OPEN SOURCE DISTRIBUTED SPECTRAL CLUSTERING

by

ANKITA PRASHANT JOSHI

(Under the Direction of Shannon Quinn)

ABSTRACT

In this thesis, we propose an approximation of the Spectral Clustering algorithm to cluster high-dimensional large datasets. Spectral Clustering is limited in its application as it suffers from a scalability problem in both memory use and computational time when the size of the data is very large. In this work, we modify the spectral clustering algorithm by developing an open source framework for approximate spectral clustering on a distributed level using Apache Spark. To perform clustering on large data sets we implement a distributed design of the algorithm and we construct a sparse affinity matrix using approximate nearest neighbors. The design reduces the affinity matrixs construction time and storage requirements without significantly affecting the accuracy of clustering. Experimental results on synthetic and real datasets demonstrate the effectiveness of our framework with respect to both time consumption and accuracy.

INDEX WORDS: Spectral Clustering, Distributed computing, Apache Spark, Approximate Nearest Neighbors.

OPEN SOURCE DISTRIBUTED SPECTRAL CLUSTERING

by

ANKITA PRASHANT JOSHI

B.E., University of Pune, India, 2013

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2017

© 2017

ANKITA PRASHANT JOSHI

All Rights Reserved

OPEN SOURCE DISTRIBUTED SPECTRAL CLUSTERING

by

ANKITA PRASHANT JOSHI

Major Professor:	Shannon Quinn
Committee:	Walter D.Potter
	Khaled Rasheed

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
December 2017

DEDICATION

To Paa, Maa, Dada and Payal.

ACKNOWLEDGMENTS

I would like to extend my heartfelt thanks and gratitude to my major advisor Dr. Shannon Quinn, for the patience, guidance, encouragement and advice he has provided in the past two years. His insights in research have been an immense help in my graduate study which has in turn inspired and motivated me to pursue higher studies. The completion of this thesis would not have been possible without him.

I would like to thank Dr. Walter D. Potter and Dr. Khaled Rasheed for being on my thesis committee and reviewing this thesis.

I would like to thank my friends for their companionship in the past two years, especially my classmate, mentor and friend, BahaaEddin AlAila. They created an amiable environment in the lab and it was a great pleasure to work with them.

Last but certainly not the least, I would like to thank my parents, my elder brother and my sister-in-law, for their unreserved love and endless support. I would not have made it without them and my deepest gratitude to my family is beyond words.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS v

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER

1 INTRODUCTION 1

2 BACKGROUND & RELATED WORK 2

3 PROBLEM STATEMENT & THESIS CONTRIBUTIONS 15

4 PROPOSED OPEN SOURCE DISTRIBUTED SPECTRAL CLUSTERING ALGORITHM 17

5 EXPERIMENTS 26

6 CONCLUSION 38

REFERENCES 40

LIST OF TABLES

4.1 Pseudocode for computing the similarity matrix. 20
4.2 Pseudocode for broadcasting the diagonal matrix. 21
4.3 Pseudocode for computing the Laplacian matrix. 22
4.4 Pseudocode for computing the SVD. 23
4.5 Pseudocode for clustering the matrix Y using K-Means clustering. 23
5.1 OSDSC experiment setup. 27
5.2 Processing time comparison from 1024 to 16384 points (in seconds). 28
5.3 Processing time comparison from 1024 to 16384 points (in seconds). 34
5.4 Processing time comparison from 1024 to 16384 points (in seconds). 35
5.5 Processing time comparison from 1024 to 16384 points (in seconds). 35

LIST OF FIGURES

2.1 Spark Architecture 3

5.1 Processing time comparison from 1024 to 16384 points (in seconds) for OSDSC, DASC [27], PSC [28] and SC [27]. It can be seen that OSDSC algorithm does consistently better than the other algorithms. 29

5.2 Processing time for datasets ranging from 1024 to 1048576 points (in seconds). All the experiments were conducted on a cluster of 5 nodes on the AWS EMR framework. 30

5.3 Memory consumption of the OSDSC algorithm for points ranging from 1024 to 1048576 points in MB-sec. 31

5.4 Processing time (in seconds) for varying dimensions of the datasets. 32

5.5 Processing time (in seconds) for dimensions ranging from 2_4 to 2_{12}. The number of datapoints in all cases was a constant of 65536. This experiment was conducted on a cluster of 5 workers and 1 master. The number of nearest neighbors for the affinity matrix construction was 10. 33

5.6 Processing time (in seconds) for different cluster configurations. The number of datapoints in all cases was a constant of 65536. This experiment was conducted on clusters with workers ranging from 4 up to 8. The number of nearest neighbors for the affinity matrix construction was 10. 34

5.7 Visualization of the clustering results for the MNIST dataset. This dataset has 10 classes for 0-9 digits. The algorithm was conducted using a cluster having 5 nodes. The number of neighbors selected for the sparse affinity matrix construction is 10. 36

5.8 Visualization of the clustering results for the CIFAR10 dataset. This dataset has 10 classes. The algorithm was conducted using a cluster having 5 nodes. The number of neighbors selected for the sparse affinity matrix construction is 10. 37

CHAPTER 1

INTRODUCTION

In this chapter, we provide a brief introduction to clustering algorithms. The organization of the thesis is given at the end of this chapter.

1.1 Introduction

The goal of clustering algorithms is to partition a set of data into groups of similar objects [1]. Among the clustering algorithms, Spectral Clustering, a class of methods based on eigen-decomposition of matrices, often gives a superior performance as compared to other algorithms [2]. Because of its effectiveness in finding clusters [3], spectral clustering has been widely used in several areas such as computer vision, information retrieval and pattern recognition. (e.g. [4], [5], [2], [6]). Unfortunately, spectral clustering suffers a computational bottleneck with the increase in the number of data points to cluster [7][8].

The proposed algorithm is open source and the implementation can be found here.

1.2 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we introduce the Apache Spark framework, and provide a brief review of clustering algorithms. We also discuss the challenges in clustering large datasets in this chapter. In Chapter 3, we briefly talk about the Approximate Nearest Neighbor library ANNOY, and how it is used in our implementation of the distributed framework. We also discuss the proposed framework in this chapter. In Chapter 4 and Chapter 5, we discuss the experimental set up and experimental results using synthetic and real-world datasets. We conclude the thesis in Chapter 6.

CHAPTER 2

BACKGROUND & RELATED WORK

In this section, we discuss the Apache Spark framework and provide an overview of clustering algorithms. After that we summarize the previous work related to our work.

2.1 Apache Spark Framework

Apache Spark is a powerful open source distributed processing engine originally developed at UC Berkeley in 2009 [9]. The main feature of Apache Spark is the memory cluster computing. This feature increases the processing speed of an application. Spark is designed to cover a vast range of workloads such as batch applications, iterative algorithms, interactive queries and streaming.

2.1.1 Spark Architecture

Apache Spark follows a master/slave architecture with two main daemons and a cluster manager.

1. Master Daemon (Master/Driver process)
2. Slave Daemon (Worker process)

Resilient Distributed Datasets (RDDs) are the lowest level data structures in Spark.RDDs and their transformations describe how to compute operations. Directed Acyclic Graphs (DAGs) represent transformations and dependencies between RDDs. A Spark Application at a high level consists of a SparkContext and user code which interacts with it creating RDDs and performing a series of transformations to achieve the result. These transformations of

RDDs are then translated into a DAG and submitted to a Scheduler to be executed on a set of worker nodes.

A Spark cluster consists of a single master and any number of workers. Let us discuss the Spark architecture [17] as seen in Figure 2.1.

1. The Driver Program runs on the master node of the Spark cluster and schedules the job execution and negotiates with the cluster manager. It translates the RDDs into the execution graphs and stores their metadata and partitions.

2. Workers are the distributed agents responsible for the execution of tasks. Workers usually run for the entire lifetime of a Spark application. Workers perform the data processing tasks and interact with the storage system. They read data from and write data to external sources.

3. The Cluster Manager is responsible for acquiring resources on the Spark cluster and allocating them to a Spark job. There are three different types of cluster managers a Spark Application can leverage for the allocation and deallocation of various physical resources such as memory for client Spark jobs, CPU memory, etc. Hadoop YARN, Apache Mesos or the simple standalone Spark cluster manager, either of them can be launched for a Spark application to run. Choosing a cluster manager for any Spark application depends on the goals of the application because all cluster managers provide different set of scheduling capabilities.

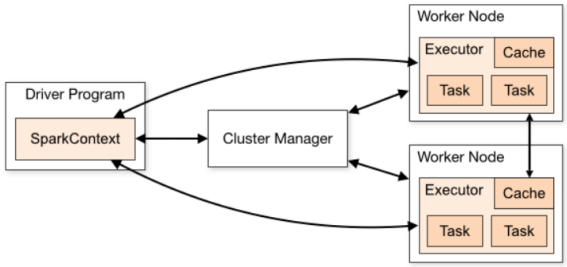

Figure. 2.1: Apache Spark Architecture [17] consists of a Driver program, a Cluster Manager and Worker Nodes.

