RICE UNIVERSITY

Some Rare lSH Gems for Large-Scale Machine Learning

by

Chen Luo

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Anshul Shrivastava, Chair
Assistant Professor,
Computer Science, Rice University

Christopher Jermaine
Professor,
Computer Science, Rice University

Lin Zhong
Professor,
Electrical and Computer Engineering,
Rice University

Yingyan Lin
Assistant Professor,
Electrical and Computer Engineering,
Rice University

Houston, Texas
September, 2019
ABSTRACT

Some Rare LSH Gems for Large-Scale Machine Learning

by

Chen Luo

Locality Sensitive Hashing (LSH) is known as a popular algorithm for approximate nearest neighbor (ANN) search in high dimensional space. In this thesis, we instead take an alternate view of LSH as samplers. This view came in light recently in early 2016. We investigate the possibility of using LSH for addressing the computational and memory challenges in machine learning tasks. We show some rare ‘gems’ of locality-sensitive hashing that can shed important light on large-scale learning systems.

We first show the power of LSH for high-speed anomaly detection. Anomaly detection is one of the frequent and important subroutines deployed in large-scale data processing applications. Even being a well-studied topic, existing techniques for unsupervised anomaly detection require storing significant amounts of data, which is prohibitive from memory, latency and privacy perspectives, especially for small mobile devices that have ultra-low memory budget and limited computational power. We Introduce ACE (Arrays of (locality-sensitive) Count Estimators) algorithm that can
much faster than most state-of-the-art unsupervised anomaly detection algorithms with very low memory requirements.

Secondly, we show a novel use of LSH for scaling-up Split-Merge MCMC Inference. Split-Merge MCMC (Monte Carlo Markov Chain) is one of the essential and popular variants of MCMC for problems when an MCMC state consists of an unknown number of components. It is well known that state-of-the-art methods for split-merge MCMC do not scale well. Here, we leverage some unique properties of weighted MinHash, which is a popular LSH, to design a novel class of split-merge proposals which are significantly more informative than random sampling but at the same time efficient to compute.

In the end, we show a practical implementation and usage of LSH for Indoor Navigation tasks. In this chapter, we developed the first camera-based (privacy-preserving) indoor mobile positioning system, CaPSuLe, which does not involve any communication (or data transfer) with any other device or the cloud. The system only needs 78.9MB of memory and can localize a mobile device with 92.11% accuracy at a very fast speed. The ability to run the complete system on the mobile device eliminates the need for the cloud, making CaPSuLe a privacy-preserving localization algorithm by design as it does not require any communication.
Acknowledgments

First, I would like to thank my advisor Dr. Anshumali Shrivastava. He is a great mentor as well as a great friend. In these 4 years, he did not only teach me the domain knowledge of locality sensitive hashing, but also improved my skills of communication, presentation, writing, and all other things a Ph.D. candidate should learn. At the first year of coming to Rice, my writing skill and presentation skill is totally bad, Anshu teaches me patiently how to write and how to do the presentation. Besides all the research collaboration with Anshu, we are also great Friends, Anshu gives me a lot of advice on how to choose life paths and how to be a good man, he’s one of the most important role models in my life.

Then, I would like to thank my committee members: Dr. Christopher Jermaine, Dr. Lin Zhong, and Dr. Yingyan Lin. They spent their precious time to attend my talks, and they provided many valuable comments on my Ph.D. proposal, which made my thesis work much better. Also, I would like to thank Dr. Krishna Palem. He played an important part during my second-year Ph.D. study, I learned how to prepare slides and how to have a wide vision of research directions.

I would like to thank all my collaborators and members in our research group, Beidi Chen, Ryan Spring, and Tharun Kumar Reddy, etc. My thesis would not have been possible without their friendship and encouragement. I would also want to thank all my collaborators: Zhengzhang Chen, Lu-an Tang, Cheng Cao, James Caverlee,
Zhichun Li, Haifeng Chen, Jieping Ye, J. Jose Gonzalez E., Yongshik Moon, Soonhyun Noh, Daedong Park, and Seongsoo Hong, etc.

Last but not least, I would like to thank my parents, Bingguo Luo and Jinghua Chen, I left home at the age of 18, I’m already 29 now. For the past 11 years, I travel and gaining knowledge, the only thing I feel guilty is that I didn’t spend enough time for them. I thank them for their support and for letting me choose the path of my life.

Work in this thesis has been supported by the National Science Foundation IIS-1652131, BIGDATA-1838177, RI-1718478, AFOSR-YIP FA9550-18-1-0152, Amazon Research Award, ONR BRC grant on Randomized Numerical Linear Algebra.
Contents

Abstract ii
List of Illustrations x
List of Tables xii

1 Introduction 1

1.1 Contributions of the thesis .............................................. 1
  1.1.1 High-Speed anomaly detection on the edge using LSH .......... 1
  1.1.2 Scaling up Bayesian Inference using LSH ...................... 2
  1.1.3 CaPSuLe: Image-based indoor navigation using LSH .......... 4

2 Background 6

  2.1 Locality Sensitive Hashing ........................................... 6
  2.2 Some Popular Locality Sensitive Hash Functions .................. 7
    2.2.1 Signed Random Projections .................................... 7
    2.2.2 Weighted (or Generalized) MinHash ............................ 8
    2.2.3 Progress in Making Locality Sensitive Computations Faster .. 9
  2.3 Locality Sensitive Sampling (LSS) and Unbiased Estimators ...... 10

3 High Speed Anomaly Detection using LSH 13

  3.1 Introduction and Background ....................................... 13
3.1.1 Challenges for Anomaly Detection on the Edge .............. 14
3.2 Literature Review ............................................. 17
3.3 ACE: Arrays of Locality Sensitive Counts Estimator ............. 19
3.4 Our Proposal ..................................................... 19
   3.4.1 Can it Discriminate Outliers? ......................... 20
   3.4.2 ACE (Arrays of (locality-sensitive) Counts Estimator) Algorithm 21
   3.4.3 Theory: Analysis and Superiority over Random Sampling ... 24
   3.4.4 Implementation Details, Running Time, Cache Utilization and Memory ..................... 30
3.5 Discussions: Privacy Preserving Anomaly Detection ............ 33
3.6 Experimental Evaluations ..................................... 34
   3.6.1 Datasets .................................................... 34
   3.6.2 Baselines .................................................. 36
   3.6.3 Methodology and Results ................................. 40
   3.6.4 Accuracy Comparison. ................................. 41
   3.6.5 Running Time Comparison. ......................... 41
   3.6.6 Memory Analysis. ...................................... 42
3.7 Discussion: Effects of K and L .............................. 42
3.8 Discussion ..................................................... 43

4 Scaling Up Bayesian Inference via Locality Sensitive Hashing 49
   4.1 Introduction and Background ................................ 49
   4.2 Literature Review .......................................... 53
4.2.1 Weighted (or Generalized) MinHash ........................................ 53
4.2.2 Split-Merge MCMC .......................................................... 54

