Approximate Clustering Algorithms for High Dimensional Streaming and Distributed Data

A dissertation submitted to the

Division of Research and Advanced Studies
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

DOCTOR OF SCIENCE

in the School of Electric and Computing Systems
of the College of Engineering and Applied Sciences

by

Lee A Carraher
BSCE, University of Cincinnati, 2008
MSCS, University of Cincinnati, 2012

Thesis Advisor and Committee Chair: Dr. Philip Wilsey

Abstract

Clustering data has gained popularity in recent years due to an expanding opportunity to discover knowledge and collect insights from multiple widely available and diverse data sources. Data clustering offers an intuitive solution to a wide variety of unsupervised classification problems. Clustering solutions to problems arise often in areas in which no ground truth is known, or when the arrival frequency of a data source exceeds the labeling capabilities of a human expert. Due to the continuous, fast moving nature of many common data streams, such as those from IoT (Internet of Things) devices, social network interactions, e-commerce click-streams, scientific monitoring devices, and network traffic, noise robust and distributed clustering algorithms are necessary.

Often, current clustering methods suffer from one or more drawbacks when applied to these demanding problems. For this reason, we propose a new method for data clustering that is noise resilient, and distributed with a predictable overall complexity. The principal claim of this research is that while many clustering algorithms rigorously optimize a loss function, their convergence often results in finding a local minima that is indistinguishable from a less computationally rigorous optimization on an approximation of the data. We propose that by removing the rigorous optimization requirement, we can achieve better scalability, and parallelism with comparable performance. In this work we design a clustering algorithm along these lines that uses dimensional reduction and hashing to reduce the problem size while still attaining comparable clustering performance to other clustering algorithms. Our proposed method is more robust to noise with a lower runtime requirement, and greater opportunity for shared and distributed memory parallelism.

This work presents a set of methods for clustering high dimensional data for a variety of data source and processing environments. The proposed RPHash algorithms share a commonality in that they all utilize locality sensitive hash (LSH) functions and random projection (RP) to identify density modes in

data sources. They differ however in the operation space and cluster region identification method. The algorithms presented are the RPHash algorithm, the streamingRPHash algorithm, and the Tree Walk RPHash (TWRP) algorithm.

We analyze the results of our developed algorithms on both streaming and at-rest data input environments. Experiments on real and synthetic data demonstrate the advantages of our proposed clustering methods for streaming and at-rest data against common clustering algorithms. Furthermore our theoretical analysis shows that our runtime and memory complexities are effectively linear and sub-linear respectively, in terms of input. Our principal claim that approximate clustering results are not substantially different than exact clustering methods with guarantee convergence to a local minima, is confirmed by these results. In addition we demonstrate the potential gains in processing speed and parallelism.

Acknowledgments

For My Friends, Family and Mentors

I would like to acknowledge the following people for their hard work and research effort on this work. People and organization who without which none of this would have been possible. First and foremost, my extreme gratitude goes out to my advisor: Prof. Philip Wilsey. Prof. Wilsey's guidance not only in academic pursuits, but in so many aspects of my life, are truly appreciated. His adherence to relentless delivery in terms of research development, publication and testing, are a testament to his own commitment to scholarship. Throughout our time working together, I feel that we have truly forged a bond of friendship and colleague that far exceeds my expectation. With my deepest gratitude I would like to say thank you Professor Wilsey. I would also like to thank my fellow student researchers and co-authors: Sayantan Dey, Anindya Moitra. My co-developers and fellow student researchers: Nick Malott, Tyler Parcell, and the Go-Lang RPHash group. Finally I would like to thank my friends and family, my parents, brothers and sisters, and my wife Alex. Alex your support and encouragement has been an inspiration to me. Your courage and commitment in your own academic pursuits are things that I constantly try to emulate and apply to my own research. I'd also like to thank the University of Cincinnati, Department of Engineering and Applied Sciences, and the National Science Foundation for financial, equipment and facilities support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Humans and animals are often taxed with the task of categorizing things to augment their understandings beyond past experience alone. In essence humans and animals are generating models. Model generation is a primary cognitive skill that underlies our ability for prediction and understanding. A primitive examples of this would be to decide whether an animals is dangerous or not, based on past observations of the same species or similar looking animals. A solution to this decision problem would require a clustering of observations along with the outcome of dangerous or not. A clustering establishes a model for a given set of observations based on a metric of closeness in some parameter space. In our example deciding whether an unseen animals is dangerous or not would likely include an evaluation of teeth shape, relative size, eye location relative to head, etc. In effect, forming two clusters of attributes relating to the predator/prey nature of a given animal.

Human capabilities for implicit observation, and subsequent model generation far exceed anything possible with today's computing systems. The problem however arises from the limitations of human observations in both size and dimension. While humans are good at building models for the tangible things around them, they are often unable to build similar models for complex systems that reside in less intuitive spaces. Mathematics and computing give us a probe into this unattainable world through the use of numeric and graph based parameter embedding. However high dimensional, numerical spaces are still not very natural concepts for humans; while computing machinery on the other hand is quite at home in with these concepts, in fact it's often a prerequisite for their use.

The development of mathematics and computing allow humans to augment abilities of prediction to develop models for massive, unintuitive and fast changing systems. These models allow humans to predict droughts, genetic predisposition to disease, financial fraud, expose the underlying principles of particle physics, and many other useful areas that improve our understanding, the quality of our lives, and protect the state of our planet.

Following the problem with correspondence between machines and concepts, the implicit parameter space is often encoded into a more explicit objective space, such as a high dimensional euclidean space. In this setting data can be rigorously scrutinized for correlation and correspondence between observations. Such correlation can then be distilled into a model from which decisions can be made.

1.1 Clustering

Clustering has long been the standard method used for the analysis of labeled and unlabeled data and is a principle occupation of statistical classification. The effect of clustering data allows for identifying dissimilar and similar observation in a dataset — often unattainable by standard single pass statistical methods. Single pass, data intensive, statistical methods are often the primary workhorses for database processing of business logic and scientific domains, while clustering is often overlooked due to issues of scalability [131] and perceived complexity. A multitude of surveys [131] have been made available comparing different aspects of clustering algorithms in regard to accuracy, complexity, and application domain. Due to this importance in machine learning, data clustering has generated extensive interest in the computing and mathematics fields [19, 80, 96, 99, 115, 131, 134].

Despite rigorous exploration, advancements in computing and countless variations, the Lloyd-type iteration k-means algorithm [77, 98] remains the base for some of the most successful parametric clustering algorithms [24]. Of those the most successful clustering algorithm is the k-means algorithm. The k-means algorithm seeks an optimal partitioning of a dataset into k subsets that minimizes the within or inter-cluster distance. The Lloyd-type/Lloyd-step k-means algorithm is an iterative algorithm for solving the k-means problem, that alternates between a cluster assignment step and centroid update step. These types of algorithms are important to consider because they not only embody many clustering procedures, but they also exemplify an intrinsic barrier to parallelism, namely the cluster update step, that is difficult to remove.

The success of k-means and similar clustering methods have facilitated enormous advances in nearly all scientific fields as well as many social, business, and financial occupations. Advances in clustering, in either scalability or speed have a direct impact on the quality and timeliness of clustering results. However, the unavoidable architectural migration from single core processors with ever increasing clock rates, to static clock rates and increasing numbers of cores, has introduced unique challenges for algorithm developers and data scientists. Unfortunately, Lloyd-type k-means is a data centric algorithm, with numerous sequential bottlenecks that do not lend themselves to parallel implementation.

1.2 Big Data Analysis

Data analysis has a long been able to discovery latent data structures that were otherwise inaccessible by human observation alone [36]. Recent advances in data aggregation along with the continuous roll-out of progressively more advanced communication infrastructures has provided an unbounded sources of information. This torrent of data, spurred by industries insatiable appetite for deeper insights and user personalization, has pushed data analysis tasks well into the realm of high performance computing. While the processing of large-scale datasets is not new (VLDB is 42 years old this year), the emergence of platforms for managing and frameworks for processing in the distributed setting has created a new commercial market for large scale computing. The trend is often referred to as *Big Data*. The development of distributed frameworks is a natural progression of the parallel processing required in facilitating the transition from single to multi core CPUs. Furthermore the cost of large scale computing has decreased considerably thanks to cloud services making it available to a larger community of researchers.

Where the general framework of distributed computing consists of a single data archive and attached processing nodes, more recent frameworks attempt to take advantage of in place computing at the site of the data itself, taking advantage of data locality and avoiding costly communication overhead. The common models for big data analysis consist of the MapReduce [58] and MPI [68] frameworks. Both models take decidedly different approaches to parallelism. While MPI, or message passing interface, assists developers in the communication aspects of parallelism, the particular programming structure is mainly up to the developer. Although not strictly enforced, the common MPI interface consists of a master process that distributes data to worker processors, who then compute a partial solution that is then returned to the master process for aggregation or further processing. MapReduce in contrast forces a more strict processing paradigm based on the functional programming structure. MapReduce tries to avoid costly data communication overhead, by enforcing data locality through functional programming structure. This structure inverts the standard data distribution mechanism, by distributing the function to the data, instead of distributing the data to functions. These frameworks also often take care of other common but difficult to optimize, system level tasks, such as fault tolerance, low level communication and shared data storage architectures. Both of these processing structure are amenable to our proposed algorithms.

1.3 Security

Another emerging requirement for data clustering especially in the distributed setting is security. Patient privacy is a high priority [1], and de-anonymization attacks have made it more difficult to provide anonymization assurances beyond what is provided by data scrubbing alone. Further, the decentralized geographic distribution of medical care facilities prevents many tangential security measures often found in more centralized network topologies (user$\rightarrow$organization).

1.4 RPHash

For these reasons we developed an algorithm to solve a parametric k-means clustering problem on distributed and streaming datasets. In addition to the goal of distributed clustering, due to the ever increase possibility of data breach, or method also provides reasonable security for data in transit and at rest. We show that our solution offers comparable clustering performance to off-line solutions on a variety of synthetic and real world datasets. Our solution uses bounded memory and achieves asymptotic linear scalability for a chosen set of configuration parameters.

Our algorithm, called *Random Projection Hash* (RPHash), was expressly created to provide algorithmic scalability on distributed datasets. Many clustering algorithms have been converted to function efficiently on distributed data, however they often have potential issues regarding asymptotic scalability [7], dimensionality in which they are effective [109], and robustness to noise. Although many algorithms have been proposed for parallel clustering, many issues are still present when applied to very large, high dimensional, distributed datasets [7,59,109,112,121]. RPHash combines approximate and randomized methods in a new way to solve these issues of scalability and data security under the assumption that approximate clustering is qualitatively similar to exact clustering methods, due to noise, redundancy, data loss, and the curse of dimensionality [30].

In addition, we propose a variant of RPHash called StreamingRPHash that is similarly effective for the streaming data setting. Data streams in todays high tech environment can arrive continuously at a very fast rate. They are considered to be potentially unbounded and the user is primarily interested in recent data that keep track of the emerging trends [5, 35, 117, 119]. Data streams are an increasingly common data model owing to the pervasiveness of always on-line computing, and continuously updated measurements. The stream clustering data model puts two additional restrictions on the standard big data clustering model. First the complexity of the on-line step, in which vectors are added, must be sub-linear in n, the input size. The other requirement is that the data can only be seen once, with some exception for temporary storage in a sub-linear in n data sketch.

1.5 Thesis Structure

The structure of this the consists of presentation of required background information and related work, followed by our problem motivation, followed by a description of our proposed solution. We then test our proposed solution, and discuss its strengths and weaknesses compared to other algorithms. More specifically, the remainder of this dissertation is detailed in the following paragraphs.

In order to establish a common understanding of the clustering problem, various tools, notation and approaches, we give a brief survey of related work, and background information. Chapter 2 provides some background information on material necessary for covering various preliminaries important to the description of our RPHash algorithm. The background section is organized into subsections corresponding to the problem decomposition of the RPHash algorithm, in which each procedure corresponds to a section. Chapter 3 reviews some of the recent work with similar methods providing more scalable data clustering. We begin by discussing scalable solutions to low dimensional clustering, followed by scalable solutions to high dimensional clustering.

To construct our reasoning and motivation for RPHash we present the next two chapters on motivations and algorithm description. We follow the chapter with an added description of a related clustering method that attempts to use the data to define the algorithm structures. Chapter 4 introduces the initial goal of RPHash, and our early attempts at creating a scalable clustering algorithm. This section describes a drawback of the LSH based kNN problem that was realized to be useful for the k-means clustering problem. Chapter 5 provides a detailed description of the RPHash algorithm. We describe the various components, and the configurations of those components. Next we describe an extension of RPHash, streamingRPHash designed for the streaming environment. The following chapter Chapter 6 first introduces a problem with LSH for data that is tightly clustered, but not uniformly distributed throughout the data space. We use this understanding to design a new LSH algorithm that uses the input data to adaptively modify the specificity of the hash function. We then propose a tree based clustering method that uses this type of function to define the clusters as well. We then extend this solution to describe a general compressed space data structure with applications beyond clustering.

We conclude the dissertation with our experimental chapter, followed by a short chapter on the theoretical results, concluded by a set of conclusions, based on the findings from the previous two sections. Chapter 7.3.1 provides both clustering performance and timing results for RPHash and its variants. The chapter begins with a description of the method that we compare RPHash against, the datasets used for the comparison, and the metrics on which comparisons are made. Then an optimal baseline is established through an exhaustive test of all possible configurations, to determine which is the best overall configuration. The section continues by describing tests for each of the three proposed clustering methods, with plots and results comparing them to other common clustering methods. We also include a brief test of vector anonymity, that demonstrates the security of our RPHash algorithms, as well as a test of the parallel speedup potential for our RPHash algorithms. Chapter 8 provides theoretical asymptotic analysis for the proposed RPHash algorithms. We also include analysis for error propagation and techniques for decreasing the memory footprint of our tree based clustering method. Finally, Chapter 9 contains a detailed analysis of the results found in Chapter 7.3.1, followed by concluding remarks about the nature of clustering and the RPHash approach to clustering. We conclude by presenting some ideas for future directions to for our line of research.

Chapter 2

Background

The beginnings of computing machinery, up until the introduction of the von-Neumann architecture, were narrowly focused on the ambition of mechanizing rote data processing and analysis work. Although today we find ourselves in a multifaceted computing landscape arguably not likely envisioned by computing's forefathers, data analysis still occupies much of the computing research space. Perhaps rightly so, as advances in data analysis have had a profound impact on virtually every field of scientific, economic, and social interest. While the computing performance has grown exponentially over the years many principal problems remain unsolved. One such problem is the statistical classification problem, with the implied definition of identifying an observations classification as induced by other observations. Classes in the computing parlance are more commonly referred to as clusters, and likewise the classification problem is often referred to as cluster analysis. Principal among clustering problems is the k-means clustering problem defined in MacQueen's 1967 paper "Some methods for classification and analysis of multivariate observations" [98] boasting over 18,000 direct citations.

2.1 k-Means Problem

By restricting observations to a vector space, the clustering problem can be described as a partitioning problem in which the goal is to optimize the demarcations bounds to group similar observations. The k-means clustering problem extends this definition to allow a user defined number of partitions, k that are specified, to separate the observations into k distinct groups. Extending this concept we get a simple heuristic for the performance of a particular clustering. Given a set of vectors $x \in X$ partition the data space into k clusters $\{C_1, C_2, ..., C_k\}$ of vectors such that the *Within-Cluster Sum of Squares Error* (WCSSE) metric is minimized over all possible partitioning. More formally,

$$\underset{C}{\text{argmin}} \sum_{i=1}^{k} \sum_{x \in C} ||x - \mu_i||^2,$$

where μ_i is the mean of the points in a partition or centroid of the cluster C_i.

Figure 2.1.1 displays an example of a standard k-means algorithm run on Gaussian data clusters with nine random starting instances. The above optimization is WCSSE statistic is given with each clustering attempt. The example shows the success of the WCSSE criterion in identifying the ground truths of a dataset while variability in correspondence between estimated centroid partitions and ground data points demonstrates a key drawback of hard margin data clustering. In RPHash, a focus on high clusters in high dimensional space, minimizes the risk of overlap, discussed more formally in the section on occultation (Section 5.1.3).

For points on a line, the k-means problem can be solved exactly in polynomial time using dynamic programming techniques. Unfortunately, this is a special case, and there is no known polynomial time algorithm for optimizing the k-means criterion for generalized $\mathbb{R}^d$. Following, if P$\neq$NP, optimizing the k-means heuristic is Max-SNP/APX hard [56,100]. The two classes Max-SNP and APX are subsets of the NP class that deal specifically with graph theoretic problems, and approximate optimization problems respectively. These problems often have solutions that are can solve real world problems in polynomial(P) time, but are none-the-less in NP for adversarially crafted problems. Despite this somewhat dire outlook, APX problems such as k-means have polynomial time bounded approximate solutions. k-means not withstanding, $1 + \epsilon$-approximate k-means can be solved in linear time [89]. Furthermore, even simpler Lloyd-type algorithms tend to converge quickly on many real-world datasets [83].

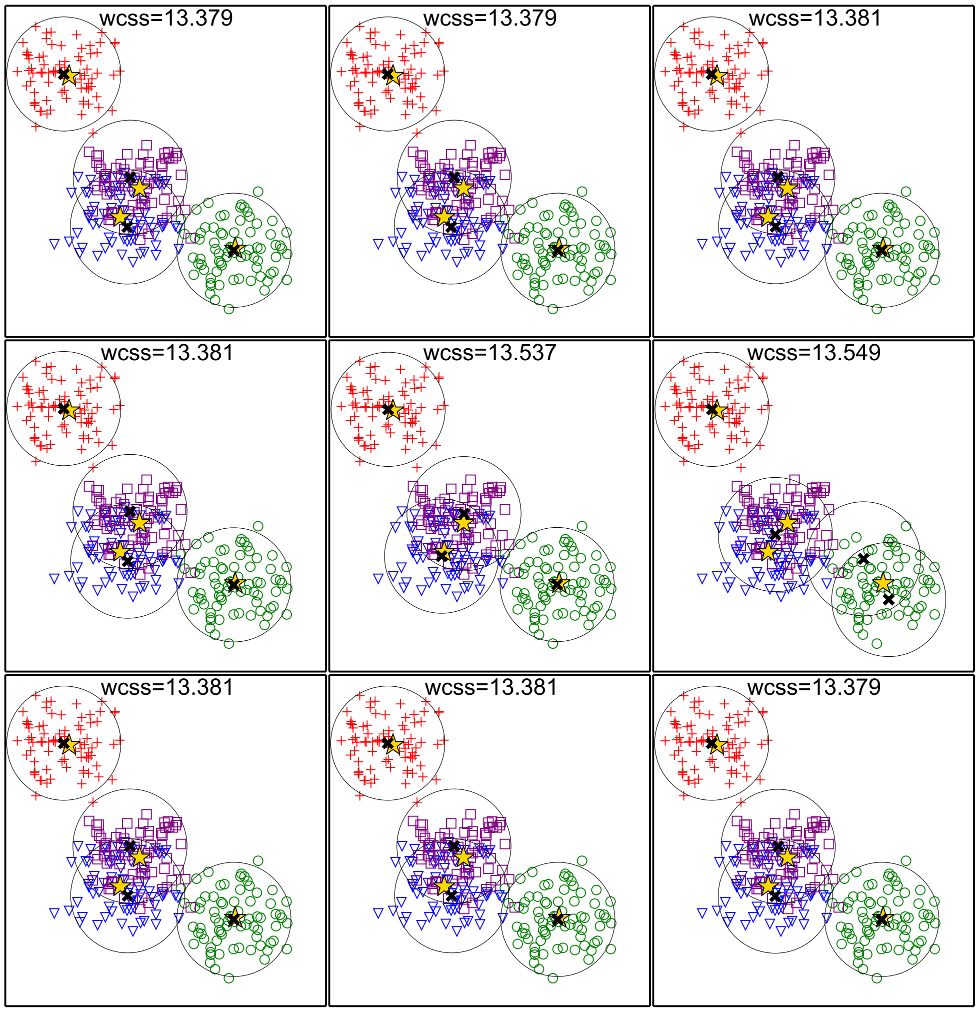


Figure 2.1.1: Example k-means

Example k-means on a dataset of 4 generated Gaussian clusters. Gold stars mark the ground truth, and cluster symbol and coloring correspond to the vector's true membership. Black X's mark the k-mean centroid estimates. The within-cluster sum of squares (WCSSE) is provided at the top of each plot.

2.2 Distributed Data

Given the success of k-means algorithms and their relatively well behaved run time complexities, what need do we have for another clustering algorithm? While incremental algorithmic improvements result in shorter run-times they do not often attack the fundamental issue of k-means's distributed scalability. For this reason we propose an algorithm that is specifically designed to overcome this issue that forestalls the communication requirement until the final processing step.

2.2.1 Distributed k-means

A principal occupation of parallel computing research is speedup and the desire for a linear, or sub-quadratic scaling. Parallel speedup is the ratio of single compute node processing time versus parallel processing time or $Speedup = T_s/T_p(n)$, where T_s is the time to process on a single node and $T_p(n)$ is the time to process in parallel on n compute nodes. Of particular interest, is the shape of the ratio curve as $n \to \infty$, which often is embedded in the underlying algorithmic structure. The structure of the most common Lloyd-step k-means clustering algorithm iteratively alternates between two steps, consisting of an assignment step followed by an update step. The assignment step, where centroids are assigned to their nearest representative cluster, can be made parallel by processing data vector assignments in parallel. However the update step, in which the cluster centroid is computed and distributed to processors for the following assignment round, seems to be inherently sequential. In distributed processing this issue is referred to as a sequential bottleneck or barrier synchronization. Amdahl's Law gives the optimal speedup potential for parallel processing based on this ratio of required sequential (s) code to parallel (p) code. More formally:

Theorem 2.2.1 (Amdahl's Law)**.**

$$Speedup = \frac{1}{(1-p)+\frac{p}{s}}.$$

In the case of Lloyd-step, k-means we can view the vector assignment step as parallel and the new centroid computation as a sequential step. Although this is perhaps an oversimplification, as centroid computation could also be parallelized, Amdahl's Law speedup assumes contention-less accesses to shared memory, which does not exist in practice as the number of processing nodes grows. Instead additional communication overhead would be required, making the centroid processing step sequential at best. In Figure 2.2.1 optimal speedups are shown for various ratios of parallel to total code.

As discussed, even optimizing sequential to parallel code, will not overcome requirements due to network latency, memory contention and cache conflicts. For these reasons, recent work has focused on creating clustering algorithms with lower communication overhead and fewer sequential bottlenecks.

2.2.2 Streaming Data

In addition to being poorly suited for distributed analysis on static data, the standard k-means is not directly applicable to processing a continuous data stream. This is because, standard k-means requires a pass through all of the data, to compute the final cluster centroids. A requirement that is not available in the streaming models. Next we define the streaming data model, its specific attributes and requirements. In particular, the unbounded nature of the stream, implies that passing over all of the data, is not an option for a streaming algorithm.

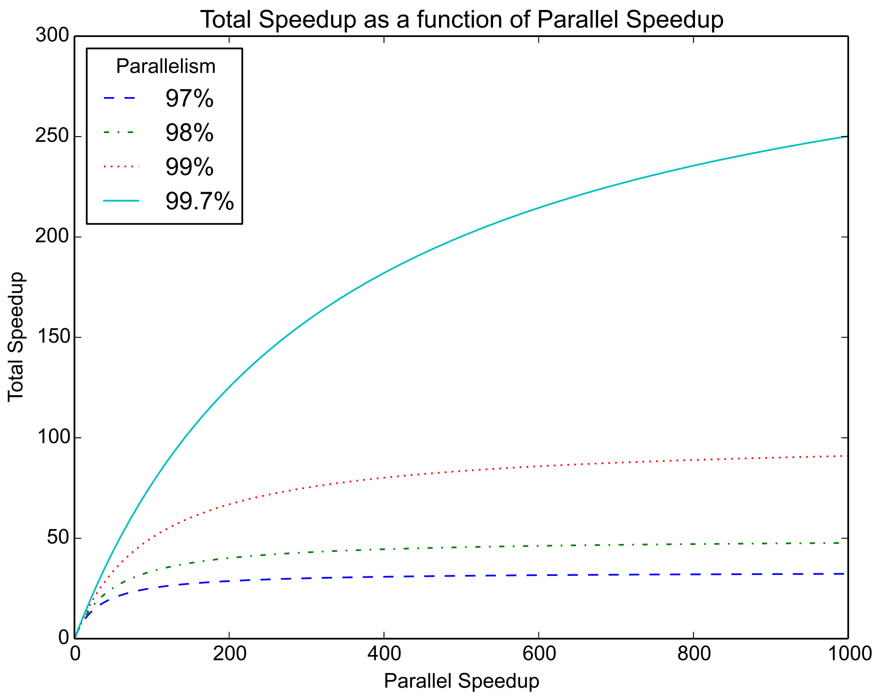


Figure 2.2.1: Amdahl's Law Speedup
Amdahl's Law optimal Theoretical Speedup In a contention-less shared memory system for various parallel to total code implementations as a function of compute nodes

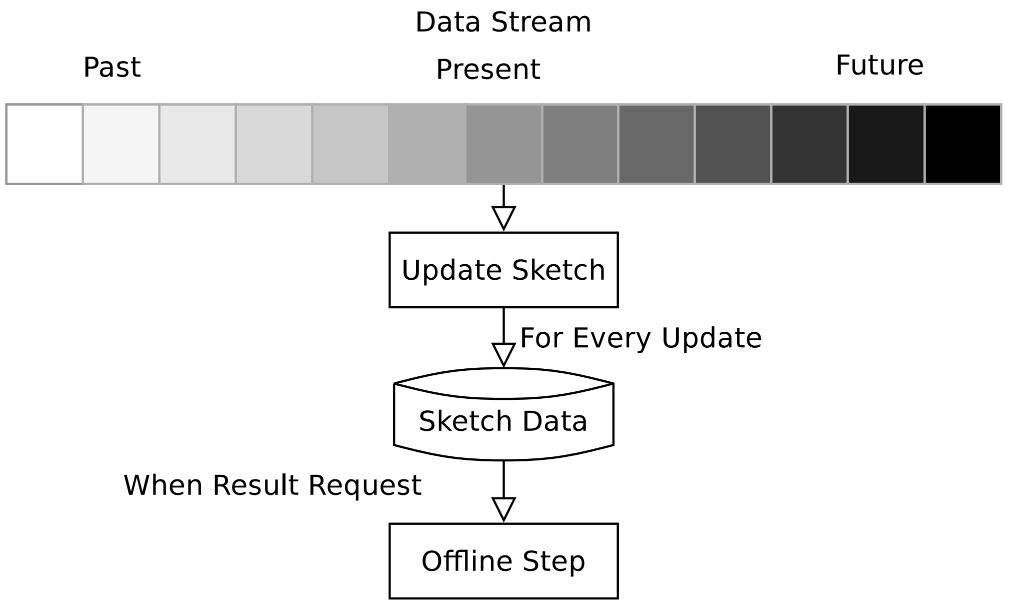


Figure 2.2.2: Streaming Model

Definition 2.2.2 (Data Stream)**.** *A data stream S is a sequence of data vectors $v_1, v_2, ...v_n \in V$ that arrive sequentially in time t where $t(v_1) < t(v_2) < ... < t(v_n) \forall v_i$, and the number of vectors n, is potentially unbounded ($n \to \infty$). Each vector v is of constant length, or is a variable length decomposable to constant length vector (e.g. words in sentences embedded in a sparse dictionary space).*

Figure 2.2.2 provides an overview of the data stream model along with common attributes of many streaming algorithms. Data streams present a variety of challenges in both computational and storage complexity to clustering algorithms. The iterative nature of k-means requires that an entire dataset be accessible at any point in time. Data streams, however are potentially unbounded sequence of observations, with irregular arrival times. Due to the unbounded nature of streaming data, the entire set of vectors cannot realistically be stored in main memory.

In general there are three requirements that we regard as essential for streaming algorithms with n observations, namely:

- **Bounded Memory:** Due to the constraints of memory bandwidth in real machines, memory access patterns for data streams must be sequential allowing for one or a very small constant number of passes over the data. The bound on the stored data, is sub-linear, and is usually regarded as either a predefined constant based on memory availability or $\theta(log(n))$.

- **Sub-Quadratic Processing Time:** Because the data stream is unbounded, it is infeasible to have an algorithm whose complexity is greater than $\theta(nlog(n))$ in regard to processing the entire data stream. Or for each vector, no more than some constant$*log(n)$ steps may be performed.
- **Off-line-Step Complexity:** Due to the unbounded nature of the data stream, there is no reasonable start and end to the data stream. As such, a streaming algorithm must be able to produce a result at any time throughout the evolution of the data stream, and in a reasonable amount of time. While this amount of time is not strict, it invariably may only operate on the stored $\theta(log(n))$ data.

In addition to the machine constraints presented by the streaming data model, streaming data often provides meaningful temporal aspects. One such attribute is changing trends over time. Such trends as concept drift, occur often in the semantic data space, and are a specific form of re-baselining in many other fields. For this reason the concept of the data stream, and subsequent clustering problems have been proposed [119] to capture these latent features.

2.3 Sketching

Following the requirements of streaming algorithms to have a bounded memory footprint, as with many big data problems in computing accuracy is traded for memory. One such method of lossy compression is the sketch data structure. The goal of a sketch is to maintain an approximate record of the salient features of a dataset while not requiring that the entire dataset be stored in main memory. Sketch data structures have seen recent resurgence following the success of the frequent directions algorithm [72], hyperloglog [65], counting [50], precision sampling, and solutions to the k-Heavy-Hitters problem [28].

Sketching tends to offer one of two trade offs for memory, either lossy and sampling methods which older elements ignored or forgotten or space saving, in which elements sketches lose accuracy over time. RPHash resorts to the latter exchange, and accept loss of accuracy over time.

2.3.1 Count-Min Sketch

One such data structure for space saving sketching is the Count-Min Sketch [50]. The general idea is to partition the element range into a finite set of discrete hash buckets, then update the bucket's value instead

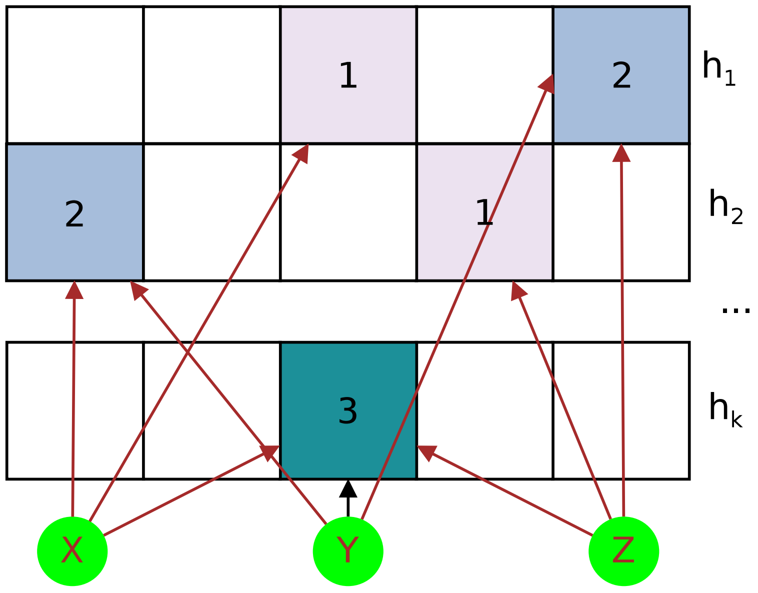


Figure 2.3.1: Count-Min Sketch

of storing each element. Due to the likelihood of hash collision in the element range, multiple copies with different hash functions from the same family are updated in parallel. To query an item's sketched value, all bucket sets are searched, and the minimum of the searched buckets is returned. The result of the minimum over multiple inaccurate searches results in an amplification of accuracy for the searched element following, similar to minimum over chained bloom filters. Buckets update false positively often, but are never false negative for a given element, therefore the minimum is the closest to the real count value. Figure 2.3.1 gives a diagram of this behavior for a toy example. Rigorous mathematical proof of the accuracy for this data structure has been studied in [52].

For a positive only counter, the error in counting corresponds to the best of the d approximate counts with error due to noise from other counts, of $1/w$. For d hash functions if we chose $d = \log 1/\delta$ and $w = 2/\varepsilon$ we have an estimate for the count that is at most εN with probability greater than $1 - \delta$. Following a proof in [51] using the Chernoff bound, the probability of returning a 'bad' estimate is exponential in d.

2.4 Dimensional Reduction

Dimensional Reduction is a key occupation for high dimensional data analysis. Yet another face of the Curse of Dimensionality, high dimensional data suffers from a loss of contrast between near and far points, making euclidean metrics and even non-euclidean metrics useless as dimensionality grows. This can be seen in the limit of the volume occupied by a distance metric and an orthogonal space embedding approaching 0 as the embedding dimension grows. Here we give an informal proof of this approach. The euclidean metric essentially carves out hyper-spheres in high dimensional space as we can see from the distance function $d(x, y) = \sqrt{\Sigma(x_i - y_i)^2}$. However, orthogonal embeddings with many dimensions often distribute volume in coordinate space along rectangles. We can see the ratio of the volumes of these two geometric objects as dimensionality grows by taking the dimensional parameterization of the volume equation for spheres and cubes with radius r and side length $2r$. Formally this derivation is

1. Sphere_Volume$(d) = \frac{r^d \pi^{d/2}}{\Gamma(d/2+1)}$

2. Cube_Volume$(d) = 2r^d$

3. dropping constants, we get the asymptotic ratio:

$$\theta\left(\frac{\text{Cube\_Volume}}{\text{Sphere\_Volume}}\right) = \frac{(2r)^d \Gamma(d/2)}{\pi^{d/2} r^d}$$

4. replacing Γ with Stirling's Approximation:

$$= \frac{(2r)^d \sqrt{\pi d}(\frac{d}{2e})^{d/2}}{\pi^{d/2} r^d}$$

5. dropping constants and simplifying:

$$= \frac{d^{d/2}(2/e)^{d/2}}{\pi^{d/2}}$$

6. taking the limit:

$$\lim_{d\to\infty} d^{d/2}\left(\frac{2}{e\pi}\right)^{d/2} = \infty.$$

Even non-euclidean metrics exhibit this behavior. This can be attributed to a more complicated analysis of the ratio of the difference between the maximum distance and minimum distance divided by the maximum distance, often referred to as contrast, approaching 0 as $d \to \infty$. A thorough investigation of this is given in [6, 30, 136]

2.4.1 Conventional Dimensional Reduction

Conventional dimensional reduction techniques consist of the classic Principal Component Analysis (PCA) reduction which consists of computing the directions of maximum variance in the data, and projecting the data along the a subset of vectors corresponding to the greatest variance vectors. PCA based reductions are useful and provide an embedding that optimizes the error in the L_2-norm. PCA reductions however require considerable computation, and are not suited for streaming environments. Frequent Directions [72] tend to mitigate these limitations but is are still fairly new in terms of adoption.

Another technique for dimensional reduction is the t-SNE [97] technique. t-SNE is similar to PCA in its minimization of an objective, however instead of minimizing the L_2-norm difference, it optimizes the Kullback-Leibler divergence. Similarly it is not well suited for the streaming environment, as the minimization requires all data, and follows a gradient descent approach to minimize the objective function.

2.4.2 Random Projection

Random projection is a method in which high-dimensional data vectors are projected down to a lower dimensional space by applying a projection following some specifically crafted matrix. One such example of an optimal distance preserving projection would be the result of the truncated principal component projection. Unfortunately PCA is computationally complex and requires multiple scans over the entire input dataset. However as the original embedding dimensionality grows, the distortion between PCA projection and random projection tends to converge. A random projection matrix is composed of orthogonal vectors sampled from some random or quasi-random distribution. Random projection can achieve a bounded error distortion factor very close to the optimal L_2 norm subspace embedding that would otherwise result from the principal component decomposition based projection [32]. The resurgence of the random projection method of Johnson and Lindenstrauss was reinvigorated with the work of Achlioptas on Database Friendly Projection that provided good subspace embeddings requiring minimal computation costs [2].

2.4.3 JL Lemma

The Johnson-Lindenstrauss Lemma defines the error bounds of random projection. This formal bound for our projections $f(\cdot)$ that preserves the distance between any two vectors u and u' with error ϵ is given as:

Theorem 2.4.1 (Johnson-Lindenstrauss Lemma [129]).

$$(1-\epsilon)\|u-u'\|^2 \leq \|f(u)-f(u')\|^2 \leq (1+\epsilon)\|u-u'\|^2$$

In Figure 2.4.1 results are given for a randomly generated dataset from $\mathbb{R}^{10000}$ of vectors that are unit distance apart. The vectors are randomly projected down to $\mathbb{R}^{10000}$, $\mathbb{R}^{1000}$, $\mathbb{R}^{100}$, and $\mathbb{R}^{24}$ and the **L2** distance is calculated. As the bound suggests, average distance is consistently unit. However, higher degrees of projection result in a greater occurrence of outliers and overall increase in overall variance.

