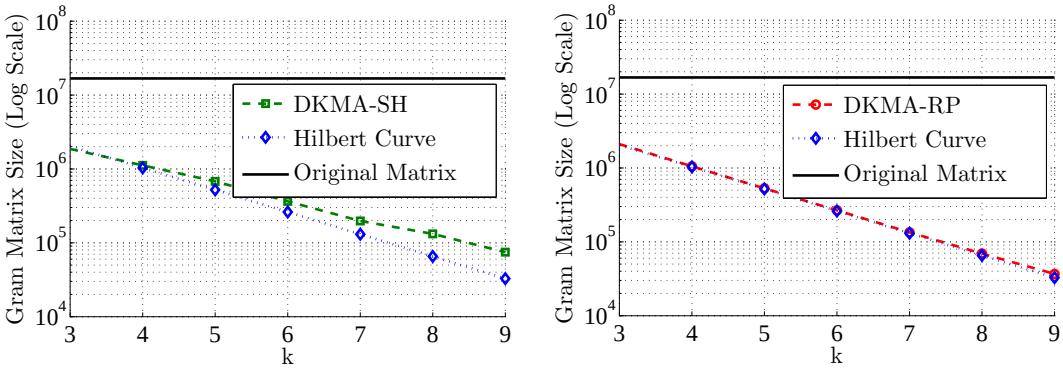


Figure 4.2: Results for accuracy on DKMA Algorithm, it achieves high accuracy using spectral hashing and Hilbert curve.

Results for accuracy using 4000 items from the USCensus data set. We apply different techniques; spectral hashing, random projection hashing and Hilbert space filling curve. Figure (a) and (b) show the Frobenius norm using different values of k, where number of buckets is 2^k for LSH and window width is $n/2^k$ for Hilbert curve (k is number of hash bits and n is the data set size). Figure (c) and (d) show the clustering error using $k = 5$, this is represented by the average distance between each point to its assigned exemplar, this metric is drawn against number of exemplars obtained by changing the self similarity values $s(i, i)$ in the affinity propagation.



(a) (b)

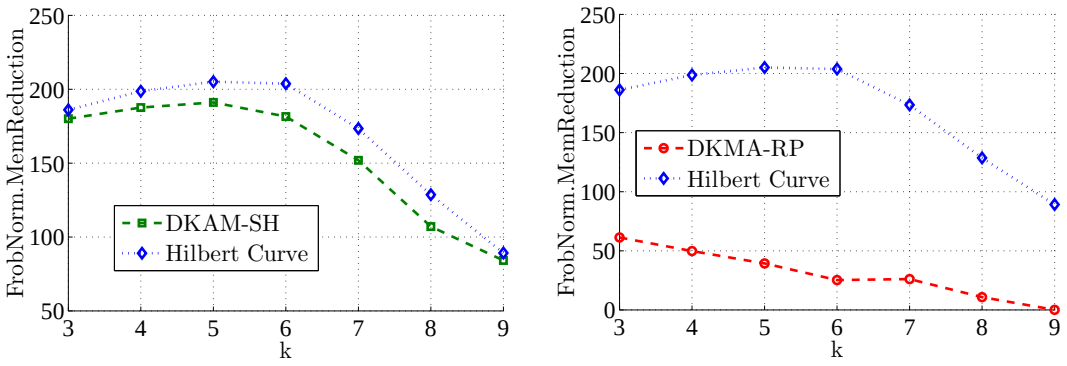
Figure 4.3: Results for memory consumption.
The approximated Gram matrix size is a small fraction of the original matrix.

4.4.3 Results for *FrobNorm.MemReduction* Product

Figure 4.4 shows the *FrobNorm.MemReduction* product. This metric gives the optimal value of k, which is 6 in this case. No need to compute the Frobenius norm for the full Gram matrix here. From Figure 4.2(a), the maximum memory reduction with a high accuracy happens at $k = 6$.

4.4.4 Results for Large-Scale Data

In Figure 4.5, the results of processing one million data items from US census data 1990, this is done using MPI on cluster as been explained before. In the naive way, 1 million $\times$ 1 million = 1 trillion of space and computations are required for the Gram matrix. Hilbert curve fails to process, as it needs the whole approximated Gram matrix stored in place for the clustering processing. In LSH, we have independent sub-problems to be processed, which are the buckets. Since the naive way and the method using Hilbert curve are not scaled to this size, we show only the results for our method using locality sensitive hashing. From the figure, spectral hashing out performs random projection, and both give us significant reduction im memory requirements..



(a) (b)

Figure 4.4: Results for $FrobNorm.MemReduction$ product on DKMA algorithm, it gives us the optimal value of k which is 6 in this case.