4.3 LSS based Split-Merge MCMC ................................................. 55
  4.3.1 Naive LSS based Proposal Design ................................. 56
  4.3.2 MinSM: MinHash based Split-Merge MCMC .................. 62

4.4 Empirical Study ................................................................. 70
  4.4.1 Gaussian Mixture Model .............................................. 70
  4.4.2 Competing Algorithms .............................................. 71
  4.4.3 Dataset ................................................................. 71
  4.4.4 Speed Comparison and Analysis .................................. 73
  4.4.5 Clustering Accuracy Comparison .................................. 74

4.5 Discussion ................................................................. 76

5 CapSule: Image based Indoor navigation using LSH 79

5.1 Introduction ................................................................. 79

5.2 Devise Positioning via Image Matching ............................... 81
  5.2.1 Dataset and Settings .............................................. 82
  5.2.2 Device and Platform .............................................. 84
  5.2.3 The Image Matching Problem and its Computational
        (Energy) Barrier .................................................. 84
  5.2.4 Clustering (Bag-of-Words (BoW), sparse coding, etc.) does
        not seem to help .............................................. 86

5.3 Hope: Probablistic Hashing Algorithms ............................... 87
5.4 The CapSule System: Near-Cloud Performance with on-Device Computation

5.4.1 Dynamic Updates

5.5 Experimental Result

5.5.1 Performance Summary

5.6 Discussion

6 Conclusion

6.1 Moving Forward
Illustrations

3.1 **Discriminative power of** $S(q, D)$: We can see from the figure that the value of $S(q, D)$ for an Outlier is significantly lower (different) compared to that of non-outliers. .................................................. 21

3.2 We use the LSH hash of the data points to increment corresponding counters into different (independent) hash arrays. We do not save anything, we only increase the value by 1 for each bucket and then forget the data. ................................................................. 24

3.3 Comparison of ACE estimator with random sampling estimator on three datasets. The x-axis denotes the number to arrays and size of samples for ACE estimator and random sampling estimator respectively. ACE estimator is not only more accurate but also cheaper compared to random sampling estimators from the computational perspective. ........................................... 29

4.1 Three way Minwise Hashing. .................................................. 65
4.3 The time and iteration wise comparison of the likelihood for difference methods on the two real dataset. It is obviously that our proposed MinSM algorithm can be at least 6 times faster than the state of the art algorithms in the real large dataset.

4.4 The time and iteration wise comparison of the likelihood for difference methods on the Synthetic Dataset. MinSM outperforms the other baselines by a large margin. It is also clear that requiring less iteration does not mean faster convergence.

5.1 Example Query and returned matches image by CaPSuLe system. The match is with varying pose and orientations showing the complexity of our dataset.

5.2 Example Query and returned matches image by CaPSuLe system. The match is with varying pose and orientations showing the complexity of our dataset.

5.3 The required memory/(main memory) with varying K. We load the buckets in main memory on demand.

5.4 The accuracies of bruteforce, BoW and CaPSuLe.

5.5 Energy consumption Vs response time.
3.1 The statistics of the three datasets. .............................................. 36
3.2 Comparison Algorithms and Their Parameter Values Recommended
   for These Benchmark Datasets. .................................................. 45
3.3 Comparison Results on Statlog Shuttle Dataset ............................. 46
3.4 Comparison Results on Image Object Dataset ............................... 47
3.5 Comparison Results on KDD-CUP 99 Dataset ............................... 48
4.1 The statistics for the two real world dataset ................................. 73
4.2 Clustering Accuracy for Different Methods ................................. 75
5.1 Target System Description ....................................................... 84
5.2 Evaluations for CaPSuLe and Bruteforce .................................... 93
Chapter 1

Introduction

Locality Sensitive Hashing (LSH) is a widely used algorithm for approximate nearest neighbor (ANN) search in high dimensional space. In this thesis, we instead take an alternate view of LSH as samplers. This view came in light recently in early 2017 [SS17a]. We investigate the possibility of using LSH sampler for addressing the computational and memory challenges in machine learning tasks. We show some rare 'gems' of locality-sensitive hashing that can shed important light on large-scale learning systems.

1.1 Contributions of the thesis

1.1.1 High-Speed anomaly detection on the edge using LSH

In chapter 2, we introduced a high-speed anomaly detection technique using hashing. The problem of anomaly (or outlier) detections is the task of identifying instances (or patterns) in data that do not conform to the expected behavior [CBK09]. These non-conforming examples are popularly referred to as anomalies, or outliers, sometimes interchangeably.

In this chapter, we are dealing with the task of anomaly detection on the edge devices using Locality sensitive hashing. There are many challenges for dealing with large-scale high dimensional anomaly detection tasks on edge devices: (1) Data are
high-speed and drifting Data. (2) Ultra-Low Memory Budget on edge devices (3) Limited computational power on edge devices (4) Privacy issues on edge devices.

As a result, we propose a family of statistics which provides a “sweet” spot between the ability to discriminate anomalies and the resource efficiency. These special statistics, due to their form, can be efficiently computed in ultra-low memory, and they do not require storing even a single data sample. Furthermore, any updates to the data can be incorporated on the fly making our proposal ideal for high-speed data applications. The proposed algorithm, besides, has strong privacy properties making it ideal for IoT (Internet of Things) settings.

We provide a comparison of our algorithm with 11 different methodologies, which include some of the fastest and most popular anomaly detection algorithms. Our experiment shows that we are around at least 60x faster than of the best performing competitor on the largest benchmark KDD-cup99 HTTP dataset. This disruptive speedup is not surprising given the computational simplicity of our algorithm and ultra-low memory print.

1.1.2 Scaling up Bayesian Inference using LSH

In chapter 3, we show a novel sampler view of LSH and propose to use LSH for scaling-up Split-Merge MCMC Inference.

Bayesian mixture models are of great interest due to their flexibility in fitting a countably infinite number of components that can grow with the data [MYB04]. The growth of model complexity with the data is also in agreement with modern progress in machine learning over massive datasets. However, the appealing properties of Bayesian modeling come with hard computational challenges. Even with simple
mixture models, the mathematical problems associated with training and inference are
intractable. As a result, recent research focuses on developing tractable computational
techniques. In particular, the use of Markov chain Monte Carlo (MCMC) methods, to
sample from the posterior distribution [And+03; Nas07; WB12] is widely prevalent.