2.1.2 Resilient Distributed Datasets [RDDs]

Resilient Distributed Datasets (RDDs) are a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. [10]. According to the scaladoc of org.apache.spark.rdd.RDD: *A Resilient Distributed Dataset (RDD), the basic abstraction in Spark, represents an immutable, partitioned collection of elements that can be operated on in parallel.* RDDs are immutable objects. They read only abstraction and cannot be changed once created. One RDD can be transformed into another RDD using transformations like join, map, filter, etc. The immutable nature of RDDs helps to make them consistent. RDDs are distributed, which means that they are present on multiple nodes in a cluster.

Apache Spark supports two types of operations:

1. Transformations

   Spark Transformation is a function that produces new RDD from the existing RDDs. It takes RDD as input and produces one or more RDD as output.

2. Actions

   Actions are Spark RDD operations that give non-RDD values. The values of action are stored to drivers or to the external storage system. It brings laziness of RDD into motion.

Spark makes use of lazy evaluation for computing RDDs. The data inside an RDD will not be transformed unless an action triggers the execution of the transformation. Spark maintains the record of which operation is being called through DAGs. Since transformations are lazy in nature, any operation can be executed at any time by calling an action on the RDD. RDDs are also cacheable i.e. they can hold the data in the desired persistent storage. RDDs in Spark process data in parallel. Spark RDDs have various types RDD [int], RDD [long], RDD [string].

2.1.3 Directed Acyclic Graphs [DAGs]

(Directed Acyclic Graph) DAG in Apache Spark is a set of Vertices and Edges, where vertices represent the RDDs and the edges represent the Operation to be applied on RDD. It contains a sequence of vertices such that every edge is directed from earlier to later in the sequence. On calling of Action, the created DAG is submitted to DAG Scheduler which further splits the graph into the stages of the task.

DAG in Apache Spark is an alternative to the MapReduce. It is a programming style used in distributed systems. In MapReduce, we just have two functions (map and reduce), while DAG has multiple levels that form a tree structure. Hence, DAG execution is faster than MapReduce because intermediate results are not written to disk.

2.2 Other parallel distributed programming models

Apache Hadoop is a software framework that supports data-intensive distributed application and was inspired by Googles MapReduce and Google File System (GFS). Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data in-parallel on large clusters in a fault-tolerant manner. One of the main limitations of MapReduce is that it persists the full dataset to HDFS after running each job. This is very expensive, because it incurs both three times (for replication) the size of the dataset in disk I/O and a similar amount of network I/O. Spark takes a more holistic view of a pipeline of operations. When the output of an operation needs to be fed into another operation, Spark passes the data directly without writing to persistent storage. The main innovation of Spark was to introduce an in-memory caching abstraction. This makes Spark ideal for workloads where multiple operations access the same input data. Users can instruct Spark to cache input data sets in memory, so they don't need to be read from disk for each operation. The primary advantage Spark has is that it can launch tasks much faster. MapReduce starts a new JVM for each task, which can take seconds with loading JARs, JITing, parsing configuration XML, etc. Spark keeps an executor JVM running on

each node, so launching a task simply takes in the single digits of milliseconds. Also, Hadoop provides batch processing, whereas Spark is a good fit for both batch processing and stream processing. Spark speeds up batch processing via in-memory computation and processing optimization. Spark is also easier to use than Hadoop as it comes with user friendly APIs for Scala, Java, Python and Spark SQL.

Message Passing Interface (MPI) [11] is an Application Programming Interface specification that allows processes to communicate with each other by sending and receiving messages. It is a standard for parallel programs running on computer clusters and supercomputers, where the cost of accessing the non-local memory is high. In order for parallel computing to work, the various computers need to be able to communicate with each other to pass messages back and forth. MPI Message Passing Interface was created to facilitate this communication. MPI needs to be implemented by the hardware vendor, and any hardware that has an MPI implementation built into can be accessed by connected systems via the MPI protocol.

OpenMP (Open Multi-Processing) [12] is an API that supports multi-platform shared memory multiprocessing programming. Therefore, programming module entails that all threads have access to the same globally shared memory. Data can be shared or private. However, this model only runs efficiently in shared-memory multiprocessor platforms and its scalability is limited by memory architecture [12].

2.3 Overview of clustering algorithms

Clustering is an unsupervised learning problem which deals with finding a structure from a collection of unlabeled data. Data clustering algorithms can be hierarchical or partitional. In the following sections, we introduce some important classes.

2.3.1 Hierarchical algorithms

Hierarchical algorithms [13] find successive clusters using previously established clusters. Hierarchical algorithms can be agglomerative (bottom-up) or divisive (top-down). Agglomerative algorithms begin with each data point as a separate cluster and merge them in successively larger clusters. Divisive algorithms begin with the whole set in the same cluster and proceed to divide it into successively smaller clusters. Hierarchical clustering procedures have the advantages of an easily interpreted graphical representation and are robust to poor algorithm initialization and getting stuck in local minima [14]. However, they only consider local neighbors when merging/splitting clusters and cannot incorporate a priori knowledge about the global shape or size of clusters [14]. The complexity of agglomerative clustering is $O(n^2 log(n))$ [15], which makes it too slow for clustering large data sets. Divisive clustering with an exhaustive search is $O(2^n)$ which is even worse [15].

2.3.2 Partitional algorithms

Partitional algorithms generally determine all cluster at once, and can also be used as divisive algorithms in the hierarchical clustering. Partitional clustering procedures do incorporate global characteristic knowledge through appropriate distance measures in the objective function. Another advantage of partitional clustering methods is that they are dynamic in the sense that pixels can be assigned different clusters to reduce the objective function [14].

2.3.3 K-means Clustering Algorithm

K-means [16] follows a simple and easy way to classify a given data set into a certain number of clusters (assume k clusters fixed a priori). This algorithm aims at minimizing an objective function, in this case a squared error function. The objective function is:

$$\sum_{j=1}^{k} \sum_{i=1}^{N} ||x_i^{(j)} - c_j||^2 \tag{2.1}$$

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 K-means algorithm is composed of the following steps:

1. Place k points into the space represented by the objects that are being clustered. These points represent initial group centroids.
2. Assign each point to the group that has the closest centroid.
3. Recalculate the positions of the k centroids when all the points have been assigned.
4. Repeat step 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.

K-Means algorithm suffers from many weaknesses as follows:

1. The number of clusters, k, must be determined before hand.
2. Initial seeds may have a strong impact on final results. Different initial condition may produce different result of cluster. The algorithm may be trapped in local optima. For every different run of the algorithm on the same dataset, if different set of initial centers are chosen, it may lead to different partitions or clustering results on different runs of the algorithm. Therefore, it is very hard to get consistent and reliable clustering results.
3. K-Means does not work well with non globular clusters as K-Means will tend to pick spherical clusters.

2.3.4 Spectral Clustering Algorithm

This section explains the spectral clustering algorithm and describes the resource bottlenecks inherent in this approach. We consider the most commonly used normalized spectral clustering [17].

Spectral clustering algorithm constructs a pairwise similarity matrix S by using a similarity function, computes the Laplacian matrix L and then computes the 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 [17]. An example of the similarity function is the Gaussian kernel:

$$S_{ij} = exp - \frac{||x_i - x_j||^2}{2\sigma^2} \tag{2.2}$$

where σ is the scaling parameter.

Given a set of points in $\{X_i\}_{i=1}^{N}$ and d is the dimensionality, the Spectral clustering algorithm is composed of the following steps:

1. Form the similarity matrix, $S \in R_{NxN}$ defined by $S = exp - \frac{||x_i - x_j||^2}{2\sigma^2}$ if $i \neq j$ and $S_{ij} = 0$.

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 $L = D^{-1/2}AD^{-1/2}$.

3. Find $V_1, V_2, ..., V_k$ the top k eigenvectors of L, and form the matrix $X = [V_1, V_2, ..., V_k] \in R^{Nxk}$ by stacking the eigenvectors in columns.

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

5. Treat each row in Y as a point as a point in R^k and cluster them into clusters using K-means [16].

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. Let A be an NxN matrix. If λ is an eigenvalue of A then a corresponding nonzero vector v is called an eigenvector of A corresponding to λ if $(AI)v = 0$. The advantages of Spectral Clustering can be summarized as follows:s

1. Spectral clustering is a mathematically well-formed clustering algorithm and performs well for non-Gaussian clusters [17].

2. As opposed to K-means clustering, Spectral clustering does not make assumptions on the form of the cluster. This property comes from the mapping of the original space to eigenspace.