Under the optimal ϵ-preserving mapping $f(\cdot)$, the Johnson-Lindenstrauss lemma results in a tight bound for $u, u' \subseteq U$ and $n = |U|$, of $d \sim \Theta(\frac{log(n)}{\varepsilon^2 log(1/\varepsilon)})$. The bound was later applied to random orthogonal projections in Frankl and Maehara, and found to have a similar order on the bound for the projected subspace dimensionality [69]. Vempala gives a relaxation of the JL-bound for random orthogonal projections, arriving at $d \sim \Omega(log(n))$ [129], with a scaling factor of $\frac{1}{\sqrt{d}}$ to preserve approximate distances between projected vectors. An additional benefit of using Random Projection for mean clustering is that randomly projected asymmetric clusters tend to become more spherical in lower dimensional subspace representations [31]. Mean and medoid clustering algorithms, such as RPHash, are predisposed to spherical clusters.

2.4.4 DB-Friendly Projection

Using the intuition that the dot product with a vector composed of zero mean Gaussian random variates, amounts to a great deal of multiplication by values near zero, D. Achlioptas describe a far more efficient projection matrix with surprisingly more favorable attributes than multiplication by a dense Gaussian matrix [2]. These so-called DB-Friendly projections provide dimensional reduction that adheres to the JL-Bound, and in many cases provide lower error embeddings than the more computationally intensive Gaussian matrix projection [2]. Computed efficiently, the DB-Friendly Projection method requires $\theta(dm/3)$

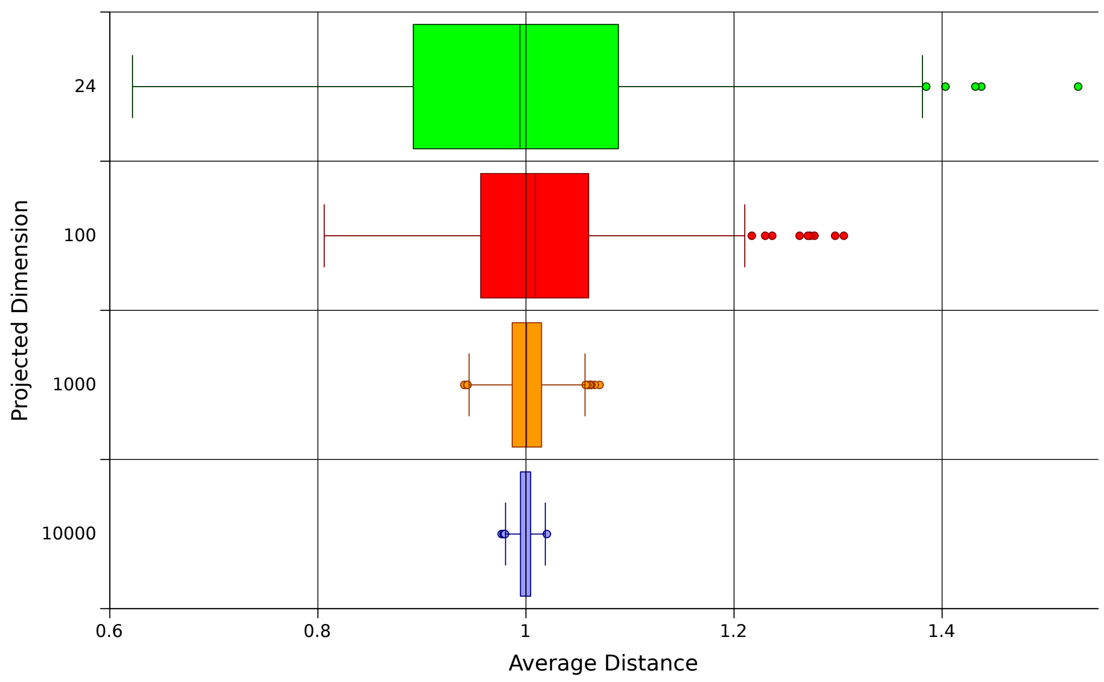


Figure 2.4.1: Example random projection dimension and expectation of distances for vectors that are unit distance apart.

steps. The DB friendly matrix is constructed as: $r_{ij} \in \mathbf{R}$ is $m \times d$ as follows:

$$r_{ij} = \sqrt{1/d} \begin{cases} +1, & \text{with probability } \frac{1}{6} \\ 0, & \text{with probability } \frac{2}{3} \\ -1, & \text{with probability } \frac{1}{6} \end{cases} [2]$$

2.4.5 Fast Johnson Lindenstrauss Transforms (FJLT)

Advances in compressed sensing and Restricted Isometry Property (RIP) have pushed the bounds of random projection to nearly optimal and work efficient samplings of the input data following a realization that the underlying goal of random projection is to evenly represent a mixture of the input space vectors. To optimize this mixture, FJLT looks to the Heisenberg principle in harmonic analysis, stating that the spectrum and signal cannot both be concentrated [11]. The FJLT consists of a projection matrix product $\Phi = PHD$ that consists of two random matrices, P a sparse $m \times d$ Gaussian, D a $m \times m$ diagonal "coin-flip" matrix, and H, the rank m Hadamard matrix. The resulting matrix projection requires only $\theta(m \log m)$ time where m is the original dimension.

2.5 Space Quantization

Switching from the continuous spaces of random projections, we now consider the discrete space partitions induced by lattices. Optimal implementations of grid based clustering algorithms such as DBSCAN [62] and DStream [37], require that the data space be partitioned as evenly as possible. Furthermore to avoid costly interprocess communication overhead, a universally generative naming scheme must be established. For known datasets, a perfect partition of the data space can be produced by the Voronoi diagram [87]. In 2-dimensional space, Voronoi Diagrams can be generated in $\Theta(nlog(n))$-time [67]. However, higher dimensional algorithms for Voronoi partitioning have far less favorable runtime complexities [71], making them inefficient for partitioning arbitrarily high dimensions. A solution to this partitioning problem is to sacrifice the optimal partitioning and accept a probabilistically close partitioning. Such is the case with Locality Sensitive Hash (LSH) functions. LSH functions use collision probability dependent on distance to chain regular partitionings together to approximate optimal partitions. Now we present a formal definition for LSH.

Definition 2.5.1 (Locality Sensitive Hash Function [57])**.** *let* $\mathbb{H} = \{h : S \to U\}$ *is* $(r_1, r_2, p_1, p_2)-$*sensitive if for any* $u, v \in S$

1. *if* $d(u, v) \leq r_1$ *then* $Pr_{\mathbb{H}}[h(u) = h(v)] \geq p_1$
2. *if* $d(u, v) > r_2$ *then* $Pr_{\mathbb{H}}[h(u) = h(v)] \leq p_2$

Lattices, are an alternative, mathematical structure, that can partition infinite, fixed dimensional data spaces in a generative way. We give an example of the A_2 Lattice in Figure 2.5.1. Furthermore, lattices formed from binary codes (such as E_8 and Leech Lattice) have extremely efficient nearest neighbor decoding algorithms. A formal definition of a lattice constructed from a linear code is:

Definition 2.5.2 (Lattice in $\mathbb{R}^n$ [81])**.** *let* $v_1, ..., v_n$ *be* n *linear independent vectors where* $v_i = v_{i,1}, v_{i,2}, ..., v_{i,n}$. *The lattice* Λ *with basis* $\{v_1, ..., v_n\}$ *is the set of all integer combinations of* $v_1, ..., v_n$ *the integer combinations of the basis vectors are the points of the lattice.*

$$\Lambda = \{z_1v_1 + z_2v_2 + ... + z_nv_n | z_i \in \mathbb{Z}, 1 \leq i \leq n\}$$

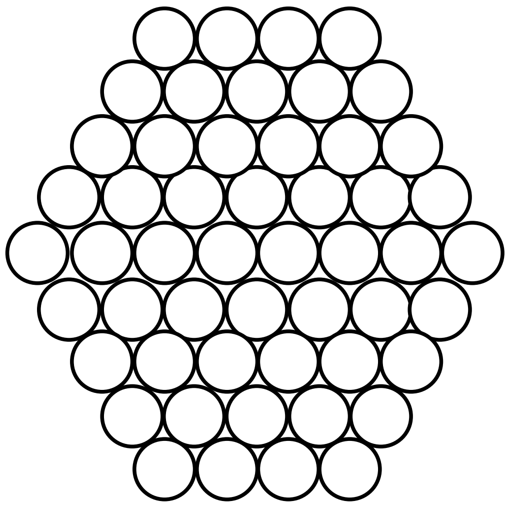


Figure 2.5.1: A_2 Lattice Constellation

2.5.1 E_8 Lattice Decoder

The E_8 Lattice or Gosset's Lattice provides an optimal sphere packing in 8 dimensions. Furthermore, the nearest neighbor decoding complexity of E_8 is relatively low, owing to its decomposition into $E_8 = D_8 \cup < \frac{1}{2} > + D_8$. D_8 decoding entails a simple rounding technique among the vectors in $\mathbb{R}^8$ and an overall parity check. Further advancements [49] in the decoding complexity of E_8 result in a decoder requiring only 72 steps. However, due to E_8's relatively low dimensionality, we must also apply a rudimentary outer decoding step which simply splits a projected vector of dimension higher than 8 into multiple E_8 decodings.

2.5.2 Leech Lattice Decoder

The Leech Lattice is a unique 24 dimensional lattice with many exceptional properties [49, 53]. Of particular interest to this work is the Leech Lattice's packing efficiency. The Leech Lattice defines an optimal regular sphere packing of 24 dimensional space [90] and serves nicely as a space quantizer for RPHash. Furthermore, the Leech Lattice, being the cornerstone of many intriguing mathematical concepts, has various relationships with other sub-lattices that offer useful algorithmic decompositions.

Among some of the most efficient decoders for the Leech Lattice, Amrani and Be'ery's [18] decoder has a worse-case decoding of only 331 floating point operations. Although higher dimensional lattices with comparable packing efficiency exist, the decoding complexity, in general, scales exponentially with dimension [10, 122].

2.5.3 Spherical Locality-Sensitive Hashing

Another method of space partitioning applicable to data where the vector norm is 1, or rather lies on the surface of a hypersphere, is the Spherical LSH technique of Terasawa [123]. Spherical LSH uses an inscribed regular polytope to partition the surface of the sphere where vertexes of the polytope correspond to partition regions. Although other regular d-polytopes such as the d-simplex and d-hypercube exist, we focus on the d-orthoplex, following the favorable collision probability per distance, which results in Table 1 of [123]. The d-orthoplex is a regular d-polytope with $2d$ vertexes corresponding to the positive and negative axis per dimension of a 0 centered unit hypersphere embedding. The nearest vertexes can be searched among the $2d$ vertexes following the max dot product method given in [123], resulting in a search time complexity of $\theta(d^2)$. The d-orthoplex has the benefit over E_8 and Leech Lattice decoders of allowing for arbitrary projection dimensions, with the disadvantage that it is only strictly applicable to vectors lying on the surface of a hypersphere.

2.5.4 Collision of LSH Functions

As stated above, a desired trait of an LSH function is a discriminative collision probability curve with respect to inter-point distance. In Figure 2.5.2 we give the collision probability curve for set of LSH functions: The Leech Lattice(Λ_{24}), Multi-E_8, Spherical LSH (SLSH), and the Gaussian p-Stable distribution. Notable, multi-E_8 gives the tightest overall curve, with λ_{24} following closely in terms of discrepancy ratio for distance $p = \frac{r_1}{r_2}$.

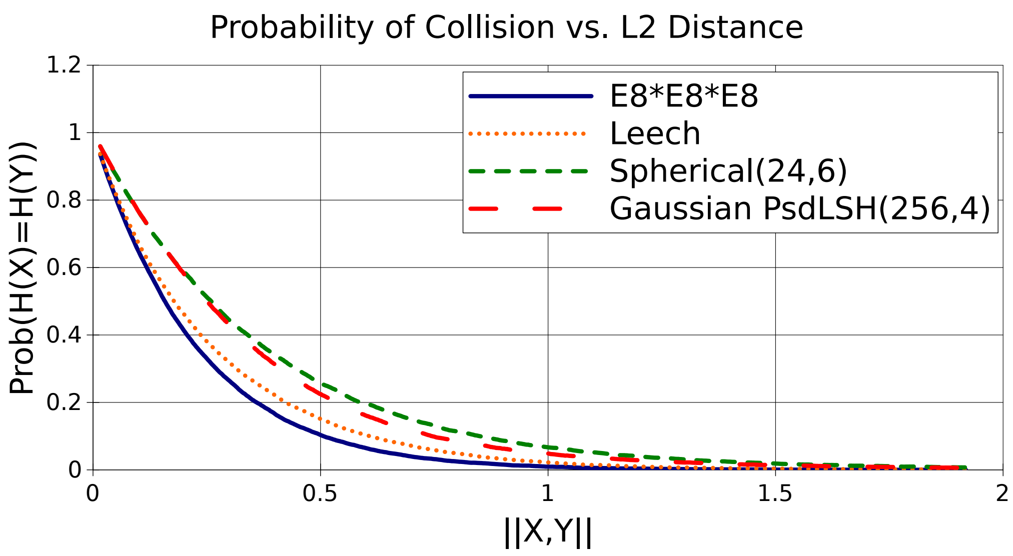


Figure 2.5.2: Probability of collision as a function of distance for various decoders for vectors in $\mathbb{R}^{24}$

2.6 Differential Privacy

Security is another issue that arises in the distributed clustering problem. Medical, financial, and proprietary commercial data are often targets for clustering that have security requirements not addressed in the local clustering setting. Data privacy is often important in machine learning due to the sometimes sensitive nature of records in a dataset. Distributed and streaming data analysis applications have additional security requirements due to the sometimes insecure transport mechanisms between processing nodes. The cryptographic concept, differential privacy, provides some solutions to these problems. The result is often a trade-off between privacy and data loss or additional processing requirements. This general concept is referred to as Differential Privacy [61].

2.6.1 k-anonymity

k-anonymity is privacy technique that defines a number k of records that are equivalent based on the record's attributes. Generalization and suppression are employed, such that data utility is not lost, and the k-anonymity constraint is met for some k [8]. An intuitive example would consist of a set of patient records with various attributes. One such attribute could be age, employing the rounding technique, one could coerce two records to being equivalent, by changing the exact age to a range of ages such as 18-25, instead of 19, and 23. Such a obfuscated dataset would have $k = 2$ -anonymity.

2.6.2 l-diversity

An extension of k-anonymity, l-diversity attempts to protect the attributes as well as the records. l-diversity overcomes some of the flaws in k-anonymity, by requiring that a dataset has at least l different values for each attribute. Common ways to achieve l-diversity are random sampling and additive noise.

2.7 MapReduce

MapReduce [58] and similar programming structures have become popular in recent years with the so called big data revolution. These architectures abstracted out the often tedious and difficult management tasks of maintaining a fault tolerant, distributed, processing system. Furthermore, they made heterogeneous computing on commodity hardware the norm for big data processing that it is today. This change allowed more people and businesses to take advantage of large scale processing while not requiring specialized hardware or in the case of Platform as a Service (PaaS) architectures removed the need for hardware altogether.

MapReduce is a distributed processing framework in which the focus is data locality facilitated by a functional programming approach. Common to functional programing, the Map and Reduce functions perform specific operations on data instead of the data-flow approach in which data is distributed to processors. The map function is an operation that maps data to a discrete set of keys, while the reduction function aggregate the keys. From the distributed computing context this process scales well so long as the key-set and reduced data are not exceedingly large compared to the original data. By strictly adhering to Map and Reduce functions, data can be distributed horizontally across a set of systems during the data collection phase, and small data bandwidth operations can be broadcast (scatter) to be performed on the processing node's local data. The key reduction step is also optimized in this framework, as key aggregation can be performed on a peer-to-peer basis (gather). While these processing method are not new, perhaps the more long lasting result is the proliferation of freely available distributed processing frameworks.

Hadoop from the Apache Foundation is a popular version of the map-reduce framework written in Java. Hadoop also includes a distributed file system (HDFS [118]) as well as monitoring tools, and extensions various computing tasks such as machine learning, namely the Mahout package.

Spark [43, 133] is an extension of map reduce that performs in memory data manipulation while also

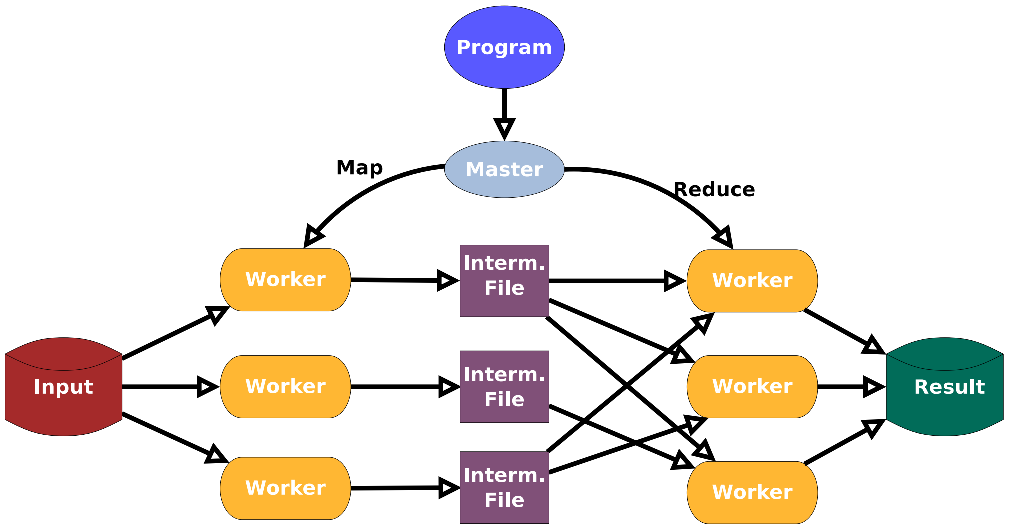


Figure 2.7.1: Map Reduce Step

allowing for more flexibility in the processing sequence. In standard map-reduce, only a single shot, map (read from disk) and reduce (write to disk) operations are preformed. Various meta-frameworks have mapped the map-reduce task, however they strongly incurred the disk IO processing bottleneck. Spark continues the functional approach to distributed processing, namely map and reduce, but extends it to utilize in memory storage and multiple applications of map and reduce. In addition to supporting a more generic map-reduce distribution, Spark also support streaming data processing. Streaming data processing consists of a continuous stream of data distributed to processing nodes, with periodic ‘off-line’ synchronization processes. Figure 2.2.2 depicts a point in time in the streaming model with earlier and past time represented by a darkening of the data stream squares. In this work it is adequate to interpret each block as not necessarily uniform sized, batch of data vectors evolving over time. For each batch of vectors, a sub-linear sketch of the data is updated. Periodically or at the request of a user, the sketch can be analyzed by an ‘off-line’ step that yields a result given the current state of the data stream. In the case of a clustering algorithm this would be a set of centroids. We refer the reader to Silva [119] and Aggarwal [4] for an extensive analysis and definition of various goals and methods in data stream clustering. In this work we will focus on finding fixed k non-moving centroids.

Chapter 3

Related Work

Clustering is a historically well developed and long lived field in mathematics and computation. The modern computational incarnation of clustering likely started with Stuart Lloyd's work in 1957 at Bell Lab's [95]. In the 60 years since this development, volumes of work have been devoted to developing new methods for identifying clusters and testing their performance.

Despite an exponential worse-case complexity of k-means (where $P \neq NP$) [128], many real-world problems tend to fair much better under Lloyd's type of solutions to k-means optimization than theoretically optimal solutions. For this reason, clustering massive datasets with k-means, although suffering from unbounded complexity guarantees, often yields qualitatively good results close to the optimal k-means solution. Due to the approximate solution's real-world proclivity towards revealing useful results, randomized methods such as sampling and random dimensional reduction are often utilized in overcoming complexity growth. The use of these methods in clustering began with density and grid based scanning algorithms.

Clustering algorithms have a variety of applicable taxonomies often differentiated by data type, cluster shape, inference model, and so on. In this work we will focus mainly on a particular type of clustering distributed algorithms for clustering high dimensional Gaussian clusters. Below we list some similar clustering methods to RPHash, split by type and process setting.

Distributed clustering algorithms:

- canopy clustering [103]

Classic algorithms:

- k-Means [77]

- k-Means++ [24]

3.1 Density Based Clustering

The first set of clustering algorithms began with density based scanning methods. These methods tend to work well on spatially separated datasets with relatively low dimension. A common clustering problem for these types of algorithms would be on geo-spatial data, in geographic data systems (GIS) and image segmentation. The algorithms DBScan [62], Clique [9], and CLARANS [109], respectively, represent a successful progression of the density scanning techniques.

DBSCAN proceeds in a conceptually similar manner to RPHash in regard to partitioning the data space and counting the number of data vectors within a partitioned region. The first set of parallel clustering algorithms began with density based scanning methods as the process of building out clusters can take advantage of memory locality (*e.g.,* spatially near vectors are also near in memory). Although density scan algorithms are an example of parallel designed algorithms such as RPHash, they often show weaknesses in accuracy when scaling the number of dimensions. A proposed solution mentioned below to this problem is Proclus [7].

3.2 Clustering in Projected Space

Another important set of algorithms related to RPHash are projection based clustering algorithms. The natural commonality among these algorithms is that instead of clustering over the entire data space, they deal with a projected subset of the data. The immediate benefit is a reduced computational complexity. Various other benefits of random projection are discussed in Section 2.4. Clustering by random projection, similar to RPHash, are explored in [14, 25, 64], but they often strongly violate the limits of the *JL*-lemma, resulting in occultation. The concept of random projection clustering is not new, having been explored in a variety papers involving high dimensional data clustering.

Proclus [7] was an even early example of projection clustering that used random 1-dimensional projections to reduce the dimensionality of a clustering problem. The 1-dimensional projections were then ensembled to create a consensus of clusters. A proof of the convergence of projected k-means clustering is given in Boutsidis [33]. The merits of random projection are further discussed in [55] who suggest that random projection not only compresses sparse datasets making them computationally more tractable but also may help overall accuracy by alleviated round-off issues caused by non-homoscedastic variance by generating more spherical clusters in a more dense subspace.

In addition to Proclus, various other methods and analysis have been proposed for clustering with random projections that provide bounds on the convergence and limits of random projection clustering. Florescu gives bounds on the scaling and convergence of projected clustering [66]. Their results closely follow the logic of Urruty [126], and find that the number of orthogonal projections required, is logarithmic in n, the number of vectors to be clustered. Related to the arctangent of the angle between any two distinct clusters the probability of a random projection plane offering a good partitioning increases exponentially as the number of dimensions in the projected subspace increases. Bingham *et al* provide examples of projected clustering well below the JL bound [31] and Bartal *et al* make these assertions mathematically rigorous showing that the projected subspace is independent of the data's original dimensionality [26]. Proclus used an assumption similar to RPHash regarding high dimensional data's sparseness. Feature subset selection offers a way of compacting the problem as well as removing artifacts. Many subset selection algorithms have been explored [34, 86, 88, 132]. They generally require difficult to parallelize iterative gradient descent [16] or tree traversal [70]. Random projection is performed on the data to compress it to spherical clusters [55] in a dense subspace. An iterative method for k-medoid search, similar to CLARANS [109] is then used to identify clusters. Although the proposed method is successful in accelerating larger dimensionality problems ($d = 20$) it does not have the overall proposed scalability of RPHash. This is due to Proclus being based on an ultimately iterative algorithm, containing inevitably sequential code, and therefore lacking indefinite scalability [16].

Other clustering by random projection algorithms have been explored that are similar to RPHash, but for projections on the line. These so called cluster ensemble approaches [14, 25, 64] use histograms to identify cluster regions of high density much like RPHash. Although, as suggested in Florescu and Urruty, the single dimension approach may be plagued by issues of occultation, and exponential convergence as d increases. Figure 3.2.1 shows a brief example of projection occultation for Gaussian clusters in $\mathbb{R}^3$ space, projected to

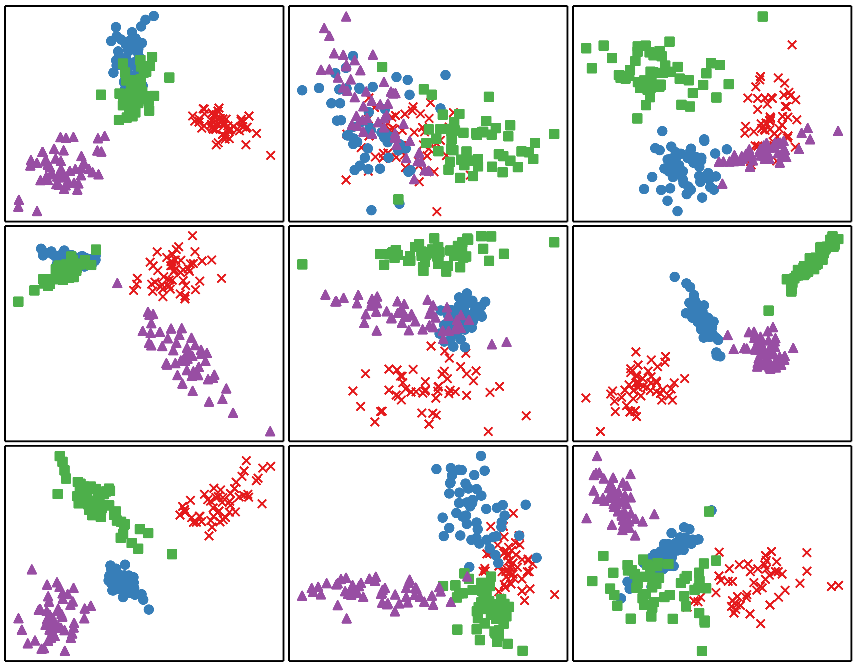


Figure 3.2.1: Random Projection of Gaussian Clusters in $\mathbb{R}^3 \rightarrow \mathbb{R}^2$

$\mathbb{R}^2$ space. In Figure 3.2.1 it is clear that even modest reductions of 3 dimensions to 2 dimensions can yield unwanted results.

3.3 Streaming Algorithms

Streaming Clustering is subset of clustering that consists of clustering data that arrives in a streaming fashion or as batches overtime. The goal is to perform a similar partitioning as the static case for other clustering methods with the requirement that we cannot randomly access seen data, nor see future data. CluStream [42] is a framework for clustering dynamic streaming data. CluStream stores micro-clusters, consisting of a over sampling of the k desired clusters. The k desired partitions are emitted at chosen time intervals, through some off-line clustering method, thus leading to on-line and off-line clustering phases. In addition addition, CluStream allows for dynamic clustering by introducing Pyramidal Time Frames that allow micro-clusters to split and merge over time. The CluStream framework is applicable to a variety of streaming clustering methods, and its on-line/off-line phase decomposition is the general framework adopted by Tree-Walk RPHash.

A common baseline algorithm for streaming clustering, called streaming k-Means [12, 35] derives much of its update process from the original Lloyd-type k-Means iteration. However, similar to CluStream, an over-estimate of micro-clusters is maintained, to be further clustered in an off-line step. Streaming k-Means' similarity to k-Means, is in its process of assigning incoming vectors to their nearest representative cluster. However due to the ground state of not having clusters, the decision whether to aggregate with an existing cluster or form a new one must be made. In general this can be done with some sort of inter-cluster similarity or intra-cluster dissimilarity metric. Some of the drawbacks to streaming k-Means is that it is highly dependent on the input order of the data stream.

In addition to random projection methods, recently streaming algorithms for data clustering are also considered [35, 76]. The general setup of these algorithms consists of a diminishing objective bound that tightens as the stream is processed. The processor updates the core-sets of pseudo-centroids when data are within the objective bound, and disregards data outside that bound. Streaming k-means algorithms perform well on data streams, but have the drawback of requiring that data be 'well-clusterable'. streamingRPHash uses a similar core-set approach, but instead of choosing core-sets based on computed distances from current centroids, it maintains a data structure of dense partitioned regions.

CSketch is a streaming algorithm for generating clusters over massive-domain datasets [5]. It shares much in common with streamingRPHash. In particular, CSketch applies the Count-Min Sketch data structure to update candidate centroids by updating the centroid location and not just a count. Incoming centroids must then search for candidate clusters in order to find the nearest centroid. In streamingRPHash we use the LSH decoding of projected points to immediately find the nearest candidate cluster in the Count-Min Sketch data structure. Furthermore, our hashing algorithm is intrinsic to the decoding step, and does not require a supplementary hashing scheme.

3.4 Tree Based Clustering

A later enhancement to RPHash referred to as adaptive LSH (ALSH), and its subsequent alternative tree-walk, off-line-step shares similarities with the decision tree clustering discussed in "Clustering Through Decision Tree Construction" [94]. Liu *et al* describe an implicit clustering approach that instead of grouping observations by inter-cluster similarity (such as WCSSE), attempts to differentiate dense clusterable data from uniformly distributed background data. Like decision trees, the CLTree algorithm recursively splits the data set into groups of similar observations. An unsupervised optimization condition is required to overcome the discord between unsupervised clustering, embedded in a supervised learning algorithm like decision trees. The condition, mention previously is to assume points belong to either a dense cluster or are part of some random uniform background noise. To differentiate clusters, from noise Liu *et al* optimize the information gain criterion C4.5 at each split. As such, CLTree iterates over the dimensions in the data space, splitting the dataset with the hyperplane that maximizes the information gain criterion for that dimension. A further modification of the information gain criterion, by which the algorithm “looks forward” a set number of levels of the tree, prevents decision plane from directly cutting clusters in half. The method is primarily concerned with lower dimensional clustering tasks for less than 20 dimensions, and has roughly linear complexity over dimensionality and dataset sizes up to 20 dimensions and 500000 records respectively. These results are on par with the results for RPHash. A caveat with their results is that each cluster was specifically generated to occupy a completely independent subspaces from the other clusters, and that subspace spans less than five dimensions.

Chapter 4

Motivations and Precursors of RPHash

In this chapter we will describe the motivations behind the RPHash algorithm, and give an overview of the RPHash algorithm's goals. The general idea of RPHash is motivated by a particularly degenerate case of the LSH-based k nearest neighbors (kNN) problem known as cr-NN. Briefly, the nearest neighbor problem is the problem of finding the closest analog in a database to an unknown data vector.

4.1 Bad kNN

In one-dimension, the NN problem is solvable in log-time via a binary search of the dataset. Similarly, in two-dimensions it is also solvable in log-time via the point location algorithm over the Voronoi partitioned dataset. The log bound unfortunately fails as dimensionality grows greater than 2, mainly due to the exponential growth of the Voronoi region set $n^{\theta(d)}$. Data structures such as k-d Trees [27] can be constructed to provide log-time average search complexity for uniform datasets up to about 32 dimensions ($N >> 2^d$, $N =$ size of Database). Beyond that however, k-d tree search tends to degenerate into the linear scan search algorithm. Naively this problem can be solved by the linear scan algorithm (shown in Algorithm 1) which consists of sorting a list of distances between the query vector and all vectors in the dataset then returning the nearest k vectors.

Definition 4.1.1 (Nearest Neighbor)**.** *[115]*
Given a set of vectors P in R^d and query vector q return k vectors $p \subseteq P$ such that $p =$ Argmin$\{dist(p', q)\}$, where dist is some metric function.

Algorithm 1: Linear Scan, Query q, dataset X

```
$k$NN= $X[0:k]$
forall the $x \in X[k:n]$ do
    if $\|x, q\|_2 < \|q, kNN[0]\|_2$ then
        $k$NN$[0] = x$
        $k$NN= $sort(k$NN$)$
Return: $k$NN
```

This algorithm requires that we compute distances between every vector in the dataset and our query vector θnd. If we could somehow omit vectors that we know are not anywhere near our input vector, we could certainly speed up our calculation and possibly solve kNN in a sub-linear time. This is the intuition of the LSH-kNN algorithm. Previously discussed LSH functions can be used to winnow the field of candidate near neighbors that need to be searched via the linear scan method. A considerable amount of computation can be avoided by performing this large volume reduction on the query dataset provided some error is tolerable. The basic outline of LSH-based kNN is given in Algorithm 2 and Algorithm 3.

Algorithm 2: Preprocessing on Dataset X

```
forall the $x \in X$ do
    id = LSH_Hash(x)
    $T$[id].add($x$)
Return: $T$
```

Algorithm 3: Query k nearest to q in X, using hashtable T

```
id = LSH_Hash($q$)
Return: linear_scan($T$[id],$k$)
```

Assuming the LSH_Hash function is sufficiently discriminative, an appropriately chosen locality sensitive hash functions can achieve a complexity bound of $\theta(n^\rho + log(n))$ where ρ is determined by the selectivity of the hash function. LSH-kNN is a highly successful solution to the nearest neighbor problem in high dimensions. However, a degenerate case of the LSH-kNN structure occurs when the candidate field becomes quite large. These degenerate cases represent an outlier condition for cr-NN algorithm, and often contributes to the algorithm's worse-case performance. While this outlier example presents a problem to cr-NN, viewed under a different problem requirement, it becomes beneficial for quickly identifying candidate density modes.

In Figure 4.1.1 we give give a toy example of the basic concept for or proposed Random Projection Hash

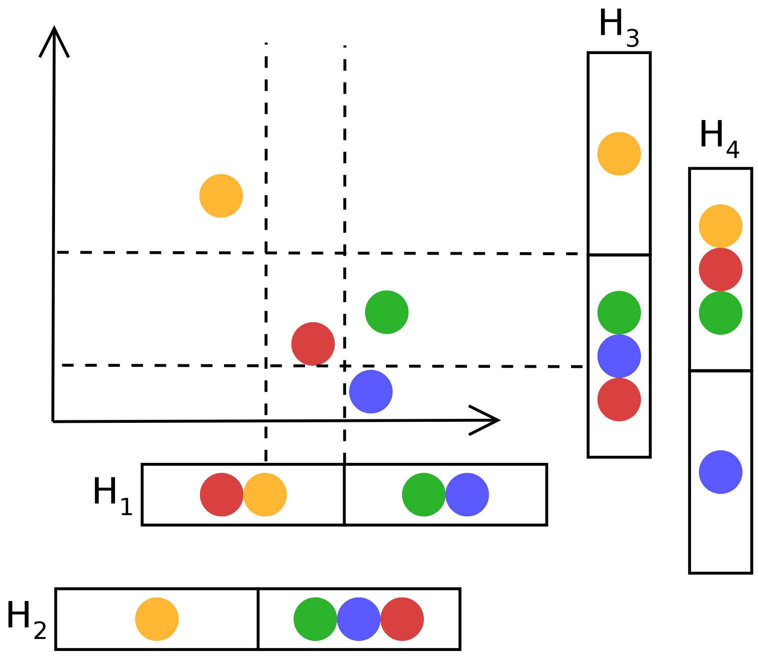


Figure 4.1.1: Example of an orthogonal LSH on $\mathbb{R}^2$ Data

algorithm. The example contains 4 vectors, 3 of are somewhat close to one another in $\mathbb{R}^2$. The space is randomly bisected along 4 partitioning plains, corresponding to 4 hash functions and 8 regions. Inspecting the high cardinality hash buckets, we see they often, but not always, correspond to high density regions. In this case our small cluster of 3 items. The correspondence between high cardinality hash buckets and high density partitions, forms the basis of our algorithms.

4.2 Cardinality Shift Clustering

Previous work on graphics processing unit (GPU) accelerated cr-NN [38] lead to a generous speedup per GPU core for bulk NN search. One problem that could take advantage of this acceleration to bulk NN search is the LSH accelerated Mean Shift algorithm of Li *et al* [91]. Although this still remains an interesting use case, a more interesting case was investigated where the generative nature of decoder functions could be used to optimize interprocess communication in distributed systems. In this dissertation, vectors would be assigned to processing nodes using the LSH function, resulting in better vector locality. Furthermore this proposed method would use LSH bucket cardinality as a proxy for the computational and communication inefficient mean shift step shown in Figure 4.2.1.

$$\vec{x} = \frac{\sum_{x_i \in N_(x)} K'(x - x_i)\vec{x}}{\sum_{x_i \in N_(x)} K'(x - x_i)}$$

Figure 4.2.1: Mean-Shift Iteration [91]

This algorithm is referred to as *Cardinality Shift Clustering* (CSC). The overall iteration of CSC is similar to the mode finding step in the mean-shift clustering algorithm [41,48,63] with a focus on scalability and low compute node interdependency. To get better vector-node locality and minimize communication overhead a two step course-grain and fine-grain shift is considered.

4.2.1 Fine-Grain Iteration

The fine-grain iteration is similar to mean-shift clustering update step except that it occurs on a finite set of vectors contained within the bounds of a the partition of the well known Leech Lattice. Due to this lattice's fixed dimensionality, conversion between vectors of $\mathbb{R}^d$ and $\mathbb{R}^{24}$ must be performed by random projection and the pseudo-inverse(or ϵ-suitable inverse) of the projection [31]. Furthermore, this clustering occurs on a per-compute node basis (Figure 4.2.2, region **II**) and each compute node is assigned a subset of lattice regions (Figure 4.2.2, region **I**). The lattice region IDs are generative and can be assigned either by truncating the ID mod the number of available compute nodes or some other more advanced load balancing technique.