In this chapter, we aimed at the problem of split merge MCMC. In the seminal
work of [JN04], split merge MCMC procedure was proposed. Split-merge MCMC
is one type of MCMC that the state of MCMC consists of a set of components or
clusters, and there are two major operations for transition: split or merge. Split
merge MCMC also has the same problem of slow convergence. To mitigate the slow
progress, RGSM is proposed. The idea of RGSM was to use restricted Gibbs sampling
to generate proposals with a higher likelihood of acceptance, instead of a random pro-
posal. Thus, a less number of MCMC iterations were sufficient for convergence due to
fewer rejections. However, the cost of restricted Gibbs is very high. As a result, even
though the iterations are less, each iteration is costly making the overall algorithm
slow, especially for large datasets. Our experiments confirm this slow convergence of
RGSM. In this chapter, we show that it is possible to construct informative proposals
without sacrificing the per-iteration cost. We leverage a simple observation that while
designing proposals we can favor configurations where entities similar are likely to be
in the same component. We use standard notions of vector similarity such as cosine or
Weighted Jaccard. To perform such sampling efficiently, we capitalize on the recent
advances in LSH sampler [LS18; SS17a; CS17a] that can perform adaptive sampling
based on similarity. Our proposed algorithms obtain a sweet tradeoff between the
number of iteration and computational cost per iteration. As a result, we reduce the
overall convergence in time, not just in iterations.
On two large public datasets, our proposal MinHash Split-Merge (MinSM) significantly outperforms other state-of-the-art split-merge MCMC algorithms in convergence speed as measured on the wall clock time on the same machine. Our proposed algorithm is around 6x faster than the second-best baseline on synthetic datasets as well as real-world datasets without loss and accuracy.

1.1.3 CaPSuLe: Image-based indoor navigation using LSH

In chapter 4, we show a practical usage of LSH for Indoor Navigation tasks. At the beginning of the chapter, we first introduce several tricks of designing good LSH systems, then, we introduce the CapSuLe system, a first camera-based (privacy-preserving) indoor mobile positioning system. CapSuLe does not involve any communication (or data transfer) with any other device or the cloud. Indoor localization technology is expected to be a 4 billion dollar industry by 2018 [Liu+07]. Increased demand for accurate indoor localization market is due to venue-based marketing, the poor performance of GPS in indoor environments [Mau09], and government initiatives in developing positioning systems for public safety and urban security segments. We propose CaPSuLe for image-based device positioning, a first of its kind system, which is free from all the three problems of communication, energy consumption, and privacy. At the heart of CaPSuLe lies an approximate image matching algorithm, based on fast locality sensitive hashing, which is more than 500x times cheaper than any state-of-the-art image matching algorithm. Such a significant gain in computation and energy cost is a result of careful choices of hash tables, hash functions, and related operations. This massive reduction allows us to perform end-to-end image matching on the mobile device itself. Our algorithm takes 1.92 seconds requiring 3.78 Joules
energy on Samsung Galaxy S4 archiving 92.11% accuracy in estimating the location. Since all computations are local and are performed on the device, our algorithm is free from privacy infringements as no information is transmitted. We hope that our work will lead to many new energy-efficient machine learning algorithms where the need for cloud computing can be eliminated.
Chapter 2

Background

In this chapter, we briefly review Locality Sensitive Hashing (LSH) families and several variants of LSH. Then we provide a new sampling view of LSH, which will be used for most of the works introduced in this thesis. The concepts described in this chapter will be heavily used throughout this thesis.

2.1 Locality Sensitive Hashing

Locality-Sensitive Hashing (LSH) \cite{indyk1998approximate} is a popular technique for efficient approximate nearest-neighbor search. LSH is a family of functions, such that a function uniformly sampled from this hash family has the property that, under the hash mapping, similar points have a high probability of having the same hash value. More precisely, consider $\mathcal{H}$ a family of hash functions mapping $\mathbb{R}^d$ to a discrete set $[0, R - 1]$.

**Definition 1 Locality Sensitive Hashing (LSH) Family** A family $\mathcal{H}$ is called $(S_0, cS_0, u_1, u_2)$-sensitive if for any two points $x, y \in \mathbb{R}^d$ and $h$ chosen uniformly from $\mathcal{H}$ satisfies the following:

- if $\text{Sim}(x, y) \geq S_0$ then $\Pr_{\mathcal{H}}(h(x) = h(y)) \geq u_1$
- if $\text{Sim}(x, y) \leq cS_0$ then $\Pr_{\mathcal{H}}(h(x) = h(y)) \leq u_2$
A collision occurs when the hash values for two data vectors are equal, meaning that \( h(x) = h(y) \). The probability of a collision for a LSH hash function is generally proportional to some monotonic function of similarity between the two data vectors, i.e.,

\[
Pr[h(x) = h(y)] \propto f(\text{sim}(x, y)),
\]

where \( \text{sim}(x, y) \) is the similarity under consideration and \( f \) is some monotonically increasing function. Essentially, similar items are more likely to collide with each other under LSH mapping.

### 2.2 Some Popular Locality Sensitive Hash Functions

LSH is a very well studied topic in computer science theory and database literature. There are many well-known LSH families in the literature. Please refer [G+99] for details. We briefly some popular ones in this section.

#### 2.2.1 Signed Random Projections

Signed Random Projections (SRP) is an LSH for the cosine similarity measure, which originates from the concept of randomized rounding (SRP) [GW94; Cha02; LS17].

Given a vector \( x \), SRP utilizes a random \( w \) vector with each component generated from i.i.d. normal, i.e., \( w_i \sim N(0, 1) \), and only stores the sign of the projection. Formally SRP family is given by

\[
h_w(x) = \text{sign}(w^T x). \tag{2.1}
\]

It was shown in the seminal work [GW94] that collision under SRP satisfies the following equation:

\[
Pr_w(h_w(x) = h_w(y)) = 1 - \frac{\theta}{\pi}, \tag{2.2}
\]
where $\theta = \cos^{-1}\left(\frac{x^Ty}{\|x\|_2\|y\|_2}\right)$. $\frac{x^Ty}{\|x\|_2\|y\|_2}$ is the cosine similarity.

If we generate $K$ independent SRP bits, by sampling $w$ independently $k$ times, and use the generated $K$-bit number as the new hash function $H$, then the new collision probability is

$$Pr(H(x) = H(y)) = (1 - \frac{\theta}{\pi})^K$$

(2.3)

by the simple multiplicative law of probability. We will be using this observation heavily in our work.

### 2.2.2 Weighted (or Generalized) MinHash

Minwise hashing [LRU14] is the LSH for resemblance similarity. The minwise hashing family applies a random permutation $\pi : \Omega \rightarrow \Omega$, on the given set $W$, and stores only the minimum value after the permutation mapping. Formally MinHash is defined as $h_{\pi}^{\min}(W) = \min(\pi(W))$. Given sets $W_1$ and $W_2$, it can be shown by elementary probability argument that:

$$Pr(h_{\pi}^{\min}(W_1) = h_{\pi}^{\min}(W_2)) = \frac{|W_1 \cap W_2|}{|W_1 \cup W_2|}$$

Weighted Minwise Hashing is a known LSH for the Weighted Jaccard similarity [LRU14]. Given two positive vectors $x, y \in \mathbb{R}^D$, $x, y > 0$, the (generalized) Weighted Jaccard similarity is defined as $J(x, y) = \frac{\sum_{i=1}^{D} \min\{x_i, y_i\}}{\sum_{i=1}^{D} \max\{x_i, y_i\}}$, where $J(x, y)$ is a frequently used measure for comparing web-documents [LRU14], histograms (specially images), gene sequences, etc.
Weighted Minwise Hashing (WMH) (or Minwise Sampling) generates randomized hash (or fingerprint) $h(x)$, of the given data vector $x \geq 0$, such that for any pair of vectors $x$ and $y$, the probability of hash collision (or agreement of hash values) is given by $Pr(h(x) = h(y)) = \frac{\sum_{i} \min\{x_i, y_i\}}{\sum_{i} \max\{x_i, y_i\}}$.