$FrobNorm.MemReduction$ product, which is the Frobenius norm multiplied by the memory reduction normalized, this metric takes in account both Frobenius norm and matrix size, so it is Gram size independent, and can be used to find the optimal value of k.

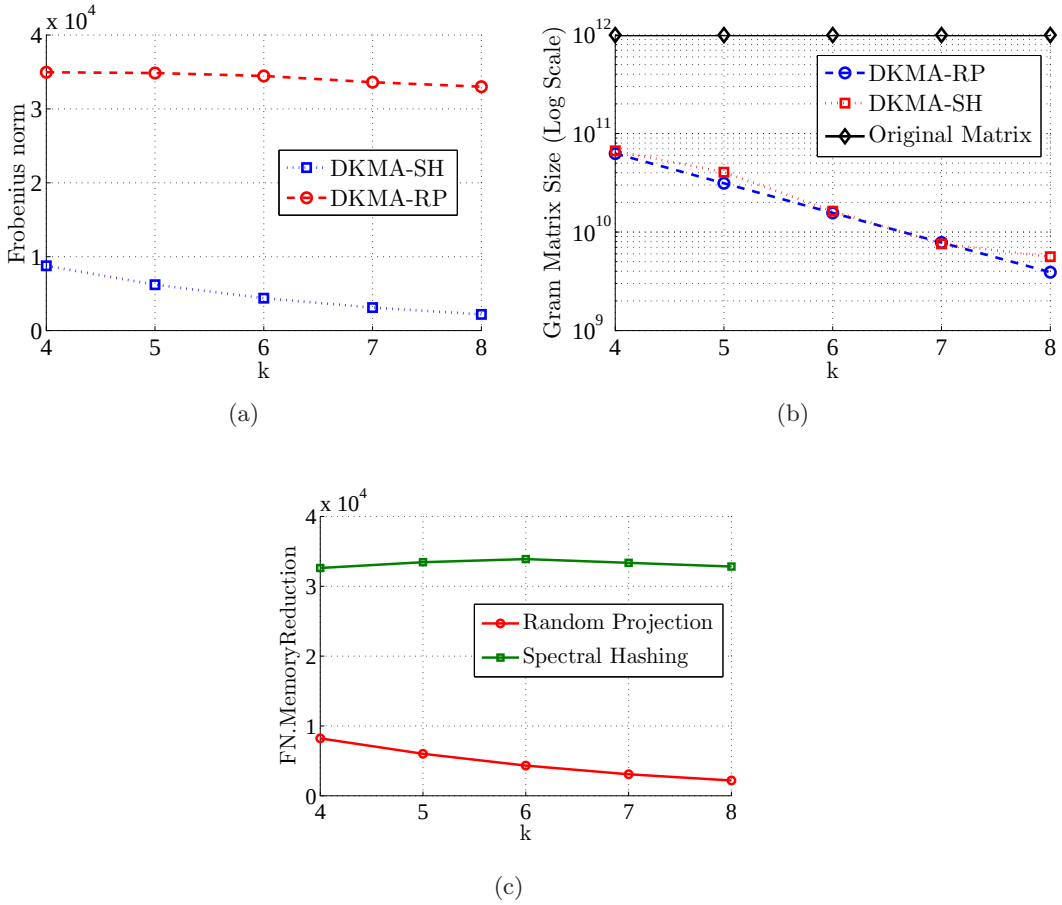


Figure 4.5: Results for large-scale data set on DKMA algorithm, it scales well and it achieves substantial memory saving and high accuracy.

One million items from the USCensus data are processed, by applying DKMA-SH and DKMA-RP. Hilbert curve failed, as it cannot handle such size. Figure (a) shows the Frobenius norm using different values of k. Figure (b) shows the approximated Gram matrix size obtained which is the same for both techniques at the same value of k. Figure (c) is the $FrobNorm.MemReduction$ product.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Kernel-based machine learning algorithms require $O(n^2)$ to compute and store the Gram matrix, where n is the number of input points. The Gram matrix is the similarity matrix that uses a kernel function to compute the similarity between each pair of points. This complexity is infeasible and unscalable when dealing with massive data sets. Some previous works pay attention to the computation and/or space complexity without working on the scalability problem itself. We proposed a method to approximate the Gram matrix which reduces this quadratic complexity down to sub-quadratic. Our proposed method is distributed as well that solves the scalability problem.

Based on the fact that the kernel functions are radial basis functions, that its value depends on the Euclidean distance between the input points, approximation can be done by finding the close points and then computing the kernel function between them. Finding the close points in the space can be done using space filling curves and locality sensitive hashing. Our proposed method uses LSH, which enables to scale to massive data sets. We succeeded in scaling **millions** of points using spectral hashing with an accuracy very close to the full Gram matrix. Spectral hashing out performs random projection hashing and have a close performance to the method using Hilbert space filling curve.