3. Spectral clustering does not intrinsically suffer from the problem of local optima [17]. It performs significantly better than the linkage algorithm [18] even when the clusters are not well separated.

2.4 Performance Metrics for Clustering

An important task after computing clusters for a given dataset is to measure the quality of the clusters obtained. An evaluation measure is needed to compare how well different data clustering algorithms perform on a set of data. To measure the quality of clustering results there are two types of validity scores, external indices and internal indices [19]. An external index is a measure of agreement between two partitions where the first partition is the a priori known clustering structure, and the second results from the clustering procedure [20]. Internal indices are used to measure the goodness of a clustering structure without external information [21]. For external indices, we evaluate the results of a clustering algorithm based on a known cluster structure of a data set (or cluster labels). For internal indices, we evaluate the results using quantities and features inherent in the data set. The optimal number of clusters is usually determined based on an internal validity index. For external indices, we need to have the ground truth for a given data set, i.e., we need to have the set of correct labels to the given dataset. Rand Index score is one example of an external validity score. Silhouette Score is an example of an internal validity score. These two metrics have been explained in detail below.

2.4.1 Silhouette Score

The silhouette score is a measure of how similar a point is to its own cluster (cohesion) compared to other clusters (separation) [22]. The silhouette score ranges from -1 to +1 where a high value indicates that the object is well matched to its own cluster and poorly

matched to neighboring clusters [22]]. If most objects have a high value, then the clustering configuration is appropriate and if many points have a low or negative value, then the clustering configuration may have too many or too few clusters [22]. The silhouette score can be calculated as:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} \tag{2.3}$$

where $s(i)$ is the silhouette score for each data i, $a(i)$ be the average dissimilarity of i with all other data within the same cluster. $b(i)$ is the lowest average dissimilarity of i to any other cluster, of which i is not a member. The average $s(i)$ of a cluster will give us a measure of how tightly coupled all the data in the cluster are.

2.4.2 Rand Index score

The Rand Index or Rand measure [23] is a measure of accuracy between two data clustering results. Given a set of n elements $S = \{o_1, o_2, ..., o_n\}$ and two partitions of S to compare, $X = \{X_1, X_2, ..., X_r\}$, a partition of X into r subsets, and $Y = \{Y_1, Y_2, ..., Y_s\}$, a partition of Y into s subsets, define the following [23],

- a, number of pairs of elements in S, that are in the same subset in X and in the same subset in Y.
- b, number of pairs of elements in S, that are in different subsets in X and in different subsets in Y.
- c, number of pairs of elements in S, that are in the same subset in X and in different subsets in Y.
- d, number of pairs of elements in S, that are in different subsets in X and in the same subset in Y.

The Rand Index, R, is given as [23],

$$R = \frac{a + b}{a + b + c + d} \tag{2.4}$$

The Rand Index has a value between 0 and 1, with 0 indicating that the two data clusterings do not agree on any pair of points and 1 indicating that the data clusterings are exactly the same [23].

2.5 Challenges in Clustering large datasets

There are some fundamental issues when using spectral clustering algorithm on large datasets. We identify the following stages at which the spectral clustering algorithm experiences crucial bottlenecks.

1. Similarity Matrix Computation: The full similarity matrix having N2 dimensions cannot be stored in main memory. Therefore, techniques which reduce the storage capacity will be very helpful. Instead of computing the full affinity matrix, we compute a sparse affinity matrix using ANNOY [8], a library for finding approximate nearest neighbors, which uses random projections to split the search space and finds the approximate nearest neighbors. A sparse representation effectively handles the memory bottleneck.

2. The Singular Value Decomposition step: This step is computationally expensive and is cubic to the number of points in the dataset. To tackle this issue, we use the distributed implementation of Singular Value Decomposition provided by Apache Spark.

3. Clustering step: K-Means is also expensive which adds to being another bottleneck in the process. We make use of the distributed implementation of the K-Means clustering algorithm provided by the Apache Spark framework.

In this work, we try to address each of these bottlenecks in order to improve the performance of Spectral clustering and enable it to work on large-scale multidimensional datasets.

2.6 Related Work

Clustering large scale data is an actively researched area [24][25][26]. In [26], Kulis and Grauman generalize locality-sensitive hashing to accommodate kernel functions and use this method for large-scale scalable image search. BIRCH [24] is also another example, in which the method pre-clusters large datasets by incrementally grouping the data as tightly as possible based on the similarity of the attributes of the data points. It constructs as many representatives of the data as the available memory can contain. In [17] they develop a distributed spectral clustering algorithm for large scale datasets using the MapReduce framework of Apache Hadoop. However, Hadoop MapReduce does not leverage the memory of the Hadoop cluster to the maximum whereas Apache Spark executes jobs 10 to 100 times faster than the Hadoop MapReduce. Also, they have not made their code openly available, so their results cannot be reproduced easily.

Apache Mahout also has an open source implementation of spectral clustering; however, they assume that the user will provide the affinity matrix of the data points as an input, which greatly reduces the computational complexity of the affinity matrix construction step. Several methods are available for sparsifying the affinity matrix [3].

Another popular approach to speed up spectral clustering is rank reduction which refers to a large class of methods in numerical algebra in which a matrix is replaced with a low rank approximation. The affinity matrix of spectral clustering is a natural target for rank reduction. Nystrom approximation has been used by [7], which samples columns of the affinity matrix and approximates the full matrix by using correlations between the sampled columns and remaining columns. A variety of sampling procedures can be used. A drawback of these methods is they do not incorporate any information of the affinity matrix while choosing the columns to sample and they do not provide performance guarantees. [27] proposes a method that does not use eigenvectors, but they assume the availability of the similarity matrix.

In this work, we aim at developing a distributed spectral clustering package which provides users an open source environment to apply spectral clustering on large-scale multidimensional datasets. We achieve this by approximation of the dense similarity matrix by using ANNOY, a library for approximate nearest neighbors.

CHAPTER 3

PROBLEM STATEMENT & THESIS CONTRIBUTIONS

Our goal is to present a distributed open source framework for spectral clustering to provide the user the freedom to provide the required input parameters and be able to spin up clusters in order to use spectral clustering on large scale and high dimensional data.

3.1 Problem Statement

As discussed in [28] and in Section 2.5 of Chapter 2, the computational resources required by Spectral Clustering in processing large scale data are often prohibitively high. The computation of the similarity matrix takes time and space to compute and store. If faced with large datasets, this algorithm will simply fail to run, either due to memory insufficiency or unacceptable long processing time. We propose to use an approximated similarity matrix instead of the dense similarity matrix construction to reduce the computational complexity of this algorithm. Furthermore, we provide a distributed implementation of the Spectral Clustering algorithm with the help of the opensource Apache Spark framework.

3.2 Thesis Contributions

Our contributions can be summarized as follows:

1. We present a novel, distributed, open source for the Spectral Clustering algorithm.
2. The design of the algorithm reduces the computational time complexity and space complexity of the algorithm by approximating the similarity matrix construction and also by using the Apache Spark framework.

3. The experimental results show that the proposed method also maintains the clustering accuracy.

CHAPTER 4

PROPOSED OPEN SOURCE DISTRIBUTED SPECTRAL CLUSTERING ALGORITHM

It is essential to conduct spectral clustering in a distributed environment to alleviate both memory and computational bottlenecks. In this section, we discuss these challenges and then propose our distributed algorithm. In the following section, we will first review the concept of random projections for finding nearest neighbors in a given dataset. Then we will present the ANNOY library used in our framework. We will then describe our proposed framework and implementation details.

4.1 Nearest Neighbors using Random Projections

Nearest neighbor search is an important part of many machine learning algorithms. Nearest Neighbors is a type of instance-based learning, it does not construct a general model, but simply stores instances of the training data. For every new point, the class with a majority vote of the nearest neighbors is assigned to the new point. For example, for a given query point, if neighborhood is set to k = 5 data points, then the label of the query point is the majority label of the nearest 5 points. The optimal choice of k is highly data dependent. Under some circumstances, a variation of nearest neighbors called weighted nearest neighbors is used, where nearest neighbors are weighted by how far away from the query point they are.

It is obvious that it is computationally expensive to find the k nearest neighbors when the dataset is very large. Also, the performance depends on the number of dimensions of the data.

Random projections [29] is a dimension reduction technique and is used to approximate the cosine distances between the vectors. Random projections are used to construct compressed versions of high dimensional data by computing linear combinations of dimensions weighted by random coefficients. The idea is to choose a random hyperplane at the outset and use the hyperplane to hash input vectors. The reason random projections work is because the dimensions and distribution of random projection matrices are controlled so as to preserve the pairwise distances between any two samples of the dataset. Also, the directions of the projections are independent of the data. Thus random projection is a suitable approximation technique for distance based method [30].