A unique property of Minwise Hashing is that there is a natural extension of $k$-way collision [SL13]. In particular, given vectors $x^{(1)}$, $x^{(2)}$, ..., $x^{(s)}$, the simultaneous collision probability is given by:

$$Pr(h(x^{(1)}) = h(x^{(2)}) = ... = h(x^{(s)})) = \frac{\sum_{j} \min\{x^{(1)}_j, x^{(2)}_j, ..., x^{(s)}_j\}}{\sum_{j} \max\{x^{(1)}_j, x^{(2)}_j, ..., x^{(s)}_j\}}$$  \hfill (2.4)

Minwise hashing can be extended to negative elements using simple feature transforms [Li17], which essentially doubles the dimensions to 2D. In this thesis, MinHash and Weighted MinHash denote the same thing.

### 2.2.3 Progress in Making Locality Sensitive Computations Faster

Over the last decade, there has been a significant advancement in reducing the amortized computational and memory requirements for computing several LSH signatures of the data vector. For random projections based LSH, of which signed random projection is a special case, we can calculate $m$ LSH hashes of the data vector, with dimensions $d$, in time $O(d \log d + m)$, a significant improvement over $O(dm)$. This speedup is possible due to the theory of Fast-Johnson-Lindenstrauss transformation [AC06; DKS11]. On the orthogonal side, even better speedup of $O(d + m)$ has been obtained with permutation-based LSH, such as minwise hashing, using ideas
of densification [SL14a; SL14b; Shr16; Shr17]. These drastic reductions in hashing
time have been instrumental in making LSH based algorithms more appealing and
practical.

2.3 Locality Sensitive Sampling (LSS) and Unbiased Estimators

LSH was considered as a black-box algorithm for similarity search and dimensionality
reduction. Recent research [SS17c; SS17a; CS17a; LS18; CSS17] found that LSH can
be used for something more subtler but useful. It is a data structure that can be used
for efficient dynamically adaptive sampling. We first describe the sampling algorithm
of [SS17a; CS17a; LS18; CSS17] and later comment on its properties crucial to our
proposal.

The algorithm uses two parameters - \((K, L)\). We construct \(L\) independent hash
tables from the collection \(C\). Each hash table has a meta-hash function \(H\) that is
formed by concatenating \(K\) random independent hash functions from some appropri-
ate locality-sensitive hash family \(\mathcal{H}\). The candidate sampling algorithm works in two
phases [SS17a; CS17a; LS18; CSS17]:

1. **Pre-processing Phase:** We construct \(L\) hash tables from the data by storing
   all elements \(x \in C\). We only store pointers to the vector in the hash tables
   because storing whole data vectors is very memory inefficient. This is a one-
time linear cost.

2. **Sampling Phase:** Given a query \(q\), we collect one bucket from a randomly
   selected hash table and return a random element from the bucket. If the bucket
is empty, we reselect a different hash table again. Keep track of the number of
different tables $T$ probed.

It is not difficult to show that an item returned as a candidate from a $(K, L)$-
parameterized LSH algorithm is sampled with probability exactly $1 - (1 - p^K)^L \times \frac{1}{S}$, where $p$ is the collision probability of LSH function and $S$ is the number of elements in
the bucket [SS17a; CS17a; LS18; CSS17]. The LSH family defines the precise form of
$p$ used to build the hash tables. Specifically, when $L = 1$ and $K = 1$, the probability
reduced to the collision probability itself ($p$). Our proposal will heavily rely on this
unusual probability expression $1 - (1 - p^K)^L \times \frac{1}{\text{size}}$ to design an informative and
proposal distribution.

This sampling view of LSH was first utilized to perform adaptive sparsification
of deep networks in near-constant time, leading to efficient backpropagation algo-
rithm [SS17c]. A year later, [SS17a] demonstrated the first theory of using these
samples for unbiased estimation of partition functions in log-linear models. More
specifically, the authors showed that since we know the precise probability of sampled
elements $1 - (1 - p^K)^L$, we could design provably unbiased estimators using importance
sampling type idea. This was the first demonstration that random sampling could be
beaten with roughly the same computational cost as vanilla sampling. [LS18] used
the same approach for unbiased estimation of anomaly scoring function. [CSS17]
used the sampling in a very different context of connected component estimation for
unique entity counts. [CS17b] showed improvements in sample complexity of kernel
density estimation problems.

The most important observation made in [SS17a], is that the expression $1 - (1 -
$p^K \cdot L$ is monotonically increasing function of $p$, which in turn is a monotonic function of cosine similarity if we use SRP as hashing scheme. Thus, given a query $q$, points with higher cosine similarity with $q$ are more likely to be sampled. Similarly, points dissimilar with $q$ is less likely to be sampled. It should be noted that querying cost only involved few (like 5-10) hash computations followed by a couple of memory lookups which is $O(1)$, very similar to random sampling. Capitalizing on this unique efficiency [CXS18a] proposed LSD (locality-sensitive descent), which was the first gradient descent algorithm that can beat the popular SGD, and any of its variants, on running time breaking what they call the chicken-and-egg loop in adaptive sampling.
Chapter 3

High Speed Anomaly Detection using LSH

In this chapter, we introduce a novel usage of LSH for high dimensional anomaly detection on the edge.

3.1 Introduction and Background

The problem of anomaly (or outlier) detections is the task of identifying instances (or patterns) in data that do not conform to the expected behavior [CBK09]. These non-conforming examples are popularly referred to as anomalies, or outliers, sometimes interchangeably.

Anomaly detection can be either supervised [LL05] or unsupervised [Gör+13]. Supervised anomaly detection leverages machine learning algorithms, such as classification, over datasets labeled as anomalous or non-anomalous. However, there are three major issues with supervised anomaly detection algorithms: 1) In most applications, label information about anomalies is not available; 2) Anomalies are rare, and hence there is a huge class imbalance, and 3) Supervised algorithms need to be re-trained for drifting data distributions with new label information. Drifting data distribution is quite common in big-data systems, where supervised re-training is prohibitive. Therefore, we are interested in unsupervised anomaly detection which does not require any label information, and which can automatically deal with changes in
data distributions over time. We briefly describe some of the modern challenges for unsupervised anomaly detection that we will address in this chapter.