Using techniques from Andoni [20] and an addition from Carraher [39], computation of the function $N(\cdot)$ of near neighbors for a given point can be performed in nearly optimal $\Theta(n^\rho)$-time. For a Leech lattice hash function where $d \leq 24$, $\rho \approx 0.2671$ [20], while linear search requires $\Theta(dn)$-operations. This gives an internal clustering complexity of $\Theta(kn^{1+\rho})$ where k is the maximum neighborhood size. By adding the random projection aspects to this iteration, it is believed we will achieve some tempering to the error surface across multiple projections which may help avoid some local minima of the density modes by smoothing out point discontinuities. Furthermore, the sigma value for an Gaussian, $N(\mu, \sigma)$ distribution in the random projection may allow us to control tempering.

4.2.2 Course Grain Iteration

Due to the $\Theta(n^2)$ complexity of mean-shift clustering, we attempt to mitigate vector to vector communication scalability by replacing it with a lattice-region to lattice-region course communication step. All lattice-regions contain a finite set of vectors. In CSC we use the cardinality of the sets as weighted components for generating the course-grain shift vectors, again using the standard kernel mean shift algorithm (Figure 4.2.2, region **III**). The shift vectors are then broadcast to all computing nodes in the system.

4.2.3 Update Step

Per compute node, the update step consists of updating all vectors current positions by applying the results of the course-grain mean-shift vectors within a lattice region for a random projection (Figure 4.2.2, region **IV**). After this, we invert our random projection matrix and apply it to all the vectors giving us a set of (number of random projections) vectors for each vector in our original $\mathbb{R}^d$ space. We then average the vectors various random projections to attain an d dimensional representation that is near the actual result of a full mean-shift

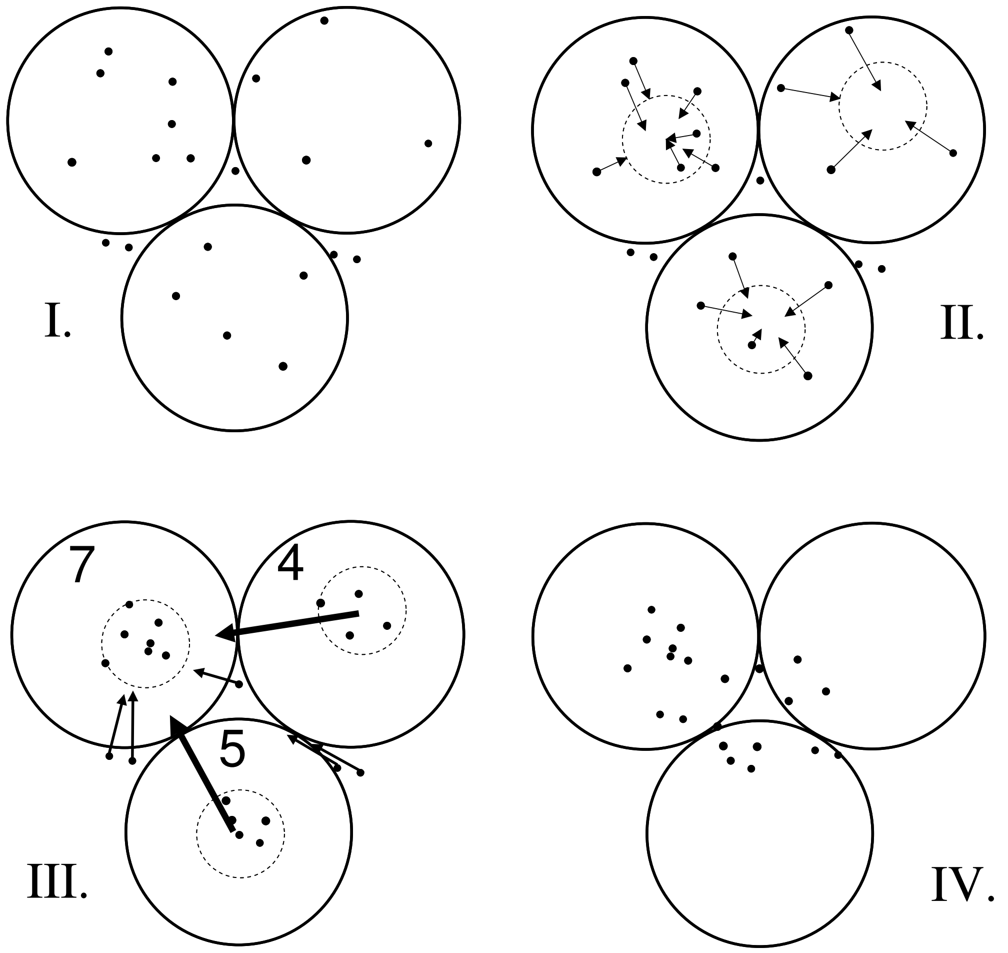


Figure 4.2.2: Cardinality Shift Clustering
I. Initial lattice assignment, **II.** Fine Grain Iteration, **III.** Course Grain Iteration, **IV.** Updated Positions

clustering iteration for that vector. Once the representations of the shifted vectors are averaged in $\mathbb{R}^d$, we again project the vectors down to 24-dimensional space. The assumption here is that the global shift will have altered the lattice-regions for vectors near the borders of a region. The vectors that have new lattice-region assignments first check the current compute node for the new lattice-region assignment(a mod lattice ID can easily be tested here). If they are not contained in the current compute node, the vector must be sent to the compute node containing its new lattice-region assignment. The amount of transfer needed is hopefully minimal and point-to-point.

4.2.4 Stopping Condition

The number of lattice-region assignments should naturally decrease as vectors are pulled toward higher density lattice-regions. By counting the number of inter-compute node vector communications (the number of vectors whose lattice-region assignments have changed), we can get an estimate of the overall entropy remaining in the system on a per step basis. As with other machine learning algorithms, the entropy will likely decrease with an inverse power-distribution. Suitable stopping conditions are well studied and have been defined for distributions based on derivative and statistical analysis.

While this method may still have some use as a communication optimization technique for more robust clustering algorithms such as distributed mean-shift and filtration algorithms for topological data analysis, its complexity may be unneeded. The first step of CSC alone turns out to be a sufficient process for general data clustering. We will describe the setup of this technique in the following section.

4.3 Big Bucket Search

In the initial development of CSC came a simple test application. In the test application, a test set $S = \{S_1, ...S_c\}$ of clusters where $S_j = \{x_{j,1}, ..., x_{j,(n/c)}\}$ and $x_{ij} \in \mathbb{R}^{24}$ were generated from C Gaussian distributions with $\mu = 0, \sigma = \frac{1}{\sqrt{24}}$. The vectors from the C clusters were processed using the Lookup-DB generation step of Andoni's [20] Leech Lattice based cr-NN algorithm. The full DB was then scanned through to identify the largest ϵC buckets, corresponding to a representative hash ID. For low variance, the largest buckets corresponded directly to the nearest points of the generated centroids. This simple test demonstrated a basic principle of the RPHash algorithm: locating low variance Gaussian Centroid. As

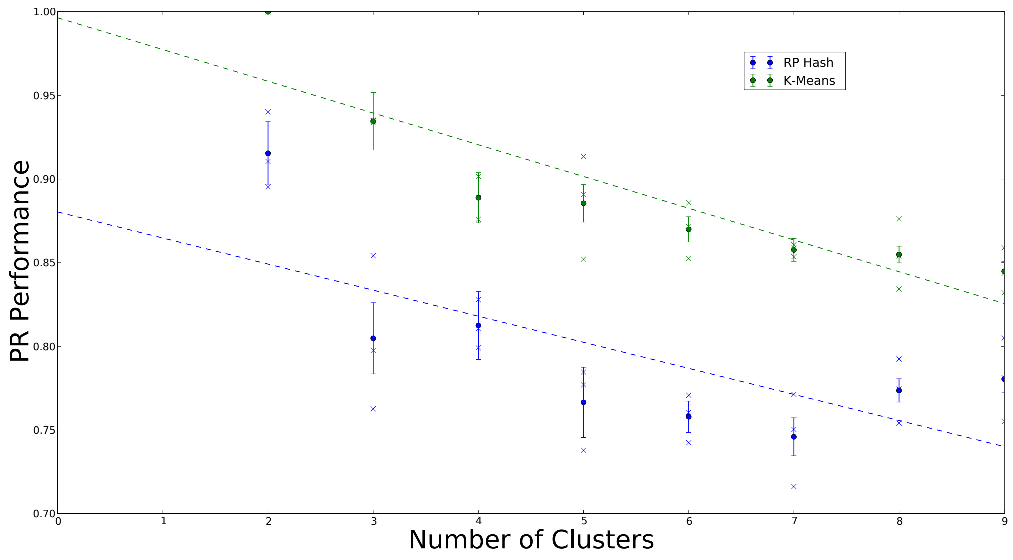


Figure 4.3.1: PR "Big Bucket" Count

Precision Recall For "Big Bucket" and Projected K-Means As a Function Inter-Cluster Variance

shown in Figure 4.3.1 this basic algorithm works for low-variance Gaussian data, some modifications are needed for clustering real world data.

In signal data, a similar process making use of windowing and kernel functions to estimate kernel density is the Parzen's kernel density estimator [113]. Kernel density estimation and clustering are related through the similar concepts of classes and clusters. While clusters are often defined by the data, classes can be defined by a set of constraints, both of which can be resolved to the other. Thus, Parzen's offers a solution to clustering, however dynamic programming solutions such as this are restricted to 1-dimensional data.

Due to our goal of providing a general clustering solution to dimensions beyond what Parzen's is applicable to, we must extend the kernel windowing method to arbitrary space partitioning. Immediately, one could apply a metric partitioning scheme such as $\mathbb{D}_n$ to the data space, and keep track of the largest partitions. In fact this is the principle concept behind many density scan based methods. These methods are efficient for the lower dimensional regimes but tend to break down as d increases beyond 10-18 dimensions. This result is similar to the efficiency breakdown of k-d tree solutions to kNN, and is often regarded as an effect of the curse of dimensionality (*COD*)— distance metric loosing its specificity. We again adopt another technique from approximate kNN, and apply an ensemble of projections to high dimensional data. These observation along with various modifications, are the basis for the RPHash Algorithm whose description and analysis constitute the remainder of this work.

Chapter 5

Implementation

In this section we will describe the two main variants of the RPHash algorithm. First we discuss the original 2-Pass RPHash algorithm designed for distributed processing on platforms such as Hadoop MapReduce. Next we discuss the streamingRPHash variant for streaming data designed for platforms such as Spark.

5.1 Implementation Overview

RPHash is a distributed algorithm for dense region and micro-cluster identification suitable as a precursor to more robust clustering algorithms (k-Means, LDA, Mean Shift) or as a standalone approximate clustering algorithm. In the (RPHash) algorithm, both approximate and randomized techniques are employed to provide a stochastic element to our clustering algorithm. To combat the curse of dimensionality (COD), RPHash performs multi-probe, random projection with Gaussian blurring of high dimensional vectors to the unique partitions of the Leech Lattice (Λ_{24}) [20].

5.1.1 Overview of the RPHash algorithm

The basic intuition of RPHash is to combine multi-probe random projection with discrete space quantization. Following this intuition, near-neighbor vectors are often projected to the same partition of the space quantizer, which is regarded as a hash collision in LSH parlance. As an extension, multi-probe projection ensures that regions of high density in the original vector space are projected probabilistically more often to the same partitions that correspond to density modes in the original data. In other words, partitions with high collision rates are good candidates for cluster centroids. To follow common parameterized k-means methods, the top k densest regions will be selected.

According to the *JL* lemma, the sub-projections will conserve the pairwise distances in the projected space for points with ϵ-distortion in which the size of the dataset is proportional to the logarithm of the number of dimensions in the randomly projected subspace. In addition to compressing a dataset to a computationally more feasible subspace for performing space quantization, random projection can also make *eccentric* cluster more spherical [55, 129].

Discrete space quantizers play a central role in the RPHash algorithm. The sequential implementation of the RPHash algorithm will rely on the efficient Leech lattice decoder of Vardy, Sun, Be'ery, and Amrani [17, 18, 120, 127] used as a space quantizer. The lattice decoder implementation relies on the decomposition of the binary $(24, 22, 8)$ extended Golay Code into 4 partitions of the $(6, 3, 4)$ quaternary hexacode and its relation to the Leech Lattice as a type B lattice construction. This decomposition and relation to the Golay Code provides a real worse case decoding complexity well below the asymptotically exponential bound for trellis decoders as the dimension d increases [122]. The Leech lattice is a unique lattice in 24 dimensions that is the densest lattice packing of hyper-spheres in 24 dimensional space [49, 90]. While the Leech lattice has many exceptional properties, of particular importance to RPHash is that it provides the densest regular lattice packing possible in 24 dimensions. It was shown to be nearly optimal among even theoretical non-regular packings [46] and more recently proven to be the densest packing achievable in $\mathbb{R}^{24}$ [47]. The 24 dimensional subspace partitioned by the Leech Lattice is small enough to exhibit the spherical clustering benefit of random projection. Low distortion random embeddings are also feasible for very large dataset ($n = \Omega(c^{24})$) to lower dimensions (shown in [26]), however for projective clustering at much lower embeddings, these embeddings risk exhibiting the the occultation problem of [126]. Furthermore, the decoding of the Leech lattice is a well studied subject with a constant worse case decoding complexity of 331 operations [127].

Space quantizers have hard margin boundaries and will only correctly decode points that are within the error correcting radius of its partitions. This is an an issue found in approximate nearest neighbor search [20, 111] and is overcome in a manner similar to Panigrahy [111] — by performing multiple random projections of a vector and then applying the appropriate locality sensitive to provide a set of hash IDs. Using multiple random projections of a vector allows the high dimensional vector to be represented as ‘fuzzy’ regions that are probabilistically dependent on the higher dimensional counterpart. From Panigrahy [111], the requirement of ($\Theta(log(n))$) random projection probes is given to achieve c-approximate hash collisions for the bounded radius, r-near vectors. Random projection probing adds a $\Theta(log(n))$-complexity coefficient to the clustering algorithm. The top k cardinality set of lattice hash ID vector subsets represent regions of high density.

Projected clustering of representative cluster centroids will not in general be correlated with other projections of data into projected cluster centroids. To recover data from the projection step, we must map projected vectors back to their original un-projected data space counterparts. The original data space vectors can then used to compute centroids corresponding to the clusters in the projected space. Figure 5.1.2 shows an example of this process for 3 projection probes of $\mathbb{R}^3 \rightarrow \mathbb{R}^2 \rightarrow \mathbb{R}^3$. Any off-line clustering algorithm can be performed to resolve the overestimate of k the number of desired clusters, effectively merging the $k \times$ number of projections representations of centroids in the original data space.

An outline of the steps in the main steps of the RPHash algorithm (Algorithm 4) is given below. One way that the RPHash algorithm achieves scalability is through the region assignment. Clustering region assignments are performed by decoding vector points into partitions of the Leech Lattice.

In most cases the problem space will not be exactly 24 dimensional. The Johnson-Lindenstrauss (*JL*) lemma and subsequently, random projection provides a solution to this problem and provides additional benefits (See Section 2.4). *JL* states that for an arbitrary set of n points in m dimensional space, a projection exists onto a d-dimensional subspace such that all points are linearly separable with ϵ-distortion following $d \propto \Omega(\frac{log(n)}{\epsilon^2 log 1/\epsilon})$. Although many options for projections exists, a simple and sufficient method for high dimensions is to form the projection matrix $r_{ij} \in \mathbf{R}$ is $m \times d$ as follows:

$$r_{ij} = \begin{cases} +1, & \text{with probability } \frac{1}{6} \\ 0, & \text{with probability } \frac{2}{3} \\ -1, & \text{with probability } \frac{1}{6} \end{cases} \quad [2]$$

Although the Leech lattices partitions provide optimal sphere packing in 24 dimensions for regular lattices, (an unrelated version of the curse of dimensionality) the overall density of the lattice is sparse at 0.001930.

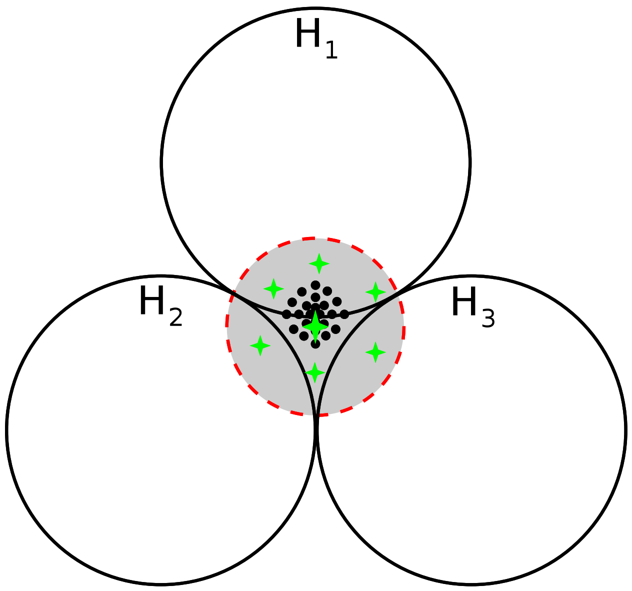


Figure 5.1.1: Gaussian Blurring to Fill Sparse Lattices.

To overcome this, RPHash "blurs" the projected data by apply shifts to projected vectors to more fully cover the $\mathbb{R}^{24}$ subspace and performing multiple probes of the Leech Lattice partitions in addition to the vector v_{24} : $v_{24} = \{v_{24}, v_{24} + N(0,1)_{24}...\}$. The approximation of a random projection is computationally efficient for large datasets, and unlike a truly Gaussian projection matrix, yields a semi-positive definite projection matrix, that is useful in showing the convergence of RPHash.

5.1.2 Blurring

Another way to overcome the hard-margins region assignments induced by a discrete lattice is to generate random points in an $r/2$ radius around the projected vector. This perturbation about the projected vector results in an approximate probability density region around the data point. In order to achieve an approximate radius probing in all dimensions, we add a Gaussian vector having elements $\{n_1, n_2, ..., n_d\} \in R$, where $\|R\| \approx r$ and is composed of elements $n \in N(0, r)$.

5.1.3 Multi-Projection Re-Association

Re-association with the original vectors is used to recover the final centroids as shown in Figure 5.1.2. However, we must now resolve the common case where low dimensional representations of re-associated vectors are unrelated to those of other projections. A simple solution found in [35] and in other streaming

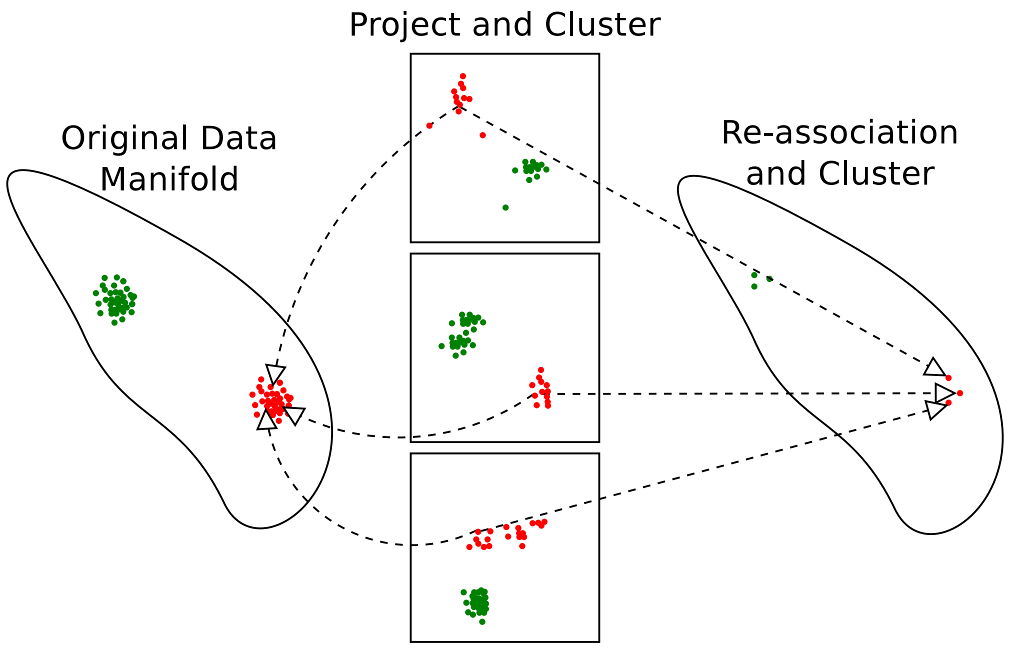


Figure 5.1.2: Multiple projections $\mathbb{R}^3 \to \mathbb{R}^2 \to \mathbb{R}^3$

algorithms is to merge the over-estimated centroid set in an off-line step such as weighted k-means. The off-line step merges the $k \times$ number of projections representations of candidate centroids but in the original dimensional embedding.

A problem known as occultation arises when disjoint clusters in a d-dimensional space have non-disjoint projections on lower-dimensional subspaces of $\mathbb{R}^d$. Specifically, the probability of a u-occultation for the d-dimensional space projected to 1-dimensional subspace is: $pr(u) = 1 - \frac{2}{\pi} arcos(\frac{r_1+r_2}{d-c})$, where r_1, r_2 are radii of the respective clusters, and $d - c$ is the distance between them [126]. The method employed in [126] is to repeat the projection until a non-occulting projection is found. The rate of convergence for finding a non-occulting projection is given as:

$$\lim_{d\to\infty} 1 - \left(\frac{2(r_1 + r_2)}{\pi\|d - c\|}\right)^d = 1$$

which is exponential in d. Recognized in [126], this bound is slightly more favorable for an orthogonal projection than the convergence rate of multiple single projections, as the distinct random projections are not always linearly independent. For certain decoders like the Leech Decoder and Spherical LSH, the required

24 and 32-dimensional projection is nearly orthogonal when the projection matrix is constructed of Gaussian random variables [129].

Algorithm 4: 2-Pass RPHash Algorithm

```
Data:
$K$ - number of clusters
$X = \{x_1, ..., x_n\}$, $x_k \in \mathbb{R}^m$ - data vectors
$C$- a $k$-HH counter
$\mathbb{H}$ - LSH Function
$\mathbb{P} = \{p_1, ...p_n\}$ - set of projection matrices
$L = \{\{\varnothing\}...\}$
$M = \{C, [0, ...0]\}$
forall the $x_k \in X$ do
    forall the $p_i \in \mathbb{P}$ do
        $\tilde{x_k} \leftarrow \sqrt{\frac{m}{d}} p_i^\intercal x_k$
        $t = \mathbb{H}(\tilde{x_k})$
        $L[k][i] = t$
        $C$.add($t$)
forall the $x_k \in X$ do
    forall the $c_i \in C.top(K)$ do
        if $L[k] \cap M[i][0] \neq 0$ then
            $\Delta = M[k] - x_k$
            $M[k] = M[k] + \Delta/count$
            $L[k]$.add($M[i][0]$)
Result: $M$
```

In addition to Achlioptas efficient random projection method for databases, a further reduction in the number of operation required for random subspace projection called the Fast Johnson-Lindenstrauss Transform (FJLT) [11, 13, 54] is currently an active area of research. FJLT, and similar nearly optimal projection methods, utilize the local-global duality (Heisenberg Principle) of the Discrete Fourier Transform to precondition the projection matrix resulting in a nearly optimal number of steps to compute an ϵ-distortion random projection [11, 13, 54]. A sub-linear bound on the number of operations required for the dominant projection operation may further improve RPHash's overall runtime complexity. However it only becomes beneficial when the vector dimensionality d is very large. For many clustering problems, this level of dimensionality is uncommon.

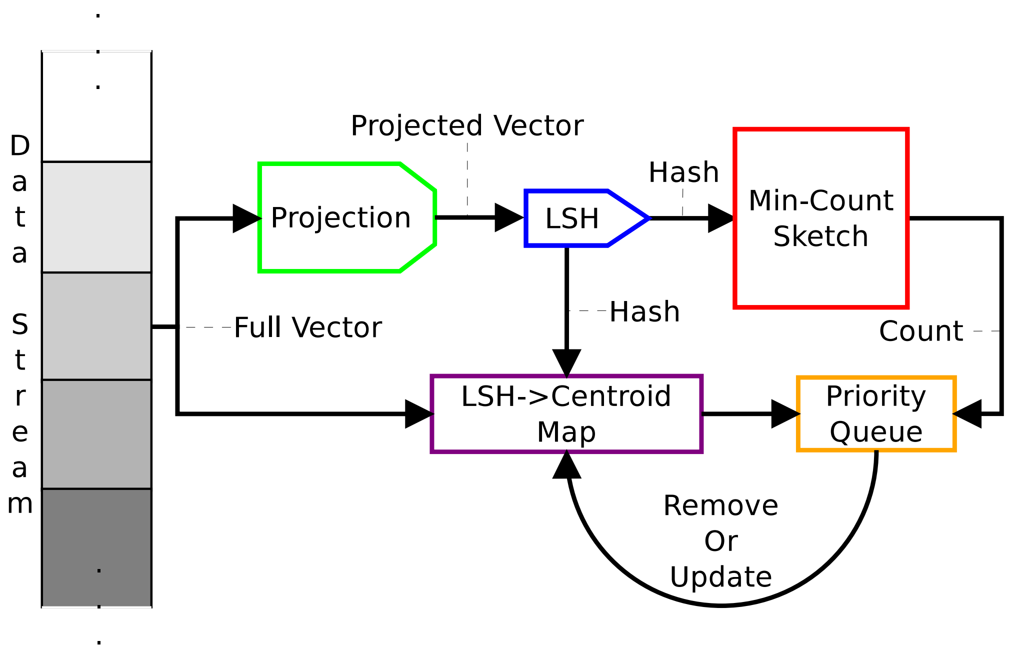


Figure 5.1.3: Streaming RPHash Diagram

5.1.4 Overview of streamingRPHash

streamingRPHash is a stream clustering variant of the original 2-Phase RPHash algorithm of [40] which employs both approximate and randomized techniques to solve the k-means clustering problem. To combat the curse of dimensionality, streamingRPHash performs multi-probe, random projection with Gaussian blurring on high dimensional data vectors [20].

The general idea of streamingRPHash is motivated by a particular bad-case result in LSH based Nearest Neighbor (LSH-NN) search where a single bucket contains an unprecedented number of candidate nearest neighbors. This is a problem in LSH-NN as the bucket then has to be linearly scanned for the optimal solution. However, these degenerate buckets can also be viewed as locally dense regions or density modes in the data. Histograms offer a familiar 1-d analog of our clustering solution. A further advance of this considers not just the points within an arbitrary histogram boundary, but relaxes the boundaries with a kernel based window function to smooth the overall population density estimation. A staple in kernel density estimation, the Parzen's window is a key motivation of the streamingRPHash algorithm.

Due to the goal of providing an approximate solution to $\mathbb{R}^d$ for $d \gg 1$, we must extend the windowing and estimation techniques of Parzen's window to high dimensional space. We could immediately apply the d-dimensional hypercube partitioning and then count partition memberships. But this method would suffer from many complexities arising from the curse of dimensionality. Similar to the issues encountered with k-d Tree NN [102], it will also be computationally prohibitive and restricted by memory limitations. streamingRPHash offers to mitigate these issues by optimizing the approximate solution instead of the exact.

streamingRPHash, being motivated in part by LSH-NN, uses locality sensitive hash functions to partition the data space evenly. After both accuracy and time complexity tests, the 32-orthoplex spherical hash function was found to be optimal across a wide range of datasets and system configurations.

An outline of the steps in the streamingRPHash algorithm are discussed below and summarized in Algorithm 5. Some of the aspects of its function in regards to randomness and approximation are highlighted here. First, we discuss the generative nature of its region assignment. Dense regions are identified by decoding data vectors using a set of LSH functions. In general the problem space will not match the locality sensitive hashing subspace. The previously described Johnson-Lindenstrauss (*JL*) lemma offers a solution to this problem at the cost of a matrix vector product. Although *JL*-lemma requires an $m \times d$ matrix of Gaussian random variables be formed, [2] suggests an approximate method to form the random projection matrix whose elements $r_{ij} \in \mathbf{R}$ have values from $\{1, 0, -1\}$ with probabilities $\{\frac{1}{6}, \frac{2}{3}, \frac{1}{6}\}$ respectively. A further computational reduction is then achieved by treating the matrix-vector product as a linear scan over the non-zero elements of the projection matrix, allowing us to effectively skip $\frac{2}{3}$ of the projection computation, with the remaining $\frac{1}{3}$ operations consisting only of scalar additions. Following this computation, a scaling factor of $\frac{1}{\sqrt{n}}$ is also needed to preserve approximate distances between projected vectors.

The subspace spanned by the 32-orthoplex is well below the requirements suggested by the *JL*-bound for individual vector discrepancy. Fortunately, the effectiveness of projected k-means has been demonstrated successfully for projections well below the *JL*-bound (and well below $d = 32$) [26, 31]. However, internal tests show precision-recall performance drops as projection space dimensionality decreases. To boost the overall data retention of our clustering, we opt to perform multiple projections on original data vectors, followed by a reconstruction step. Similarly, a 'blurring' step is used to increase the overall hash collision rate for nearby vectors. streamingRPHash 'blurs' the projected data by applying shifts to projected vectors which help to overcome the hard margins of the LSH decoder.

5.2 streamingRPHash Algorithm

streamingRPHash follows the same structure as the original RPHash algorithm such as multiple random subspace projections and their Re-Associations, Gaussian Blurring and LSH Functions. However, the difference is in the way in which streamingRPHash identifies and updates frequent density modes. Due to the streaming data model's requirements that an incoming data stream cannot be randomly accessed, or completely stored, we can only store a sketch of the incoming data. Furthermore, streamingRPHash cannot store the entire dataset, so the set of candidate centroids must be bounded. Fortunately, the Count-Min Sketch and its k-HH implementation fulfill both of these requirements. The sequential version of the streamingRPHash algorithm is given in Algorithm 5.

Algorithm 5: streamingRPHash Algorithm

```
Data:
$k$ - number of clusters
$X = \{x_1, ..., x_n\}, x_i \in \mathbb{R}^m$ - data vectors
$M$- lsh_key $\rightarrow$ centroid map
$C$- cm-sketch data structure
$T$- CM-Sketch based $\epsilon k$ bounded priority queue
$R = \{r_1, ..., r_{\log_2 d}\}, r_i \in \mathbb{R}^d$ - Gaussian blurring vectors
$\mathbb{H}(\cdot)$ - LSH Function
$\mathbb{P} = \{p_1, ...p_n\}$ - set of $n, m \times d$ projection matrices w/ JL property
forall the $x \in X$ do
    forall the $p \in \mathbb{P}$ do
        $\tilde{x} := \sqrt{\frac{m}{d}} p^\intercal x$                    // Random Projection
        forall the $r \in R$ do
            $t := \mathbb{H}(\tilde{x} + r)$                  // LSH Hashing
            $C$.add($t$)
            if $t \in M$.keys then
                $M[t].wadd(x)$                  // Weighted add
            else
                $M[t]$ =new centroid($x$,C.count(t))
                $T$.insert($M[t]$)
            M.remove(T.pop())                  // remove least
Result: OfflineWeightedClusterer($M$.items)
```

5.3 k-HH

Streaming Data and Approximate k-Heavy Hitters

To apply streamingRPHash to data streams, the candidate centroid set must be bounded and able to quickly respond to most frequent itemset queries. Formally, the k-Heavy Hitters (k-*HH*) problem is the problem of identifying the k most frequent items in a dataset [50]. It can trivially be solved using a histogram with $\theta(n)$ space. In the case of data streams, space requirements are potentially unbounded, making the histogram approach infeasible. In [15], the related distinct counting problem for streams is shown to require $\theta(n)$ space in the worst case. As with many cases involving unbounded memory requirements, approximate approaches have been proposed such as [85, 101, 106]. A thorough survey of approximate distinct item counting can be found in [52] where counting algorithms are divided into two categories: *count based* and *sketch based*.

One of the earliest streaming solutions for the approximate k-*HH* algorithms is the Count-Min Sketch data structure [51]. Figure 2.3.1 shows the primary data structure for 3 elements (X, Y, Z) and k hash functions. Count-Min sketch can be used to solve the approximate k-*HH* problem with only the addition of a priority queue using $\theta(\frac{1}{\epsilon}log\frac{1}{\delta})$ space with error $(1-\epsilon)f$, where f is the minimum frequency $\frac{m}{k}$ to be considered frequent.

While the code base for streamingRPHash contains implementations for several LSH methods (including E_8, Leech, and Spherical), our experimental studies show that the best overall performance is achieved with Spherical. Hence the experiments in this dissertation report results with the spherical 32-orthoplex decoder of [123]. The spherical orthoplex decoding consists of a projection to $32d$ space followed by a vector normalization step in order to have the vector lie on the surface of a hypersphere. Next, a random rotation is applied to the 64 basis vectors of the 32-orthoplex. The basis vector nearest to the projected vector is then used as the vector's representative, and the corresponding bits resulting from the natural ordering of basis vectors are used to define a $log_2(64)$ partial decoding. Subsequently, two similar additional hashings are applied to generate a total 18 bits of entropy.

5.4 Data Security

Data Security: At no added cost

Recent United States government initiatives pushing for the large scale availability of data resources have made vast quantities of de-identified health information available to the public. These resources however have prompted advances in attacks on de-identification of whole genome sequence data. Such attacks have been used to associate, thought to be, anonymous medical records with specific individuals [75]. Similar de-anonymization attacks [60, 108] along with a presidential commission (privacy and progress in WGS) have prompted a need for better data security of medical records data. The RPHash algorithm provides an intrinsic solution to this problem in both the distribution of data among servers as well as during the communication steps required by the algorithm.

While attempting to mitigate communication restrictions, RPHash intrinsically provides some security in the data it exchanges. Previous attempts at securing data in distributed systems required additional cryptographic steps [93]. Namely, the randomly projected centroid IDs, and the aggregation of only the k-largest cardinality vector sets. Non-distributed data clustering requires the entire dataset to reside on the processing system and distributed methods often require communication of full data records between nodes.

In the subspace projection step of RPHash, nearly-orthogonal random projection is utilized as a destructive operation, providing vector anonymity. As a consequence of projecting the real data vectors to a random subspace via a near, but not completely orthogonal matrix, destructive data loss occurs providing a cryptographic "trapdoor" function. The data loss is an intrinsic part of the RPHash clustering algorithm that has little adverse effect on its model generation and subsequent recall accuracy. Given the likelihood that RPHash is applied to a dataset where the number of vectors n is much greater than the desired k centroids, recovering an individual's private information would require knowledge of $\frac{n}{k}$ (on average) records in the representative centroid.

5.5 Software Implementation Details

RPHash and Variants of RPHash are provided in a variety of languages (Go, Python, and Java), however the primary testing and development is done on the Java variant. All software development and collaboration was tracked using the git version control system and is made available on-line:

- `https://github.com/wilseypa/rphash-java`
- `https://github.com/wilseypa/rphash-spark`
- `https://github.com/wilseypa/rphash-golang`
- `https://github.com/leecarraher/PyRecRPHash`

The RPHash test frameworks is designed in `java` following an object-oriented approach. Additional performance metrics and algorithms for comparison are provided by various Matlab packages. Common functional units with the similar functions are designed as implementations of general abstract classes (i.e `ArrayList` implements `List` functions) Examples of abstract functional units that are described in more detail are `AbstractProjector`, `AbstractClusterer` , `AbstractDecoder`, and `AbstractCounter`. The methods contained in each interface pertain to their function within the RPHash `java` Framework.

The abstract class of all RPHash variants, which includes other clusterers for comparison is the `AbstractClusterer` class. For algorithms specific to streaming datasets, a further abstraction of the `AbstactCluster` method is provided, `StreamingClusterer` which amends the abstract clusterer with function prototypes for adding vectors in a stream and invoking the off-line process. Implementing clusterers in this framework allows us to easily design tests that are applicable to all RPHash variants and other clustering algorithms.

For implementation of the offline clustering and post processing clustering steps, multiple standard clustering algorithms were implemented. These methods all implement the `AbstractClusterer` interface. In particular the following offline algorithms are included in the code base: Hartigan and Wong k-Means [77], agglomerative clustering with single, complete average, and wards method link strategies, Mean-Shift [48], Maximum Likelihood [78], DBScan [62], k-Means++ [24], Braverman *et al* Streaming k-Means [35] , Ailon *et al* Streaming k-Means [12], as well as a dummy clusterer that simply samples the dataset. All clustering methods can be wrapped in a multi-reseeding method that optimizes within-cluster sum of squared error.

Multiple LSH algorithms were implemented for testing and comparison as implementations of the `AbstractDecoder` class. Decoders include the P-Stable-Distribution Decoder [57], Spherical LSH

Decoder [123], Leech Decoder [18, 21], E_8 Decoder [49], the classic Dihedral Group Decoder [49], Origin Hamming Decoder [82], Binary Golay Decoder [18], and or proposed Depth Probing Decoder. All Decoders are composable into multi-level decoders, through implementation of the multi-decoder wrapper.

`AbstractProjectors` consist of the DB-Friendly random projection method of [2], a the Gaussian variate based projection [129], a bitwise DB-Friendly random projection method, the Fast Johnson-Lindenstauss Transform [11], and the classic Principle Component projection.

A set of counting and sketching algorithms are also implemented in the RPHash test framework and are implementations of the `AbstractCounter` class. The included counting algorithms are: the Count-Min sketch [50], and its k-Heavy Hitter implementation [51], Lossy counting [106] and a hash table based exact counter. The Count-Min implementation is adapted to also support vector sums and decay rates for streaming data in which clusters are regarded as temporal structures.