Given two vectors x and y, the cosine of the angle between them is the dot product $x.y$ divided by the $L2$ norm of x and y, for instance, the Euclidean distances from the origin. Given and input vector v and a hyperplane defined by, r, we let, $h(v) = sng(v.r) = \pm 1$. Each possible choice of r defines a single function. Let H be the set of all such functions and D be the uniform distribution. For two vectors u and v we have:

$$Pr[h(u) = 1 - \theta(u, v)/\pi] \tag{4.1}$$

Where $\theta(u, v)$ is the angle between u and v. Given functions in this family, there will be bits in the result for every input data point. These 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.

4.2 ANNOY (Approximate Nearest Neighbors, Oh Yeah)

ANNOY library was built by Erik Bernhardsson during his work at Spotify to find music recommendations. Annoy is a C++ library with Python bindings, and also has a Spark Data Frame version, to search for points in space that are close to a given query point. Annoy uses random projections to split up the search space by building up a tree, and find approximate nearest neighbors.

At every intermediate node in the tree, a random hyperplane is chosen, which divides the space into two subspaces. This hyperplane is chosen by sampling two points from the subset and taking the hyperplane equidistant from them. We do this times so that we get a forest of trees. has to be tuned to your need, by looking at what trade-off you have between precision and performance.

4.3 Open Source Distributed Spectral Clustering(OSDSC) Algorithm and Implementation

The proposed algorithm in the distributed environment approximates the similarity matrix using the ANNOY library and speeds up the spectral clustering algorithm. Because of the distributed setting, the memory footprint is also reduced. We present the design of the proposed OSDSC algorithm by implementing it on the open source, distributed computing environment, Apache Spark.

The algorithm implementation on Apache Spark is explained in the following steps. Each step has a pseudo code explaining each step of the algorithm in detail.

The user inputs the dataset in the form of an RDD of datapoints. As already seen, an RDD, the fundamental data structure of Spark, will store the data by partitioning it. The partitioned collection of points can be operated on in parallel.

1. Building the similarity matrix.

   For each data point, we compute k nearest neighbors using the ANNOY library. ANNOY library returns k neighbors for each point in the dataset. We only compute the similarity between the given point and the k neighbors returned. This step will give us an RDD which holds the similarity of n points with k other points. Each point is stored by Apache Spark as an insance of IndexedRow. The IndexedRow in Spark will be able to hold a collection of SparseVectors. In Spark, "Row" is a generic row object with an ordered collection of fields that can be accessed by an index, in which case it will

be an IndexedRow. An IndexedRowMatrix can be created from an RDD[IndexedRow] instance. The base class of local vectors is Vector, and Spark provides two implementations: DenseVector and SparseVector. Since, the affinities computed will be sparse, each point with just k other points, we use the SparseVector in order to store the affinities computed. A SparseVector will only store the values, and the locations in the vector at which the value is present, eliminating the zeros in the vector. This greatly reduces the storage requirements of the affinity matrix for large datasets.

The similarity function used is:

$$S_{ij} = exp - \frac{||x_i - x_j||^2}{2\sigma^2} \tag{4.2}$$

where σ is a scaling parameter to control how rapidly the similarity S_{ij} reduces with the distance between X_i and X_j. Table 4.1 shows the pseudo code for this step.

Table. 4.1: Pseudocode for computing the similarity matrix.

Pseudocode 1:	Computing the similarity/affinity matrix, A, using ANNOY library
/*Input:	RDD of data points,number of dimensions of data points, d, number of neighbors, k.
Output:	RDD of points with affinities computed with each of the k neighbors of the point. */
1	For the given input, train an ANNOY model and fit the given dataset. The result will be a trained annoyModel.
2	Pass the parameter for the number of neighbors required to this trained annoyModel and obtain k neighbors for each point in the data set.
3	Compute the affinities of each point with its neighbors by using the equation in (3.1).
4	The affinities computed in step 3, are stored using a sparse Vector, the final results a distributed sparse affinity matrix in Apache Spark.

2. Broadcasting the diagonal matrix.

The diagonal matrix is computed using the affinity matrix computed in step 1. The diagonal matrix is actually an RDD consisting of one value for each point in the affinity matrix. Each value in this RDD will be calculated by performing a transformation on

the RDD of the affinity matrix. The diagonal matrix value for each point is merely the sum of the affinity matrix columns. In order to optimize the implementation of the Laplacian matrix, we also precompute the inverse square root value of the diagonal matrix, which will be useful in Step 3. Since we are working on RDDs, this task will be done in parallel on all worker nodes and a new RDD for the diagonal matrix will be computed.

$$affinity.map(v => 1/Math.sqrt(v.vector.toArray.sum) \tag{4.3}$$

The result of the above operation will result in a single value for each point in the affinity matrix. This resulting RDD obtained, the diagonal matrix, will be of size equal to the number of datapoints. It will be broadcasted to all the nodes in the cluster. This is termed as broadcasting. Each node in the cluster will need access to this diagonal matrix for computing the Laplacian matrix step 3. When broadcasted, the master of the cluster, will collect all the partitions of the RDD, and then distribute it as a Vector to each node in the cluster. It should be noted that if we have a million points, then assuming we are using float point precision, we will need only 4MB storage space on each node to store the distributed matrix. It is safe to assume that the diagonal matrix will fit on each machine. Table 4.2 shows the pseudo code for this step.

Table. 4.2: Pseudocode for broadcasting the diagonal matrix.

Pseudocode 2:	Computing the Diagonal matrix: D
/*Input:	Affinity matrix, A, computed in Step 1.
Output:	Inverse square root of Diagonal Matrix, $D_i^{-1/2}$ */
1	For each row in the affinity matrix, a. Compute the sum. b. Take the square root of this sum value. c. Invert the value.
2	For each point value obtained in (1) is broadcasted (distributed) to each node in the cluster.

3. Building the Laplacian Matrix

The Laplacian matrix will be computed using the Affinity and Diagonal matrices computed in step 2 and step 3 respectively. As seen in step 1 and step 2, the Affinity Matrix is of the instance RDD[IndexedRow], whereas the Diagonal Matrix is a broadcasted Vector. The computation of the Laplacian matrix will be triggered with a map transformation on the RDD of the Affinity Matrix. The laplacian is computed as:

$$L = D^{-1/2} A D^{-1/2} \tag{4.4}$$

The broadcasted $D^{-1/2}$ is collected at every node in the cluster as an array. For each IndexedRow in affinity matrix, which consists of an index, values and the locations at which the values are present, the values are multiplied (a simple dot product) with the respective diagonal element present at the location given by the SparseVector. In this way we skip the computation of the zero entries. Table 4.3 shows the pseudo code for this step.

Table. 4.3: Pseudocode for computing the Laplacian matrix.

Pseudocode 3:	Computing the Laplacian matrix: W
/*Input:	Affinity matrix A, computed in step 1, Diagonal matrix D, computed in step 2
Output:	Laplacian matrix W. */
1	Using the affinity matrix and diagonal matrix, compute the Laplacian matrix using the equation: $L = D^{-1/2} A D^{-1/2}$.
2	The Laplacian matrix is stored using an IndexedRowMatrix in Apache Spark. This is basically a row distributed data structure, which means the different rows of the computed matrix, will be stored on different nodes in the cluster.

4. Eigen-decomposition

   In this step, we compute the eigen vectors of the Laplacian matrix. We use the distributed computeSVD implementation from the MLLib package of Apache Spark. Table 4.4 shows the pseudo code for this step.

5. K-Means clustering

Table. 4.4: Pseudocode for computing the SVD.

Pseudocode 4: /*Input: Output:	Computing the SVD of the input matrix. Laplacian Matrix W, computed in step 3, Number of eigenvectors to compute: k Eigen vectors, U, of matrix W */
1	We use the computeSVD function available on distributed indexed row matrices in Apache Spark, to obtain the matrix U, which consists of the eigenvectors of W.
2	Find $V_1, V_2, ..., V_k$, the first K_i eigenvectors of L_i and form the matrix $X_i = [V_1, V_2, ..., V_k] \in R^{N_i x K_i}$ by stacking the eigen vectors in columns.
3	Form the matrix Y by renormalizing each of Xs rows to have unit length, $Y_{ij} = \frac{X_{ij}}{\sqrt{\sum_j X_{ij}^2}}$

We use the points obtained in matrix Y, from step 4. We use the distributed K-Means implementation from the MLLib package of Apache Spark. Table 4.5 shows the pseudo code for this step.

Table. 4.5: Pseudocode for clustering the matrix Y using K-Means clustering.