### 3.1.1 Challenges for Anomaly Detection on the Edge

**Challenge 1: High-Speed Drifting Data:** Many applications demand fast-response and real-time inference from dynamic and drifting high volumes of sensor data over time. Most anomaly detection applications, for example over the web-network servers, require dealing with unprecedented amounts of data in a fraction of seconds. The data distribution is constantly changing, and it is often bursty [Kle02; Luo+14]. Detecting anomaly events in real-time, such as DDoS (Distributed Denial of Service) attacks, network failures, etc., is highly beneficial in monitoring network performance.

**Challenge 2: Ultra-Low Memory Budget:** In many high-speed streaming applications, such as High Energy Physics (HEP) and network servers, any algorithm requiring to store and process a significant fraction of data is prohibitive. Another critical pushing need for ultra-low memory algorithm is anomaly detection on mobile phones or smart sensors. Algorithms which require significant resources are prohibitive for low-resource platforms.

**Challenge 3: Anomaly Detection on the Edge (Mobile Devices):** Anomaly detection on portable devices or mobile devices often requires dealing with high-speed drifting data, low-memory, and in addition ultra-low power. A modern smartphone usually only have a relatively small memory capacity (1,2 Gigabytes) and limited computational power. Battery life is a significant concern, and transmitting data to cloud for analysis has privacy risks as well as are not sustainable due to their energy demands. With the accelerated adoption of 4G (or 5G) technologies includ-
ing WiMAX and LTE, cellular devices will become the primary means of broadband
Internet access for many users. According to the report from Cisco, Global mobile
data traffic reached 7.2 exabytes per month at the end of 2016 [Cis14]. Thus, the
traffic data that is monitored, or generated, by the mobile devices are extremely high-
speed and enormous, and it is hopeless to rely on anomaly detection methods which
require consulting a significant fraction of the data. Unfortunately, most unsuper-
vised anomaly detection techniques are near-neighbor based and require querying,
and hence the prohibitive storage of the historical data.

**Challenge 4: Privacy:** As the IoT becomes more widespread, consumers must
demand better security and privacy protections that do not leave them vulnerable to
corporate surveillance and data breaches. Thus, storing a significant fraction of data
for finding anomalous behavior is prohibitive. Privacy-preserving anomaly detection
is a challenging problem in itself [burkhart2010sepia].

To resolve all the challenges above, in this chapter, we propose a family of statistics
which provides a “sweet” spot between the ability to discriminate anomalies and
the resource efficiency. These special statistics, due to their form, can be efficiently
computed in ultra-low memory, and they do not require storing even a single data
sample. Furthermore, any updates to the data can be incorporated on the fly making
our proposal ideal for high-speed data applications. The proposed algorithm, in
addition, has strong privacy properties making it ideal for IoT (Internet of Things)
setting.

Our proposed family of statistics are derived from the collision probability of
locality sensitive hashing (LSH) functions. We show that these classes of statistics
have strong discriminative property for identifying outliers and most importantly,
it can be accurately estimated using Arrays of Count Estimators (ACE), a novel and tiny LSH based data structure. Designing these estimators requires using the sampling view of LSH rather than the widely popular near-neighbor search view. To the best of our knowledge, this is the first use of LSH counts as unbiased estimators of outlierness.

We demonstrate, empirically and theoretically, that the proposed LSH based count estimators are significantly more accurate than random sampling approaches. Our ACE algorithm only requires computing few locality sensitive hashes of the data and a small set of count array lookups to estimate the proposed statistics sharply. Our approach does not require even a single distance computation. The theory and the class of estimators presented in the work, could of independent interest in itself.

We demonstrate rigorous experimental evidence on three public outlier detection benchmarks including the largest publicly available benchmark dataset KDD-cup99 HTTP dataset having more than half a million labeled instances. Empirically, our algorithm only requires around 4MB of memory and near-constant amount of computations, for all the three benchmark datasets. Thus, we can exploit fast L3 caches (Level 3 caches), which can be significantly faster than dealing with main memory.

We provide a comparison of our algorithm with 11 different methodologies, which include some of the fastest and most popular anomaly detection algorithms. Our experiment shows that we are around at least 60x faster than of the best performing competitor on the largest benchmark KDD-cup99 HTTP dataset. This disruptive speedup is not surprising given the computational simplicity of our algorithm and ultra-low memory print.
3.2 Literature Review

There are numerous methods for unsupervised anomaly detection in literature. We review and compare with 11 of these popular methods in our experiments. Unsupervised anomaly detection can be broadly categorized into two categories: 1) Near-Neighbor (NN) Based and 2) Aggregate Statistics (or score) based. NN based approaches typically define the outlier score of a point $q$ based on the difference between $q$’s own behavior and the behavior of $q$’s near-neighbors. The first category is the most common category.

Aggregate statistics based methods, on the other hand, define the outlier score of a point $q$ based on the expected behavior of a global function $S(q, D)$ of the data $D$, relative to $q$. A notable method among them is ABOD (Angle-based outlier detection) [PP12]. ABOD computes the variance of the angle formed by different pairs of points, in the dataset, incident on the point of interest $q$. It is expected that the outliers will have a small variance [PP12].

There are several implementations of existing anomaly detection algorithms. A notable among them is the ELKI package [Ach+09] which is currently one of the most efficient and popular packages for outlier detection because of highly optimized implementations.

Both categories of anomaly detection algorithms require storing the complete dataset to either compute near-neighbor or the desired statistics from the data. The bottleneck computational cost is at least one pass over the data to either calculate the near-neighbor or the statistics. Thus, these methods have poor computational and memory requirements. Furthermore, change in the distribution of data requires
storing and processing a larger set of observations.

To work around the computational requirements it is natural to resort to fast alternatives [Cha+15]. There are plenty of techniques which exploits efficient near-neighbor capabilities to speed up NN. However, they still require storing the data in the memory. Even with the computational speedups, the methodologies are still slow for ultra-high speed data mining, as an accurate near-neighbor search over large data is costly.

Relying on random sampling and projections of the data to estimate the aggregate statistics efficiently is not new [CBK09]. For example, recently, [PP12] showed that using smart random sampling and hashing algorithms, we can speed up the anomaly detection and also reduce the memory requirement. Instead of storing all the data points, we only need few random samples and their quantized projections. They proposed FastVOA which uses a modified ABOD statistics that can be estimated in near-constant time and is as good as ABOD for anomaly detection.

However, these approximation methods still require storing a significant number of data samples, which makes the algorithm slow and prohibitive from privacy perspective. FastVOA involves various computation of medians and other costly statistics. Our experiments show that the sampling based FastVOA approach is significantly slower than fast NN based alternatives.

There is a third category of anomaly detection algorithms over a sliding window in data streams [TFS16]. The notions of anomalies in these algorithms are confined to a given fixed-size window over time. Not surprisingly, if the size of sliding window is increased to take into account large amounts of data, we again observe the same memory and latency issues.
The focus of this chapter is on unsupervised outlier detection, where the notion of anomaly is with respect to the complete data and not constrained to a limited sliding window.