For testing RPHash, a framework of statistical performance metrics and data generators and readers is provided. The generators consist mainly of Gaussian cluster generators, that can be sparse, heteroscedastic, and full or streaming. All data readers implement a simple matrix format that supports both binary and ASCII data with uncompressed and LZ compressed data.

Chapter 6

Data Aware and Adaptive LSH

In this section, we explore the drawbacks of fixed lattice and self-normalized partitioning schemes previously presented in Chapter 4. We first define the problems that arise due to non-data aware LSH partitioning, present real world analysis of these problems, and then discuss some techniques for mitigation. Finally, we present a new LSH function that dynamically adjusts the partitioning regions, minimizing false positive collisions and improving the selectivity and stability of the RPHash algorithm.

6.1 Problems with Uniform Partitioning

LSH schemes work best when the data range is known beforehand. A noticeable drawback in our use of fixed and lattice based partition regions, is that the scaling of the partitioning constellation must be "useful". By "useful" we mean that the size of the partitioned regions must provide a region assignment that is not overly dense: in which no two vectors occupy the same partition, while also not being too sparse: in which all vectors occupy the same region. Figures 6.1.1, 6.1.2, 6.1.3 contain an example partitioning of Gaussian, homoscedastic cluster data for a sparse lattice (Figure 6.1.1), "useful" (Figure 6.1.2), and dense lattice (Figure 6.1.3).

6.1.1 Shaping

One issue encountered with using a uniformly distribution decoding lattice was a misappropriation of decoding symbols in the Quadrature Amplitude Modulation (QAM) constellation (Figure 6.1.4). The goal of

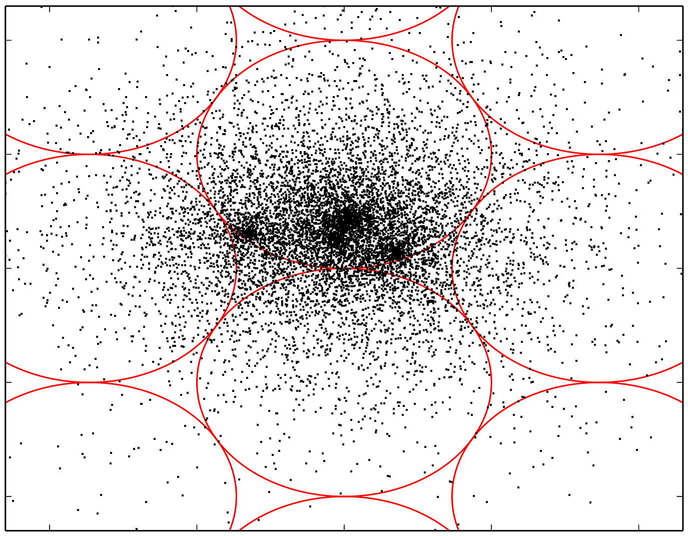


Figure 6.1.1: Spares Region Partitioning

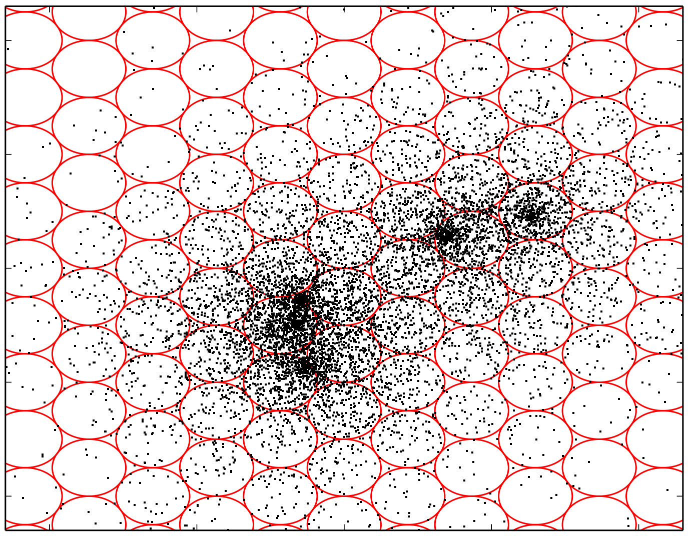


Figure 6.1.2: “useful” Region Partitioning

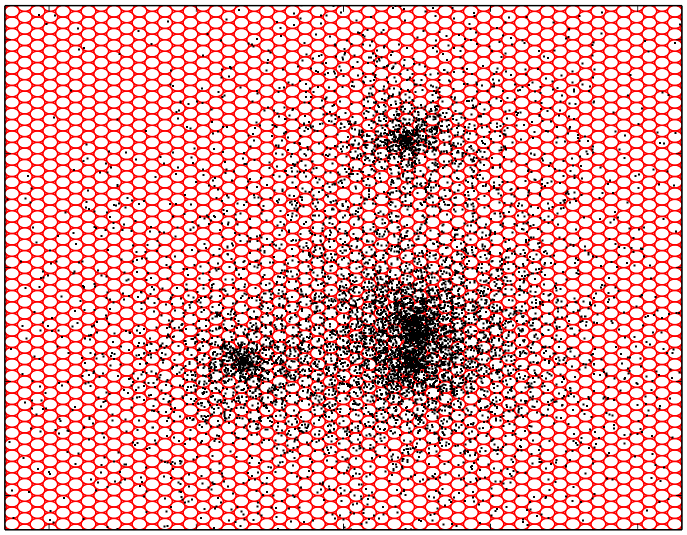


Figure 6.1.3: Dense Region Partitioning

QAM is to evenly distribute symbols in a metric space along some corresponding axis. In electromagnetic signal propagation, these axis are often the amplitude and frequency of a signal. Unfortunately Symbols that are toward the edges of the signal constellation correspond to larger regions than symbols toward the center of the constellation. This discontinuity between the symbol coverings results in a power efficiency loss for EM signals called the shaping gap. RPHash exhibits a similar gap in-efficiency, due to the discordance between the projected vector's spherical distribution and the constellation's rectangular distribution.

Commonly in communication theory, a shaping code would be applied to data in order to redistribute symbols to minimize the signal energy needed to encode symbols in regard to the AWGN perturbed communication channel [36]. In RPHash a similar circumstance is encountered resulting in data that does not evenly utilize the symbols of the lattice. Following that projected data tends to generate spherical distributions in the lower dimensional embedding, we can illustrate this issue. Various shaping procedures help to assure that the data is at least within the range of the symbols, in particular Z-score normalization. However, even with normalization we can continue to require a shaping solution. We would prefer that our lattice be symmetric about both X and Y axis. To achieve this RPHash employs an inverse Gaussian transformation to the QAM constellation points, resulting in a rounded QAM.

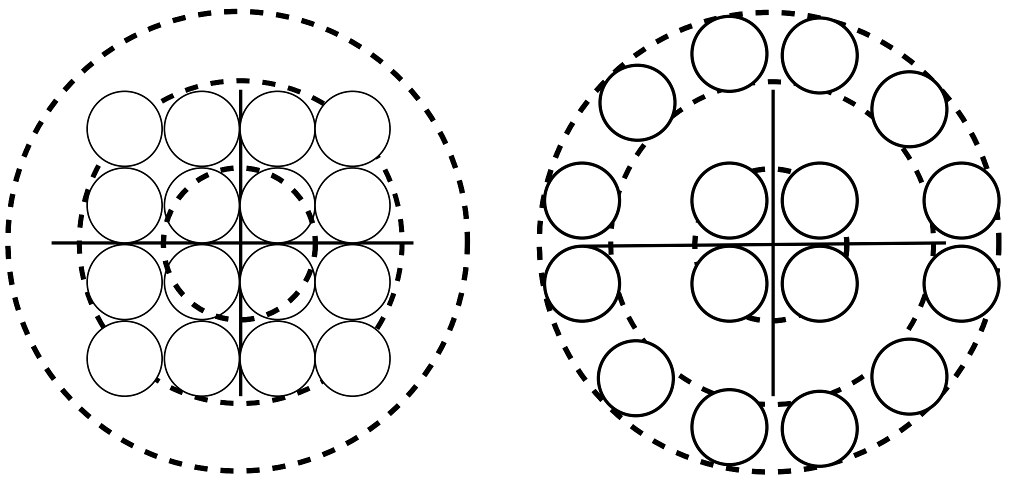


Figure 6.1.4: QAM Constellation

Although Gaussian distributions tends to have tightly centered point clouds around their respective centroids. Increasing the density of the partitioning will result in single member buckets, instead of the desired clusters. A simple estimate for the scaling factor of constellation vectors is the sampled on-line variance of the projected data stream. This is effective for spherically distributed clusters, but fails on ellipsoidal and more complicated cluster distributions. However issues still arise leading to irregular structured data, that we will address in the following section.

6.2 Where are clusters in projected space?

We begin by looking at some distributions of projected data. Our initial hypothesis is that projected data will roughly cluster around the origin of the projected plane, following the central limit theorem. Our hypothesis however was incorrect for various synthetic and real datasets. Figures 6.2.1, 6.2.2, 6.2.3 show the kernel density contours for projected data in 8 dimensions, plotted sequentially on x and y axis. The red line marks the origin, the figures are synthetic Gaussian clusters, data from the Human Activity Recognition dataset [92], and the Indoor Localization dataset [29], respectively. From the plots it is clear that only the Indoor Localization dataset seems to adhere to our hypothesis, and in which case is a good candidate for scaled uniform partitioning. The Gaussian Data and the HAR dataset however exhibit a more correlated structure even in the projected space.

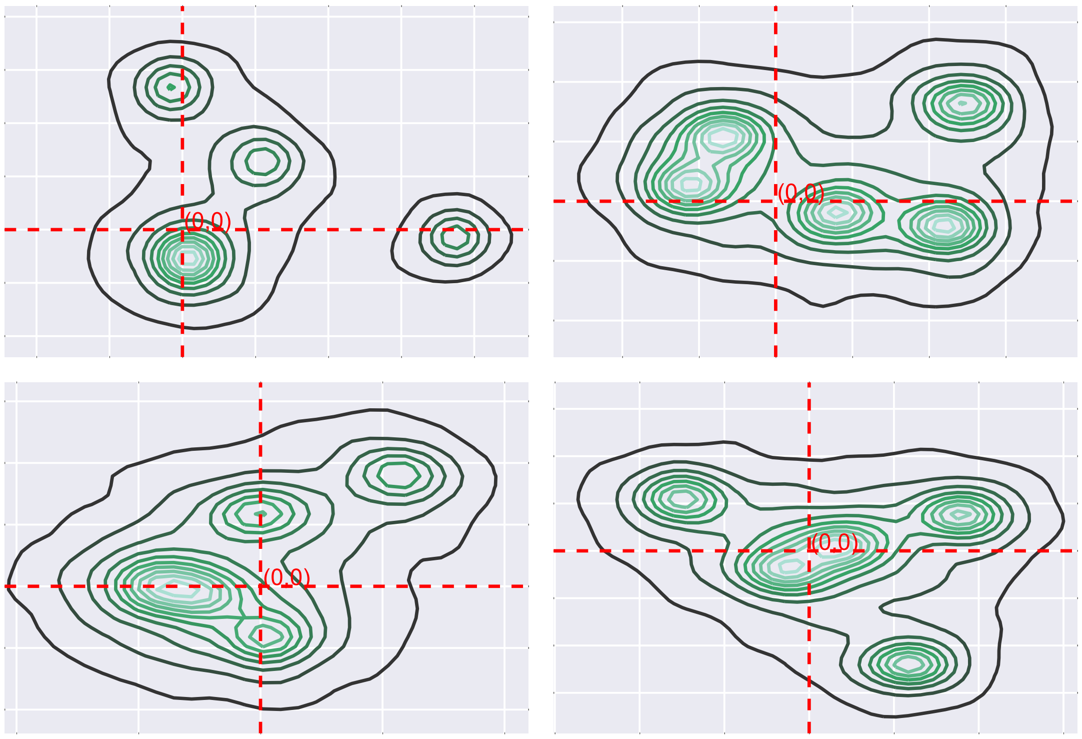


Figure 6.2.1: Projected Gaussian Cluster Kernel Density Estimates Contours

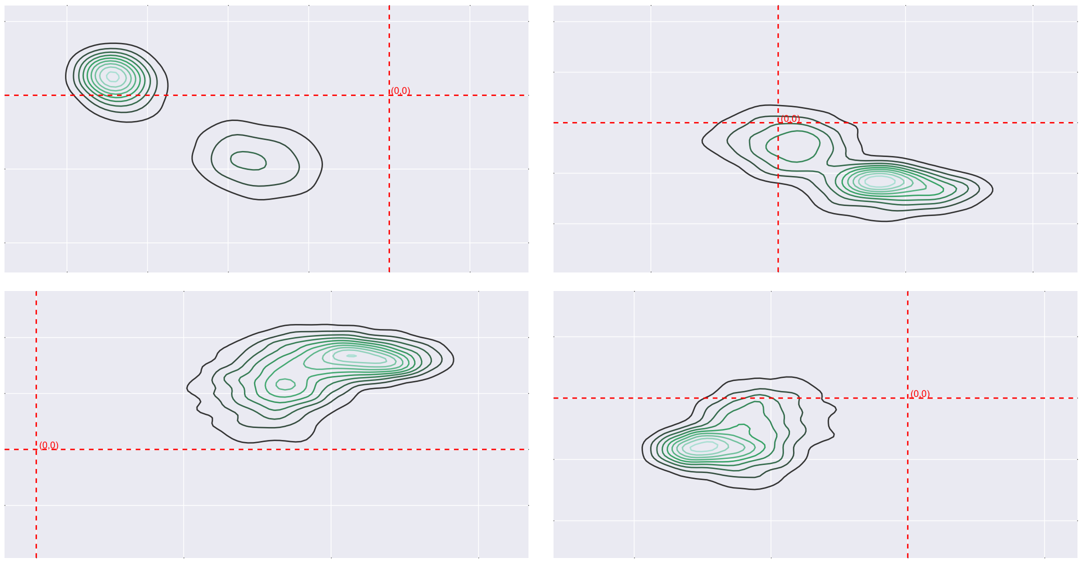


Figure 6.2.2: Projected Human Activity Recognition Kernel Density Estimates Contours

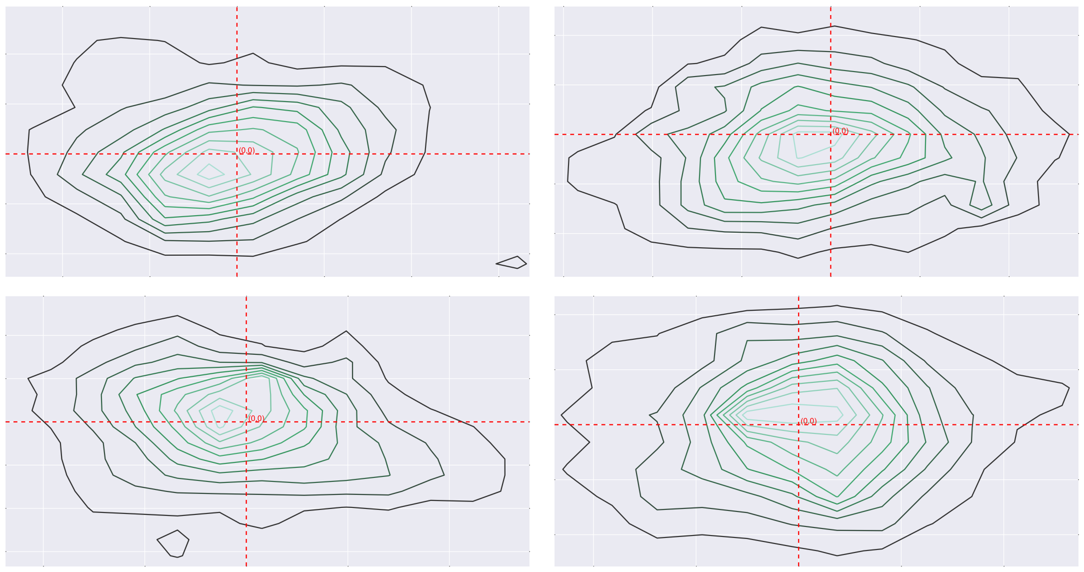


Figure 6.2.3: Projected Indoor Localization Dataset Kernel Density Estimates Contours

6.3 Adaptive LSH

Adaptive LSH is a proposed LSH function that uses the hash bucket counts from RPHash to drive hyperplane partitioning. The counts can be used to adaptively direct the LSH hash key lengths to correspond to the variance of candidate clusters. The hyperplane partitions are obtained by dividing regions about the origin for a given projected column space. Allowing a hierarchical comparison between nested subspaces and the application of information based criterion to drive bisection adaptive LSH should optimize our hash function for the input data.

Following the result of this metric, the LSH is either returned or deepened with a successive hyperplane cut. Figure 6.3.1 demonstrates this process for using the relative density metric. The relative density metric requires that the successive cutting planes yields a denser region than the parent region. adaptive LSH proceeded by taking a simple LSH function similar to the p-stable distribution LSH [82]. A key attribute to this particular type of LSH function is the immediate relationship between adjacent depths of the hash.

Definition 6.3.1 (LSH Composability)**.** *An LSH function $\mathbb{H}^n(x)$ that maps $x \in \mathbb{R}^n \rightarrow \mathbb{Z}_2^n$, is composable if there is a related function $\mathbb{H}^{n-1}(x_{n-1})$ that maps $x_{n-1} \in \mathbb{R}^{n-1} \rightarrow \mathbb{Z}_2^{n-1}$ where $\mathbb{H}^{n-1}(x_{n-1}) = \big(\mathbb{H}^n(x) + 1\big) \bigcup \big(\mathbb{H}^n(x) + 0\big)$ for all $x_n \in \mathbb{R}^n$*

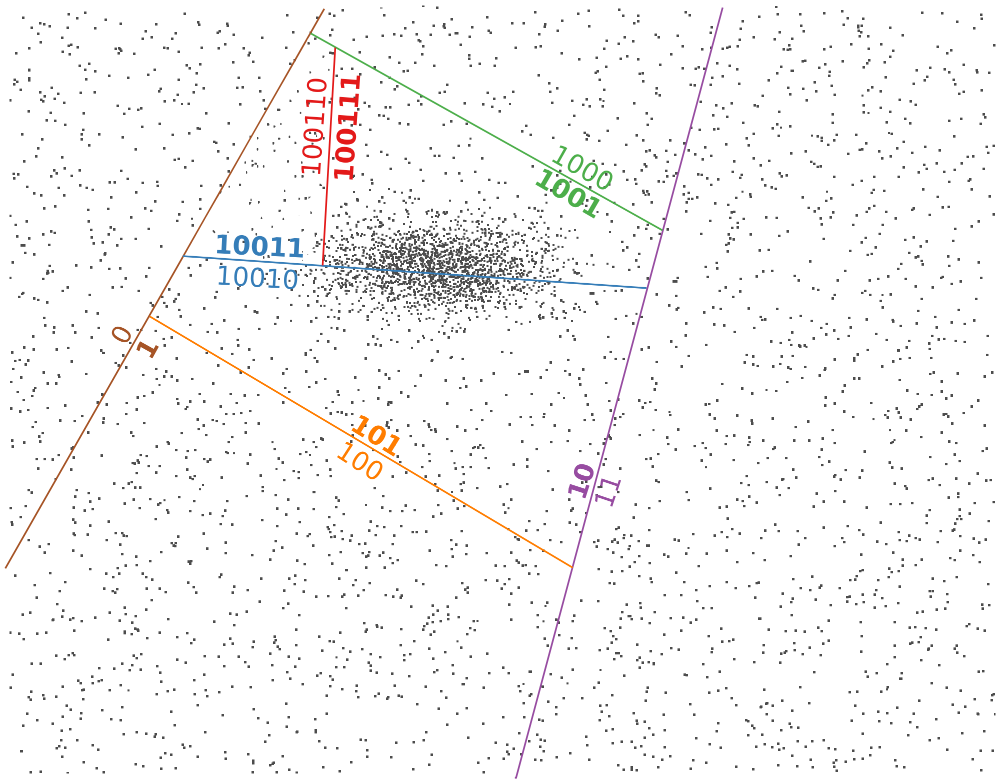


Figure 6.3.1: Density Based Adaptive LSH Generation

A useful property of composable LSH functions is that they allow for direct access to the parent and sibling hash buckets in the Count-Min Sketch (available through inverting or removing the final bit of the hash ID). This relationship between hashes can then be used to balance the hash ID allocation. Although a variety of metrics are feasible, one of the simplest is to continue to extend the hash depth, so long as the subsequent hash count is greater than half of the parent hash count. This method comprises the basis for our adaptive LSH function.

```
Algorithm 6: Adaptive LSH
  $i = 1$
  $ct, ct\_prev = C\big(\mathbb{H}^{i+1}(x)\big), C\big(\mathbb{H}^{i}(x)\big)$
  while $i < n$ and $2ct > ct\_prev$ do
      $ct\_prev, i = ct, i + 1$
      $ct = C\big(\mathbb{H}^{i}(x)\big)$
  return $\mathbb{H}^{i}(x)$
```

From the Figure 6.3.1 we can see that space partition continues so long as the relative density of a region increases, with respect to its parent partitioning. In this case the vectors with the label 100111 comprise the

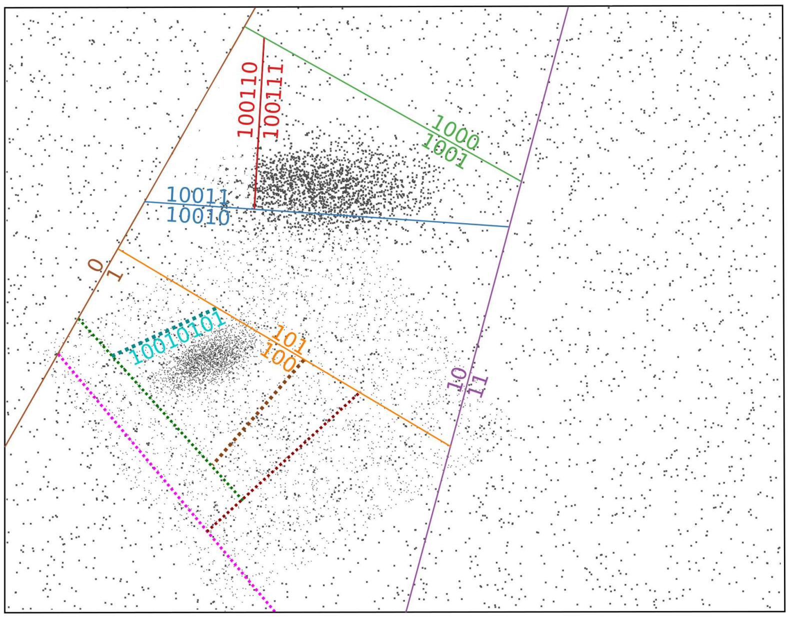


Figure 6.3.2: Density Based Adaptive LSH Generation with Two Dense Regions

densest region, and are likely good cluster centroid candidates, while vectors with 0 and 11 appears to contain mostly noise vectors. If we extend this concept, as in Figure 6.3.2 with a denser cluster, we immediately see that the LSH key length grows, increasing the selectivity of the hash function, while also maintaining the original cluster's partitioning.

To direct our cutting plane procedure, we must have access to a function that can quickly provide us important characteristics regarding the induced partitions. To do this, we simply maintain either a complete hash of cutting regions to parameters, or a Count-Min Sketch of parameters. The choice will depend on the desired space / accuracy trade-off.

In Figure 6.4.1 a region is partitioned by a set of random hyperplanes. The region parameter maintained is the count of objects in the induced regions and the storage structure is a complete hash. After generating the complete hash data structure through the Adaptive LSH, the data structure can be used to query the appropriate length hash for incoming objects, or be used in off-line analysis to discover clusters directly. The determination of the appropriate length and cluster discovery can optimize a variety of useful heuristics, depending on what data is available in the complete hash structure. In Figure 6.4.1 only the partition's size is known. However using this data, we consider a simple heuristics for determining clusters assignment. Rule 6.3.2 is the simplest rule for deciding whether to accept or reject a cluster. Rule 6.3.2 always chooses the node containing more data points and set the parent cluster and sibling cluster counts to zero. Rule 6.3.3 follows a similar pattern, but instead of simply observing the count of clusters as the split metric, it uses the inter-cluster variance, favoring lower variance clusters over higher variance clusters. Rule 6.3.4 defines the splitting problem in the context of a supervised learning algorithm C4.5. A further analysis of this metric is given in [94]. Rule 6.3.2 and Rule 6.3.4 offered slightly better clustering performance than Rule 6.3.2 for synthetic datasets, but the difference was not great enough to warrant the additional computation and memory requirements.

Rule 6.3.2 (Denser than parent Rule).

$$\begin{aligned} \textbf{\textit{Given:}}\quad & S_{2,1}, S_{2,2} \subseteq S_{1,1} \textit{ and} \\ & |S_{2,1}| + |S_{2,2}| = |S_{1,1}| \\ \textbf{\textit{if:}}\quad & |S_{2,1}| > |S_{2,2}| \\ & \textbf{\textit{Remove }} S_{2,2}, S_{1,1} \end{aligned}$$

Rule 6.3.3 (Decreasing Partition Variance).

$$\begin{aligned} \textbf{\textit{Given:}}\quad & S_{2,1}, S_{2,2} \subseteq S_{1,1} \textit{ and} \\ & Var(S_{2,1}) + Var(S_{2,2}) = \tfrac{Var(S_{1,1})}{2} \\ \textbf{\textit{if:}}\quad & Var(S_{2,1}) < Var(S_{1,1}) \\ & \textbf{\textit{Remove }} S_{2,2}, S_{1,1} \end{aligned}$$

Rule 6.3.4 (Information Gain).

$$\begin{aligned} \textbf{\textit{Given:}}\quad & S_{2,1}, S_{2,2} \subseteq S_{1,1} \textit{ and} \\ & H(S_{2,1}) + H(S_{2,2}) < H(S_{1,1}) \textit{ and} \\ & H \textit{ is Entropy} \\ \textbf{\textit{if:}}\quad & H(S_{2,1}) < H(S_{2,2}) \\ & \textbf{\textit{Remove }} S_{2,2}, S_{1,1} \end{aligned}$$

Alternative adaptive LSH techniques have been designed previously for optimizing kNN search problems [22, 110]. These adaptive methods however do not optimize for the our RPHash goal The goal in RPHash is to find dense clusters, while kNN adaptive LSH tries to evenly partition data such that the LSH algorithm exhibits high specificity. In the case of RPHash, this would likely result in getting only dense sub-regions of a cluster. While such adaptive methods may be useful for RPHash -like algorithms, we did not explore it in this work.

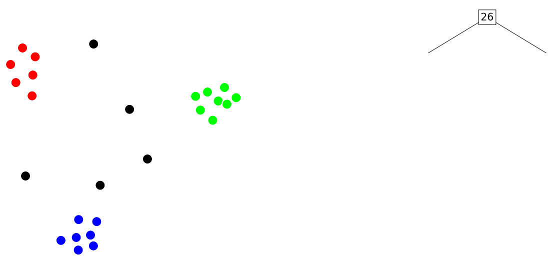
6.4 Tree Walk RPHash (TWRP)

The adaptive LSH function performs comparably to well tuned Leech and Spherical LSH on on synthetic and real world data as discussed further in Chapter 7. However a modification of the depth like search method, over the dataset yields one of the more important contributions of this research. The tree walk method executes as either a 2-pass or streaming streaming RPHash algorithm, but instead of only updating buckets corresponding to the maximum density bucket as in Adaptive LSH, TWRP increments the counts of all sub-hashes as well. This bares a resemblance to the approach in [94] while adapting their algorithm to work in a streaming and memory limited setting.

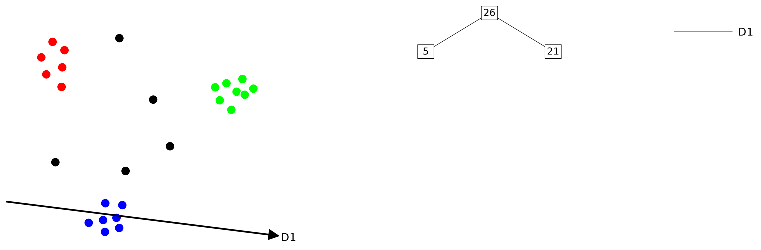
As in [94], TWRP can use a variety of metrics for the tree splitting condition. Figure 6.4.1 shows a toy example of this procedure in which the metric is to traverse the path where a node is significantly larger than its sibling. The example depicts a dataset containing 3 labeled clusters of vectors and some labeled outlier vectors. In step (a) a random vector is drawn across the plane, that partitions the blue cluster poorly. Step (b) similarly partitions the plane, however, in this case it does not split a candidate cluster. This process continues 2 more times, sometimes intersecting clusters and sometimes not intersecting clusters. The tree on the right hand side of Figure 6.4.1 gives the cardinality of the partitions following the cuts. By searching the tree in a greedy depth first traversal, we get the red cluster, and a noise point. If we then deflate the above partition by the identified cluster size, we can restart our traversal from the root node that is now size 19. This process continues until we get the desired k number of clusters, or the root of the tree is 0. It is intended that this example be contrasted with the initial RPHash example given in Figure 4.1.1. In that example cutting a cluster, would have a negative effect on locating density modes, because the corresponding hash bucket would be missing roughly half of its population. TWRP remedies this situation, by allowing the density search to disregard partitions that might intersect a dense region.

6.4.1 Count-Min Cut Tree

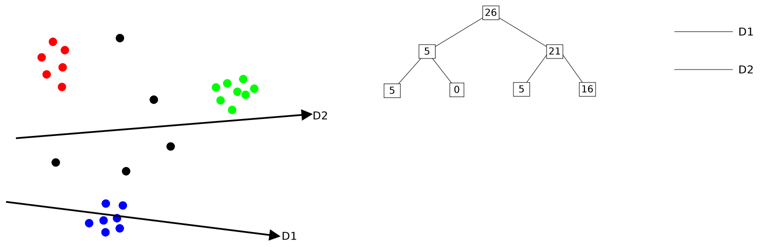
The Figure 6.4.1 shows a process familiar to [94], but it is unfortunately a toy example. This structure will clearly exhibit exponential growth in the input and dimensionality of the dataset. In TWRP we introduced a new structure that is not only novel and applicable to a variety of other subspace analysis and compression algorithms, it also directly solves our memory problem. TWRP achieves this through the use of our often



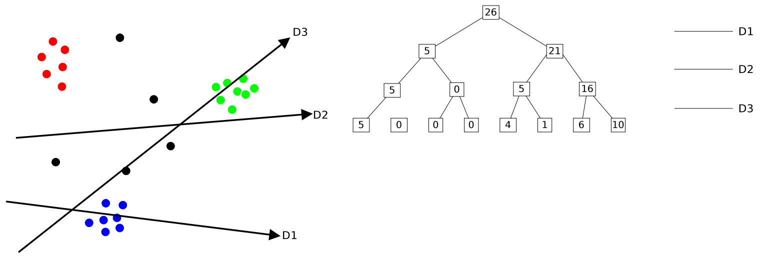
(a)



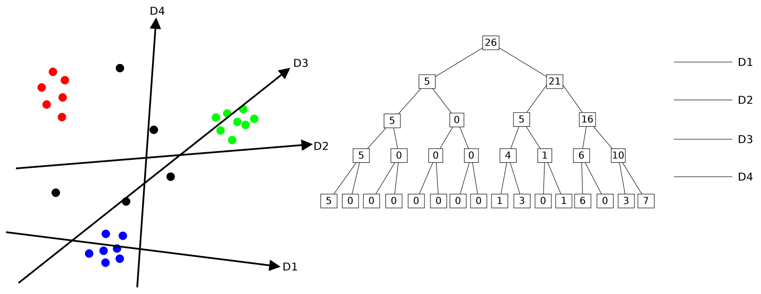
(b)



(c)



(d)



(e)

Figure 6.4.1: (a),(b),(c),(d),(e) show the splitting procedure and subsequent count tree generation for the Adaptive LSH function. Note that leaves with count equal to 0 are shown only to illustrate the breadth-first search tree traversal, while the algorithm would not actually explore these nodes.

utilized Count-Min Sketch. Instead of storing the exponential set of all possible hyperplane splits, TWRP stores a bounded sketch of splits to be explored in a depth first search traversal during subsequent or periodic off-line step. The traversal uses a simplification of Liu et. al.'s C4.5 metric, in which the branch direction favors the larger cardinality node. The result of this method was depicted in Figure 6.4.1, but differs in the tree construction step. Instead of constructing and storing the binary tree in it's entirety, TWRP expands only the important paths of the tree in a depth first search traversal, using the Count-Min sketch counts as an oracle for the approximate cardinality of each visited node.

Our approximate oracle structure shown in Figure 6.4.2 is called the Count-Min Cut tree. Count-Min Cut tree combines the Count-Min sketch with a binary space partitioning tree similar to k-d Trees [27], based on locality sensitive hash functions. Although applicable to any composable LSH function, we use the simple LSH function proposed in Indyk and Motwani's original work on LSH for approximate near neighbor search [82]. We apply the hamming space LSH function to projected euclidean space by observing the signs of the projected vectors with respect to the origin. We formally define this in Definition 6.4.1.

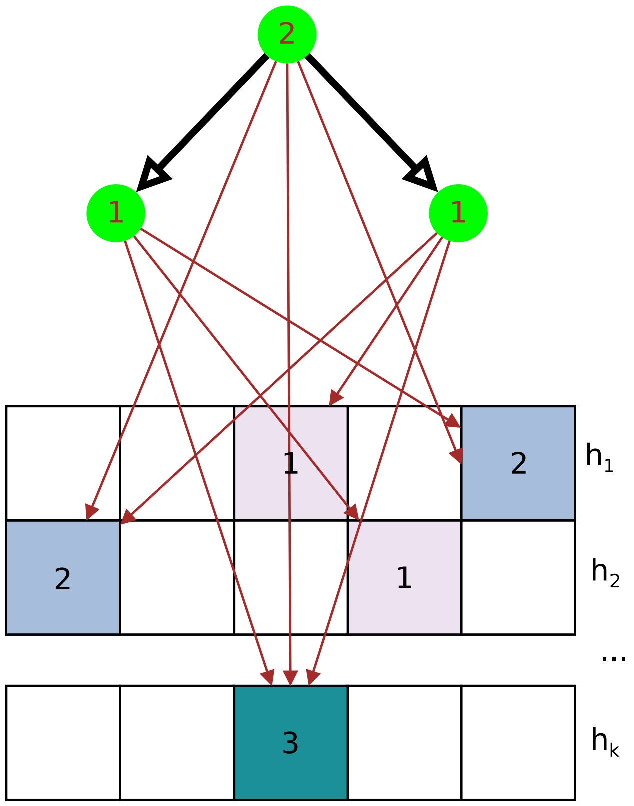


Figure 6.4.2: Count-Min Sketch with Cut Tree

Definition 6.4.1 (Sign based Projected LSH)**.**

$$H(X) = \sum sign(P(X))2^n$$

In the same process as Adaptive LSH RPHash. Figure 6.4.2 shows the basic structure of the count tree. An immediate consequence of this structure is a reduction in the failure probability for mis-counting collisions. This consequence is provided by the tree structures adherence to triangle inequality, such that mis-counted collisions can be verified or reject and updated by structurally related nodes.

Unlike Liu et. al. TWRP does not concern itself with the more complicated to compute C4.5 entropy method, or the possibility of splitting clusters with random hyperplanes. TWRP, avoids the latter restriction by virtue of its intended application on high dimensional datasets. We assert that the splitting likelihood falls exponentially with the original embedding dimension. We formally state this in Theorem 6.4.2 and prove the probability of splitting a cluster converges to zero probability of splitting clusters as the data dimensionality grows large.

Theorem 6.4.2 (Hyper-rectangle Splitting)**.** .
The probability of splitting a hyper-rectangular region into two equal mass clusters where subsequent dimensional cuts are always of the smaller region is 0 as the dimensionality grows to infinity.

$$\lim_{d \to \infty} \frac{Vol(R) - Vol_{removed}(R)}{Vol(R)} = 0, \textit{ R is a hyper-rectangle in } \mathbb{R}^d$$

Proof:

1. Let the vector X s.t. $x_j = [0, ...c, ...0] \in \mathbb{R}^d$ is orthogonal where $c \in [0, 1)$
2. $\sum_i^n x_i = \mathbb{P}$ is a plane in $\mathbb{R}^d$
3. R is a unit hyper-rectangle in $\mathbb{R}^d$ with volume $Vol(R) = 1$
4. let $S_1(p)$ be the volume of the projection of R on x_p
5. If we restrict $S_1(p) + \tilde{S_1}(p) = 1$, $S_1(p) \leq \tilde{S_1}(p)$ for all p

   we get a dimension-wise construction for the volume of the smaller region of a hyper-rectangle split by a hyperplane.