Pseudocode 5: /*Input: Output:	Cluster the points in matrix Y, using K-Means. Matrix Y, computed in step 4, Number of clusters to compute, k cluster label: C for each point in the dataset. */
1	Treat each row in Y as a point in R^k, cluster them into K_i clusters using the K-Means clustering algorithm.
2	Train the input data (Y).
3	This will create a K-Means Model, upon calling the fit() method, we will receive the output as an RDD in the form of (point, class), which will basically mean for each point, we will get the associated cluster, class, the point belongs to.

4.4 Time Complexity Analysis of OSDSC

Let us consider each step of the algorithm described in section 3.3 and analyze the complexity at each step. Let us suppose we are clustering N points:

1. Computing the similarity matrix for all N points will be $O(N^2)$. However, we only compute the similarity for every point and its k approximate nearest neighbors. This

reduces the complexity of this step to $O(kN)$ where k is the number of approximate neighbors specified by the user.

2. For computing the inverse square root of the diagonal matrix D is $O(N)$ and the complexity of multiplying an NxN diagonal matrix with a NxN matrix is $O(N^2)$. 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 the similarity matrix. For a similarity matrix that contains U non-zero entries, the complexity of this step is $O(U)$.

3. The complexity of computing the Laplacian matrix, is the same as explained in Step 2, and will be $O(U)$.

4. The SVD step on Spark requires $O(Nkn)$, where ,k is the number of nearest neighbors and n is the number of eigen vectors needed.

5. Complexity of the K-Means step, for clustering points into K clusters is $O(Ndc)$ per iteration, where computing the distance is $O(d)$ and c is the number of clusters.

Therefore, the total complexity of the proposed algorithm is:

$$TimeComplexity = O(kn) + 2O(U) + O(Nkn) + O(Ndc) \tag{4.5}$$

4.5 Performance Metrics in Spark for OSDC

The distributed algorithm was implemented on the open source Apache Spark framework. Spark.mllib provides a suite of metrics for the purpose of evaluating the performance of these machine learning models. However, machine learning algorithms fall under broader types of machine learning applications like classification, regression, clustering, etc.

Currently there are no evaluation metrics available for clustering machine learning models on spark.mllib. This required us to implement our own evaluation metrics on Apache Spark. We provide these metrics as a part of the OSDC framework, but they are generic enough to be called by any clustering algorithm provided under spark.mllib.

For evaluating cluster results, we implemented two evaluation techniques namely, silhouette score [22] and rand index [23]. These techniques have been explained in detail in Section 2.4 of Chapter 2.

CHAPTER 5

EXPERIMENTS

In this chapter, we evaluate the proposed algorithm on a cluster. We explain the evaluation metrics, then discuss the setup of the experiments. After that we present the evaluation of the results.

5.1 Experimental Setup

This section discusses the experimental setup done and the dataset used for running the algorithm.

5.1.1 Cluster Setup

The experiments were conducted on a cluster using the Amazon Web Service (AWS) Elastic MapReduce (EMR). Each of the machine is equipped with high frequency Intel Xeon E5-2670 v2 (Ivy Bridge) processors and 8 vCPU, 15 GiB memory and 80 SSD GB storage. One machine in the cluster serves as the master (job tracker) and the others are slaves (task trackers). The cluster sizes are varied starting from 1 master-4 slaves up to 1 master-8 nodes. The software stack on each machine of the cluster and the property of the dataset are given in Table 4.1.

5.1.2 Dataset

The dataset used in the experiment is 64-dimensional 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 [1]. The size of the dataset ranges from 1024 to a million data points.

For real world dataset, the CIFAR10 dataset [31] was used. This dataset comprises of 50000 32x32 color images in 10 classes, with 5000 images per class. This dataset consists of each point (an image) with 3072 dimensions, each channel will comprise of 1024 values for a 32x32 image, and there are three channels namely, red, green and blue channels for each image.

We also used the MNSIT dataset [32]. The original black and white (bilevel) images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. The images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field. This gives us a dataset which consists of each point (an image) with 784 dimensions. The MNIST training set is comprised of 60000 training points too.

For both the above datasets we have the ground truth labels available and could therefore use the Rand Index score [23] in order to evaluate the clustering accuracy.

Table. 5.1: OSDSC experiment setup.

OS	Ubuntu Linux
Spark Version	1.6.2
Java Version	1.7.0_111 (Oracle Corporation)
Scala Version	2.10.5

We implement the proposed method OSDSC and compare the method against three other methods. One is the basic distributed Spectral Clustering (SC) [33] which is the existing implementation in Mahout, Distributed Approximate Spectral Clustering (DASC) [33], and the Parallel Spectral Clustering (PSC) in [34]. PSC is implemented by Chen [34] in C++ using PARPACK library [35] as underlying eigen value decomposition package and F2C to compile fortran code. The parallelization is based on MPI. All the experimental runs were done three times for each set up, and the final values of results given are an average of the three runs.

5.2 Results and Analysis for Synthetic Datasets

5.2.1 Scalability

Processing time is the running time of the program. Table 4.2 shows the processing time of our proposed algorithm OSDSC compared with DASC [33], PSC [34] and SC [33]. It can be observed that OSDSC considerably improves the running time. It should be noted that for PSC, one big drawback is that, in order to find out a group of nearest neighbors for a data point, it still needs to do a pair-wise computation. Figure 5.1 shows the clear improvement in the processing time that OSDC has over the other algorithms.

Table. 5.2: Processing time comparison from 1024 to 16384 points (in seconds).

	OSDSC	DASC	SC	PSC
1024 points	78	90.1	79.2	12.1
2048 points	86	187.5	256.3	42.2
4096 points	92	346.1	826.9	160.5
8192 points	104	703.8	3041.2	610.2
16384 points	120	1562.6	12328.8	2318.8

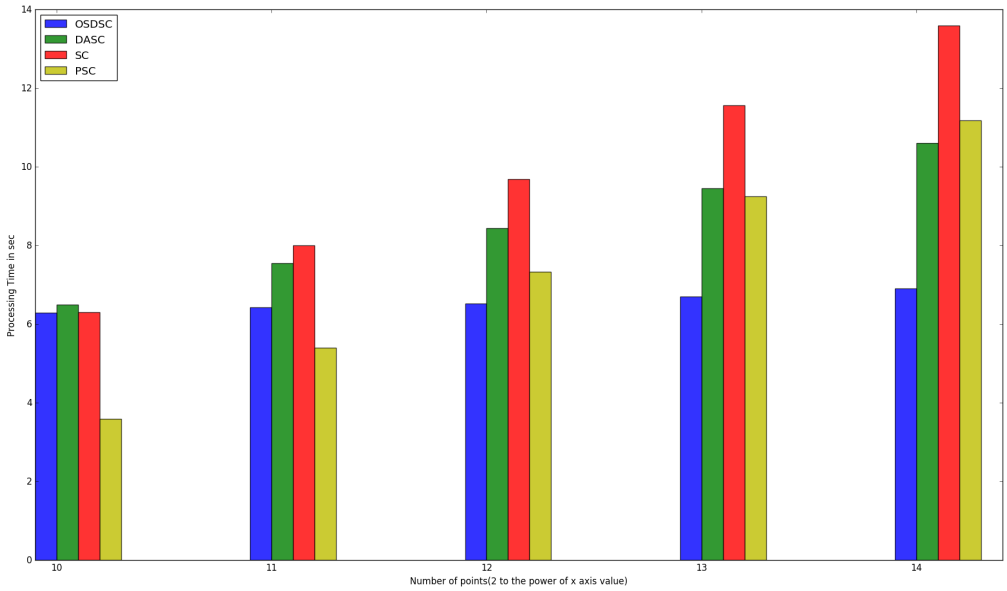

Figure. 5.1: Processing time comparison from 1024 to 16384 points (in seconds) for OSDSC, DASC [27], PSC [28] and SC [27]. It can be seen that OSDSC algorithm does consistently better than the other algorithms.

Figure 5.2 shows the processing time in seconds for points ranging from 1024 to 16384. We can see a linear growth in time. This shows that the distributed implementation of the OSDSC algorithm greatly improves the performance and can be used for clustering large scale datasets.