We have implemented our method on a cluster using the Message Passing Interface (MPI)

framework. Our implementation has minimum synchronization and communication overheads among processes, because we divide the data into buckets which are processed independently. Moreover, our implementation is independent of the size of the hardware resources. Hence, our algorithm can get as much speedup as the available hardware resources. This speedup will be limited to the total number of buckets, which is 2^k, where k is the total number of hash functions. Our accuracy is dependent on the value of k, as k increases accuracy decreases, but we showed that we can fix k at a maximum value of $\log n$, where n is the data set size, and in the same time, we maintain an accuracy that is more than 90%.

5.2 Future Work

We have explored and succeeded in processing million of data items, we used data from the US census of 1990 data set, this data set is relatively a low dimensional (< 100). One of the next steps is to explore the high dimensional challenge. In particular, the bio-informatics data or the text and image documents, where we may have thousands of dimensions. Moreover, exploring extremely large scale data sets is one of the next steps as well, this scale could range from billions to trillions of data items.

Random projection works well on high-dimensional data, but spectral hashing may have limitation on such high-dimensional and very-large scale data sets. This can be solved by using a hierarchical spectral hashing, where the main data sets are segmented into smaller data sets and so on, each time we have to hash using the dimensions that have the most variance, which can be done using the principle component analysis as a pre-processing step before the spectral hashing step.

Another future work can be on improving the fault tolerance of the proposed algorithm. As well as a comparison between two implementation, MPI and MapReduce.

Bibliography

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.

[2] K. Andreev and H. Räcke. Balanced graph partitioning. In *Proc. of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '04, pages 120–124, New York, NY, USA, 2004. ACM.

[3] N. O. Andrews and E. A. Fox. Recent developments in document clustering. 2007.

[4] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. Optics: ordering points to identify the clustering structure. *SIGMOD Rec.*, 28:49–60, June 1999.

[5] M. Belkin and P. Niyogi. Towards a theoretical foundation for laplacian-based manifold methods. *Journal of Computer and System Sciences*, 74(8):1289 – 1308, 2008.

[6] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21 –29, June 1997.

[7] C. Carpineto, S. Osiński, G. Romano, and D. Weiss. A survey of web clustering engines. *ACM Comput. Surv.*, 41:17:1–17:38, July 2009.

[8] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, New York, NY, USA, 2002. ACM.

[9] C. Cortes and V. Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, Sept. 1995.

[10] S. Dasgupta and A. Gupta. An elementary proof of a theorem of johnson and lindenstrauss. *Random Structures and Algorithms*, 22(1):60–65, 2003.

[11] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. of the twentieth annual symposium on Computational geometry*, pages 253–262, New York, NY, USA, 2004. ACM.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, Jan. 2008.

[13] D. Decoste and B. Schölkopf. Training invariant support vector machines. *Mach. Learn.*, 46(1-3):161–190, Mar. 2002.

[14] W. Dong. Lshkit: A c++ locality sensitive hashing library. `http://lshkit.sourceforge.net/`, 2009.

[15] P. Drineas and M. W. Mahoney. On the nystrom method for approximating a gram matrix for improved kernel-based learning. *J. Mach. Learn. Res.*, 6:2153–2175, December 2005.

[16] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[17] M. Ester, H.-p. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. *Computer*, 1996(6):226231.

[18] C. Fowlkes, S. Belongie, F. Chung, and J. Malik. Spectral grouping using the nystrom method. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(2):214 –225, feb. 2004.

[19] A. Frank and A. Asuncion. UCI machine learning repository. `http://archive.ics.uci.edu/ml`, 2010.

[20] P. Frankl and H. Maehara. The johnson-lindenstrauss lemma and the sphericity of some graphs. *J. Comb. Theory Ser. A*, 44:355–362, June 1987.

[21] B. Frey and D. Dueck. Affinity propagation. `http://www.psi.toronto.edu/index.php?q=affinity%20propagation`, 2007.

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

[23] F. Gao and M. Hefeeda. Distributed approximate spectral clustering for large-scale datasets. Master's thesis, Simon Fraser University, December 2011.

[24] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. of the 25th International Conference on Very Large Data Bases*, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[25] G. H. Golub and C. F. V. Loan. *Matrix Computations*. JHU Press, 1996.

[26] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *Solid-State Circuits, IEEE Journal of*, 31(9):1277 –1284, sep 1996.

[27] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.

[28] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, 1999.