3.3 ACE: Arrays of Locality Sensitive Counts Estimator

In this section, we introduced our proposed ACE algorithm for anomaly detection for high dimensional data.

3.4 Our Proposal

Denote the dataset with $\mathcal{D} = \{x_i| i \in [1, n]\}$, where $n$ is the number of data points in $\mathcal{D}$. By definition, outliers are significantly separated from an average data point. Therefore, any reasonable statistics of $x_i$ with respect to all other $x_j \in \mathcal{D}$ will deviate significantly for outliers compared to a normal data point. Even an average distance of $x_i$ with all other elements of $\mathcal{D}$ is a reasonably good statistics [RRS00]. However, as noted before, computing these statistics requires storing the complete data $\mathcal{D}$. In general, calculating every single $S(x_i, \mathcal{D})$ requires one complete pass over the dataset $\mathcal{D}$. Besides, our experiments show that alternative estimations based on random sampling and random projections still lead to significant computational overheads.

We instead focus on classes of scoring functions $S(\ldots)$ over the dataset that can be estimated efficiently using a tiny (memory efficient) data structure that can easily fit fast processor cache. Furthermore, we also want to update the data structure on the fly. In particular, any change in data from $\mathcal{D}$ to $\mathcal{D}'$ requires no change, and the estimates get dynamically adjusted.
We show that a class of scoring functions of the following form have the aforementioned property:

\[ S(q, D) = \sum_{x_i \in D} p(q, x_i)^K, \]  

(3.1)

where \( p \) is the collision probability of any LSH family and \( K \geq 1 \) is an integer.

The analysis of this work extends naturally to any LSH scheme. For this work, we will focus on the popular signed random projections (SRP) as the LSH because of its simplicity. Furthermore, advances in fast SRP have lead to some very lightweight hashing variants. With SRP, the collision probability \( p(q, x_i) \) is given by:

\[ p(q, x_i) = 1 - \frac{1}{\pi} \cos^{-1}\left(\frac{q^T x_i}{\|q\| \|x_i\|}\right) \]

which will also be the value of \( p(q, x_i) \) for the rest of this chapter.

### 3.4.1 Can it Discriminate Outliers?

To demonstrate the discriminative power of the scoring function in Equation 3.1, we do a simulation experiment similar to the one performed in [PP12]. We first generate a simple dataset with an outlier point. Figure 3.1 (left) shows the snapshot of the data. There are two sets of data points. The outlier and the general data points. For the general data points, in addition, we make a distinction between the border points and inner points as illustrated in the figure.

In Figure 3.1 (right), we plot the value of our proposed statistics \( \frac{1}{n}S(q, D) \), given by Equation 3.1, for different sets of data points as a function of \( K \). We can see from the figure the value of \( \frac{1}{n}S(q, D) \) for an outlier point is near zero. In particular, it is significantly lower compared to the values of the same statistics for inner points and
Figure 3.1: **Discriminative power of** $S(q, D)$: We can see from the figure that the value of $S(q, D)$ for an Outlier is significantly lower (different) compared to that of non-outliers.

even border points. This behavior is expected. Note that our statistics is a sum of collision probabilities of the LSH mapping over all the data points $x_i \in D$. From the theory of LSH, the collision probability $p(q, x_i)$ indicates the level of similarity between $q$ and $x_i$. If $q$ is an outlier, $p(q, x_i)$ is expected to be significantly low. We will further demonstrate the usefulness of these statistics in the experiments section.

### 3.4.2 ACE (Arrays of (locality-sensitive) Counts Estimator) Algorithm

For the ease of explanation, we first describe the procedure of our proposed ACE algorithm. We later show that this procedure is an efficient statistical estimator of our proposed outlier score $S(q, D)$ defined by Equation 3.1.

The overall process of ACE is summarized in Algorithm 1. Our ACE algorithm, uses $K \times L$ independent SRP hash functions $h_i$, each given by Equation 2.1. $K$ and $L$ are hyperparameters that are pre-specified. Note, this is analogous to the traditional
Algorithm 1 Arrays of (locality-sensitive) Count Estimator (ACE) Algorithm

1: **Input:** Dataset \( D \), Number of Hashes \( K \), Number of Hash tables \( L \), \( \alpha \)

2: **Hash Initialize:** Generate \( L \ H_j(.) \) using \( K \) independent SRPs each.

3: for \( i = 1 \) to \( L \) do

4: \( A_j = \text{new short}[2^K](0) \) (Short Arrays)

5: \( \mu = 0, n = 0 \)

6: end for

7: **Online Addition Phase**

8: for \( x_i \in D \) do

9: \( \mu_{\text{incre}} = 0 \)

10: for \( j = 1 \) to \( L \) do

11: \( A_j[H_j(x_i)]++ \)

12: \( \mu_{\text{incre}} = \mu_{\text{incre}} + \frac{2A_j[H_j(x)]+1}{L} \)

13: end for

14: \( \mu = \frac{1}{n+1} \left( n\mu + \mu_{\text{incre}} \right) \)

15: \( n++; \)

16: end for

17: **Query (Detection) Phase:** Given query \( q \)

18: \( \hat{S}(q, D) = 0 \)

19: for \( j = 1 \) to \( L \) do

20: \( \hat{S}(q, D) = \hat{S}(q, D) + \frac{1}{L} A_j[H_j(x_i)] \)

21: end for

22: if \( \hat{S}(q, D) \leq \mu - \alpha \) then

23: report \( q \)

24: end if

(\( K, L \)) parameterized LSH algorithm for near-neighbor search. However, we do not perform any retrieval which requires heavy hash tables with buckets of candidates for each hash index. For near-neighbor, we further need to compute the distances of these candidates to identify the best.

On the contrary, our algorithm does not require a single distance computation. Our method only needs to check the value of a simple counter at each index. We only
need arrays of counters. The process is significantly efficient, both in memory and speed, compared to a single LSH near-neighbor query.

We use Signed Random Projections (SRP) $h^{\text{sim}}$ (Equations 2.1) which gives one-bit output. Using these 1-bit outputs, we then generate $L$ different meta-hash functions given by

$$H_j(x) = [h_{j1}(x); h_{j2}(x); \ldots; h_{jK}(x)]$$

of $K$ bits each. The $K$ bits are generated by concatenating the individual bits. Here $h_{ij}, i \in \{1, 2, \ldots, K\}$ and $j \in \{1, 2, \ldots, K\}$, are $K \times L$ independent evaluations of the SRP.

The overall algorithm works in the following two phases:

1) **Counting Phase**: We construct $L$ short arrays, $A_j, j = \{1, 2, \ldots, L\}$, of size $2^K$ each initialized with zeros. Given any observed element $x \in D$, we increment the count of the corresponding counter $H_j(x)$ in array $A_j$, for all $j$s. Thus, every counter keeps the total count of the number of hits to that particular index (See Figure 3.2). The total cost of updating the data structure for any given $x$ is $KL$ SRP computations followed by $L$ increments.