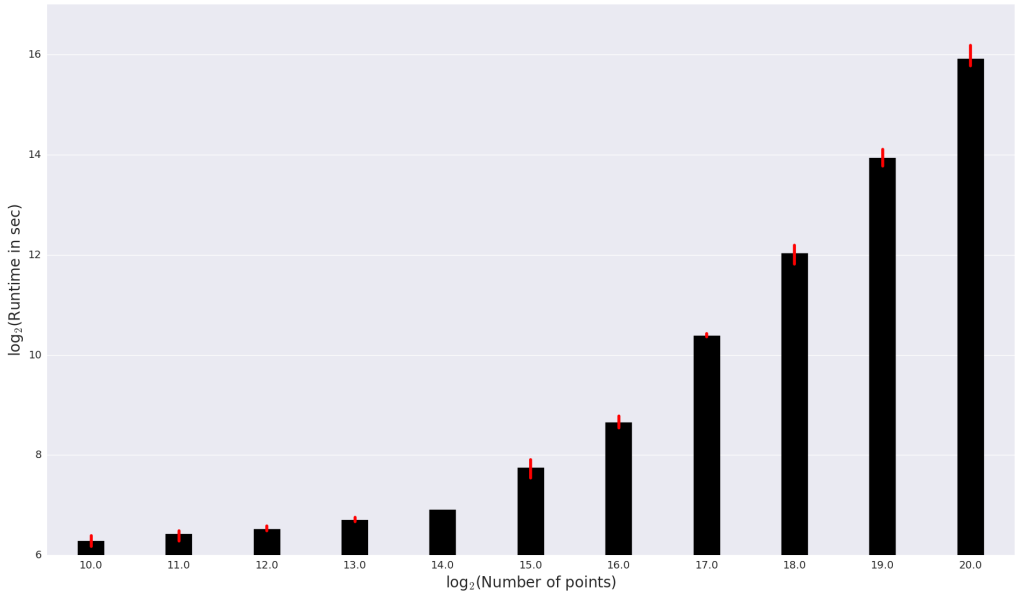

Figure. 5.2: Processing time for datasets ranging from 1024 to 1048576 points (in seconds). All the experiments were conducted on a cluster of 5 nodes on the AWS EMR framework.

Figure 5.3 shows the memory consumption of the OSDSC algorithm for points ranging from 1024 to 1048576 points in MB-sec. It can be seen that when the dataset grows, the memory consumption increases rapidly.

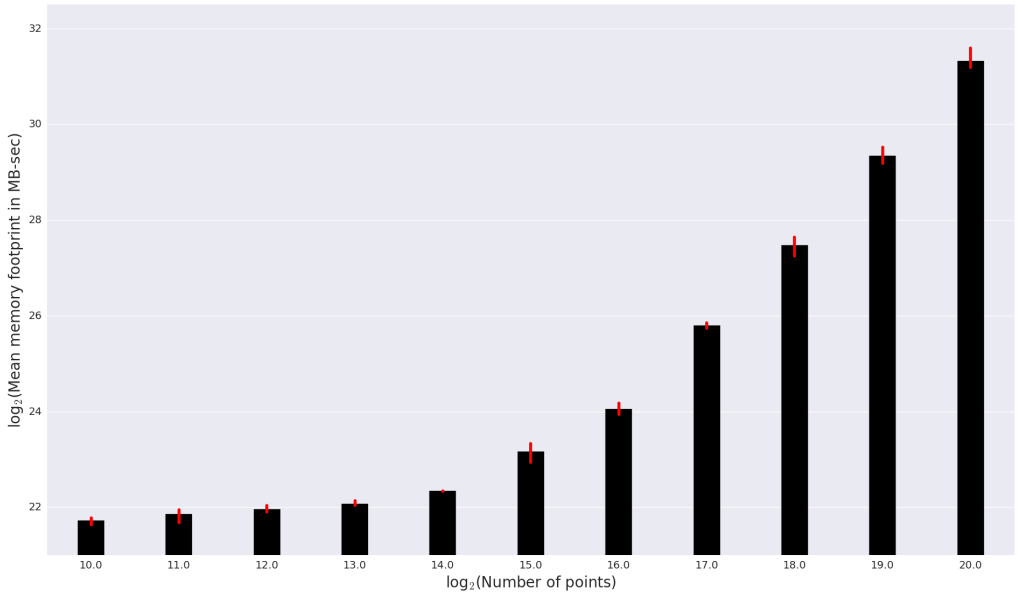

Figure. 5.3: Memory consumption of the OSDSC algorithm for points ranging from 1024 to 1048576 points in MB-sec.

Figure 5.4 shows the effect of changing the number of dimensions of the datasets on the processing time of the algorithm. The graph shows that the algorithm can scale well for increasing number of dimensions.

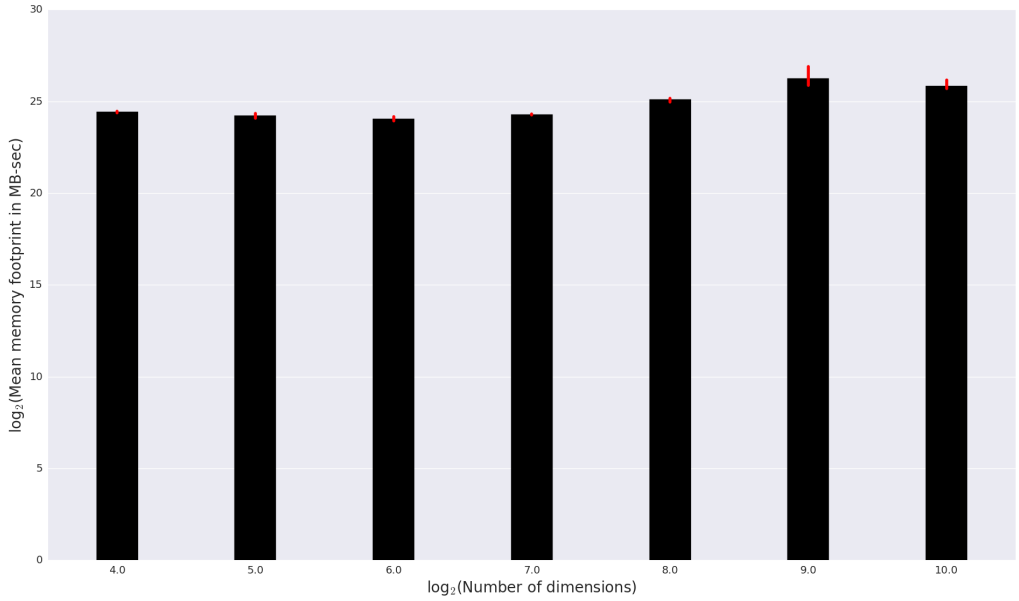

Figure. 5.4: Processing time (in seconds) for varying dimensions of the datasets.

Figure 5.5 shows the processing time in seconds for different dimensions of the dataset. In this case we used a dataset of 65536 points and with dimensions ranging from 16 to 4096.

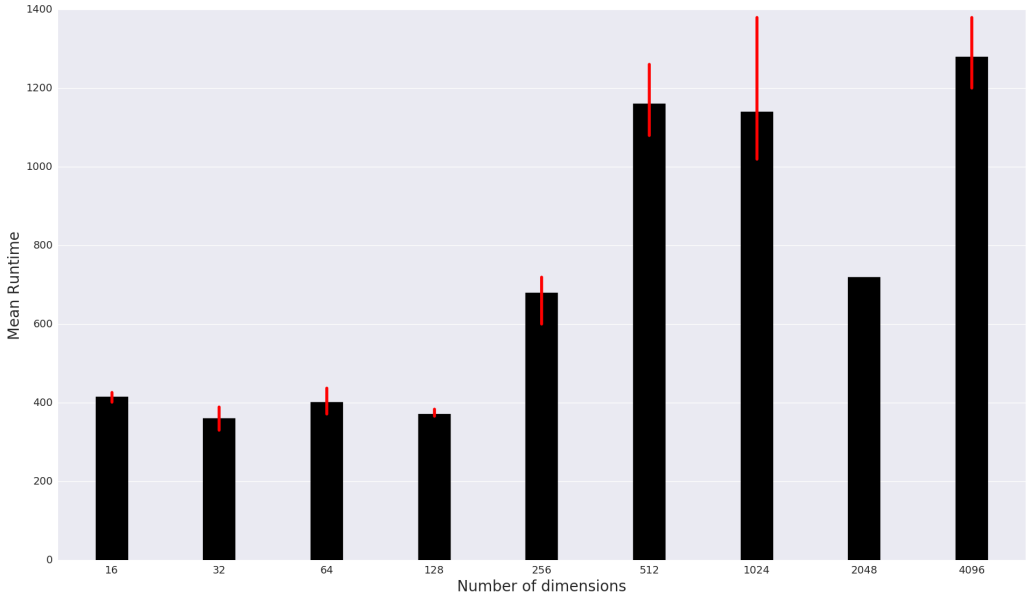

Figure. 5.5: Processing time (in seconds) for dimensions ranging from 2_4 to 2_{12}. The number of datapoints in all cases was a constant of 65536. This experiment was conducted on a cluster of 5 workers and 1 master. The number of nearest neighbors for the affinity matrix construction was 10.

Figure 5.6 shows the processing time (in seconds) for various cluster configurations. In order to understand if the algorithm scales well, the processing time should decrease with increase in the number of workers i.e. increase in the cluster size. However, we also need to take into account the overhead of a distributed network architecture, and this overhead can be seen in Figure 4.6. After the number of workers are increased beyond 6, there is a slight increase in the processing time.