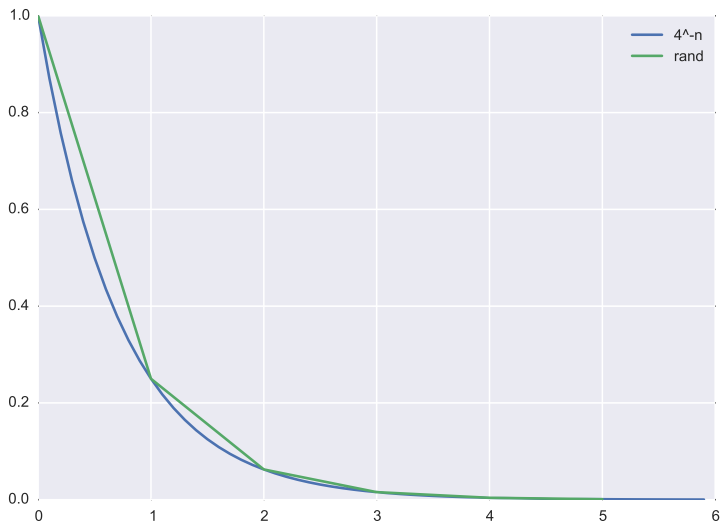


Figure 6.4.3: Volume of Mass To Cut Mass Ratio

6. Next consider the volume of such a hyper-rectangular region $V(R_{s(X)})$.
   $V(R_{s(X)}) = \prod_{p}^{n} S_1(p)$ where $S_1(p) \in U[0, \frac{1}{2})$
7. $V(R_{s(X)}) \leq 2^{-n}$ for all n
8. $\lim_{n\to\infty} 2^{-n} = 0$
9. $\Rightarrow V(R_{S(X)} + V(\tilde{R}_{S(X)}) = V(s)$
10. $V(\tilde{R}_{S(X)}) = V(s)$ as $d \to \infty$ ■

The bound on the expected volume of a cut region to total region is closed. Figure 6.4.3 is a plot of the average of splits drawn from a uniform ratio from $(0 : 0.5]$ and the function $f(x) = (1/4)^n$. The latter is the continuous bound for the expectation of volume of loss due to sub-optimal cutting.

Theorem 6.4.3. *Optimally Dense Spheres Packings are less dense than hyper-rectangular packings in $\mathbb{R}^n - space$.*

Proof:

1. Formally we can restate Theorem 6.4.3 with the following formula, regarding the ratio for Sphere filling of euclidean space vs rectangular cover of euclidean space.

2.
$$\theta\left(\frac{\text{Sphere\_Volume}}{\text{Cube\_Volume}}\right) = \frac{1}{d^{d/2}\left(\frac{2}{e\pi}\right)^{d/2}} < \theta\left(\mathbf{e}^{-d}\right)$$

3. Therefore we have an exponential bound of an otherwise exponentially bound probability shown in line 8 of Proof 6.4.2. ■

Next we formally introduce the Tree Walk RPHash (TWRP) algorithm as a set of two processing modes. The first processing mode Algorithm 8 consists of an on-line operation and data structure to be updated for each incoming date vector. The second mode is the off-line step. In the off-line step the store data structure is evaluated, to find the density modes in the data stream. Table 7 defines the variables and data structures used in both algorithm descriptions.

```
Algorithm 7: Preliminaries
  $k$ -number of clusters
  $X = \{x_1, ..., x_n\}, x_i \in \mathbb{R}^m$ - stream
  $C$- cm-sketch, counts→vector
  $\gg$ - bit shift
  $\mathbb{H}(\cdot)$ - LSH Function
  $\mathbb{P} = \{p_1, ...p_n\}$ - set of $n, m \times d$ projection matrix w/ JL property
  $+$-weighted addition
```

The on-line tree generation step consumes the input vector data stream. First the input vector is projected from m dimensions by the projection matrix to a d-dimensional vector. The LSH function is then formed for each successive dimension $[1, 2...d]$, using the sign based LSH Function Definition 6.4.1. Each of the d hashes is added to the Count-Min Sketch. In addition to counting the, the Count-Min Sketch also stores the arithmetic sum of the vectors. This process concludes the on-line tree generation step.

```
Algorithm 8: On-line Tree Generation
  forall the $x \in X$ do
      $\tilde{x} = \sqrt{\frac{m}{d}} p^{\intercal} x$          // Random Projection
      $h := \mathbb{H}(\tilde{x})$          // LSH Hashing
      while $h > 0$ do
          $h = h \gg 1$
          $x' = C[h] + x$
          $C$.add$(h, x')$
```

The off-line step can be run at any time during the data stream progression. The step consists of searching the binary tree induced by our splitting, sign-based, hash function. However, instead of storing all nodes in the tree this process queries the Count-Min Sketch as an oracle for node attributes. The binary expansion of the tree, corresponds to the LSH function which is used to query the Count-Min sketch. The traversal of the tree is directed by chosen splitting Rule 6.3.2. When the splitting rule exhausts a particular traversal, that node is identified as a candidate cluster, and all parent nodes are 'deflated' by subtracting the emitting node's vector count. The tree traversal then repeats the expansion from the parent node. Traversals complete when a constant number of candidate cluster nodes is identified.

Algorithm 9: Off-line Tree Search

```
forall the $H \in sort(C.ids)$ do
    if $2C[H] < C[H \gg 1]$ then
        $C[H \gg 1] = 0$
$L = []$
forall the $h \in sort(C.counts)$ do
    $L \leftarrow$ medoid($C[H]$)
return $L$
```

6.4.2 Error Recoverability

In Figure 6.4.2 the tree is consistent, however a degenerate case can occur in which a short LSH, corresponding to a high dimensional subspace, collides with a long LSH, corresponding to a low dimensional subspace. An example of this occurs when a short LSH, such as the first space cut, is hashed by the Count-Min Sketch's universal hash function to the counter H_0. This results in roughly, half of the vectors being counted by counter H_0. Subsequently if a vector exists, that collides with H_0 and the depth of the Count-Min sketch table is not sufficient to correct the error, the vector will be over represented in the Count-Min Cut Tree. Fortunately, in the tree setting the failure can be detected and corrected by investigating the parent and child counts adjacent to the given count. This results in a lower bound on the sketch depth required. See Section 8.3.1 in Chapter 8 for a more thorough investigation of this concept.

Chapter 7

Performance Analysis

RPHash performance is tested against a selection of standard and state-of-the-art algorithms for clustering at rest and streaming data. The goal of our performance analysis is to provide comparable clustering performance to standard and state-of-the-art algorithms, while showcasing RPHash's processing speed compared to other algorithms. Due to the ill-posed nature of the clustering problem, we provide evaluation over a set of clustering metrics in an attempt to capture the multifaceted nature of what 'good' clustering is.

Due to the dependence between RPHash clustering and timing performance with RPHash components configuration, we first perform an evaluation of component dependence and performance. We then choose the optimal configuration, based on speed, data set stability, and clustering performance, as a baseline configuration for all subsequent RPHash experiments.

Our clustering comparison approach consists of sets of experiments where our clustering method is compared to various common clustering methods. The compared clustering methods vary by which RPHash variant we are evaluating and its intended data access structure. For our two pass algorithm, we consider other algorithms that require a finite number of passes over the data, with sub polynomial memory growth complexity. Likewise, for streaming data, we compare our streaming algorithm with other streaming clustering algorithms that have sub polynomial processing complexity, and sub linear memory growth. Therefore, the performance analysis discussion of the three RPHash algorithms will be given in three parts. Each section will define the comparison algorithms, and the corresponding data source setting, as well as any other pertinent experiment specifics.

The remainder of this chapter is organized as follows. First we describe our experimental model, data sources, test structure, metrics, and comparison algorithms. Next we provide the parameter exploration study, to establish our baseline configuration. The third and fourth section comprise the main performance results for the two pass RPHash algorithm. The following section shows results specific to the streamingRPHash algorithm, and streaming data. Followed by an evaluation of the Tree-Walk RPHash TWRP Algorithm. Lastly we give an overview of the k-anonymity security performance of random vector projection in RPHash.

7.1 Metrics and Data Sources

The RPHash framework provides a comprehensive experimental framework consisting of evaluation metrics and data generation procedures. Both real world and synthetic data sources are made available in the testing framework. The general test setup consists of data generation or ingest, into a static memory resident array. The vectors in the array are then randomized, to remove sample sequence order dependence and clustering is performed, on the data. The results of the clustering procedure are then evaluated against applicable clustering metrics. Due to the randomized nature of RPHash and many clustering algorithms, all experiments are performed 6 times to establish an average performance and variance over the applicable metrics. The system provisioned for experimentation was stable, and as such outliers were not removed. The details of the various aspect of our experimental set up follow.

7.1.1 Real World Data

RPHash is a general clustering algorithm intended for a range of high dimensional clustering problems. We designed experiments around a variety of real world datasets. To assess the clustering performance of RPHash, a set of expert labeled, real world data sets used throughout the our performance experiments is summarized in Table 7.1. Labeled datasets are useful for computing external metrics on clustering performance, however the labeling requirement is often infeasible for very large datasets. As a result, we have included a common dataset for this purpose, based on the Word2Vec transformation of semantic data [104]. As well as a dataset to test anonymization, the MIMIC II dataset [114].

Table 7.1 gives and overview of the features found in these labeled datasets and will be referenced throughout the remainder of this chapter. The two most common datasets featured in our tests are the

Data Set	**Num of Clusters**	**Num of Features**	**Num of Vectors**	**Type of Data**
Arrhythmia [73]	16	279	452	Real
CNAE-9 [44]	9	856	1080	Binary
Cora [116]	7	1433	2708	Binary
Gisette [74]	2	5000	7000	Real
Human Activity [23]	6	561	10299	Real
UJIIndoorLoc [124]	3	520	21000	Real
WebKB [116]	5	1703	265	Binary

Table 7.1: Real-World Data Sets.

human activity and location based datasets. The first dataset of these activity datasets is the *Human Activity Recognition Using Smartphones* [92]. It has 10,299 records (the first 10,000 are used in this study), and 561 attributes representing time and frequency domain variables. The dataset contains six clusters which denote the six activities (WALKING, WALKING_UPSTAIRS, WALKING_DOWNSTAIRS, SITTING, STANDING, LAYING) performed by each person. Next we have the *Indoor Localization* dataset based on device WiFi location. (`https://archive.ics.uci.edu/ml/datasets/UJIIndoorLoc`) [29]. It is a multi-building multi-floor indoor localization database to test Indoor Positioning System that rely on WLAN/WiFi fingerprint. The dataset has a total of 21,048 vectors. This study uses first 21,000 vectors each having 520 attributes which represent WiFi fingerprints composed of 520 intensity values detected by Wireless Access Points. Building ID is used as the target variable which has three possible values. Three citation and link datasets are included: WebKB (Wisconsin) Dataset [116], Cora Dataset [116], and CiteSeer Dataset [116]. These datasets consist of the sparse citation graphs of text corpus. In each, documents are organized into a set of sub-corpus clusters. CNAE-9 [45] contains natural language data represented by a sparse term-document matrix. The Arrhythmia [73] contains dense signal measurements of ECG data grouped into arrhythmia and regular heart rhythm.

In addition to our standard labeled datasets, we also include some large unlabeled datasets. The first dataset utilizes the popular word2vec [104] space representations of topic vectors. The dataset Google Word2Vec News dataset [105] is a very large publicly available dataset built from articles in the Google news corpus. It contains 300k topics represented as 300 word2vec attributes. Our next large, unlabeled dataset is the medical dataset MIMIC II [114] biometric dataset. We pre-process this dataset, prior to clustering by taking generating signatures of the biometric markers “RESP”, “ABP”, “ECG II” and “PLETH”. The signatures were created by taking the 64 most prevalent frequencies as identified via power spectral density estimation of the signals. The dataset consists of 100,000 patient time based signals, and is over 4 TeraBytes in total size. Due to the rather large size, we sampled 38 full signal observations from the 100,000 observations.

7.1.2 Data Generators

In addition to real world data, synthetic data generators are a major part of our test framework. While real world datasets provide a realistic view of clustering performance, they are not very useful in providing scalability and variability results over ranges of values. We have implemented both streaming and static data sources generators in the RPHash testing framework.

Our general vector stream consists of a user defined set of cluster centroids consisting of uniform variates in the range $(-1, 1)$. The centroids however are not present in the data set, and merely establish a ground truth mean around which synthetic vectors are generated. Test vectors are generated with equal probability of choosing any of the k ground truth centroids and adding a vector of dimensionally scaled Normal variates. While this data source is well suited for evaluating timing performance, the predictable structure of the data, is not very realistic and may favors certain clustering methods over others. To overcome these limitation, we provide a set of configuration variables.

First, we must accept that equi-probable cluster membership is unlikely. In many machine learning problems, learning unlikely classes is more important than evenly partitioning data, such would be the case in cancer screening, and Internet anomaly detection. To provide this, we allow for a non-equi-probable cluster centroid selection for the generation set. Next we consider the possibility for non-univariate data. In real world data sets, clusters often have varying size and shape. While we maintain a requirement of generating ellipsoidal clusters, our test framework can generate clusters with variable variance, by replacing the normal variate constraint, with a Gaussian variate with variance chosen uniformly in the range $(0, 1]/d$. Another important aspect of real world data, not present in the described framework, is noise. Again we provide this aspect, through a configurable noise percentage parameter.

In addition to the static data generation, we also provide a streaming data generator. All configuration parameters of the static data generation are available in the streaming version. The only difference is that vectors are generated one at a time, and are not stored. This prevents us from performing some clustering metrics, but it allows us to simulate the important unbounded nature of streaming data sources.

A generator in the R programming language was also developed to access various R language based streaming clustering algorithms. The generator uses a similar technique as DSD_Gaussian function from the 'stream' package to generate vectors from a random cluster in a continuous data stream.

7.1.3 Evaluation Metrics

The clustering problem is an ill-posed machine learning problem. While the k-Means problem description establishes inter-cluster variance as the optimization metric, it may not be well suited for many real world problems in which noise is present, or cluster shape is erratic. For this reason we provide a selection of clustering metrics that describe the 'goodness' of clustering both internally and externally.

Internal metrics are metrics that can be computed directly from the data with no known ground truth, while external metrics require cluster class labels, to compute. First we describe our internal cluster metrics. The most common internal metric for clustering, established by the k-Means problem, is the within-cluster sum of squared errors (WCSSE). This metric establishes a value of inter-cluster vector similarity. Formally, the WCSSE is:

$$\text{WCSSE}(C) = \sum_{i=1}^{k} \sum_{x \in C} ||x - \mu_i||^2$$

where μ is the mean value of each vector. WCSSE is a useful metric for evaluating clustering in which all of the vectors being clustered, are close to the mean of the cluster (compact).

Another internally calculable metric, similar to WCSSE, and the goal of k-means clustering is the Silhouette Index. Silhouette Index is less susceptible to the potential problems that WCSSE might encounter if true clusters, are not compact. Silhouette Index evaluates the ratio of inter-cluster co-similarity to the nearest intra-cluster dissimilarity. Silhouette Index includes the WCSSE, but attempts to remove the impact of wide clusters, by taking the ratio of true WCSSE with the false WCSSE of the nearest cluster centroid for a sample.

The Dunn Index (DI) is an internal clustering metric similar to the Silhouette Index and WCSSE. DI takes a blended approach toward cluster evaluation that favors tightly packed clusters that are far from other nearby clusters. The Dunn index is a ratio of the greatest inter-cluster distance between two vectors in a cluster over the intra-cluster distance between the two closest clusters.

Both of these internal clustering metrics rely on a ’useful’ (see 2.4) distance metric. However, we are concerned more with high dimensional clustering problems in which the distance metric may not be very useful. For these types of problems, we rely on having some expert labeled ground truth. For the case of synthetic sets, this can be the centroid generator’s centroids. For real world datasets, this is often expert labeled data. Cluster purity is a clustering metric that takes a supervised machine learning approach to evaluating cluster membership. Cluster purity measures the amount of correctly labeled clusters for a given partitioning. More specifically it is the sum of counts of correctly classified vectors in each cluster.

$$\text{Purity}(C) = \frac{1}{N}\sum_{c\in C} \max_{x\in X} |c \cap x_j|$$

A direct shortcoming of all of the above metrics is that they do not directly penalize misclassification. The Rand Index (RI) overcomes this shortcoming by establishing a ratio of true-positive co-occurrence, false-positive co-occurrence, true-negative co-occurrence and false-negative co-occurrence. The ratio can be expressed without the false-positive and false-negative values, by scaling it over the binomial coefficient of all possible pairs $\binom{n}{2}$. This ratio results in very small values for large datasets, so in practice, the Adjusted Rand Index (ARI) is often used. ARI scales the final ratio by the maximum ARI from the mean ARI. Formally, this computation is:

$$\text{ARI}(C) = \frac{RI(C) - \mathbb{E}(RI(C))}{max(RI(C)) - \mathbb{E}(_RI(C))}$$

Variation of Information (VI) is another performance metric considered. VI is an external metric that measures the degree of correspondence between the clustering labeling and the ground truth labeling. It is similar to mutual information between two random variables.

Finally, timing and memory usage metrics are also included in performance tests. Timing is evaluated as the time delta between the start of vector processing, and the final centroid generation. Setup time, such as VM initialization or R startup, was not included in the timing delta. Timing is provided by the system clock, for the given machine, and averaged over six experiments.

7.1.4 Comparison Algorithm Details

To evaluate RPHash Performance on static datasets we provide comparative performance data with the following common clustering algorithms:

- **k-Means:** It is implemented with 'kmeans' in R [77].
- **Four methods of Agglomerative Hierarchical clustering:** Single Linkage, Complete Linkage, Average Linkage and Ward's minimum variance method. At each stage distances between clusters are recomputed by the Lance-Williams dissimilarity update formula according to the particular clustering method being used. Ward's (1963) clustering criterion, where the dissimilarities are squared before cluster updating, is used in the implementation of Ward's method (Murtagh and Legendre 2014). All of these agglomerative clustering methods are implemented with 'hclust' in R.
- **Self-organizing Tree Algorithm (SOTA):** It is implemented with 'sota' (Package 'clValid') in R [79].
- **k-Means++** is implemented as the Hartigan and Wong [77] implementation of k-Means, with random initial seeding based on the furthest from a vector metric defined in [24]. The version used in this comparison comes from the Apache Math library (`http://commons.apache.org/proper/commons-math/`).

For the streaming version of RPHash, we also include a selection of streaming clustering algorithms for comparison. In particular we provide comparative performance results to the following algorithms:

- **Streaming k-Means:** clustering algorithm for streams based on facility location problem [35]. Implementation from S-Space [84].
- **Damped Sliding Window:** [135] window length $= 100$.
- **DStream:** [125] The size of grid cells is set at 0.8.
- **Biased Reservoir Sampling:** [3] With a window length $= 100$.

These algorithms are chosen because of their importance and availability (except for streaming k-means) in the R statistical computing framework. All of these algorithms follow the conventional two-stage (on-line/off-line) approach for data stream clustering. The weighted k-means implementation of Hartigan and Wong (with $nstart = 25$ for algorithms implemented in R) is used in the off-line stage.

7.2 Parameter Exploration

In order to find the optimal configuration of RPHash over various synthetic and real world datasets, we performed an exhaustive test of all of the various RPHash configurations, then used Partial Least Squares (*PLS*) regression to evaluate the configuration variable importance. This analysis helps to further assess the correlation and relative effect of the tunable metrics of RPHash on clustering performance and runtime.

7.2.1 Effects of RPHash Parameters and Performance Correlation

To identify an optimal configuration for RPHash, we examine the impact of a large number of possible combinations of input parameters on the performance of the algorithm (Table 7.2). The number of random projections is varied from 1 to 8. The amount of Gaussian blurring shifts applied to projected vectors is changed from 1 through 4. Over-estimated centroids at the end of phase-2 in RPHash are resolved in the 'off-line' step using one of the four standard clustering algorithms: k-means [77] and three strategies of hierarchical agglomerative clustering (single, complete or average linkage). Locality-sensitive hashing methods implemented in RPHash are E_8, Leech, Spherical, and three types of *LSH* algorithms based on p-stable (*Lévy, Cauchy and Gaussian*) distributions. During the random projection step, high-dimensional data vectors are projected to a lower dimensional subspace, and this reduced dimensionality is sometimes restricted by the particular *LSH* algorithm used in the subsequent step. For E_8 and Leech Lattice decoders,

***LSH* Algorithm**	**Projected Dimension(s)**	**# Projections**	**# Blurrings**	**Offline Clustering**
E_8	8	1, 2, 3, ..., 8	1, 2, 3, 4	k-means Single Linkage Complete Linkage Average Linkage
Multi-E_8	8, 16, 24, 32			
Leech	24			
Multi-Leech	24, 48, 72, 96			
Lévy p-stable	8, 16, 24, ..., 80			
Cauchy p-stable				
Gaussian p-stable				
Spherical				

Table 7.2: Input Parameters to RPHash.

Data Set	Configuration	RPHash				k-means			
		ARI	Purity	Runtime	Memory	ARI	Purity	Runtime	Memory
Arrhythmia	Multi-E_8/24/1/2/Complete Linkage	0.2710	0.5885	0.1277	0.1420	0.0811	0.6069	0.5287	16.4333
CNAE-9	Multi-Leech/72/5/2/k-means	0.3424	0.5707	0.7917	0.8312	0.2798	0.5312	2.7120	165.1167
Cora	Multi-Leech/96/3/3/Complete Linkage	0.1065	0.3988	2.6022	0.6660	0.1158	0.4271	52.410	227.9833
Gisette	Spherical/16/2/3/k-means	0.5675	0.6218	1.7769	0.4530	0.4610	0.6002	24.746	1485.0667
UJIIndoorLoc	Spherical/8/8/2/k-means	0.6608	0.8268	1.3840	0.2780	0.6954	0.7750	23.821	2850.6500
WebKB	Spherical/16/5/1/k-means	0.4210	0.7396	0.1117	0.9015	0.4403	0.7528	0.8087	50.1500

Table 7.3: Best Configurations on Individual Data Sets (Configuration: LSH Algorithm/Reduced Dimension/Number of Projections/Number of Gaussian Blurring/Offline Clustering).

the reduced dimensions are 8 and 24 respectively. But to have a wide range of 'projection distances', multiple E_8 and Leech Lattice decoders are combined to construct Multi-E_8 and Multi-Leech decoders respectively. For Multi-E_8, the reduced dimensionality is varied in steps of 8 from 8 to 32 (*i.e.,* reduced dimensionality = 8, 16, 24, or 32). The same for Multi-Leech is varied in steps of 24 (*i.e.,* reduced dimensionality = 24, 48, 72, or 96). We get a wider range of 'projection distances' for Spherical and p-stable *LSH*s where the reduced dimensionality is varied in multiples of 8 from 8 to 80. The Combination of all of these input parameters produces 6400 possible configurations for RPHash. Each of the configurations is then tested on seven real-world labeled data sets which are detailed in Table 7.1. Every attribute of the real-valued data sets is normalized using the standard score or z-score method. In a labeled data set, the 'correct' or ground-truth partitioning is known *a priori*. As such, the clustering accuracy of RPHash is evaluated using ARI and Cluster Purity (Section 7.1.3). Memory and Runtime performance are presented as average runtime for each algorithm ('user time' in R) and elapsed system time in Java on a computer with an Intel(R) Core(TM) i7-4770 CPU with 4 cores @ 3.40GHz supporting 8 hardware threads and 32 GB memory. The results of these tests, along with the results of k-means for reference, are presented Table 7.3.

PLS Regression, is a latent variable method that identifies the direction in the predictor variable space that explains the greatest variance in the response variable space [130]. Although it provides a linear regression model in the projected space, this analysis is only concerned with the response variance optimizing coefficients referred to in Wold [130] as the Variable Importance for the Projection (*VIP*).

In the experiment the sum over all 7 datasets given in Table 7.1 for ARI, Purity, and runtime were used as response variables for the 6400 configuration metrics for RPHash described in Figure 7.4. Categorical data was initially modeled following the Partial least squares Discriminant Analysis permutation of standard *PLS* regression, however no algorithm for *PLS* converged. *PLS* did, however, converge when categorical data was

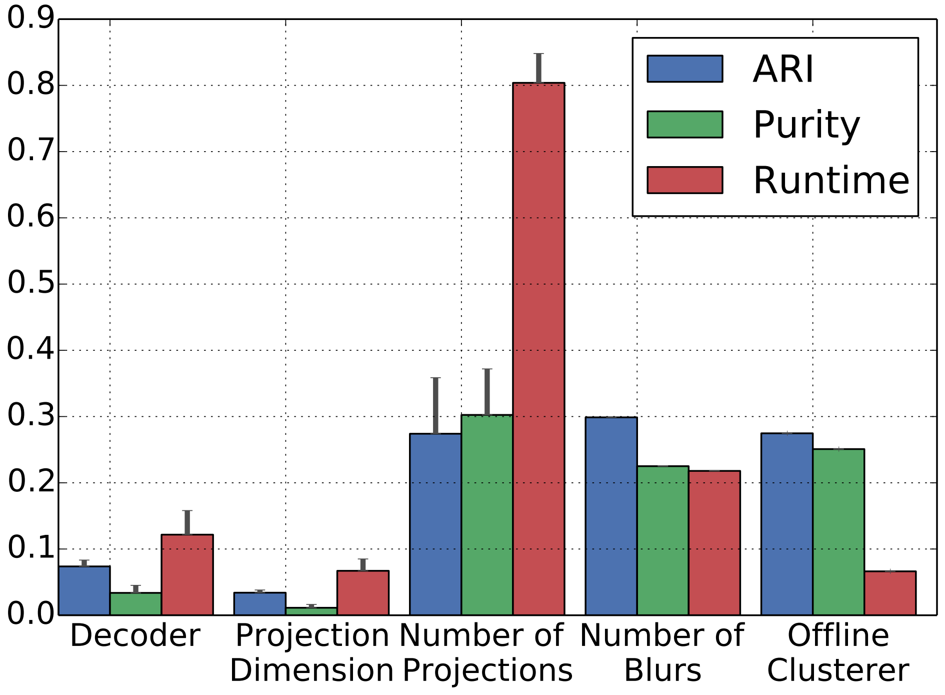


Figure 7.2.1: VIP analysis PLS regression results for 6400 configurations of RPHash on 7 datasets.

encoded as a simple linear progression. The results for this categorical data encoding are shown in Figure 7.2.1. Intuitively runtime responses should be predictable, as the number of projections increased, as will the runtime. This is corroborated in the plot. It can be concluded from an algorithm optimization standpoint that the best clustering performance versus runtime would be to increase the blur factor. Both categorical configuration parameters tend to explain some variance, and the previous analysis suggests that the off-line step of k-means and *LSH* algorithms E_8 and Leech tend to consistently perform well.

7.3 RPHash Performance

Our first performance analysis section validates the performance of RPHash over a set of Real world datasets followed by an analysis on synthetic datasets to validate the scalability of RPHash. In addition we give a detailed overview of clustering performance for various LSH functions available for RPHash including our new Adaptive LSH.

7.3.1 Experimental Approach

To assure RPHash's accuracy and performance, tests for similarity to the standard clustering algorithms, on various real and synthetic datasets are performed. The experimental approach for testing RPHash will address several major areas of RPHash's utility, namely: (i) Synthetic Algorithm Accuracy, (ii) Real World Data Set Accuracy, and (iii) Overall Scalability.

A required analysis of any k-means algorithm is its ability to correctly cluster expert labeled data. It is imperative that RPHash perform comparable to the standard clustering algorithms in regard to external clustering accuracy metrics before timing and memory metrics are considered. To evaluate the real world performance of RPHash with the optimal Leech based configuration (Section 7.2), we clustered seven different datasets and compared the results to those produced by six other well established clustering algorithms. The R implementations used for the six other clustering algorithms are listed in Section 7.1.4. In this test, we excluded k-means++ due to it not offering significantly better results than standard k-means.

For each of the seven datasets, ground-truth partitions are known *a priori*. The ground-truth labels are compared with labels produced by each clustering algorithm to compute the values of three different external measures of clustering validation metrics are provided, *Adjusted Rand Index* (ARI) and *Cluster Purity* (see Section 7.1.3 for details).

The results are given in Table 7.4 for ARI, Purity, Runtime, and Memory consumption. Memory and Runtime performance are presented are averaged for each algorithm ('user time' in R) and elapsed system time in Java, running on a computer configured with an Intel(R) Core(TM) i7-4770 CPU with 4 cores @ 3.40GHz supporting 8 hardware threads and 32 GB memory. In these data sets we see that RPHash performs, on average below standard k-means and similar to Ward's Method agglomerative. RPHash performs better than the other linkage agglomerative clustering methods, and SOTA. RPHash outperforms all other methods in regard to processing time and memory requirement by a large margin. RPHash also demonstrates a significant scalability advantage over the other methods.

7.3.2 Performance Comparison

7.3.3 Adaptive LSH

An extension to our initial RPHash work was the addition of the adaptive LSH decoder. While this decoder was not optimal, it is still an interesting result of this work. In this experiment we generated a synthetic dataset consisting of 10 Gaussian cluster uniformly distributed in 1000 dimensional space. We then vary the inter-cluster variance of the clusters from 0 to 2. The results are shown in Figure 7.3.1. The blue line is the ground truth WCSSE, while the dashed lines are Two-Pass RPHash with Leech Lattice Decoder, Two-Pass RPHash with the spherical (32,2,1) decoder, and Two-Pass RPHash with Adaptive LSH. This figure confirms the main results of our optimal configuration experiment, while suggesting that Adaptive LSH may not be the best solution to the space utilization problem discussed in Section 6.3

Data Set	**Measures**	**RPHash**	**k-means** [77]	**Single Linkage**	**Complete Linkage**	**Average Linkage**	**Ward's Method** [107]	**SOTA** [79]
Arrhythmia	ARI	0.0697	0.0811	0.0461	0.0963	0.0546	0.0889	0.0981
	Purity	0.6058	0.6069	0.5730	0.5885	0.5752	0.5951	0.6062
	Runtime	0.2709	0.5287	0.1680	0.1640	0.1680	0.1680	3.4440
	Memory	0.7070	16.4333	3.4000	3.4000	3.4000	3.4000	21.3000
CNAE-9	ARI	0.2788	0.2798	0.0000	0.0000	0.0000	0.3547	0.1730
	Purity	0.4873	0.5312	0.1185	0.1204	0.1185	0.5722	0.3657
	Runtime	0.3932	2.7120	4.3360	4.3400	4.3400	4.3440	3.7080
	Memory	1.1370	165.1167	24.2000	24.2000	24.2000	24.1000	134.2000
Cora	ARI	0.0915	0.1158	0.0001	0.0120	0.0002	0.0930	0.0647
	Purity	0.3858	0.4271	0.3039	0.3335	0.3039	0.4597	0.3342
	Runtime	0.8290	52.4100	71.9200	71.9600	71.9400	71.9720	11.7120
	Memory	1.4590	227.9833	100.7000	100.7000	100.7000	100.7000	265.4000
Gisette	ARI	0.1282	0.0615	0.0000	0.0000	0.0000	0.0018	0.1147
	Purity	0.6720	0.6241	0.5001	0.5003	0.5001	0.5216	0.6694
	Runtime	2.7363	423.7660	2280.4320	2280.4640	2280.0800	2280.5480	46.8120
	Memory	1.4300	2138.3833	829.3000	829.3000	829.3000	829.3000	2097.5000
HAR	ARI	0.3348	0.4610	0.0000	0.3270	0.3321	0.4909	0.3143
	Purity	0.4631	0.6002	0.1890	0.3770	0.3588	0.6597	0.3966
	Runtime	1.8774	24.7460	413.8800	414.3320	414.0960	414.4480	14.2440
	Memory	0.5157	1485.0667	1259.0000	1214.9000	1214.8000	1214.9000	946.2000
UJIIndoorLoc	ARI	0.5043	0.6954	0.0001	0.0001	0.0001	0.6021	0.3351
	Purity	0.7105	0.7750	0.4635	0.4635	0.4635	0.7732	0.6918
	Runtime	2.6363	23.8213	1093.9440	1094.7000	1094.8200	1095.5360	16.1880
	Memory	0.2460	2850.6500	5132.4000	5049.0000	5049.0000	5049.0000	2227.0000
WebKB	ARI	0.3205	0.4403	0.0066	0.0404	0.0066	0.3276	0.3906
	Purity	0.7063	0.7528	0.4755	0.5283	0.4755	0.7094	0.7019
	Runtime	0.1648	0.8087	0.3760	0.3760	0.3760	0.3760	2.5400
	Memory	1.2000	50.1500	6.3000	6.4000	6.3000	6.3000	44.1000

Table 7.4: Comparison of Optimal Configuration of RPHash with Other Algorithms on Real World Datasets

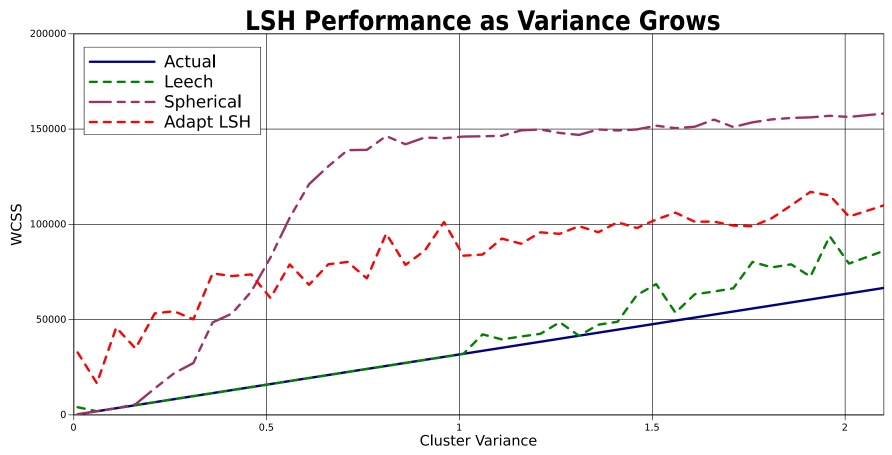


Figure 7.3.1: Decoders Comparison on Varying Cluster Variance

7.4 Streaming RPHash

In this section, we evaluate the performance of streamingRPHash and compare it against several other stream clustering algorithms, namely: Streaming k-means [35] implementation from S-Space [84], Damped Sliding Window [135], DStream [125] and Biased Reservoir Sampling [3] (described in Section 7.1.4).

streamingRPHash uses the optimal configuration described in Section 7.2. We apply the stream clustering algorithms on both real-world and synthetic data streams and evaluate their clustering results based on various external, internal and entropy based clustering validation measures. Runtime and memory usage of all algorithms are also captured on a computer having Intel Xeon X5675 CPU with 6 cores @ 3.07GHz supporting 12 hardware threads. The operating system is Debian 'stretch' (x86_64), running on 28 GB memory. In addition, we assess the impact of noise on stream clustering algorithms by injecting noise points in synthetic data streams. Finally, we perform a preliminary comparison of parallelization of streamingRPHash against streaming k-means. The results are described in the following subsections.