[29] J. A. Hartigan and M. A. Wong. A K-means clustering algorithm. *Applied Statistics*, 28:100–108, 1979.

[30] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, New York, 2009.

[31] J. He, W. Liu, and S.-F. Chang. Scalable similarity search with optimized kernel hashing. In *Proc. of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '10, pages 1129–1138, New York, NY, USA, 2010. ACM.

[32] M. Hussein and W. Abd-Almageed. Efficient band approximation of gram matrices for large scale kernel methods on gpus. In *Proc. of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–10, New York, NY, USA, 2009. ACM.

[33] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, New York, NY, USA, 1998. ACM.

[34] L. Jing, M. Ng, and J. Huang. An entropy weighting k-means algorithm for subspace clustering of high-dimensional sparse data. *Knowledge and Data Engineering, IEEE Transactions on*, 19(8):1026 –1041, aug. 2007.

[35] W. Johnson and J. Lindenstauss. Extensions of lipschitz mappings into a hilbert space. *Contemporary Mathematics*, 26:189–206, 1984.

[36] H.-P. Kriegel, P. Krger, J. Sander, and A. Zimek. Density-based clustering. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(3):231–240, 2011.

[37] H.-P. Kriegel, P. Kröger, and A. Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Trans. Knowl. Discov. Data*, 3:1:1–1:58, March 2009.

[38] N. D. Lawrence, M. Seeger, and R. Herbrich. Fast sparse gaussian process methods: The informative vector machine. In *Advances in Neural Information Processing Systems 15*, pages 609–616. MIT Press, 2003.

[39] Z. Li, X. Xie, L. Zhang, and W.-Y. Ma. Searching one billion web images by content: Challenges and opportunities. In *MCAM'07*, pages 33–36, 2007.

[40] Q. Lv, M. Charikar, and K. Li. Image similarity search with compact data structures. In *Proc. of the thirteenth ACM international conference on Information and knowledge management*, pages 208–217, New York, NY, USA, 2004. ACM.

[41] S. C. Madeira and A. L. Oliveira. Biclustering algorithms for biological data analysis: A survey. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 1:24–45, January 2004.

[42] S. Mika, G. Ratsch, J. Weston, B. Scholkopf, and K. Mullers. Fisher discriminant analysis with kernels. In *Neural Networks for Signal Processing IX, 1999. Proceedings of the 1999 IEEE Signal Processing Society Workshop*, pages 41 –48, aug 1999.

[43] D. Moore. Fast hilbert curve generation, sorting, and range queries. `http://web.archive.org/web/20041028171141/http://www.caam.rice.edu/~dougm/twiddle/Hilbert/`.

[44] K.-R. Muller, S. Mika, G. Ratsch, K. Tsuda, and B. Scholkopf. An introduction to kernel-based learning algorithms. *Neural Networks, IEEE Transactions on*, 12(2):181 –201, Mar. 2001.

[45] L. Parsons, E. Haque, and H. Liu. Subspace clustering for high dimensional data: a review. *SIGKDD Explor. Newsl.*, 6:90–105, June 2004.

[46] J. C. Platt. Fastmap, metricmap, and landmark mds are all nystrom algorithms. In *Proc. of 10th International Workshop on Artificial Intelligence and Statistics*, pages 261–268, 2005.

[47] D. A. Spielman. Spectral graph theory and its applications. *Foundations of Computer Science, IEEE Annual Symposium on*, 0:29–38, 2007.

[48] G. W. Stewart. On the early history of the singular value decomposition. *SIAM Review*, 35(4):pp. 551–566, 1993.

[49] J. Wang, S. Kumar, and S.-F. Chang. Semi-supervised hashing for scalable image retrieval. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3424 –3431, june 2010.

[50] Z. Wang, W. Dong, W. Josephson, Q. Lv, M. Charikar, and K. Li. Sizing sketches: a rank-based analysis for similarity search. *SIGMETRICS Perform. Eval. Rev.*, 35(1):157–168, 2007.

[51] Y. Weiss, A. B. Torralba, and R. Fergus. Spectral hashing. In *Neural Information Processing Systems*, pages 1753–1760, 2008.

[52] C. Williams and M. Seeger. Using the nystrm method to speed up kernel machines. In *Advances in Neural Information Processing Systems 13*, pages 682–688. MIT Press, 2001.

[53] G. Yihong and X. Wei. *Machine Learning for Multimedia Content Analysis*. Springer, 2007.

[54] A. Zien, G. Rtsch, S. Mika, B. Schlkopf, T. Lengauer, and K.-R. Mller. Engineering support vector machine kernels that recognize translation initiation sites. *Bioinformatics*, 16(9):799–807, 2000.