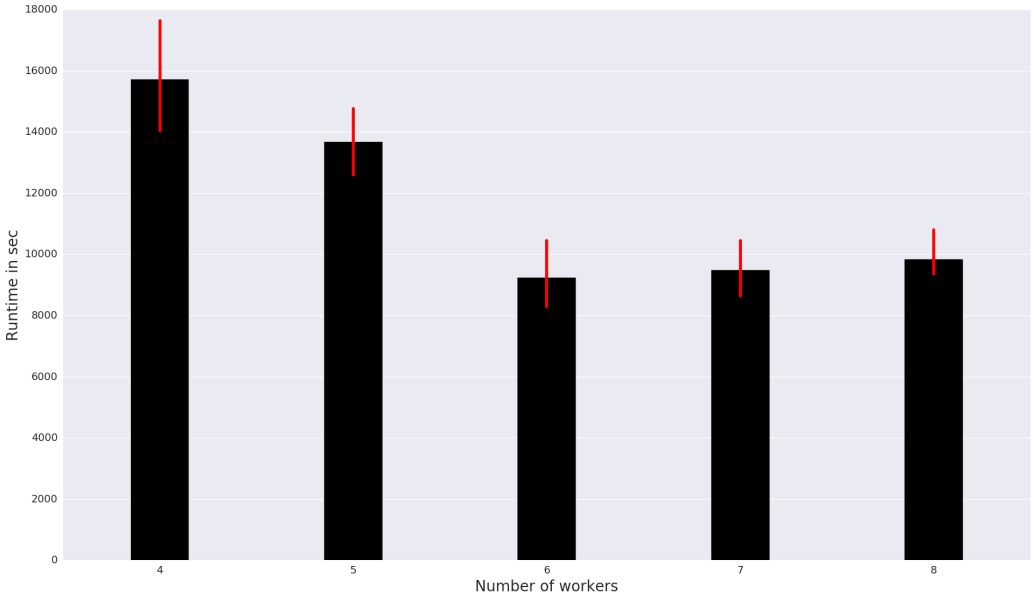

Figure. 5.6: Processing time (in seconds) for different cluster configurations. The number of datapoints in all cases was a constant of 65536. This experiment was conducted on clusters with workers ranging from 4 up to 8. The number of nearest neighbors for the affinity matrix construction was 10.

5.2.2 Accuracy

To show accuracy of the clustering results we created two datasets. The first dataset consisted of 5000 points with 100 dimensions and 5 classes. The second dataset consisted of 10000 points with 1000 dimensions and 10 classes. For the first dataset we chose number of neighbors 7, and for the second dataset we chose number of neighbors to be 10. We also generated the labels for these datasets. We then used our Rand Index score to measure the quality of our clustering results. Table 4.3 shows the Rand Index scores for each of these datasets.

Table. 5.3: Processing time comparison from 1024 to 16384 points (in seconds).

Points	RandIndex	Neighbors	Clusters	Dimensions
5000	0.86262	7	5	100
10000	0.941916	10	10	1000

5.3 Results and Analysis for Real World Datasets

5.3.1 Scalability

In order to check if our algorithm scales well on real world datasets we observe the processing time on the real-world datasets. The details of this experiment can be seen in Table 4.4.

Table. 5.4: Processing time comparison from 1024 to 16384 points (in seconds).

Number of CIFAR10 images	Time in seconds
5000	114
8000	132
50000	1500

5.3.2 Accuracy

Since we have the ground truth files for both the CIFAR10 and MNIST datasets, we used the Rand Index score [23] in order to compute the accuracy of our results. For this, we implemented the Rand Index score in Apache Spark and provide it along with the OSDSC Framework as a clustering utility. The rand index scores for both CIFAR10 and MNIST can be seen in Table 4.5.

Table. 5.5: Processing time comparison from 1024 to 16384 points (in seconds).

Dataset	Rand Index Score
CIFAR10	0.8088
MNIST	0.8126

Figures 5.7 and 5.8 show a visualization of the clustering results. We have used the T-SNE algorithm [36] in order to visualize the high dimensional data.

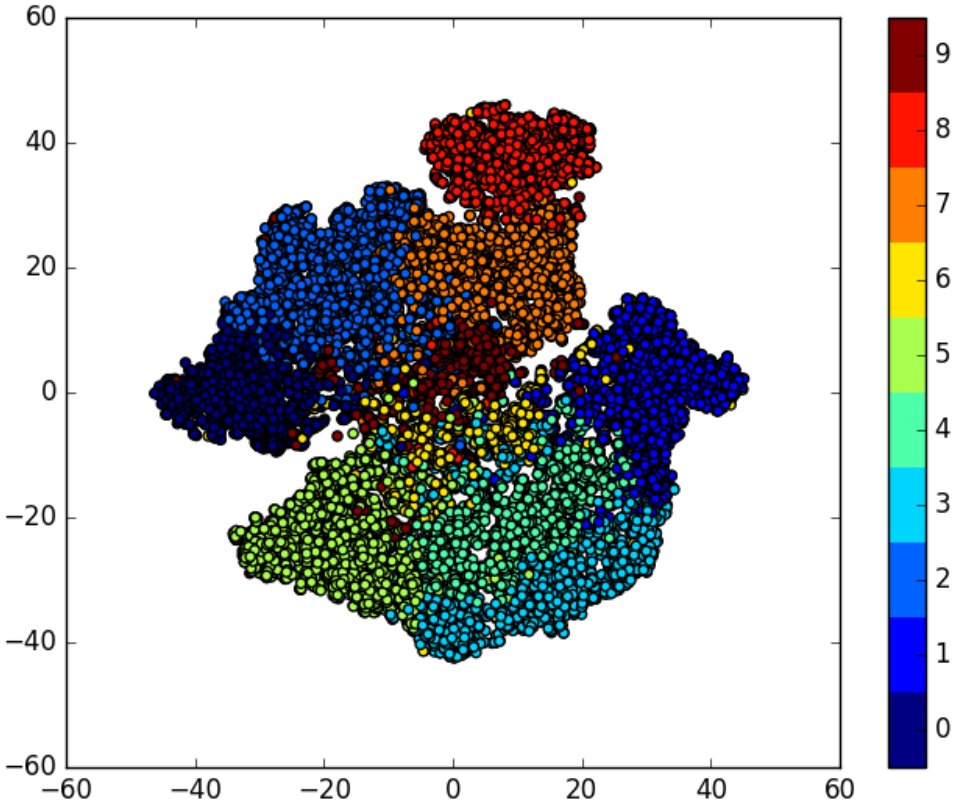

Figure. 5.7: Visualization of the clustering results for the MNIST dataset. This dataset has 10 classes for 0-9 digits. The algorithm was conducted using a cluster having 5 nodes. The number of neighbors selected for the sparse affinity matrix construction is 10.

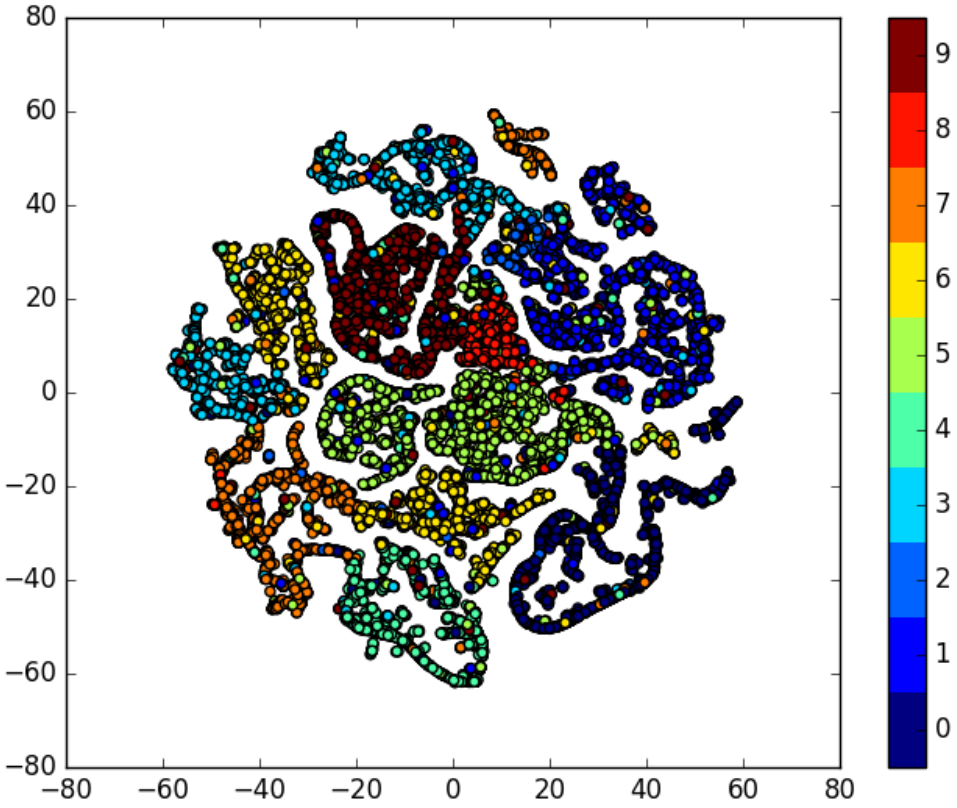

Figure. 5.8: Visualization of the clustering results for the CIFAR10 dataset. This dataset has 10 classes. The algorithm was conducted using a cluster having 5 nodes. The number of neighbors selected for the sparse affinity matrix construction is 10.