**Mean Update on Fly**: For each $x_i \in D$ our estimated score is

$$S(x_i, D) = \frac{1}{L} \sum_{j=1}^{L} A_j[H_j(x_i)]$$

We compute the mean behavior $\mu$ of the scores over all the element in dataset $x \in D$.

$$\mu = \frac{1}{n} \sum_{i=1}^{n} S(x_i, D).$$

Deviation from this mean will indicate outlierness. It turns out that we can dynami-
Figure 3.2: We use the LSH hash of the data points to increment corresponding counters into different (independent) hash arrays. We do not save anything, we only increase the value by 1 for each bucket and then forget the data.

cally update the mean $\mu$ on fly, as we read (or observe) the new data as shown in the algorithm. See Section 3.4.4 for details.

2) Real-time (query) Phase: Given a query $q$, for which we want to compute the score, we report the average of all the counters $A_j[H_j(q)] \forall j \in \{1, 2, \ldots, L\}$, i.e.,

$$
\hat{S}(q, D) = \frac{1}{L} \sum_{j=1}^{L} A_j[H_j(q)].
$$

We report $q$ as an anomaly if the estimated score $\hat{S}(q, D)$ is less than $\mu - \alpha$, where $\alpha$ is some preselected hyperparameter. The overall cost for querying is $KL$ SRP computations and $L$ lookups followed by a simple average calculation.

3.4.3 Theory: Analysis and Superiority over Random Sampling

We first define few notations needed for analysis. Given a query point $q$. For convenience, we will denote $p(q, x_i)$, the collision probability of the SRP of $q$ with that of
Due to space limitations the proofs are omitted.

**Intuition: LSH as Samplers** LSH is widely accepted as a black box algorithm for near neighbor search. We take an alternative adaptive sampling view of LSH which has emerged very recently [SS17d; SS17b; CSS17; CS17b; CXS18b]. As argued in Section 2.1, for a given query \( q \) and \( K \)-bit SRP hash function \( H_j \), the probability that any element \( x_i \) increments the count of location \( H_j(q) \) (the location of query) in array \( A_j \) is precisely \( p(q, x_i)^K \). Using this observation, we will show that the count of the number of elements, from \( \mathcal{D} \), hitting the bucket of query \( H_j(q) \) is an unbiased estimator of the \( S(q, \mathcal{C}) = \sum_{i=1}^{n} p(q, x_i)^K \). This novel use of LSH as efficient data structure for statistical estimation could be of independent interest in itself.

We define indicator variable \( \mathbb{I}_{x_i \in B_q} \) as

\[
\mathbb{I}_{x_i \in B_q} = \begin{cases} 
1, & \text{if } x_i \text{ is in the bucket of } q \\
0, & \text{otherwise.} 
\end{cases} 
\] (3.2)

Here, \( \mathbb{I}_{x_i \in B_q} \) is an indicator for the event that data element \( x_i \) and the query \( q \) are in the same bucket. It should be noted that

\[
Pr(\mathbb{I}_{x_i \in B_q} = 1) = p(q, x_i)^K = p_i^K 
\] (3.3)

Note that, \( \mathbb{I}_{x_i \in B_q} \) and \( \mathbb{I}_{x_j \in B_q} \) are correlated. If \( x_i \) and \( x_j \) are "similar" then \( \mathbb{I}_{x_i \in B_q} = 1 \) is likely to imply \( \mathbb{I}_{x_j \in B_q} = 1 \). In other words, high similarity indicates positive
correlation. Due to correlations, we may have both the cases:

\[
\mathbb{E}[\mathbb{I}_{x_i \in B_q}\mathbb{I}_{x_j \in B_q}] \begin{cases} 
\geq p^K_i p^K_j, & \text{(positive correlations)} \\
\leq p^K_i p^K_j, & \text{(negative correlation)}.
\end{cases}
\] (3.4)

Here, \(\mathbb{E}\) is the expectation.

Using the above notations we can show that, for a given query \(q\), \(\hat{S}(q, D)\), computed in Algorithm 1, is an unbiased estimator of \(S(q, D)\) with variance given by:

**Theorem 1**

\[
\mathbb{E}[\hat{S}(q, D)] = \sum_{x_i \in D} p^K_i = S(q, D)
\]

\[
\text{Var}(\hat{S}(q, D)) = \frac{1}{L} \left( \sum_{i=1}^{n} p^K_i (1 - p^K_i) \right) + \sum_{i \neq j} \left[ \mathbb{E}[\mathbb{I}_{x_i \in B_q}\mathbb{I}_{x_j \in B_q}] - p^K_i p^K_j \right].
\]

The variance of \(\hat{S}(q, D)\) is dependent on the data distribution. There are two terms in the variance \(\frac{1}{L} \left( \sum_{i=1}^{n} p^K_i (1 - p^K_i) \right)\) and \(\frac{1}{L} \sum_{i \neq j} \left[ \mathbb{E}[\mathbb{I}_{x_i \in B_q}\mathbb{I}_{x_j \in B_q}] - p^K_i p^K_j \right]\).

The terms inside summation is precisely the covariance between \(\mathbb{I}_{x_i \in B_q}\) and \(\mathbb{I}_{x_j \in B_q}\)

\[
\mathbb{E}[\mathbb{I}_{x_i \in B_q}\mathbb{I}_{x_j \in B_q}] - p^K_i p^K_j = \mathbb{E}[\mathbb{I}_{x_i \in B_q}\mathbb{I}_{x_j \in B_q}]
\] (3.5)

\[
- \mathbb{E}[\mathbb{I}_{x_i \in B_q}] \mathbb{E}[\mathbb{I}_{x_j \in B_q}]
\] (3.6)

\[
= \text{Cov}(\mathbb{I}_{x_i \in B_q}, \mathbb{I}_{x_j \in B_q})
\] (3.7)

There are \(n(n-1)\) covariance terms in the second term of variance, \(\frac{1}{L} \sum_{i \neq j} \left[ \mathbb{E}[\mathbb{I}_{x_i \in B_q}\mathbb{I}_{x_j \in B_q}] - \right] - \)
$p^K_i p^K_j$. To see why almost all of them will be negative, let $m$ be the number of elements in the buckets of the query. So only pairs $x_i$ and $x_j$ in the bucket ($O(m^2)$ pairs) of query $H_j(q)$ will contribute $1 - p^K_i p^K_j \geq 0$ to the summation (product of indicators is $1 \iff$ both are 1). Rest all pairs ($O((n - m)^2)$) will contribute negative terms $-p^K_i p^K_j$. Thus, if we choose $K$ large enough then the expected number of elements in the bucket $m$ is quite small. Hence, we can expect the variance to be significantly smaller than $\left(\sum_{i=1}^{n} p^K_i \frac{(1-p^K_i)}{L}\right)$. We observe in our experiments that $K = 15$ is a good recommended constant value.