7.4.1 Real-World Datasets

The algorithms are set up to report clustering results in batches of 500 points. Performance results for the *HAR* dataset are shown in Figure 7.4.1. The plots are organized into two side-by-side graphs. The left graph shows the computed values at each 500 points interval and the right graph is a box and whisker plot that shows the median and quartiles of all of the sample points for each algorithm. The results show that streamingRPHash performs on par with the other algorithms. The runtime performance is better than most

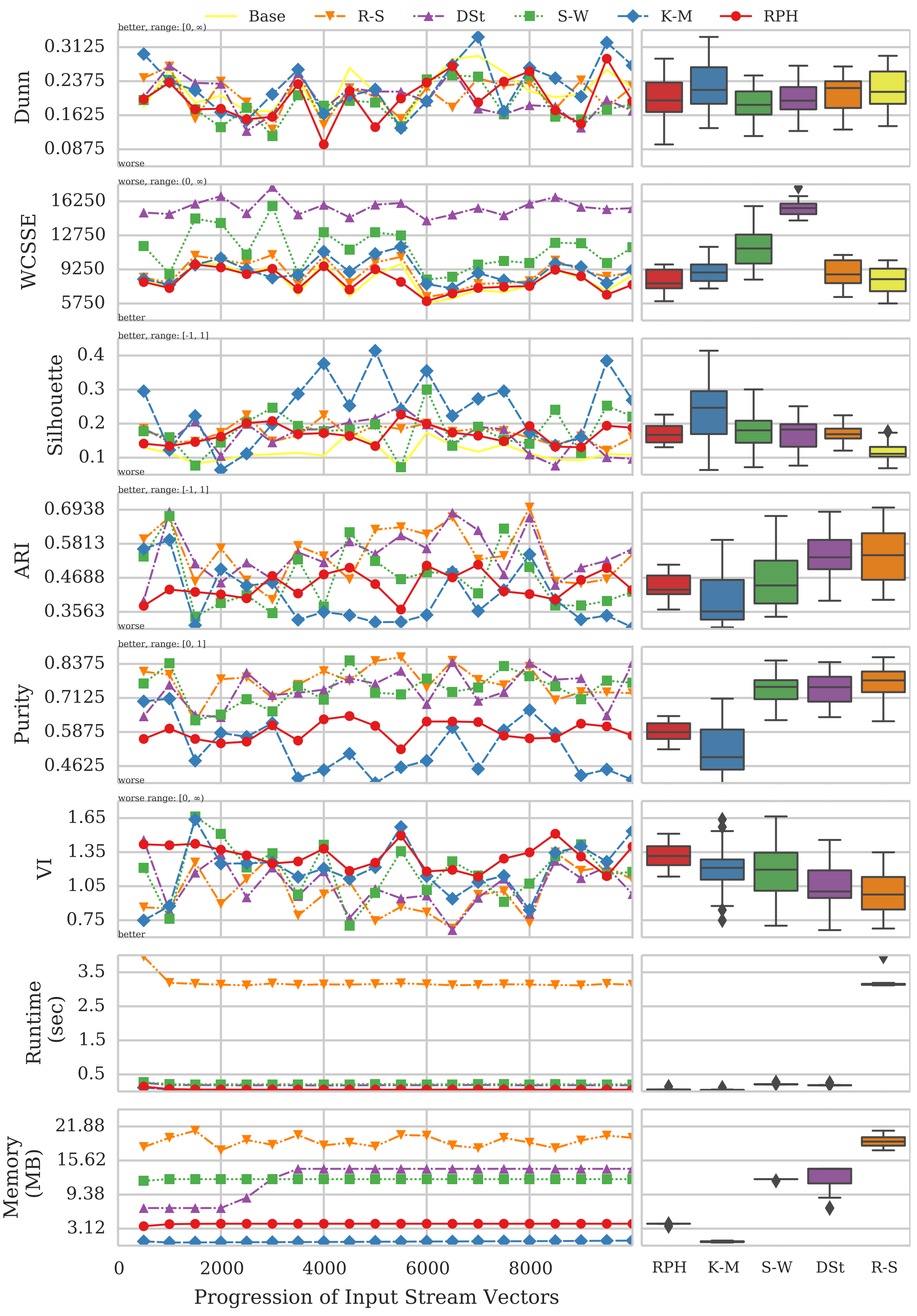


Figure 7.4.1: Clustering Smartphone Sensor Data Comparison

Clustering Results from Smartphone Sensor Data (abbreviations: RPH → streamingRPHash, K-M → Streaming k-means, S-W → Damped Sliding Window, DSt → DStream, R-S → Biased Reservoir Sampling, and Base → Base Value

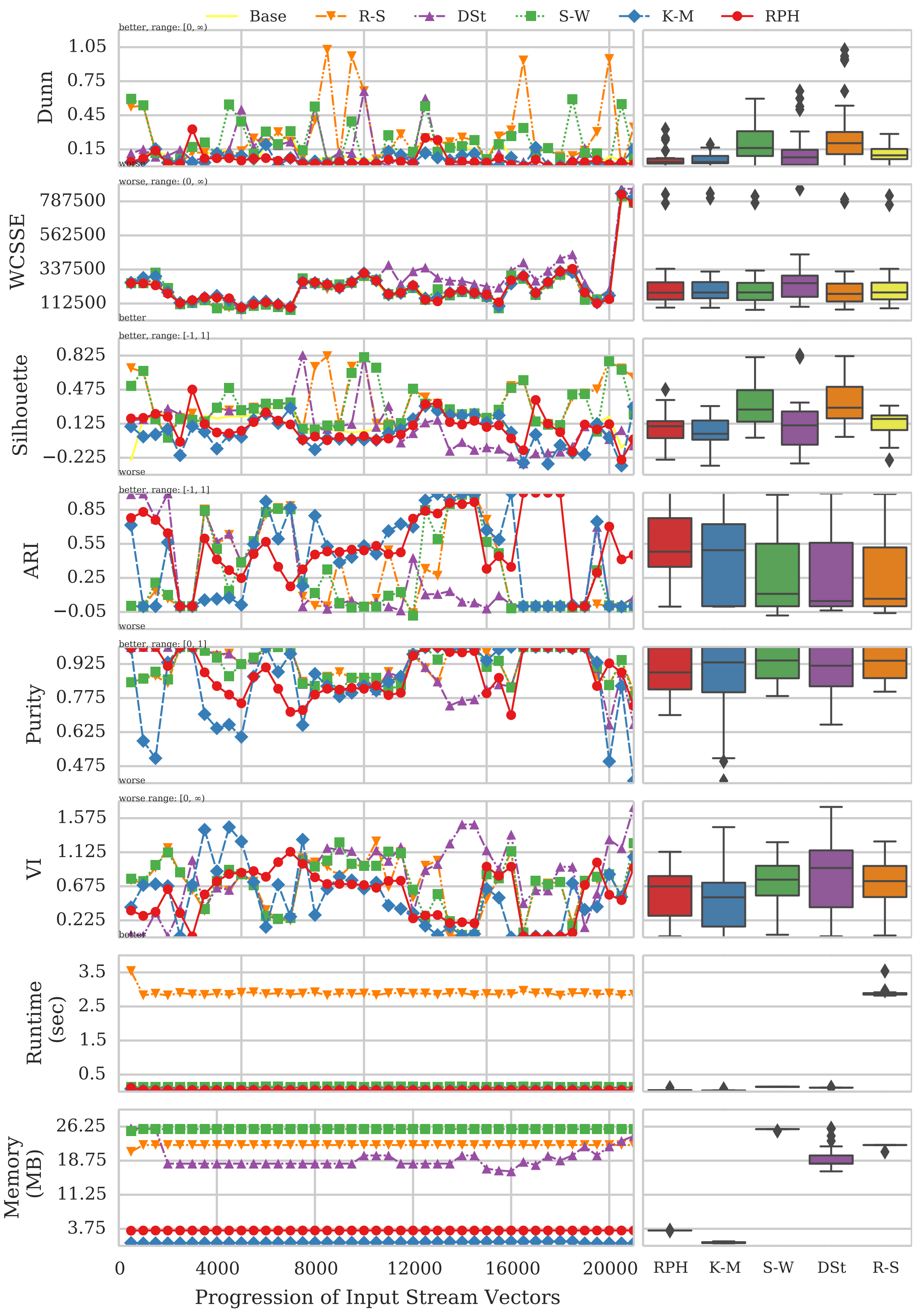


Figure 7.4.2: Clustering Results from WiFi Location Data

Clustering Results from WiFi Location Data (abbreviations: RPH → streamingRPHash, K-M → Streaming k-means, S-W → Damped Sliding Window, DSt → DStream, R-S → Biased Reservoir Sampling, and Base → Base Value and on par with streaming k-means. The memory usage is slightly worse than streaming k-means, but both are significantly better (less than half) than the other algorithms studied.

The second dataset is the *UJIIndoorLoc Data Set* [29]. Results from this dataset are shown in Figure 7.4.2. The plots have the same format as with the previous study. Again, the clustering accuracy of streamingRPHash is on par with the other algorithms while streaming k-means is the fastest and has a considerably smaller memory footprint.

7.4.2 Scalability Study

We study the dimensional scalability of stream clustering algorithms by evaluating their clustering results as the dimensionality of data streams increases. The purpose of this study is to assess and compare the accuracy, runtime, and memory usage of streamingRPHash to those of other stream clustering algorithms with increasing dimensionality. Labeled data streams of 20,000 points and 10 randomly placed clusters with random multivariate Gaussian distributions are generated using the R generator (as described in Section 7.1.2).

The dimensionality of data streams, which increases in a sequence from 100 to 5767, follows two polynomial sequences of the form $d_n = 1.5d_{n-1}$. The first sequence starts from $d_0 = 100$ and the second one from $d_0 = 125$. We do not inject noise points or outliers in these data streams as we inspect the impact of noise in the next section. The clustering validation measures are computed in batches of 1000 points and are plotted against dimensions of data streams. For ease and clarity of viewing the WCSSE plot, its values are scaled by diving by the number of dimensions.

Figures 7.4.3 and 7.4.4 show the performance results from this scalability study. The colored lines in each of these plots represent the mean value and shaded regions around the lines represent the variance of a measure over all the batches of data points. The shaded regions on the plots for DStream are wide and clearly visible as its measures have significantly higher variances than those of other algorithms. In regard to clustering performance, streamingRPHash performs optimally on all metrics with only one slight variation at 633 dimensions. This performance is on par with the other window and sampling streaming clustering algorithms, and outperforms both DStream and streaming k-means. The Runtime and memory requirements show a different story for Reservoir Sampling and Sliding window. Both of these algorithms require significantly more processing time and space, with significantly worse growth complexity. This plot shows a major strength of streamingRPHash, in that it achieves near optimal clustering in linear processing time similar to streaming k-means.

7.4.3 Impact of Noise

In this subsection, we assess the clustering results of streamingRPHash in the presence of random noise in data streams. We compare the clustering accuracy of streamingRPHash to those of four other stream clustering algorithms while gradually increasing the percentage of noise in a data stream. We arbitrarily choose a dimension = 759 from those used for the scalability study and generate several data streams with increasing amount of noise. The percentage of random noise is increased in an arbitrary sequence. All of

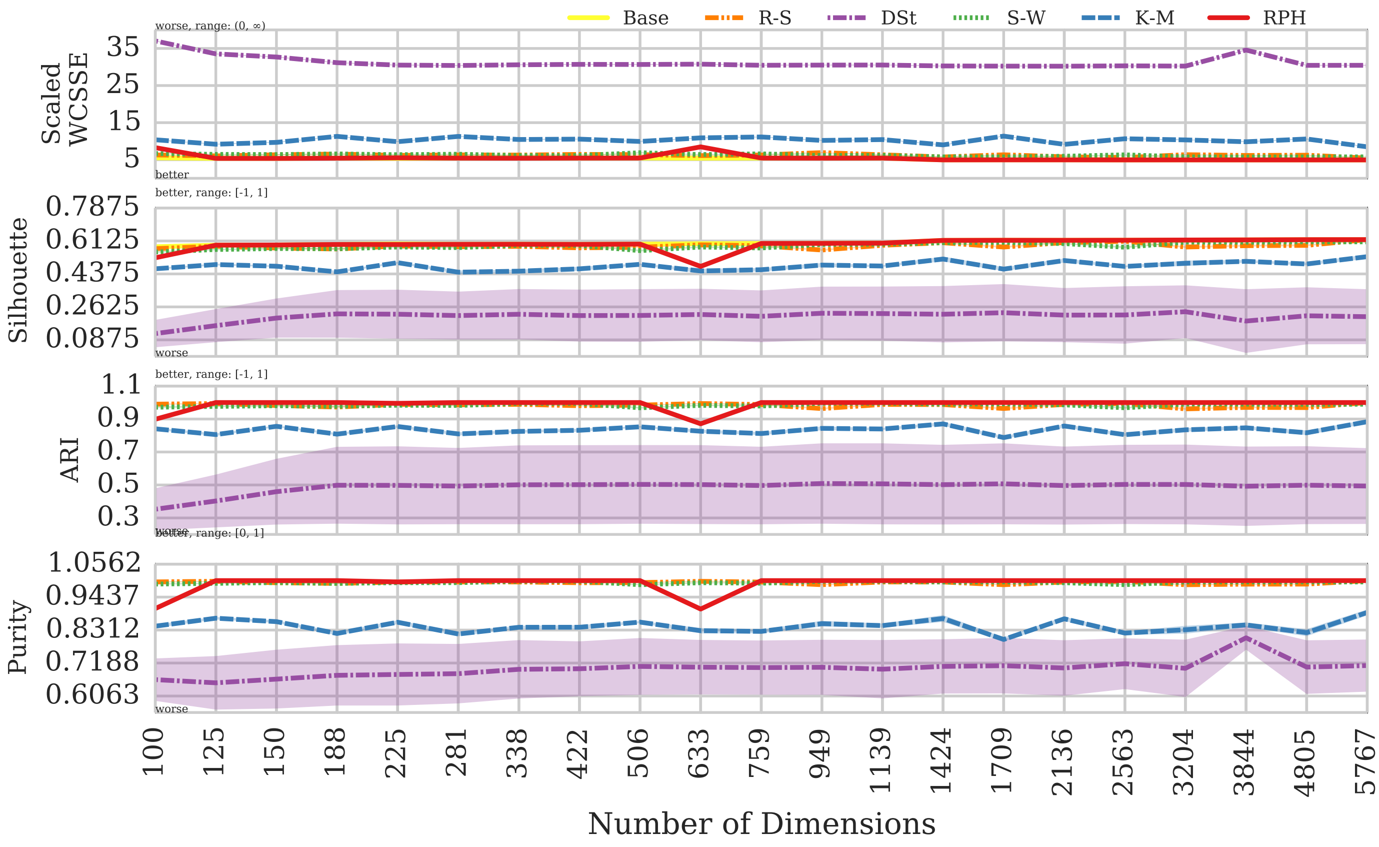


Figure 7.4.3: Scaling Comparisons External and Internal Measures

External and Internal Measures from Scaling the Number of Attributes (abbreviations: RPH $\rightarrow$ streamingRPHash, K-M $\rightarrow$ Streaming k-means, S-W $\rightarrow$ Damped Sliding Window, DSt $\rightarrow$ DStream, R-S $\rightarrow$ Biased Reservoir Sampling, and Base $\rightarrow$ Base Value

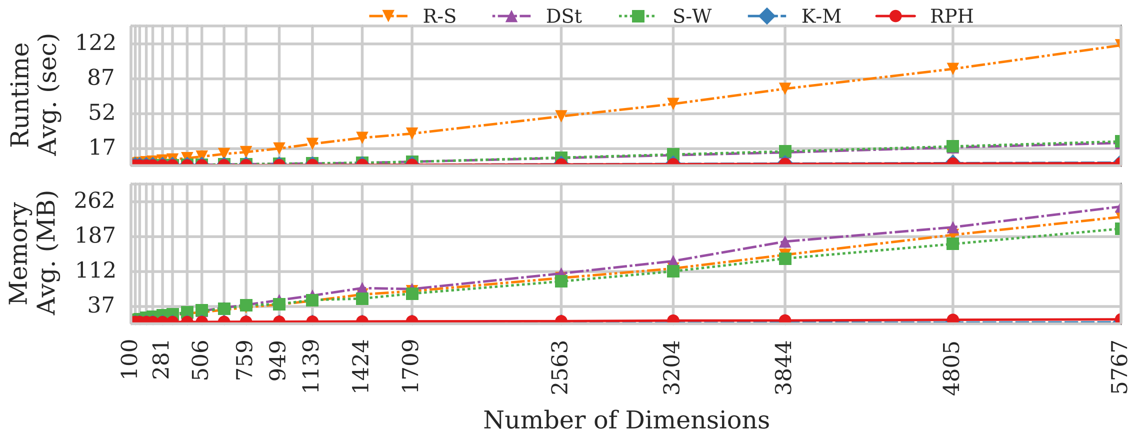


Figure 7.4.4: Runtime and Memory usage from Scaling

Runtime and Memory usage from Scaling the Number of Attributes (abbreviations: RPH → streamingRPHash, K-M → Streaming k-means, S-W → Damped Sliding Window, DSt → DStream, R-S → Biased Reservoir Sampling, and Base → Base Value

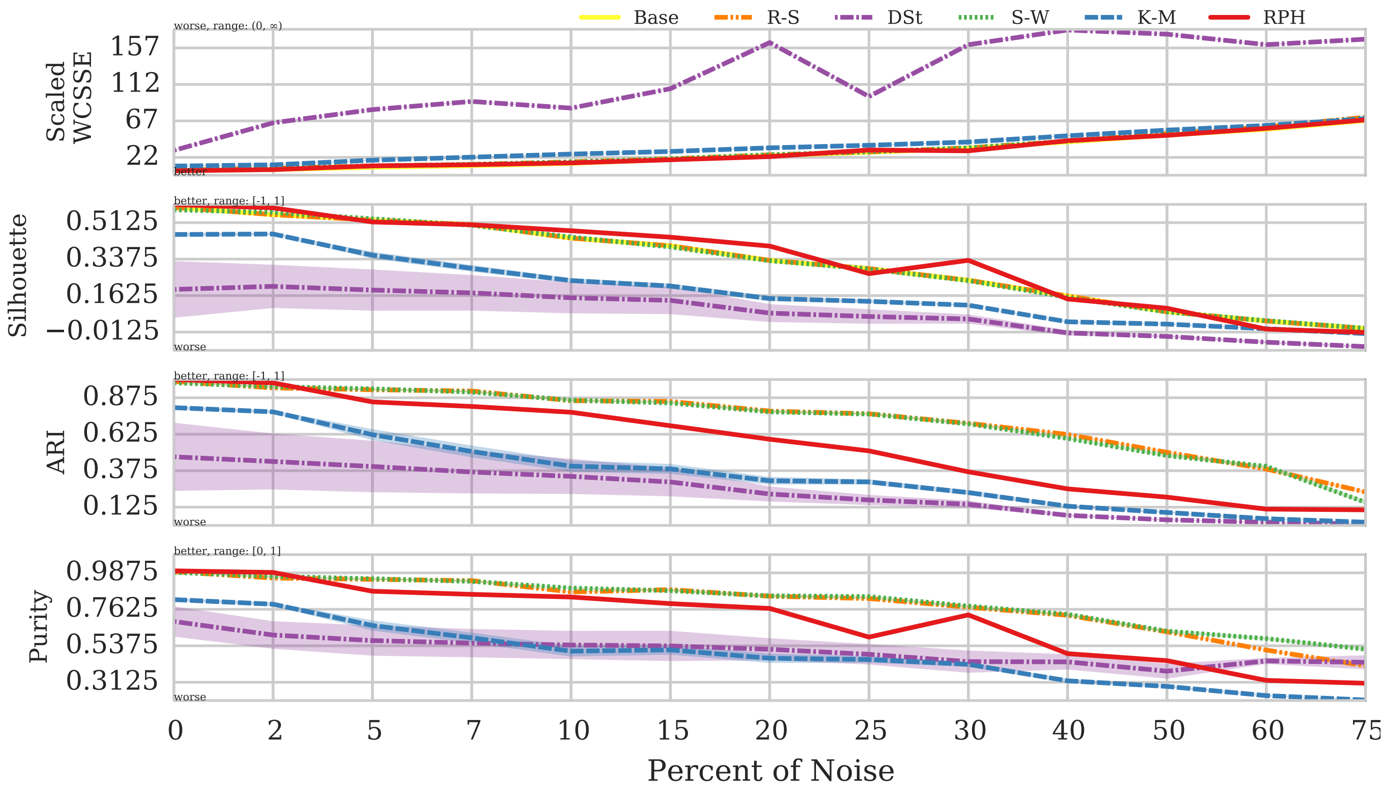


Figure 7.4.5: Injecting Noise into the Data Source.

these noisy data streams are generated using the R stream generator, where each data stream has 20,000 points and 10 randomly placed clusters with random multivariate Gaussian distributions. Noise points are assigned labels according to their nearest cluster centroids. We use the same clustering validation measures, same batch size (1000 points) and similar plots as with the scalability study. The results are shown in Figure 7.4.5. The only difference here is that the horizontal axes represent percentage of noise in data streams. In these figures we see that streamingRPHash performs optimally in regard to the other algorithms and even baseline for WCSSE and Silhouette, as noise is injected. It performs mediocre in the two external metrics ARI and cluster Purity, besting streaming k-means and DStream.

7.5 Tree-Walk RPHash Performance

In this section we explore the clustering performance of the Tree-Walk RPHash variant (TWRP). The TWRP variant of RPHash is what we would consider our best of breed RPHash algorithm. TWRP uses information about the data distribution to build adaptive LSH functions, and also more thoroughly explores the relationships between clusters and encapsulated clusters. We first compare TWRP to various other clustering algorithms on real world data, then we explore the scalability, and noise robustness of TWRP.

7.5.1 Real World Data

The largest and most stable dataset that we've explored so far is the *Human Activity Recognition* dataset [92] dataset. The attributes of this dataset are available in Table 7.1. We compare TWRP to standard RPHash as well as our usual set of standard clustering algorithms, consisting various Agglomerative methods, standard k-means, and SOTA. The results can be found in Table 7.5. We captured the runtime (in Secs) performance for all of the algorithms and results were obtained on a computer with an Intel Core(TM) i7-4770 CPU with 4 cores @ 3.40 GHz and 32 GB memory. Among the compared algorithms TWRP improves upon the performance of standard RPHash and performs on par with k-means and Ward's linkage agglomerative clustering. The main benefit however is in its timing results. TWRP is 100x faster than k-means, and 2000x faster than Ward's linkage agglomerative clustering.

Algorithm	ARI	Purity	WCSSE	Time
Kmeans	0.461	0.6	182168.7	24.746
Single Link	0	0.189	556519.1	413.88
Complete Link	0.327	0.377	222044.4	414.33
Average Link	0.332	0.359	236142.8	414.1
Ward's	0.491	0.66	191441.1	414.45
SOTA	0.314	0.397	210490.1	14.244
RPHash	0.363	0.508	210628.6	0.4838
TWRP	0.449	0.609	194688.7	0.2617

Table 7.5: Clustering Performance and Timing

Synthetic Data

In this experiment we generated datasets with successive dimensionality in $\mathbb{R}^{100}$ to $\mathbb{R}^{7000}$ space. Each dataset consists of 10000 vectors equi-probable chosen from a set of 10 univariate Gaussian clusters. Figures 7.5.1,7.5.2,7.5.3 give the clustering performance of TWRP compared to other clustering algorithms for the metrics, ARI , Purity and WCSSE on the generated datasets. TWRP performs optimally for all dimensions, equaling the agglomerative methods. It is also stable, compared to standard RPHash and k-means.

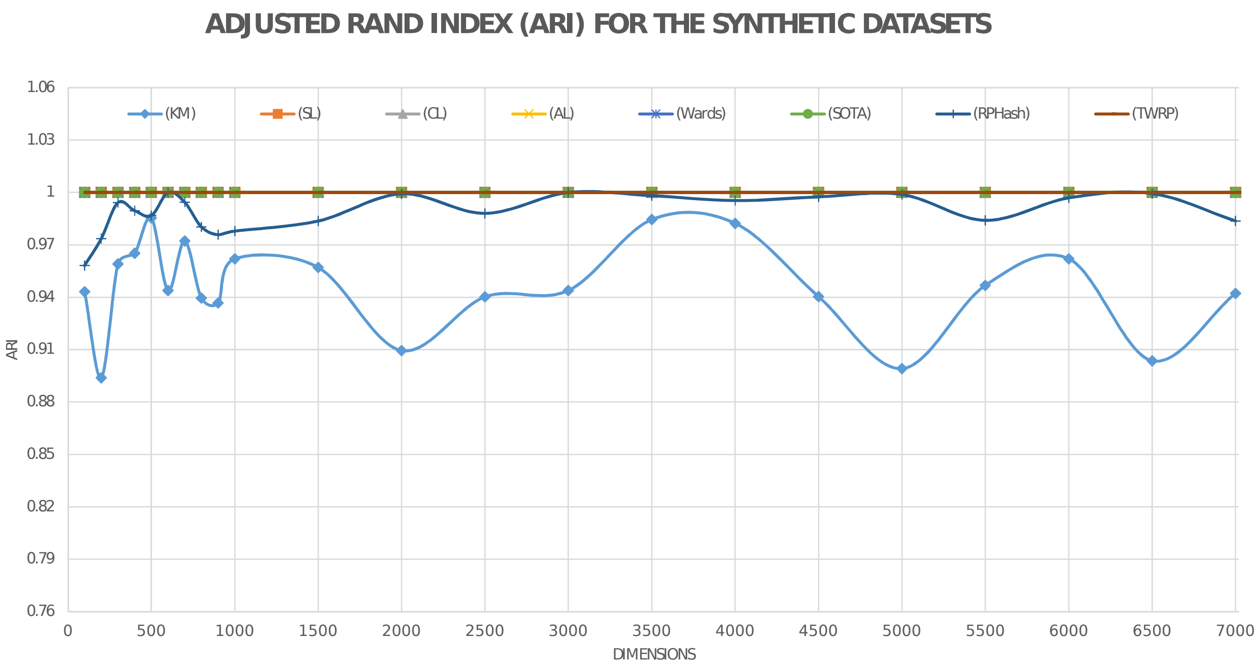


Figure 7.5.1: ARI Synthetic

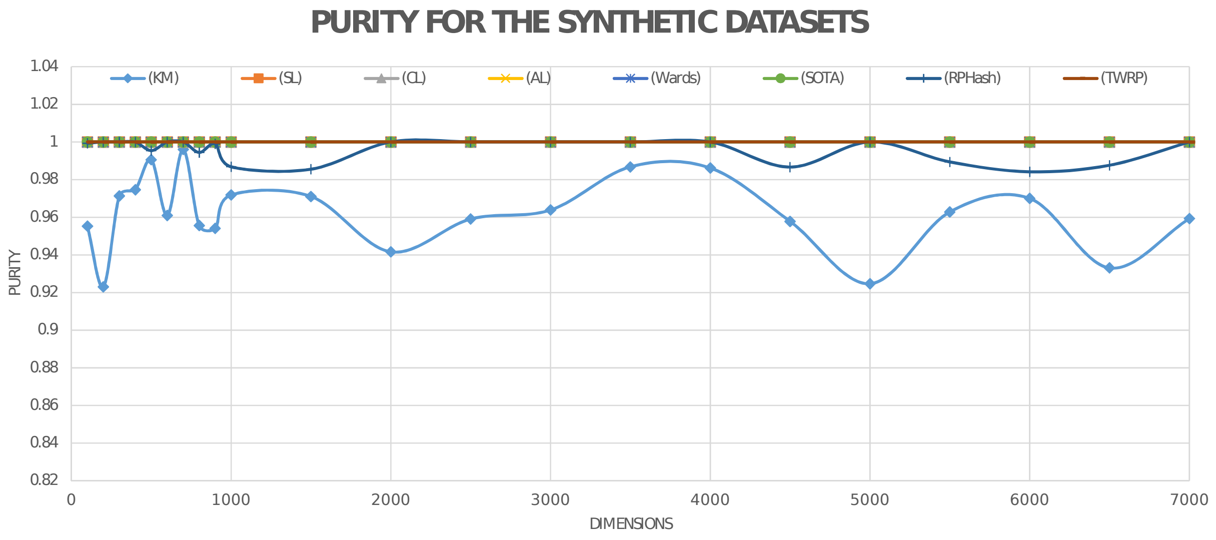


Figure 7.5.2: Purity Synthetic

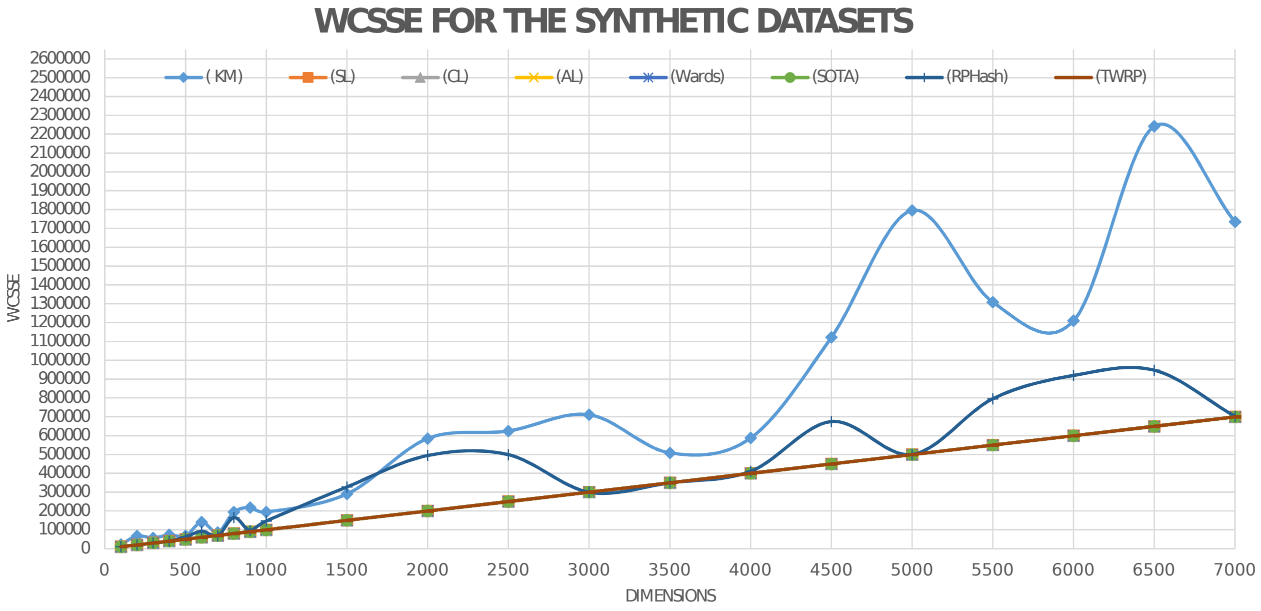


Figure 7.5.3: WCSSE Synthetic

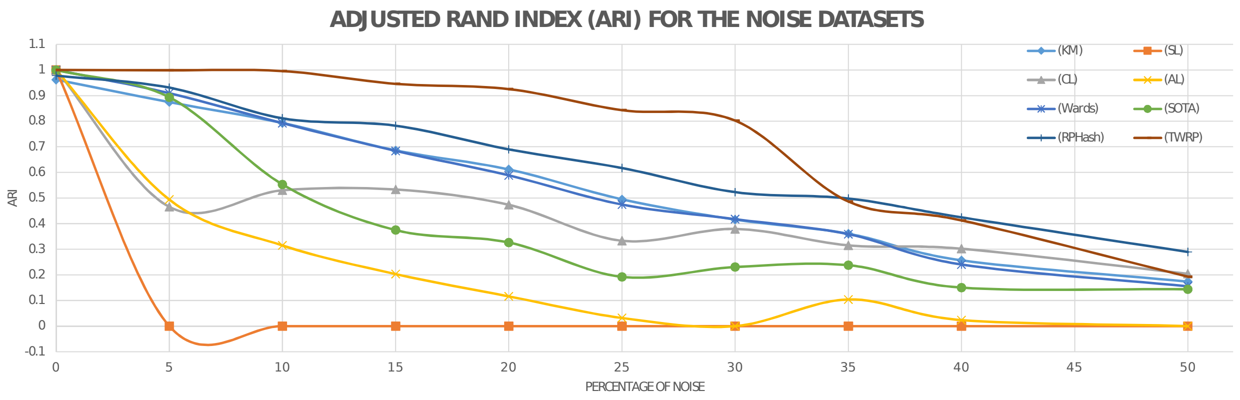


Figure 7.5.4: ARI Noise

Noise Stability

An important test of a clustering algorithm's versatility is its robustness to noise. In general, real world data will contain at least some level of noise due to systematic errors in measurement. In this experiment we will generate datasets with varying amount of noise. We chose a dimensionality of 1000 dimension and inject noise varying from 5 percent to 50 percent of the overall dataset size in increments of 5 percent. The noise vectors generated were assigned the label of the closest cluster center. The results of the noise injection tests are plotted in Figures 7.5.4,7.5.5,7.5.6, with ARI, Purity, and WCSSE being tracked respectively.

We observe that until the noise percentage reaches 40 the accuracy of both the RPhash algorithms are far better than the other six standard algorithms. Thereafter as the noise grows beyond 50 percent and the signal to noise ratio is very low, all the algorithms tend to perform poorly.

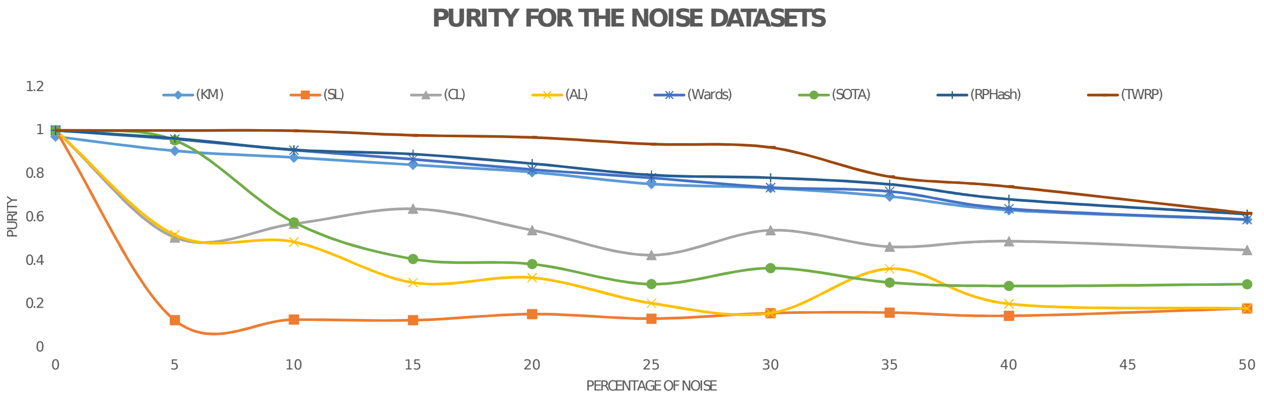


Figure 7.5.5: Purity Noise

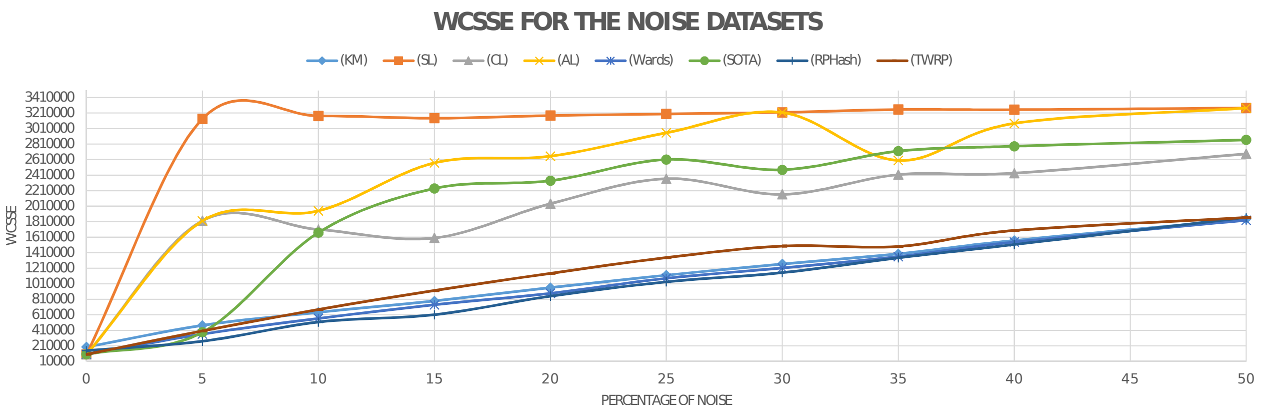


Figure 7.5.6: WCSSE Noise

Scalability

Table 7.6 summarizes the timing results on synthetic datasets. The data sets were generated using the R data generator discussed in Section 7.1.2 to generate 10000 vectors from $k = 10$ univariate clusters, with varying dimensionality. Results for all clustering algorithm timings were obtained on a computer with Intel(R) Xeon(R) E5-2670 @ 2.6 GHz with 16 cores and 64 GB RAM. While Table 7.6 gives a detailed measure of the scalability for all comparison clustering algorithms, Figure 7.5.7 shows difference between the RPHash algorithms and only the two faster comparison clustering methods. This is due to the agglomerative methods being so much slower that they would skew the output plot making it useless for comparison. In this experiment we see again that the scalability of the RPHash algorithms far exceeds the other compared methods. While SOTA and k-means achieve, what appears to be linear complexity growth, the RPHash methods, appear constant in comparison. This of course is not true and they share a similar linear complexity with SOTA and k-means. We can verify this in the scalability Table 7.6, but should note that the difference in processing time, between RPHash algorithms, SOTA, and k-means is significant.

7.5.2 Tree Walk RPHash (TWRP) Results on Word2Vec Data

In this section we used the GoogleNews [105] Word2Vec dataset to compare the clustering and timing performance of TWRP with k-means++. Word2Vec data is particularly interesting, because it often contains very large, high dimensional, and dense, data vectors that often occupy lower dimensional embeddings.

Dimension	KMeans	AverageLinkage Link	SOTA	RPHash Leech	TWRP
100	3.9656	47.46	13.758	0.4890	0.1638
300	13.3796	194.121	23.797	0.4782	0.1638
500	32.92	454.777	36.046	0.5594	0.3005
700	77.7298	672.759	48.963	0.6174	0.3642
900	117.3675	929.6	61.759	0.7052	0.4078
1000	142.2341	1064.178	68.86	0.7206	0.4318
1500	237.0007	1789.142	96.795	0.8233	0.5392
2000	366.0743	2450.813	127.972	0.8796	0.6548
2500	431.5876	3108.654	155.968	0.9997	0.8050
3000	542.0223	3771.886	185.23	1.1122	0.8917
3500	631.8423	4435.349	220.562	1.2421	1.0229
4000	741.915	5080.802	248.669	1.3157	1.1498
4500	811.3911	5725.061	274.983	1.3986	1.2873
5000	909.223	6362.574	301.314	1.5095	1.4128
5500	975.2703	7006.91	324.314	1.5977	1.5392
6000	1076.882	7645.155	359.911	1.6805	1.6157
6500	1187.6062	8278.964	400.767	1.7933	1.7266

Table 7.6: Scalability Study.