5.4 Summary

In this section, we evaluated the proposed OSDSC algorithm using synthetic and real world datasets and compared it against three closest algorithms in literature: SC [33], PSC [34] [33] and DASC [33]. The experimental results show that OSDSC can successfully retain clustering accuracy, greatly reduce computing time and significantly reduce the memory footprint too.

CHAPTER 6

CONCLUSION

6.1 Conclusion

We have proposed a novel framework for clustering large scale datasets using a distributed implementation of Spectral Clustering and evaluated its scalability. The proposed algorithm uses the approximate nearest neighbors library, ANNOY, to reduce the number of pairwise computations. This helps to speed up the the construction of the similarity matrix to facilitate the computation for large datasets.

The proposed framework can be used to significantly reduce the time and memory consumption, while at the same time retaining the clustering accuracy. We implemented the proposed algorithm using the open source Apache Spark framework, conducted extensive experiments on a cluster using the 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. The framework is generic and can be used for any type of dataset.

6.2 Future Work

The work in this thesis can be extended in multiple directions. As this is a framework developed to help users run Spectral Clustering on large scale datasets, more options can be provided to the user to make it more flexible in catering to the different needs of the user. For example, options can be provided for the types of kernel used, in the way SVD is computed, having a user defined gamma value and providing various library options to the user instead of ANNOY for finding approximate nearest neighbors. Instead of random

projections, we can use other methods to find the approximate nearest neighbors in order to construct the sparse affinity matrix. Locality-Sensitive Hashing (LSH) [37] and Spilltree [38] have been efficient in finding approximate nearest neighbors.

As finding the first eigenvectors of Laplacian matrix is computationally expensive, [27] propose a clustering approach without eigenvalue decomposition .Their method is related to but different from the standard spectral clustering. We can use random sampling as Nystrm method with an adaptive selection of samples. Several papers (e.g. [39], [40]) have developed advanced sampling techniques which can achieve comparable clustering results via a smaller subset of samples. The parallelization of these adaptive sampling approaches is also an interesting and challenging research issue.

REFERENCES

[1] Pavel Berkhin et al., "A survey of clustering data mining techniques.," *Grouping multidimensional data*, vol. 25, pp. 71, 2006.

[2] Jianbo Shi and Jitendra Malik, "Normalized cuts and image segmentation," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 22, no. 8, pp. 888–905, 2000.

[3] Ulrike Von Luxburg, "A tutorial on spectral clustering," *Statistics and computing*, vol. 17, no. 4, pp. 395–416, 2007.

[4] Inderjit S Dhillon, "Co-clustering documents and words using bipartite spectral graph partitioning," in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2001, pp. 269–274.

[5] Wei Xu, Xin Liu, and Yihong Gong, "Document clustering based on non-negative matrix factorization," in *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*. ACM, 2003, pp. 267–273.

[6] X Yu Stella and Jianbo Shi, "Multiclass spectral clustering," in *null*. IEEE, 2003, p. 313.

[7] Charless Fowlkes, Serge Belongie, Fan Chung, and Jitendra Malik, "Spectral grouping using the nystrom method," *IEEE transactions on pattern analysis and machine intelligence*, vol. 26, no. 2, pp. 214–225, 2004.

[8] Rong Liu and Hao Zhang, "Segmentation of 3d meshes through spectral clustering," in *Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on*. IEEE, 2004, pp. 298–305.

[9] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica, "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, no. 10-10, pp. 95, 2010.

[10] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[11] William Gropp, Ewing Lusk, and Rajeev Thakur, *Using MPI-2: Advanced features of the message-passing interface*, edition, 1999.

[12] Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann, "Programming distributed memory sytems using openmp," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 2007, pp. 1–8.

[13] Tony F Chan, "Hierarchical algorithms and architectures for parallel scientific computing," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 3, pp. 318–329, 1990.

[14] Hichem Frigui and Raghu Krishnapuram, "Clustering by competitive agglomeration," *Pattern recognition*, vol. 30, no. 7, pp. 1109–1119, 1997.

[15] Lior Rokach and Oded Maimon, "Clustering methods," in *Data mining and knowledge discovery handbook*, pp. 321–352. edition, 2005.

[16] John A Hartigan and Manchek A Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[17] Andrew Y Ng, Michael I Jordan, and Yair Weiss, "On spectral clustering: Analysis and an algorithm," in *Advances in neural information processing systems*, 2002, pp.

849–856.

[18] Onaiza Maqbool and Haroon Atique Babri, "The weighted combined algorithm: A linkage algorithm for software clustering," in *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on.* IEEE, 2004, pp. 15–24.

[19] Kaijun Wang, Baijie Wang, and Liuqing Peng, "Cvap: validation for cluster analyses," *Data Science Journal*, vol. 8, pp. 88–93, 2009.

[20] Sandrine Dudoit and Jane Fridlyand, "A prediction-based resampling method for estimating the number of clusters in a dataset," *Genome biology*, vol. 3, no. 7, pp. research0036–1, 2002.

[21] Anbupalam Thalamuthu, Indranil Mukhopadhyay, Xiaojing Zheng, and George C Tseng, "Evaluation and comparison of gene clustering methods in microarray analysis," *Bioinformatics*, vol. 22, no. 19, pp. 2405–2412, 2006.

[22] Peter J Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987.

[23] William M Rand, "Objective criteria for the evaluation of clustering methods," *Journal of the American Statistical association*, vol. 66, no. 336, pp. 846–850, 1971.

[24] Tian Zhang, Raghu Ramakrishnan, and Miron Livny, "Birch: A new data clustering algorithm and its applications," *Data Mining and Knowledge Discovery*, vol. 1, no. 2, pp. 141–182, 1997.

[25] Keke Chen and Ling Liu, "ivibrate: Interactive visualization based framework for clustering large datasets (version 3)," .

[26] Brian Kulis and Kristen Grauman, "Kernelized locality-sensitive hashing for scalable image search," in *Computer Vision, 2009 IEEE 12th International Conference on.*

IEEE, 2009, pp. 2130–2137.

[27] Inderjit S Dhillon, Yuqiang Guan, and Brian Kulis, "Weighted graph cuts without eigenvectors a multilevel approach," *IEEE transactions on pattern analysis and machine intelligence*, vol. 29, no. 11, 2007.

[28] 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*. ACM, 2009, pp. 907–916.

[29] Nello Cristianini and John Shawe-Taylor, *An introduction to support vector machines and other kernel-based learning methods*, edition, 2000.

[30] Ella Bingham and Heikki Mannila, "Random projection in dimensionality reduction: applications to image and text data," in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2001, pp. 245–250.

[31] Alex Krizhevsky and Geoffrey Hinton, "Learning multiple layers of features from tiny images," 2009.

[32] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[33] Mohamed Hefeeda, Fei Gao, and Wael Abd-Almageed, "Distributed approximate spectral clustering for large-scale datasets," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. ACM, 2012, pp. 223–234.

[34] 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*, vol. 33, no. 3, pp. 568–586, 2011.

[35] RB Lehoucq, DC Sorensen, and C Yang, "Arpack users' guide: Solution of large scale eigenvalue problems with implicitly restarted arnoldi methods.," *Software Environ. Tools*, vol. 6, 1997.

[36] Laurens van der Maaten and Geoffrey Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[37] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al., "Similarity search in high dimensions via hashing," in *VLDB*, 1999, vol. 99, pp. 518–529.

[38] Ting Liu, Andrew W Moore, Ke Yang, and Alexander G Gray, "An investigation of practical approximate nearest neighbor algorithms," in *Advances in neural information processing systems*, 2005, pp. 825–832.

[39] Marie Ouimet and Yoshua Bengio, "Greedy spectral embedding.," in *AISTATS*, 2005.

[40] Kai Zhang, Ivor W Tsang, and James T Kwok, "Improved nyström low-rank approximation and error analysis," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 1232–1239.