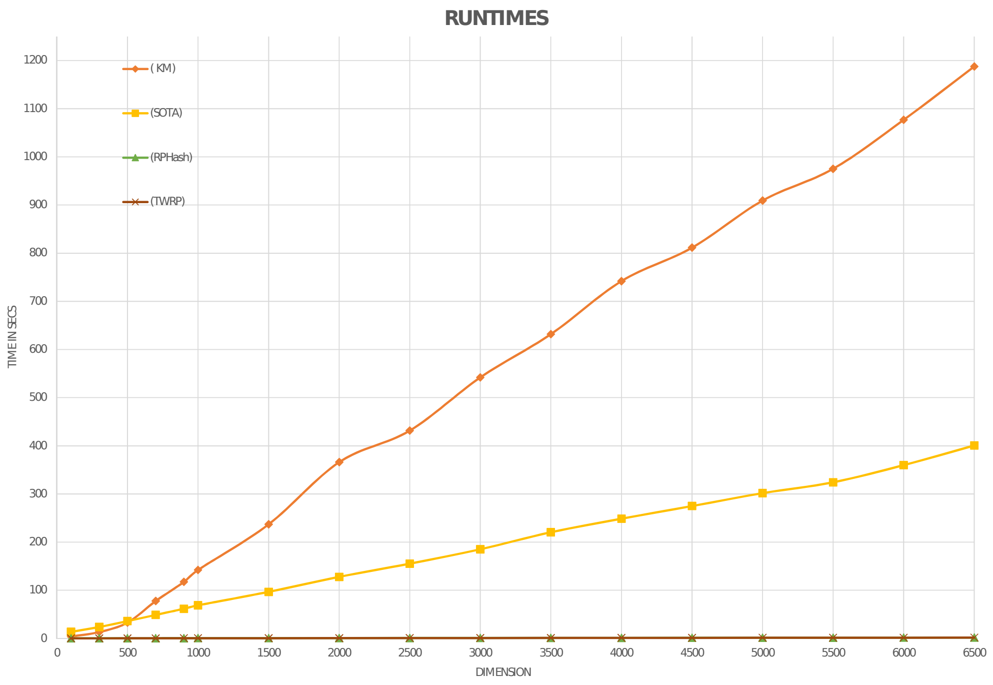


Figure 7.5.7: SCALABILITY Synthetic

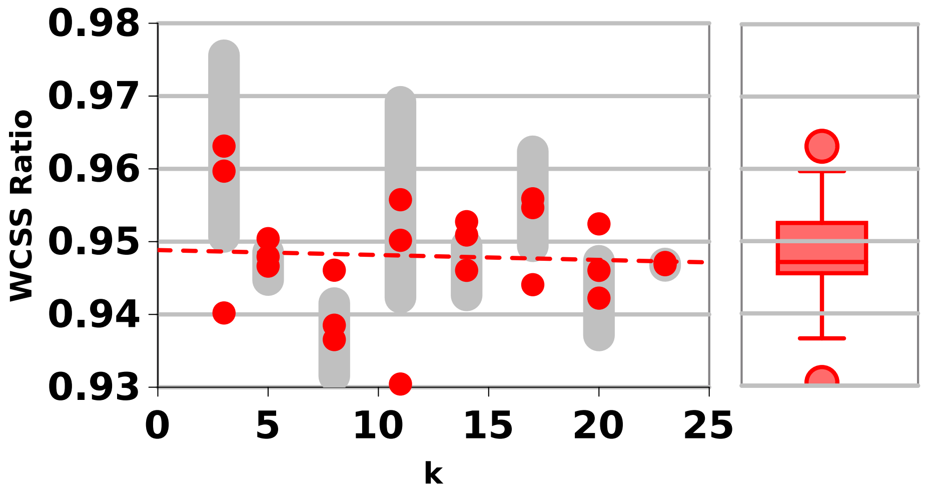


Figure 7.5.8: WCSSE KMeans++ to TWRP Ratio

However, due to the size of the observations, it is unlikely that a ground truth will be available. Observation can verify certain subjective aspects of the data clustering, such as two similar words or concepts being labeled in the same cluster, but an overall labeling is usually unavailable. For this experiment, we rely on the WCSSE only to act as a performance metric. We could theoretically compute other external metrics, such as Silhouette and Dunn Index however given the size of the GoogleNews data set, it is computationally infeasible. In addition, we do not have a defined number of clusters. Figure 7.5.9 gives the WCSSE for a sequence of possible values of k for TWRP and k-means++. Likewise, Figure 7.5.8 gives the ratio of the k-means++ WCSSE and TWRP WCSSE. Under the assumption that k-means++ finds nearly the optimal WCSSE, we can regard TWRP as an ϵ-error equivalent. As TWRP appears to be outperformed in regard to WCSSE, by k-means++, it is important to focus on some of its advantages, namely its speed. Figure 7.5.10 gives a comparison of the two algorithms in regard to processing time. As, is clear in other runtime performance tests, the TWRP algorithm scales linearly with the data size, and overall requires far less computation than k-means, and similar algorithms.

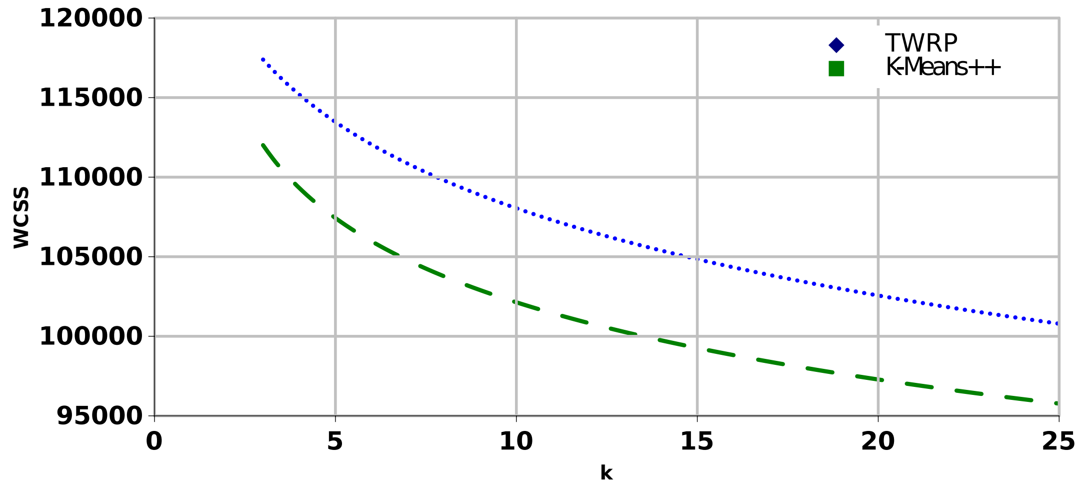


Figure 7.5.9: WCSSE Comparison Results

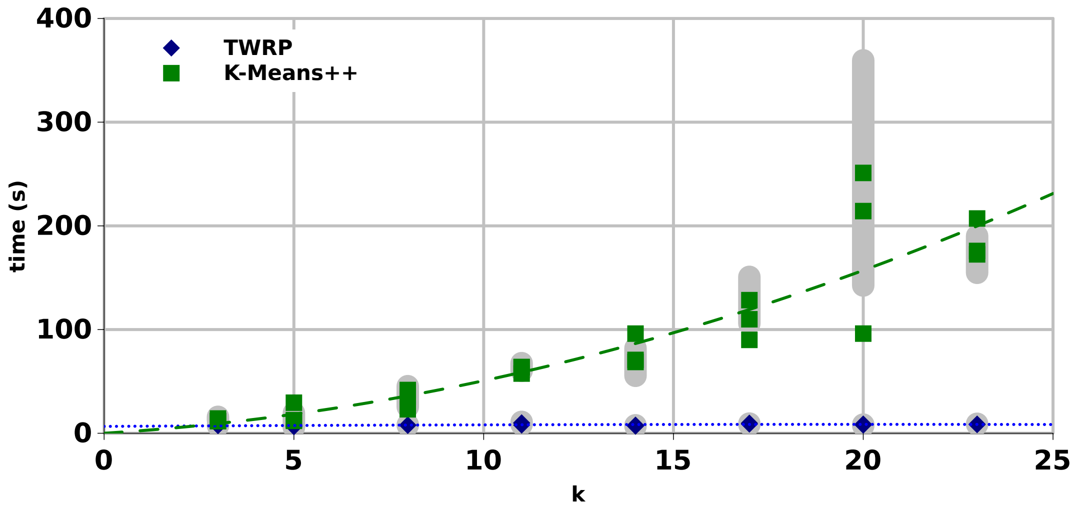


Figure 7.5.10: Processing Time Comparison

7.6 RPHash Parallelism

RPHash was designed with the intent of being naively parallelizable. As with many parallel adaptations of well-known machine learning algorithms, the influence of Amdahl's Law appears during parallelization efforts due to unavoidable sequential bottlenecks as the number of processing nodes increases.

In this section we compare the three proposed variants of RPHash in regard to parallel speedup. Our testing was done on an 8-core Intel Xeon CPU with 4 threads per core, for a total of 32 sequential threads. The parallelism was implemented similar to our target platforms of map-reduce and spark streaming interface.

Standard RPHash

The standard RPHash algorithm parallel implementation consists of an equal partitioning of vectors to processing units. In the map phase, each processing unit applied the per vector process to their respective subset of vectors in parallel. The threads then performed a log reduction sum of the counts of hash collisions in the thread's respective subset of vectors. The result of the first phase is a set of the densest hash bucket, IDs. The second phase then proceeds byt redistributing the data vectors to processing nodes, and accumulating a centroid for vectors that produce hashes that collide with the dense set of hash bucket IDs. These centroids are then reduced through a weighted sum log reduction of the centroids.

TWRP

The tree walk variant consists of a similar equal partitioning of vectors among processing nodes as the standard RPHash variant. Similarly, each node accumulates a separate MinCount data structure. Once all vectors have been processed, the MinCount structures are merged, for all indices. The result is the complete MinCount structure, from which the tree based, off-line step of TWRP proceeds sequentially. In our tests, we only tested the sequential off-line step, however, various parallel depth first search methods could be employed to improve performance and parallel speedup.

Streaming RPHash

The streamingRPHash variant is discussed last because it differs the most from the previous algorithms. Namely, the streamingRPHash algorithm has a different data access structure. streamingRPHash receives vectors one after another, and updates a Count-Min sketch. To parallelize the streaming version we process the projections and hashing of vectors in parallel, but must use a thread safe priority queue, for updating frequent centroids. While thread safety tends to inject sequential bottlenecks in otherwise parallel code, the bottleneck is often reduced, by the low probability of colliding hashes for frequent and infrequent vectors. For highly clusterable data the ratio of frequent and infrequent vectors is low. While for real world vectors with a high degree of noise, the ratio is much higher, resulting in better parallelism.

7.6.1 Parallel Speedup Results

Figure 7.6.1 shows the parallel speedup as a function of processing cores for the 3 RPHash variant algorithms proposed. The data source consists $n = 10^7$ vectors in $\mathbb{R}^{1000}$ uniformly distributed among 10 Gaussian distributed clusters, and 1 uniformly distributed 'noise' cluster. The experiment consisted of generating the vectors and storing them in main memory, then iteratively increasing the number of processing units for each of the three described RPHash algorithms. The results of these tests show that the standard RPHash has the greatest potential for distributed scalability, followed by TWRP, then streamingRPHash. All of the speedup curves show characteristic diminishing returns of Amdahl's Law curves.

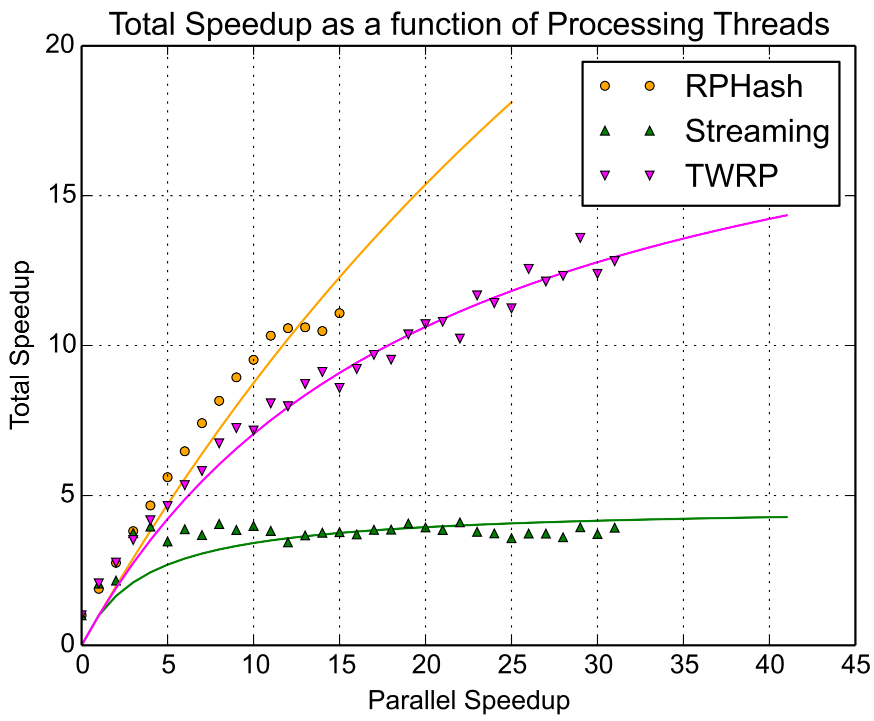


Figure 7.6.1: Speedup Comparison between RPHash Algorithms

7.7 Security Performance

Due to the inability to anticipate all possible cryptographic attacks on de-anonimization, a qualitative measure of data obfuscation is developed for comparing a fully qualified vector $v \in \mathbb{V}^d$ with its corresponding projected vectors $v' \in \mathbb{V}^d$. This metric referred to as k-anonymity allows us to evaluate the probability of de-anonymization of a vector.

v' is the inverse projection of $u \in \mathbb{V}^s$ that results from the random projection of v. Destructive data obfuscation occurs if the distance between v' and v is greater than the distance between v and some other vector $\hat{v} \in \mathbb{V}^d$. The inverse projection matrix $R^{-1}_{d \to s}$ will be used to map u back to v's original subspace. The two equations below describe the projection of v to u and the theoretical inverse of the projection from u to v' under the matrix transform $R_{d \to s}$ where $d > s$.

$$u = \sqrt{\frac{n}{k}} R^T_{d \to s} v$$

$$v' = \sqrt{\frac{k}{n}} u^T R_{d \to s}^{-1}$$

The above inversion is theoretical however due to the orthogonal projection $R_{d \to s}$ being non-square and not invertible. Therefore, for a projection matrix $R_{d \to s}$ the least squares solution $\hat{R}_{d \to s}^{-1}$ will serves as the optimal inverse of the projection. Even in the not strictly orthogonal random projection case (as in Achlioptas [2] and RPHash) the least-squares solution will result in an over-determined system of equations. Which implies that any pseudo-inverse projection of a vector in V^s to V^d will result in unrecoverable data loss for non-trivial cases (*i.e.*, $<\mathbf{0}>$,$<\mathbf{1}>$). The goal in testing the security of RPHash is to show that the data loss is sufficient to make it impossible for an attacker to re-associate the projected vectors. A formal definition of the requirement for destructive data obfuscation follows:

$$s(v, v') = ||v, v'||_2$$

$$\forall \{v, v'\} \in V, \exists \hat{v} \in V : s(v, v') > s(\hat{v}, v) \text{ where } \hat{v} \neq v \text{ or}$$

$$Pr(NN((v \cdot R) \cdot \hat{R}^{-1}, V) = v) \lesssim \frac{1}{||V||}$$

Due to the often sensitive nature of medical data, we give a real world example based on the MIMIC II [114] data set. Figure 7.7.1 shows the k-anonymity curve for vector recoverability demonstrating the effectiveness of random projection for providing vector anonimization. The overall bi-clustering in full and reduced dimensions showed little to no degradation over full and reduced subspaces down to 10 dimensions. This is further corroborated on additional data in Bingham [31]. Following the above definition of destructive data obfuscation, we constructed a test on the MIMIC II data. The Moore-Penrose inverse generated from the reciprocal of the singular value decomposition diagonal matrix is generated from the projection matrix and used as the least squares inverse projection to remap the projected vectors to their original subspace. The nearest neighbor method was then used to query the original set of vectors to see if a vector can be re-associated with itself. In the results it is clear that as dimensionality increases, the probability of re-association converges to random coincidence of a Bernoulli trial $((1 - (1/n))^n) \approx .363$. The successful

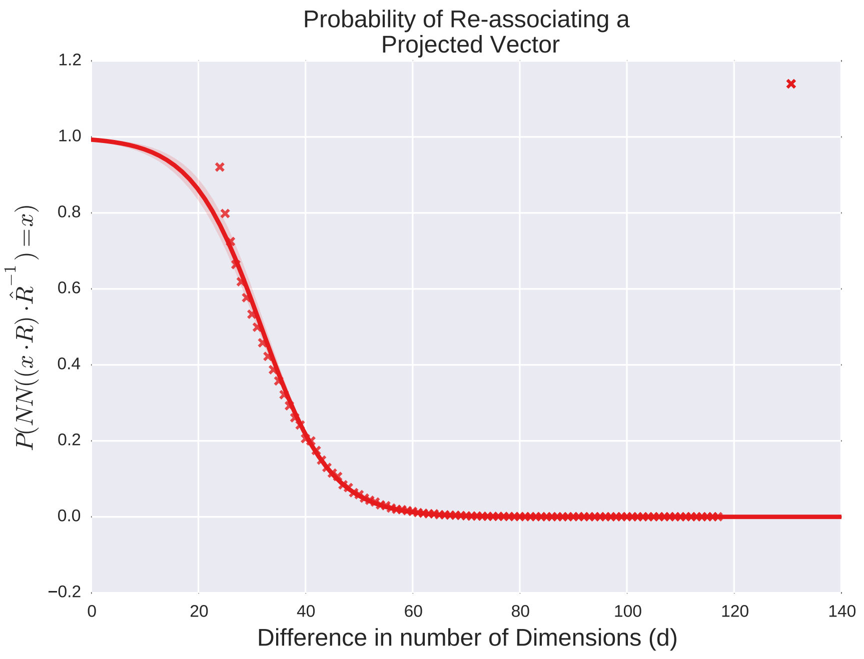


Figure 7.7.1: Probability of Vector Re-association for MIMIC II BioMetric Signatures

re-associate rate follows an inverse power-law like distribution.

In regard to security performance, it is fairly clear that data becomes unrecoverable when the difference between the original data embedding and the projected space exceeds 75 dimensions. Given that RPHash's native clustering space is 24 dimensions, this case occurs for a wide variety of high dimensional datasets, namely those that exceed 99 dimensions.

Chapter 8

Algorithmic Complexity and Theory

In this section we analyze the performance of the proposed variants of RPHash in regard to their asymptotic complexity. Although adversarialy crafted datasets can be NP-Hard for iterative update methods like Hartigan and Wong's k-Means algorithm [128], they often fair much better in practice. In common practice k-Means tends to be described as having $\Theta(n \log\log(n))$-complexity, or sometimes simply $\Theta(n)$. While this fact is useful in terms of scalability, the effects of constant factors can vary widely in practice in regard to processing time. For this reason we prefer the results presented in Chapter 7.3.1 for exact timing results. None-the-less we provide a complexity analysis for RPHash, and in all cases find them to be linear, and predictable.

8.1 RPHash Complexity

In the experimental section we empirically show that RPHash has linear complexity. However in this section we take a more theoretical approach to proving this claim. Namely we show that RPHash has asymptotic complexity, that is linear to the input size.

8.1.1 Algorithmic Complexity

The simplest form of RPHash, the two pass approach. This approach has a rather simple complexity derivation. The algorithm passes once over all vectors, performing a projection and a hash. The dimensionality of the projections m depends on the number of input vectors n and not on the dimension of the vectors d. From JL-Lemma, we set this dependence as $m = \Theta(\log(n))$, where the problem input space is $N = nd$. Projection requires $\Theta(\log(N/d)N/3)$ steps, with generally small constant factors using the db-friendly projection of [2]. When $\log(N/d)$ is near c^3,c is constant and projection is linear. Hashing on the projected vectors is performed in constant time for certain lattices (*e.g.,* Λ_{24}, E_8), or using $m = \log(d)$, $\Theta(2\log(d))$-time. Updating the counters in the first phase is performed in constant time, through a hash-table, or count sketch. Thus, the overall complexity of phase one is $\Theta(\log(N/d)N/3 + C) = \Theta N$.

The second pass of 2-pass RPHash consists of the same process as the first pass to project and look up vectors and their respective hash ids. However in the second phase we also compute the centroids for dense hash buckets. The worse case complexity of the update process is $\Theta(nd)$ or $\Theta(N)$ assuming all vectors are contained in a dense cluster. The overall complexity of the second pass is $\Theta(N + N) = \Theta(N)$. Examining both phases, we have an overall complexity of $\Theta(\log(N/d)N/3 + C) + \Theta(N) + \Theta(N) = \Theta(N)$ which when $\log(N/d) \approx 3$, is effectively linear.

8.2 Streaming RPHash Complexity

In this section we discuss the compute and memory complexities of the streaming variant of RPHash (streamingRPHash. One of the principal goals for a streaming algorithm, and basic requirements, is that the computational complexity is at least sub polynomial and the Storage complexity is sub-linear. We prove both of these claims in the following two sections.

8.2.1 Computation Complexity

One of the earliest streaming solutions for the approximate k-*HH* algorithms is the Count-Min sketch data structure [51]. Count-Min sketch can be used to solve the approximate k-*HH* problem with only the addition of a priority queue using $\Theta(\frac{1}{\epsilon}\log\frac{1}{\delta})$ space with error $(1-\epsilon)f$, where f is the minimum frequency $\frac{m}{k}$ to be considered frequent. The complexity of streamingRPHash is:

$$\Theta\Big(NP\Big(\frac{Md}{3} + \log(d)\big(\log(k) + d + 2d^2\big)\Big)\Big)$$

when using P database friendly projections from M to d, SLSH decoding complexity ($2d^2$), a balanced priority tree (with $\Theta(\log(k))$ insertion and constant time removal), and a blurring factor $\log(d)$. The projection step $\frac{Md}{3}$ tends to dominate when $M >> d$. If we remove the constant factor, constant number of projection P and subspace d, we have complexity of $\Theta(NM)$, leaving only the input and thus, the algorithm is linear in n.

8.2.2 Storage Complexity

Streaming algorithms are potentially unbounded in size. While the limit is finite, it does impose an additional storage requirement on our algorithm. In particular, our algorithm must grow sub-linear, otherwise the stream is no different than reading a large data set into memory.

Using the Count-Min sketch to store the approximate hash counts gives us a bound on storage complexity that is a product of the stream size and desired error rate. Cormode [51] shows that the storage complexity of the Count-Min Sketch with error tolerance ϵ, where $\epsilon = 1 - \delta$ is $\Theta(Bln(\frac{1}{\delta}))$, where B is the number of independent objects that need to be counted. If then use a bounded kHH-Tree of size $klog(k)$ to maintain the $klog(k)$ densest vector centroids, we come to a total space requirement of

$$\Theta((dklog(k))ln(\frac{1}{\delta}))$$

If we assume $k \approx \log\log(n)$. We get a final complexity of

$$\Theta(d \log\log(n/d) \log\log\log(n/d)) = \Theta(d \log\log(n))$$

8.3 TWRP Complexity

The complexity of TWRP is similar to that of streamingRPHash. More precisely, consider its complexity from the perspective of a per vector complexity, with an on-line and off-line step. For each vector TWRP must compute the projection, and update the sketch. Projection using the db friendly approach of Achlioptas [2] can be performed in $\Theta(dm/3)$ operations where d is the original dimensionality, m is the projection sub-dimension and $m = \Theta(\log(n))$ where n is the total number of input vectors. Assuming n is large, but not infinite, the projection step is $\Theta(d\log(C))$, where C is a constant related to $log(n)$ by the JL lemma. From the complexity analysis of Count-Min Sketch for streamingRPHash, we have an additional complexity factor of $\Theta(ln(\frac{1}{\delta}))$. Following this, the time to update all levels of the hash are consequently the tree depth and dimensionality of the projected subspace. Leading to an overall per object complexity of $\Theta(dm^2ln(\frac{n}{\delta}))$. The Count-Min Sketch update factor is effectively constant for a reasonable chosen $\delta = .01$. Furthermore, following Johnson-Lindenstrauss's bound on sub-dimension representations, reasonably $m = \Theta(\frac{\log(d)}{\epsilon})$, resulting in a complexity of $\Theta(md\log(C)\log(C)) = \Theta(md\log^2(C))$.

The off-line step consists of the min-cut tree exploration. The exploration follows a depth first search tree-traversal for candidate clusters, which has a worse case complexity when exploring all non-leaf nodes of $\Theta(2^{d-1})$.

8.3.1 Storage Complexity

The count-min tree is generally never explored in its entirety. Instead only nodes corresponding to dense hash regions are traversed. Furthermore if we store the counts approximately in the Count-Min sketch, the storage requirement can be further reduced. Therefore if we only traverse dense clusters, using a Count-Min Sketch, we get the standard k-Heavy-Hitter formulation. This formulation has storage complexity $\Theta(ln(\frac{n}{\delta}))$ for n items, and a desired error $\delta = 1 - \varepsilon$. Although the Count-Min Sketch size is parametric in the desired error bound, for acceptable error rates, we must store a sketch of centroids that is at least $\Theta(mln(\frac{n}{\delta}))$. This will violate the streaming data requirement when $\delta > n^{-n/m}$. However, we can use the tree structure to our advantage, to correct errors after the fact.

Count-Min Tree Error

Interestingly, this leads to a boost in the general Count-Min Sketch error, because we can recover errors using adjacent parent and sibling nodes in the induced space cut tree. In other words, by exploring neighboring tree nodes, we can confirm or refute the approximate count of a node. This is particularly useful for the possible occurrence of a short LSH ID, colliding with a much longer ID. Which would result in a false branch. To illustrate this, we can consider the case where we have two sibling nodes, with approximate cardinality, 5 and 100. The parent node cardinality should be near or exactly 105, otherwise we can suspect a collision error in the larger node. Using the triangle inequality, we can confirm or refute the approximate counts within this hierarchy of nodes. Furthermore, if the parent and sibling counts do not fully satisfy the triangle inequality requirement, we can query the counts of their parent and child nodes to find additional support, and correct approximate count errors.

For count min-sketch alone, we choose the number of sketches to be $ln(\frac{n}{\delta})$ [51]. However using tree count adjacency we can reduce the count error to $\sqrt{\delta}$ and set the number of sketches $\Theta(\frac{e}{\varepsilon})$. Resulting in a total storage requirement of $\Theta(m\frac{e}{\varepsilon}ln(\frac{nlog(d)}{(\sqrt{\delta})}))$.

8.4 DB Friendly Random Projection Speed Up

From [2] we know that a set of vectors can be projected into a lower dimensional embedding $\Re^d$ where $d \propto \Omega(\frac{\log(n)}{\epsilon^2 \log 1/\epsilon})$ with low ϵ-distortion if projected using a matrix formed as:

$$r_{ij} = \sqrt{3}\begin{cases} +1, \text{with probability } \frac{1}{6} \\ 0, \text{with probability } \frac{2}{3} \\ -1, \text{with probability } \frac{1}{6} \end{cases}$$

where:

n - number of vectors

m - vector dimensionality

k - projected space dimensionality

The above result allows us to avoid the computationally expensive task of matrix orthogonalization. This point is due to the high probability of randomly generated vectors being nearly orthogonal as n increases. Furthermore, the above result suggests we can avoid the costly generation of the normal variate, as the above suffices to give a nearly normal distribution about the point in the projected subspace. Computing and multiplying still naively requires nmk operations. Below we will show how to get this down to $\log(\frac{3}{2})nmk$ operations. A trick, since multiplying by 0 is has no effect on the row vector sums, is to select $\frac{n}{3}$ of the indices from a given vector, and multiplies them by $+1/-1$ evenly (thus our $\frac{1}{6}$ probability). Random selection however requires us to either store collisions and re-select indices until they do not collide. The odds of a perfect non-colliding intersection are tremendously low due to the birthday problem (yields $\approx \frac{2}{3n}$). So we need to account for intersections. The amount of intersection however is bounded. To assess this bound we use Riemann Sums and Union Bound Estimate simultaneously to obtain the integral:

$$\int_0^{\frac{n}{3}} \frac{1}{n-x}dx = ln(\frac{3}{2}),$$

For the selections

$$\lim_{x\to\infty}(\frac{1}{n} + \frac{1}{n-2}... + \frac{1}{n-n/3})$$

By changing our range from 0 to $\frac{1}{3}$, to $ln(\frac{3}{2})$, we now generate our selections and compensate for the intersections. The evenness of $+1/-1$ assures that even though we are generating more indices, the generated distribution will still be symmetric about 0. Although we can similarly calculate the probability of the additional selections colliding with the selected $\frac{2}{3} - ln(\frac{3}{2})$, the recursion converges quickly and the effects are negligible. An experiment was performed for a set of uniform random vectors $x \in X$ where $d = 5000$. In the experiment we compared shuffling a full list and selecting the first $n/3$ indices, to our proposed selection of $n\ln(\frac{3}{2})$ from the full range of indices one after another. In this experiment we get the following deviation between p_1- db-friendly projection and p_2 - sequential random selection projection.

$$\|p_1^\intercal X - p_2^\intercal X\| = .00001$$

.

Chapter 9

Conclusions

All variants of RPHash share a number of similar performance advantages over other traditional methods while also suffering similar drawbacks. Subsequent attempts to mitigate these drawbacks resulted in the various version of RPHash and configurations. In this section we discuss those common attributes as well as a breakdown of the variant specific attributes and how they compare to other clustering algorithms.

9.1 Clustering Performance

In this section we discuss how RPHash clustering performance compares to classic and state-of-the art clustering algorithms. While the primary goal of clustering is ill-posed, performing well on an ensemble of clustering metrics is generally regarded as an overall useful measure of clustering performance.

9.1.1 RPHash

Table 7.4 gives an exhaustive comparison of the standard RPHash algorithm to various other common clustering methods. In this section we focus only on the listed clustering performance. Regarding clustering performance, we report measures for *adjusted rand index* and *cluster purity*. RPHash's performance is highly dependent upon the dataset being clustered. In general it performs better than most of the agglomerative methods for all datasets. The exception for agglomerative methods is the Ward's Linkage, which performs better in some of the chosen datasets. This performance often echoes that of k-means performance, on the same datasets. While k-means tends to outperform RPHash in regard to overall performance, none of the performances are better by a significantly large margin. SOTA seems to perform most similarly to RPHash, with similar ARI and Purities for all datasets. The overall goal however is not that RPHash outperforms other metrics in terms of clustering performance, rather that it is comparable, with the an advantage in processing speed. We investigate this further in Section 9.3.1.

Table 7.3 gives more detailed performance for an optimized configuration of RPHash against k-Means. As before, RPHash and k-Means alternate between best performance, with only minor deviations between the two. The reconfigurability of RPHash is important in this case because it can be optimized for the particular dataset, while k-means is less flexible in configuration.

9.1.2 Streaming RPHash

The performance of streamingRPHash is shown in Figure 7.4.2 and Figure 7.4.1 as a function of streamed batches of data. On both datasets streamingRPHash tends to find a good local minima for the Within-Cluster Sum of Squared Error metrics. However, streamingRPHash tends to suffer on some external metrics such as the adjusted rand index, and internal metrics that tend to favor cluster separability, such as Dunn Index and Silhouette Index. Further analysis of these results show a bias in streamingRPHash that suggests it favors solutions to the facility location problem over the more general clustering problem. This may be exacerbated by the fact that streamingRPHash does not specifically label noise or outlier vectors, which can result in poorer values pertaining to Silhouette Index and Dunn Index. This is mainly due to noise clusters skewing the inter/intra -cluster distance ratio.

The performance of streamingRPHash on synthetic datasets with varying dimensions is shown in Figure 7.4.3. For synthetic datasets streamingRPHash does quite well, matching the optimal performance of the best stream clustering algorithms. In all cases it outperforms Streaming k-Means and Damped Sliding Window.

The scalability of streamingRPHash for Runtime and memory is shown in Figure 7.4.4. As expected the resource requirements of streamingRPHash and streaming k-means are substantially lower than those of other tested algorithms. This advantage becomes increasingly significant as the dimensionality of the input vectors grows. Although streaming k-means maintains a consistently better bound on both memory and processing time, the overall growth complexity is similar, which suggests optimizations to streamingRPHash could put it in reach or streaming k-means' performance.

The noise study demonstrates an addition benefit of streamingRPHash. As expected, a gradual decrease in clustering accuracy of all algorithms is observed as the amount of noise increases in the data stream. However, there are some interesting patterns in clustering results produced by streamingRPHash. Its clustering accuracy clearly exceeds that of other algorithms in the presence of 2% noise. As the amount of noise begins to increase in the data stream, Damped Sliding Window and Biased Reservoir Sampling seem to produce better clustering in terms of external measures. However, even with the increasing noise, streamingRPHash continues to give the best clustering results in terms of internal measures. In particular, its WCSSE values remain the lowest, and sometimes lower than baseline values, even in the presence of noise as high as 60% and 75%. This can be explained if we look into how labels are assigned to random noise points at the time of data stream generation. As mentioned before, each noise point is assigned to a cluster whose centroid is the closest to that noise point. Thus, a noise point, which is assigned to a cluster i, may actually be closer to the boundary of another cluster j. Therefore, it is possible that a clustering algorithm places a noise point in a *more appropriate* cluster than the data stream generator does. In other words, the clustering algorithm may be able to find better centroids to incorporate noise points, thus producing more compact clusters than ground-truth partitions. In this case, the clustering algorithm will produce better values of internal measures than the baseline values. On the other hand, since external measures try to capture the agreement between ground-truth partitions and clusters detected by an algorithm, the values of external measures will clearly remain low. This is why streamingRPHash, when applied on noisy data streams, performs much better in terms of internal measures. The apparent decrease in values of external measures is merely due to the disagreement with ground-truth labels of noise points.

To summarize our findings from the scalability study: we discover a major strength of streamingRPHash and conclude that the algorithm is capable of producing perfect clustering when the data stream consists of a mixture of Gaussian distributions without any noise. For real world, and data containing a significant amount of noise, it may be useful to augment streamingRPHash with the ability to label observations as noise.

9.1.3 Tree Walk RPHash

On real world and synthetic datasets, the Tree-walk RPHash algorithm tends to perform better than the standard RPHash and streamingRPHash algorithms. In contrast the Adaptive LSH algorithm in both the Streaming and Standard RPHash algorithm has a slightly mixed result.

In Figure 7.3.1 we see that standard RPHash and the Leech decoder outperforms the adaptive LSH algorithm and the TWRP algorithm. While this result is in opposition of our claim that the TWRP is best among RPHash variants, it is important to note that the experiment was performed on a highly synthetic dataset with no noise, and tightly clusterable data. The test in fact is very similar to the results in Figure 4.3.1 where the Leech Lattice decoder was well suited to finding tight, uniform clusters scaled between -1 and 1. It is not until we look at more realistically generated datasets and robust clustering metrics that we see the advantage of TWRP. Figures 7.5.1, 7.5.2, and 7.5.3 show TWRP performing optimally for all datasets at all dimensions, while the standard RPHash algorithm and k-Means show considerable variance in Purity, ARI, and WCSSE. In Figure 7.5.9 and Figure 7.5.8 we look at the raw WCSSE scores, and the ratio of WCSSE for TWRP and the k-Means++ algorithm on 1000 dimensional, real world, Word2Vec data. Overall, TWRP performs within 5.1% of k-Means++. This is significant if we consider the scalability difference between TWRP and k-Means++. Figure 7.5.10 shows these differences, and while k-Means++ seems to grow quadratically with the number of clusters, TWRP remains constant. This is significant, because TWRP could potentially output any number of micro-clusters to some, more robust machine learning algorithm, with no change in processing time requirement.

9.2 Noise Resilience

All variants of the RPHash algorithms are noise robust. This is primarily due to the density mode search common in all algorithms, ignoring data that resides outside the partition region. In Figure 7.4.5 we show the performance of streamingRPHash in regard to injected noise. In regard to WCSSE, streamingRPHashs outperforms all other clustering algorithms on noise resilience. While among other performance metrics, it tends to be in the top three. Standard RPHash and TWRP are compared in Figures 7.5.5, 7.5.5 and 7.5.5 on ARI, Purity and WCSSE as a function of injected noise. For ARI and Purity, TWRP outperforms standard

RPHash and the other comparison algorithms up until the highest levels of injected noise $> 35\%$. Standard RPHash outperforms TWRP on WCSSE metrics, but not by a very large margin.

9.3 Timing Results

A principal focus and motivation for this work, is to reduce the time and overall complexity of current clustering methods. As expected, RPHash and its variants outperform all other tested clustering methods on processing speed and asymptotic complexity. Time results at linear with smaller constant factors than many other clustering methods.

9.3.1 RPHash

In Table 7.4 we see that the standard RPHash algorithm often bests comparable clustering algorithms by at times orders of magnitude on larger datasets. Furthermore the growing deviation between processing times in regard to dataset sizes suggests and overall asymptotic complexity difference over other tested algorithms, which is verified in the previous sections runtime analysis. While agglomerative methods and SOTA tend to require considerably more time to process than RPHash, k-Means is sometimes regarded as a near linear runtime algorithm ($\theta(loglog(n) * n)$). However in a direct comparison shown in Table 7.3 we can see that RPHash outperforms k-Means considerably on runtime. This suggests that while k-Means may have near linear asymptotic complexity, it exhibits considerable constant factor offsets.

9.3.2 Streaming RPHash

Similar to standard RPHash, streamingRPHash performs well on runtime. While standard RPHash was not compared against streaming k-Means, streamingRPHash was. Streaming k-Means in Figure 7.4.2 and Figure 7.4.1 outperforms streamingRPHash on runtime. However both algorithms exhibit similar scalability, as suggest by Figure 7.4.4. It is likely that these performance differences could be overcome with minor implementation optimization.

9.3.3 TWRP

The scalability with respect to dimension of TWRP and standard RPHash is shown in Figure 7.5.7. In the plot we see that there is no contest between the RPHash algorithms and the comparison algorithms SOTA and k-Means. While the Table 7.6 shows an even more dramatic difference between both versions of RPHash and the agglomerative algorithms. TWRP is almost 688 and 4795 times faster than k-means and agglomerative hierarchical respectively for the noise datasets having dimensions 10000x6500. We can see the growth of time as the dimension increases in Figure 7.5.7 . RPhash and TWRP has almost no growth when compared to the other algorithms.

Figure 7.5.10, Table 7.6 and Table 7.5 provide similar evidence regarding the superiority or RPHash algorithms in general With the complexity growth of RPHash being nearly linear, with small constant factor offsets. Therefore we can see that one of the biggest advantages of RPHash is that it achieves such comparable clustering accuracy with minimal costs of runtime . This makes RPHash suitable for applications where the dimension or volume of data is very high.

9.4 Parallelism

A primary focus of the RPHash algorithms is to enable naive parallelism through the coordination of generative mathematical functions and approximate data sketches, instead of costly verbose interprocess communication. The results in Figure 7.6.1 show that our assumptions about RPHash parallelism, are to some degree, correct. For data centric algorithms like RPHash, the ratio of parallel to sequential code, per Amdahl's law, is strongly dictated by the input data size. However to not bias our parallelism with the input IO of our test machine's memory subsystem, all parallel experiments were done on in memory datasets, resulting in a limit to the overall amount of achievable parallelism. It is our belief that larger datasets may show further advantages for RPHash in regard to parallel speedup.

Standard RPHash appears to enjoy the greatest advantage in regard to parallel speedup. The results for RPHash were edited to remove, outlier data points in which memory contention for many nodes, caused errors as parallelism grew. This is more an artifact of our chosen method of parallelism, than that of an intrinsic algorithm design. Namely, standard RPHash tends to use significantly more memory per core than our sketch based algorithms. This Memory requirement presented itself when scaling to more processing nodes, and caused a memory bottleneck around 15 nodes. Fifteen nodes however is a reasonable baseline to establish the predictable parallelism. For the intended distributed implementation of RPHash in MapReduce or Spark, it is unlikely these physical memory restrictions would exist.

Interestingly, TWRP which would presumable be sequentially bound by the off-line Tree walk procedure and per vector, shared sketch update step, actually parallelizes quite well. The likely reason for this is that the off-line step does not contribute significantly to the overall number of computations required for both on-line and off-line processing steps. Furthermore, the shared memory environment of our experimental setup, allows for fast access to the Count-Min Cut Tree structure. In the distributed setting, this would not be the case. However, due to the additive nature of the Count-Min Sketch, completely independent update of the Count-Min sketch would be trivial and would only require a $\theta(log(N))$ number of steps to reduce the N sketches.

streamingRPHash suffered a poorer than expected speedup performance. streamingRPHash utilized vector parallelism, in which processing nodes were scheduled by a master node, to the first available thread. This naturally results in a sequential bottleneck at the master node. However, it additionally stresses cache prediction, in real world CPUs, due to our greedy vector-to-CPU assignment. The greedy assignments are strongly influenced by the chosen LSH algorithm, which in this experiment was the Leech Decoder, which is determined by the randomly generated data vectors. This cache pressure along with the sequential bottleneck, make streamingRPHash a poor candidate for shared memory system parallelization. However, for distributed systems in which vectors are fetched from a remote source, the parallelism should only be bound by the intermittent off-line steps. In contrast, both TWRP and standard RPHash were able to partition the problem equally among processing nodes and then work on the sets of vectors independently.

An alternate hypothesis however, is that our results may be somewhat under represented due to our test system only having 16 physical cores and 32 threads due to Hyper-Threading, instead of 32 truly independent processing nodes. Hyper-Threading is a technology that effectively shares the resources of a single processing pipeline, between two processing workloads, with the assumption that underutilized processing sub-units will achieve better overall utilization. This however is not completely equivalent to two independent processing units. The saturation of one unit, such as the Floating Point sub processing unit, could cause a bottleneck.

Furthermore, hyper-threading is also bound by there only being one physical, memory management unit per core, which could be a source of further IO saturation. Another likely candidate for our sub-linear speedup performance could be our test system's Quick Path Memory controller, resulting in non-uniform memory data access patterns. Attempts were made to optimize this access pattern through both the Java NUMA flag, and the operating system 'numactl' meta-processing environment. Both alterations failed to achieve significantly different results (timing was worse on all tests). These possibilities further suggest that a completely independent distributed processing environment may achieve better overall parallelism.

9.5 Overall Conclusion

The RPHash algorithms combines approximate and randomized methods in a new way to solve issues of scalability and data security for cluster analysis on distributed data. The runtime and clustering performance of our RPHash algorithms are similar to that of the standard k-means clustering algorithm, with the added benefit of being scalable to very large datasets. This randomized, clustering algorithm is well suited for ill-posed, combinatorially restrictive problems such as clustering and partitioning. This assertion is complemented by a similar inversion regarding clustering and computing, in which real world problems tend to converge much faster than adversarially crafted worse-case problems.

The principle assumption in this work is that approximate and exact clustering, are qualitatively similar due to noise, redundancy, data loss, and the curse of dimensionality. Furthermore, the addition of random noise to the clustering dataset resulting from the random subspace projection requirement, provides some degree of non-deterministic process, so subsequent iterations of the algorithm could potentially find better results. Making the process of finding better clustering results, a matter of available processing time and resources. Furthermore, the destructive projection process affords us a certain degree of data privacy while requiring no additional processing.

9.5.1 RPHash Conclusions

The RPHash algorithm offers a fast highly scalable algorithm for data clustering that shows potential for distributed computing that can benefit well from addition processing resources, and is secure. RPHash's simple straightforward structure results in a predictable time requirement. The processing time predictability is beneficial for time dependent cases where completion time is more important than minor errors in clustering accuracy. Overall, RPHash is much faster than other clustering algorithms. Some drawbacks of the standard RPHash algorithm, is that the results can be order dependent, and sometimes unstable. Multiple runs can fix some of these issues, but the overall stability may be an outstanding problem, that we suggest TWRP remedies.

9.5.2 Streaming RPHash Conclusion

streamingRPHash Is a fast accurate single pass clustering method for high dimensional data streams. The growing number of massive, high dimensional streaming data sources can take advantage of the benefits of quick, accurate, scalable and distributed processing of streamingRPHash. We show that our method is faster than most of our comparative streaming clustering algorithms. while performance is underwhelming on the HAR dataset, streamingRPHash does quite well on the UJII dataset. The deviation in performance between datasets suggests that streamingRPHash may not be well suited for all datasets. However, when data is well behaved, containing predominantly spherical clusters in high dimensional space, streamingRPHash outperforms all other streaming clustering methods. This may be a result of the sub-space projection, being well equipped to capture the overall data variance for these types of datasets. Another feature of streamingRPHash is that it is robust to noise, outperforming all other clustering algorithms on clustering metrics. These benefits however are overshadowed by streamingRPHash processing efficiency, scalability and memory usage. streamingRPHash compares well to Streaming k-means in timing and memory usage. While shared memory parallelism is below our expectations, the potential for distributed processing, is positive, and would likely exceed the scalability of streaming k-means which incurs a sequential bottleneck when checking and updating a shared list of nearest cluster centroids.

9.5.3 TWRP Conclusions

In this work we introduced the Tree-Walk Random Projection (TWRP) clustering technique for clustering large datasets with log-linear processing complexity. We applied our clustering technique on real and synthetic data of varying noise levels. Our method performed comparable to various other common clustering methods, while improving significantly upon the overall required processing time. TWRP improves upon standard

RPHash clustering performance and in many cases performs better than k-means and similarly to Ward's Linkage hierarchical agglomerative clustering. Importantly, it does so, while achieving a similar processing time requirement and scalability as standard RPHash. Furthermore the TWRP algorithm improves upon RPHash's already relatively good noise suppression performance. TWRP also improves upon RPHash by offering more stable results between runs, that border on deterministic behavior. Another advantage of TWRP is that is shows strong potential for shared memory and distributed parallelism. In addition, TWRP is also amenable to streaming environments, due to its base structure being the Count-Min Cut-Tree. The Count-Min Cut-Tree is an error bounded constant memory sketch, that succinctly encodes the required data structure of the off-line clustering process.

9.5.4 Code Repository

The RPHash Java source code is released under an open source license and is freely available at `github.com/wilseypa/rphash-java`. Python versions of TWRP are available on-line at `https://github.com/leecarraher/PyRecRPHash`. Plots and figures are available by request `mailto://leecarraher@gmail.com`

9.6 Future Research

In this section we give suggestions for potential future research following the RPHash clustering framework. We discuss a further distributed framework for streamingRPHash, and RPHash as a pre-clustering technique for more robust clustering methods like topological data analysis(TDA). We also suggest more practical implementation improvements like GPU processing. Lastly we discuss the potential of the general Count-Min Cut Tree as a general space partitioning sketch.

9.7 Spark Implementation

Spark is a distributed computing framework loosely based on functional parallelism that runs on Hadoop and other similar frameworks. Spark is similar in design to MapReduce functional dissection for parallel algorithms, however, unlike the standard Hadoop MapReduce algorithm, it is geared toward multi-pass processing common in machine learning algorithms, by leveraging RAM caching of persistent states between Map and Reduce phases. Furthermore, Spark allows for streaming interfaces for parallel processing, capable of extending the streamingRPHash algorithm. A future direction for streamingRPHash would be to utilize the input stream methods in spark, to implement streamingRPHash as a fully distributed stream based clustering method. The implementation could read streams independently from disconnected data streams, building independent data structures, and merging the results when requested. Similar to the streamingRPHash diagram given in Figure 5.1.3, we give a distributed extension in Figure 9.7.1.

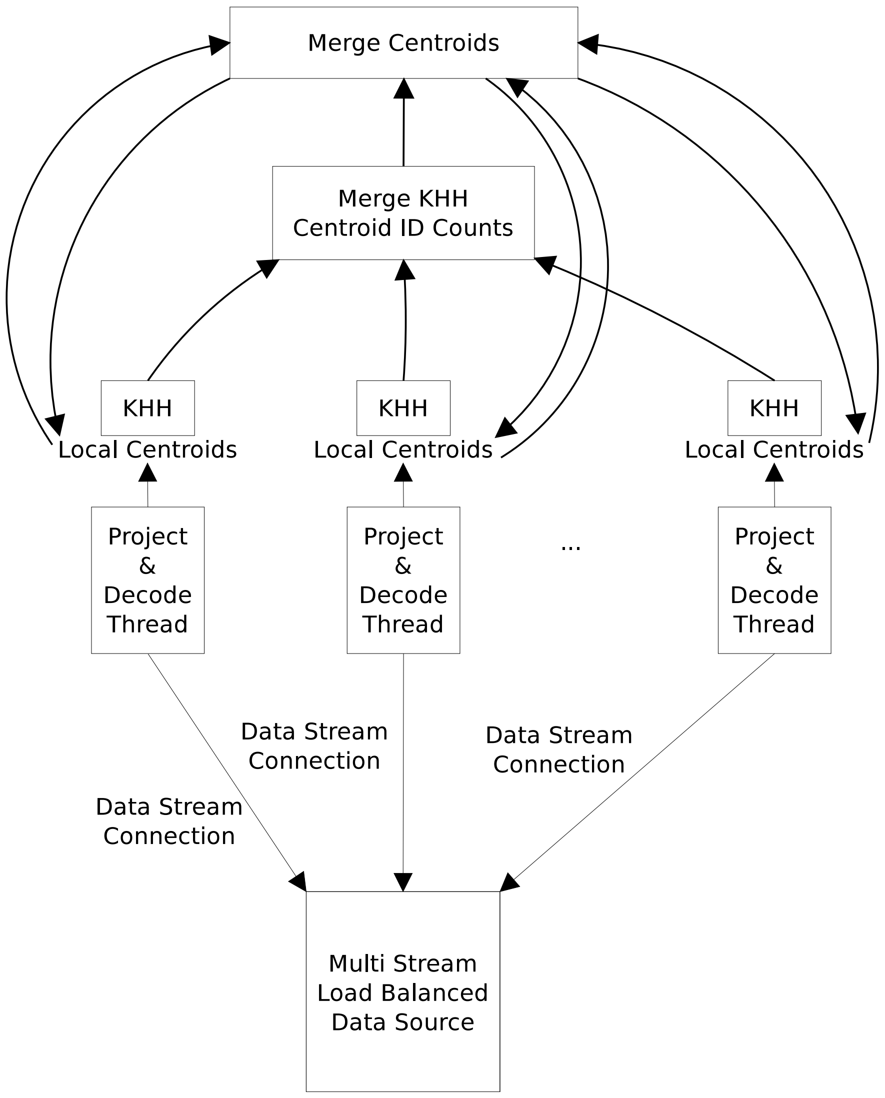


Figure 9.7.1: Spark Streaming RPHash Diagram

9.7.1 Topological Data Analysis

An interesting emerging field in machine learning is topological data analysis(TDA). TDA attempts to overcome some of the issues inherent with orthogonal metric data embeddings by regarding not just the data, but the connectivity between the data. Somewhat similar to manifold learning methods, in which data is regarded as being embedded in some complex manifold, TDA goes further, by removing the restriction that a surface be locally differentiable. In this way, surfaces can have holes, and true discontinuities.

While TDA work is still emerging, there is a significant gap between theoretical understanding and practical application. In particular, many TDA algorithms are graph based and therefore, combinatorially bound. However, by using the concept of micro-clusters, arbitrarily available through RPHash clustering, with no computation penalties, one could potentially reduce the complexity of TDA algorithms, to these representative micro-clusters, followed by subsequent refinements.

9.8 GPU Leech

Previous work [38] on a graphics processing unit (GPU) implementation of the Leech Decoder for LSH could readily be used to accelerate the RPHash algorithm. While performance monitoring showed the principal occupation of RPHash was in the projection step, accelerating just the LSH may only result in a small speedup. However, if we were to combine both the projection and a GPU based implementation of LSH, we may be able to take advantage of GPU memory locality, and achieve considerably high GPU core occupancy (a principle measure for GPU parallelism).

9.8.1 Bounded Error Compressed Cut Tree

While the results from Tree Walk Random Projection Hash are promising, a more interesting data structure is defined that may have application beyond cluster analysis. In particular, this structure, which will be referred to as a *Bounded Error Compressed Cut Tree* has the potential of accelerating and making feasible a variety of interesting learning and data space exploration problems.

We briefly review its structure here. In TWRP we partitioned a data set by a sequence of orthogonal vectors corresponding to the origin of each subsequent dimension. This partitioning results in a d length

LSH encoding for every vector. The hashes are then stored as a compressed representation in the Count-Min Sketch. As an aside, while we make the restriction that all hyperplanes intersect the origin, BECC-Trees could also be composed of any set of hyperplanes rotated any number of times. This structure which we treat as a vector density oracle for TWRP clustering could be used similarly, for a variety of other purposed. The basic setup would involve changing either the tree traversals method, or by storing some alternative metric in the nodes. The tree, again never being fully realized, uses the Count-Min Sketch as an approximate partitioning oracle. Therefore any tree based solution to a graph problem in which a cut oracle would be helpful, could be solved in $\frac{1}{\epsilon}$ space.

The true power of this structure comes of course comes from the parameterization of the Count-Min Sketch, which allows one to set the level of acceptable error, for tree based oracle problem. The Count-Min Sketch constrained to elements in a tree can compress any exponentially large space partitioning problem, and is certainly worth further investigation. Our Bounded error compressed cut trees certainly demonstrate this property for clustering. In addition, the ability of BECC Trees to consume a potentially unbounded data stream, and store only a sketch of its current structure to later be scrutinized and reconstruction is potentially applicable to off-line data stream inquiry.

Acknowledgments

Support for this work was provided in part by the National Science Foundation under grant ACI–1440420.

Chapter 10

Bibliography

[1] Privacy and progress in whole genome sequencing. Technical report, Washington, October 2012.

[2] D. Achlioptas. Database-friendly random projections. In *Proc of the twentieth ACM SIGMOD-SIGACT-SIGART symp on principles of database systems*, pages 274–281, 2001.

[3] C. C. Aggarwal. On biased reservoir sampling in the presence of stream evolution. In *Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB '06*, pages 607–618, Seoul, Korea, 2006.

[4] C. C. Aggarwal. *Data streams: models and algorithms*, volume 31. Springer Science & Business Media, 2007.

[5] C. C. Aggarwal. A framework for clustering massive-domain data streams. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 102–113, 2009.

[6] C. C. Aggarwal, A. Hinneburg, and D. A. Keim. On the surprising behavior of distance metrics in high dimensional spaces. In *Proceedings of the 8th International Conference on Database Theory*, ICDT '01, pages 420–434, London, UK, UK, 2001. Springer-Verlag.

[7] C. C. Aggarwal, J. L. Wolf, P. S. Yu, C. Procopiuc, and J. S. Park. Fast algorithms for projected clustering. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 61–72, New York, NY, USA, 1999. ACM.

[8] C. C. Aggarwal and P. S. Yu. *A General Survey of Privacy-Preserving Data Mining Models and Algorithms*, pages 11–52. Springer US, Boston, MA, 2008.

[9] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 94–105, New York, NY, USA, 1998. ACM.

[10] E. Agrell, T. Eriksson, A. Vardy, and K. Zeger. Closest point search in lattices. *Information Theory, IEEE Transactions on*, 48(8):2201–2214, Aug 2002.

[11] N. Ailon and B. Chazelle. Approximate nearest neighbors and the fast johnson-lindenstrauss transform. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 557–563, 2006.

[12] N. Ailon, R. Jaiswal, and C. Monteleoni. Streaming k-means approximation. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 10–18. Curran Associates, Inc., 2009.

[13] N. Ailon and E. Liberty. An almost optimal unrestricted fast johnson-lindenstrauss transform. *ACM Trans. Algorithms*, 9(3):21–1, Jun 2013.

[14] M. Al-Razgan and C. Domeniconi. Weighted clustering ensembles. In *Proceedings of the SIAM international conference on data mining*, pages 258–269, 2006.

[15] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29, 1996.

[16] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[17] O. Amrani and Y. Beery. Efficient bounded-distance decoding of the hexacode and associated decoders for the leech lattice and the golay code. *Communications, IEEE Transactions on*, 44(5):534–537, May 1996.

[18] O. Amrani, Y. Be'ery, A. Vardy, F.-W. Sun, and H. C. A. van Tilborg. The leech lattice and the golay code: bounded-distance decoding and multilevel constructions. *Information Theory, IEEE Transactions on*, 40(4):1030–1043, Jul 1994.

[19] M. R. Anderberg. *Cluster analysis for applications*. Academic Press, 1973.

[20] A. Andoni. *Nearest Neighbor Search: the Old, the New, and the Impossible*. PhD thesis, Massachusetts Institute of Technology, September 2009.

[21] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS '06. 47th Annual IEEE Symposium on*, pages 459–468, Oct. 2006.

[22] A. Andoni, P. Indyk, H. L. Nguyen, and I. Razenshteyn. Beyond locality-sensitive hashing. In *Proc of the Twenty-Fifth Annual ACM-SIAM Symp on Discrete Algorithms (SODA '14)*, pages 1018–1028, 2014.

[23] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. A public domain dataset for human activity recognition using smartphones, Apr. 2013.

[24] D. Arthur and S. Vassilvitskii. K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[25] R. Avogadri and G. Valentini. Fuzzy ensemble clustering based on random projections for dna microarray data analysis. *Artificial Intelligence in Medicine*, 45(2):173–183, 2009.

[26] Y. Bartal, B. Recht, and L. J. Schulman. Dimensionality reduction: beyond the johnson-lindenstrauss bound. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 868–887, 2011.

[27] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[28] R. Berinde, P. Indyk, G. Cormode, and M. J. Strauss. Space-optimal heavy hitters with strong error bounds. *ACM Transactions on Database Systems*, 35(4):26, 2010.

[29] R. Berkvens, M. Weyn, and H. Peremans. Position error and entropy of probabilistic wi-fi fingerprinting in the ujiindoorloc dataset. In *International Conference on Indoor Positioning and Indoor Navigation, IPIN 2016, Alcala de Henares, Spain, October 4-7, 2016*, pages 1–6, 2016.

[30] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. *When Is "Nearest Neighbor" Meaningful?*, pages 217–235. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[31] E. Bingham and H. Mannila. Random projection in dimensionality reduction: Applications to image and text data. In *in Knowledge Discovery and Data Mining*, pages 245–250. ACM Press, 2001.

[32] J. Bourgain. On lipschitz embedding of finite metric spaces in hilbert space. *Israel Journal of Mathematics*, 52(1-2):46–52, 1985.

[33] C. Boutsidis, A. Zouzias, and P. Drineas. Random projections for k-means clustering. *CoRR*, abs/1011.4632, 2010.

[34] P. S. Bradley, O. L. Mangasarian, and W. N. Street. Clustering via concave minimization. *Advances in neural information processing systems*, pages 368–374, 1997.

[35] V. Braverman, A. Meyerson, R. Ostrovsky, A. Roytman, M. Shindler, and B. Tagiku. Streaming k-means on well-clusterable data. In *Proceedings of the Twenty-second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '11, pages 26–40. SIAM, 2011.

[36] A. R. Calderbank. The art of signaling: fifty years of coding theory. *Information Theory, IEEE Transactions on*, 44(6):2561–2595, Oct 1998.

[37] F. Cao, M. Ester, W. Qian, and A. Zhou. Density-based clustering over an evolving data stream with noise. In *SIAM Conference on Data Mining*, pages 328–339, 2006.

[38] L. Carraher and F. Annexstein. A parallel algorithm for query adaptive, locality sensitive hash search. 2012.

[39] L. Carraher and P. A. Wilsey. A gpgpu algorithm for c-approximate r-nearest neighbor search in high dimensions. 2012.

[40] L. A. Carraher, P. A. Wilsey, A. Moitra, , and S. Dey. Multi-probe random projection clustering to secure very large distributed datasets. In *2nd International Workshop on Privacy and Security of Big Data*, Oct. 2015.

[41] M. A. Carreira-Perpinan. Gaussian mean-shift is an em algorithm. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 29(5):767–776, May 2007.

[42] Y. Chen and L. Tu. Density-based clustering for real-time stream data. In *Proc of the 13th ACM SIGKDD Int Conf on Knowledge Discovery and Data Mining (KDD '07)*, pages 133–142, 2007.

[43] M. Chowdhury, M. Zaharia, and I. Stoica. Performance and scalability of broadcast in spark, 2010.

[44] P. M. Ciarelli and E. Oliveira. UCI machine learning repository.

[45] P. M. Ciarelli and E. Oliveira. Agglomeration and elimination of terms for dimensionality reduction. In *Intelligent Systems Design and Applications, 2009. ISDA '09. Ninth International Conference on*, pages 547–552, Nov 2009.

[46] H. Cohn and A. Kumar. Optimality and uniqueness of the leech lattice among lattices. *Annals of Mathematics*, 170(3):1003–1050, 2009.

[47] H. Cohn, A. Kumar, S. D. Miller, D. Radchenko, and M. Viazovska. The sphere packing problem in dimension 24. *arXiv preprint arXiv:1603.06518*, 2016.

[48] D. Comaniciu and P. Meer. Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5):603–619, May 2002.

[49] J. Conway and N. Sloane. *Sphere Packings Lattices and Groups Third Edition*. Springer, New York, 1998.

[50] G. Cormode and M. Hadjieleftheriou. Methods for finding frequent items in data streams. *The VLDB Journal*, 19(1):3–20, 2010.

[51] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[52] G. Cormode and S. Muthukrishnan. Approximating data with the count-min data structure. *IEEE Software*, 2012.

[53] R. T. Curtis. A new combinatorial approach to m24. *Mathematical Proceedings of the Cambridge Philosophical Society*, 79(01):25–42, 1976.

[54] A. Dasgupta, R. Kumar, and T. Sarlos. A sparse johnson: Lindenstrauss transform. In *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, STOC '10, pages 341–350, New York, NY, USA, 2010. ACM.

[55] S. Dasgupta. Experiments with random projection. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, UAI'00, pages 143–151, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[56] S. Dasgupta. *The hardness of k-means clustering*. Department of Computer Science and Engineering, University of California, San Diego, 2008.

[57] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG 04, pages 253–262, New York, NY, USA, 2004. ACM.

[58] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications ofthe ACM*, 51(1):107–113, Jan. 2008.

[59] I. Dhillon and D. Modha. Lecture notes in computer science. In M. Zaki and C.-T. Ho, editors, *Large-Scale Parallel Data Mining*, volume 1759, chapter A Data-Clustering Algorithm on Distributed Memory Multiprocessors, pages 245–260. Springer Berlin Heidelberg, 2000.

[60] X. Ding, L. Zhang, Z. Wan, and M. Gu. A brief survey on de-anonymization attacks in online social networks. In *Computational Aspects of Social Networks (CASoN), 2010 International Conference on*, pages 611–615, 2010.

[61] C. Dwork. Differential privacy. In *Proceedings of the 33rd International Conference on Automata, Languages and Programming - Volume Part II*, ICALP'06, pages 1–12, Berlin, Heidelberg, 2006. Springer-Verlag.

[62] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, volume 96, pages 226–231, 1996.

[63] M. Fashing and C. Tomasi. Mean shift is a bound optimization. *Pattern Analysis and Machine Intelligence, IEEE Trans. on*, 27(3):471–474, March 2005.

[64] X. Z. Fern and C. E. Brodley. Random projection for high dimensional data clustering: A cluster ensemble approach. In *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003)*, pages 186–193, aug 2003.

[65] P. Flajolet, ric Fusy, O. Gandouet, and et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *IN AOFA 07: PROCEEDINGS OF THE 2007 INTERNATIONAL CONFERENCE ON ANALYSIS OF ALGORITHMS*, 2007.

[66] I. Florescu, A. Molyboha, and A. Myasnikov. Scaling and convergence of projection sampling. Technical report, 2009.

[67] S. Fortune. A sweepline algorithm for voronoi diagrams. In *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, pages 313–322, New York, NY, USA, 1986. ACM.

[68] T. M. Forum. MPI: A message passing interface, 1993.

[69] P. Frankl and H. Maehara. The johnson-lindenstrauss lemma and the sphericity of some graphs. *J. Comb. Theory Ser. A*, 44(3):355–362, jun 1987.

[70] J. Freeman. Parallel algorithms for depth-first search. Technical Report MS-CIS-91-71, October 1991.

[71] M. L. Gavrilova. Lecture notes in computer science. In V. Kumar, M. Gavrilova, C. Tan, and P. L'Ecuyer, editors, *Computational Science and Its Applications –- ICCSA 2003*, volume 2669, chapter An Explicit Solution for Computing the Euclidean d-dimensional Voronoi Diagram of Spheres in a Floating-Point Arithmetic, pages 827–835. Springer Berlin Heidelberg, 2003.

[72] M. Ghashami, E. Liberty, J. M. Phillips, and D. P. Woodruff. Frequent directions: Simple and deterministic matrix sketching. *SIAM Journal on Computing*, 45(5):1762–1792, 2016.

[73] H. A. Guvenir. UCI machine learning repository.

[74] I. Guyon, S. R. Gunn, A. Ben-Hur, and G. Dror. Result analysis of the NIPS 2003 feature selection challenge, 2004.

[75] M. Gymrek, A. L. McGuire, D. G. D, E. Halperin, and Y. Erlich. Identifying personal genomes by surname inference. *Science*, (339):321–324, 2013.

[76] S. Har-Peled. A replacement for voronoi diagrams of near linear size. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pages 94–103, Oct. 2001.

[77] J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. *JSTOR: Applied Statistics*, 28(1):100–108, 1979.

[78] R. J. Hathaway and J. C. Bezdek. Fuzzy c-means clustering of incomplete data. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 31(5):735–744, 2001.

[79] J. Herrero, A. Valencia, and J. Dopazo. A hierarchical unsupervised growing neural network for clustering gene expression patterns, 2001.

[80] T. Hofmann. Unsupervised learning by probabilistic latent semantic analysis. *Mach. Learn.*, 42(1-2):177–196, Jan 2001.

[81] P. V. Huffman W. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 1 edition, 2003.

[82] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[83] A. K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recogn. Lett.*, 31(8):651–666, June 2010.

[84] D. Jurgens, K. Stevens, and M. Dyer. S-space package. `https://github.com/fozziethebeat/S-Space`, 2011–2015.

[85] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions Database Systems*, 28(1):51–55, Mar. 2003.

[86] S. Kaski. Dimensionality reduction by random mapping: Fast similarity computation for clustering, 1998.

[87] R. Klein. Lecture notes in computer science. In H. Noltemeier, editor, *Computational Geometry and its Applications*, volume 333, chapter Abstract voronoi diagrams and their applications, pages 148–157. Springer Berlin Heidelberg, 1988.

[88] R. Kohavi and D. Sommerfield. Feature subset selection using the wrapper method: Overfitting and dynamic search space topology. In *KDD*, pages 192–197, 1995.

[89] A. Kumar, Y. Sabharwal, and S. Sen. A simple linear time (1+/spl epsiv/)-approximation algorithm for k-means clustering in any dimensions. In *Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on*, pages 454–462. IEEE, 2004.

[90] J. Leech. Notes on sphere packings. *Canadian Journal of Mathematics*, 1967.

[91] F. Li and R. Klette. A variant of adaptive mean shift-based clustering. In *Proceedings of the 15th international conference on Advances in neuro-information processing - Volume Part I*, ICONIP'08, pages 1002–1009, Berlin, Heidelberg, 2009. Springer-Verlag.

[92] M. Lichman. UCI machine learning repository, 2013.

[93] Y. Lindell and B. Pinkas. Lecture notes in computer science. In M. Bellare, editor, *Advances in Cryptology –- CRYPTO 2000*, volume 1880, chapter Privacy Preserving Data Mining, pages 36–54. Springer Berlin Heidelberg, 2000.

[94] B. Liu, Y. Xia, and P. S. Yu. Clustering through decision tree construction. In *Proceedings of the Ninth International Conference on Information and Knowledge Management*, CIKM '00, pages 20–29, New York, NY, USA, 2000. ACM.

[95] S. Lloyd. Least squares quantization in pcm. *IEEE Trans. Inf. Theor.*, 28(2):129–137, Sept. 1957.

[96] U. V. Luxburg and S. Ben-david. Towards a statistical theory of clustering. In *In PASCAL workshop on Statistics and Optimization of Clustering*, 2005.

[97] L. v. d. Maaten and G. Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[98] J. Macqueen. Some methods for classification and analysis of multivariate observations. In *5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[99] J. Magidson and J. Vermunt. Latent class models for clustering: A comparison with k-means. *Canadian Journal of Marketing Research*, 20(1):36–43, 2002.

[100] M. Mahajan, P. Nimbhorkar, and K. Varadarajan. Lecture notes in computer science. In S. Das and R. Uehara, editors, *WALCOM: Algorithms and Computation*, volume 5431, chapter The Planar k-Means Problem is NP-Hard, pages 274–285. Springer Berlin Heidelberg, 2009.

[101] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 346–357, 2002.

[102] R. B. Marimont and M. B. Shapiro. Nearest neighbour searches and the curse of dimensionality. *IMA Journal of Applied Mathematics*, 24(1):59–70, 1979.

[103] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the Sixth ACM SIGKDD International Conference*

on Knowledge Discovery and Data Mining, KDD '00, pages 169–178, New York, NY, USA, 2000. ACM.

[104] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[105] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Googlenews-vectors-negative300.bin.gz - efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[106] R. Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

[107] F. Murtagh and P. Legendre. Ward's hierarchical agglomerative clustering method: Which algorithms implement ward's criterion? *J. Classif.*, 31(3):274–295, Oct. 2014.

[108] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 111–125, 2008.

[109] R. T. Ng and J. Han. Clarans: A method for clustering objects for spatial data mining. *Knowledge and Data Engineering, IEEE Transactions on*, 14(5):1003–1016, 2002.

[110] Y. Okadome, K. Urai, Y. Nakamura, T. Yomo, and H. Ishiguro. Adaptive lsh based on the particle swarm method with the attractor selection model for fast approximation of gaussian process regression. *Artificial Life and Robotics*, 19(3):220–226, Nov 2014.

[111] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, SODA '06, pages 1186–1195, New York, NY, USA, 2006. ACM.

[112] B.-H. Park and H. Kargupta. Distributed data mining: Algorithms, systems, and applications. In N. Ye, editor, *Data Mining Handbook*, pages 341–358. 2002.

[113] E. Parzen. On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 33(3):1065–1076, 1962.

[114] M. Saeed, M. Villarroel, A. T. Reisner, G. Clifford, L.-W. Lehman, G. Moody, T. Heldt, T. H. Kyaw, B. Moody, and R. G. Mark. Multiparameter intelligent monitoring in intensive care ii (mimic-ii): A public-access intensive care unit database. *Critical Care Medicine*, 39:952–960, May 2011.

[115] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[116] P. Sen, G. M. Namata, M. Bilgic, L. Getoor, B. Gallagher, and T. Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93–106, 2008.

[117] M. Shindler, A. Wong, and A. W. Meyerson. Fast and accurate k-means for large datasets. In *Advances in neural information processing systems*, pages 2375–2383, 2011.

[118] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[119] J. A. Silva, E. R. Faria, R. C. Barros, E. R. Hruschka, A. C. P. L. F. d. Carvalho, and J. a. Gama. Data stream clustering: A survey. *ACM Comput. Surv.*, 46(1):13:1–13:31, July 2013.

[120] F.-W. Sun and H. C. A. van Tilborg. The leech lattice, the octacode, and decoding algorithms. *Information Theory, IEEE Transactions on*, 41(4):1097–1106, Jul 1995.

[121] T. Tamura, M. Oguchi, and M. Kitsuregawa. Parallel database processing on a 100 node pc cluster: Cases for decision support query processing and data mining. *SC Conference*, 0:49, 1997.

[122] V. Tarokh and I. F. Blake. Trellis complexity versus the coding gain of lattices. i. *Information Theory, IEEE Trans. on*, 42(6):1796–1807, Nov 1996.

[123] K. Terasawa and Y. Tanaka. Spherical lsh for approximate nearest neighbor search on unit hypersphere. In *WADS*, pages 27–38, 2007.

[124] J. Torres-Sospedra, R. Montoliu, A. Martinez-Uso, T. J. Arnau, J. P. Avariento, M. Benedito-Bordonau, and J. Huerta. Ujiindoorloc: A new multi-building and multi-floor database for wlan fingerprint-based indoor localization problems, 2014.

[125] L. Tu and Y. Chen. Stream data clustering based on grid density and attraction. *ACM Trans. Knowl. Discov. Data*, 3(3):12:1–12:27, July 2009.

[126] T. Urruty, C. Djeraba, and D. Simovici. Clustering by random projections. In P. Perner, editor, *Advances in Data Mining. Theoretical Aspects and Applications*, volume 4597, chapter Lecture Notes in Computer Science, pages 107–119. Springer Berlin Heidelberg, 2007.

[127] A. Vardy. Even more efficient bounded-distance decoding of the hexacode, the golay code, and the leech lattice. *Information Theory, IEEE Transactions on*, 41(5):1495–1499, 1995.

[128] A. Vattani. k-means requires exponentially many iterations even in the plane. In *Proceedings of the 25th annual symposium on Computational geometry*, SCG '09, pages 324–332, New York, NY, USA, 2009. ACM.

[129] S. S. Vempala. *The Random Projection Method.* DIMACS Series. American Mathematical Society, 2004.

[130] S. Wold, M. Sjstrm, and L. Eriksson. Pls-regression: a basic tool of chemometrics. *Chemometrics and Intelligent Laboratory Systems*, 58(2):109 – 130, 2001. {PLS} Methods.

[131] R. Xu and . I. I. Wunsch, D. Survey of clustering algorithms. *Neural Networks, IEEE Transactions on*, 16(3):645–678, May 2005.

[132] Y. Yang and K. Chen. Temporal data clustering via weighted clustering ensemble with different representations. *Knowledge and Data Engineering, IEEE Transactions on*, 23(2):307–320, Feb. 2011.

[133] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[134] R. Zass and A. Shashua. A unifying approach to hard and probabilistic clustering. In *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, volume 1, pages 294–301, Oct 2005.

[135] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proc of the 28th Int Conf on Very Large Data Bases*, pages 358–369, 2002.

[136] A. Zimek, E. Schubert, and H.-P. Kriegel. A survey on unsupervised outlier detection in high-dimensional numerical data. *Statistical Analysis and Data Mining*, 5(5):363–387, 2012.