As noted the variance is dependent on the data distribution. If we have all exact duplicates, then all the covariances are positive. However, for real datasets, for any randomly chosen pair $x_i, x_j$, the covariance $Cov(\mathbb{I}_{x_i \in B_q}, \mathbb{I}_{x_j \in B_q})$ will be almost always be negative.

An alternative way of estimating $S(q, D)$ is to use the random sampling. The idea is to uniformly sample a subset $S \subseteq D$ of size $L$ and report the random sampling estimator $RSE(q, D)$ as:

$$RSE(q, D) = \frac{n}{L} \left[ \sum_{x_i \in S} p^K_i \right]$$  \hspace{1cm} (3.8)

From the theory of random sampling this estimator is also unbiased and has the following variance:
Theorem 2

\[ \mathbb{E}[RSE(q, D)] = \sum_{x_i \in D} p^K_i = S(q, D) \]

\[ \text{Var}(RSE(q, D)) = \sum_{i=1}^{n} p^K_i \left( \left\lceil \frac{n}{L} - 1 \right\rceil p^K_i \right) \]

Both $RSE(q, D)$ and $\widehat{S}(q, D)$ are unbiased. For the same number of samples, the estimator with smaller variance is superior.

We can get some insights from the leading terms $\sum_{i=1}^{n} p^K_i \left( \left\lceil \frac{n}{L} - 1 \right\rceil p^K_i \right)$ and $\frac{1}{L} \left( \sum_{i=1}^{n} p^K_i (1 - p^K_i) \right)$. Generally, for any $i$ and large enough $n$, we always have $\left\lceil \frac{n}{L} - 1 \right\rceil \geq \frac{1}{L}(1 - p^K_i)$. Thus, for large enough $n$,

\[ \text{Var}(RSE(q, D)) > \frac{1}{L} \left( \sum_{i=1}^{n} p^K_i (1 - p^K_i) \right) \]

As argued before for real data we can expect $\frac{1}{L} \left( \sum_{i=1}^{n} p^K_i (1 - p^K_i) \right) > \text{Var}(\widehat{S}(q, D))$. Precise mathematical comparison between the variances of these two estimators is fairly challenging due to data-dependent correlation.

**Empirical Comparison:** As argued, we expect that for real datasets the ACE estimator to be more accurate (less variance) compared to the random sampling estimator. To validate our arguments empirically, we compare these estimators on the three benchmark anomaly detection datasets. These are the same datasets used in the experiment sections (see section 3.6.1 for details). For all the three datasets, we randomly chose 50 queries and estimate their $S(q, D)$ using the two competing estimators. We use $K = 15$ which is the fixed value used in all our experiments.
Figure 3.3: Comparison of ACE estimator with random sampling estimator on three datasets. The x-axis denotes the number to arrays and size of samples for ACE estimator and random sampling estimator respectively. ACE estimator is not only more accurate but also cheaper compared to random sampling estimators from the computational perspective.
We plot the mean square error of the estimates, computed using the actual and the estimated values on three real anomaly detection datasets, in Figure 3.3. We vary the number of samples for random sampling estimator $RSE(q, \mathcal{D})$ and the number of arrays for $S(q, \mathcal{D})$. From the plots, it is clear that on all the three real datasets, as expected from our analysis, our ACE estimator $\hat{S}(q, \mathcal{D})$ consistently outperforms the random sampling estimator $RSE(q, \mathcal{D})$ at the same level of $L$. Note, these estimators are unbiased and hence mean square error value is also the theoretical variance. These experiments indicate that the variance of our ACE estimator is superior for estimating $S(q, \mathcal{D})$ over random sampling.

In addition to providing sharper estimates, in the next section, we show that our ACE algorithm only needs $O(d \log d + KL)$ computations to calculate the score. Here, $d$ is the dimensions of the dataset. For the same number of samples $L$, random sampling estimator requires $O(Ld)$ computations. Given that $K = 15$ is a fixed constant. For high dimensional datasets, we will have $d > K$. Thus, our estimator is not only more accurate but also cheaper compared to random sampling estimators from the computational perspective.

3.4.4 Implementation Details, Running Time, Cache Utilization and Memory

Running Time: From Algorithm 1, it is not difficult to see that for a query $q$, we need to compute $KL$ hashes of the data followed by a simple addition of size $L$. The costliest step is the computations of $KL$ hashes, which for $d$ dimensional data can be accomplished in $O(d \log d + KL)$ computations using advances in fast random projections (Section 2.1). If instead, we are using minwise hashing as the LSH then it
can be done in mere $O(d + KL)$ using fast minwise hashes. However, minwise hashing is limited to binary datasets only.

Note that computing the original score $S(q, D)$ via naive calculation requires $O(nd)$ computations. It further require to store all the data for outlier detection, which for large and high dimensional datasets can be prohibitive.

In all our experiments, we use $K = 15$ and $L = 50$ for all the three datasets (see Section 3.7). Thus, with these small constant values, our scoring time negligible compared to the other algorithms which requires one pass over the full dataset $D$.

In the experiments, we see that even with such minuscule computation, our method provides competitive accuracy while being orders of magnitude faster than 11 state-of-the-art methods.

**Memory:** Since we have $2^K$ counters, it is unlikely that the counters will get too many hits. To save memory by a factor of two, we can use short integers (16 bits) instead of integer counters. The total amount of memory required by $L$ counter arrays is $2^K$ bytes each if we use short counters. The total space needed for the arrays is $L \times 2^K \times 2$ bytes. For $K = 15$ and $L = 50$, the total space required by the ACE algorithm is around $3.2MB$. In addition, we need to compute $KL = 750$ hashes, which requires storing 750 random seeds (integers) from which we can generate hashes on the fly. 750 integers require negligible space compared to $3.2MB$. In the worst case, even if we decide to store the full random projections, we only need $750 \times d \times 8$ bytes (approx $6d$ kilobytes).

**L3 Cache Utilizations:** For all of our experiments, the total memory requirement of the ACE algorithm is $\leq 4MB$ for all the datasets. Our query data structure, the arrays, can easily fit into L3 cache of any modern processor, where the memory
access can be anywhere from 2-10x faster than the main memory (DRAM) access. Detecting anomaly requires scoring which only needs reading count from the arrays. Due to all these unique favorable properties, our algorithm is orders of magnitude faster than the fastest available packages for unsupervised anomaly detection.

**Dynamic Updates**

One of the appealing features of the ACE algorithm is that data can be dynamically updated. It is straightforward to increment the counters if we decide to add any data \( x \). However, we will lose all the data information. We only store a set of count arrays, so it is not clear how we update the global mean \( \mu \) of counts. Updating \( \mu \) is an important part of Algorithm 1. Note that the updated mean, \( \mu' \), should be the average of all the estimated score of all the data in \( D' = D + x \).

It turns out that we can exactly compute the new value of \( \mu' \) from the existing count arrays. To simplify, let us convert old mean \( \mu \) to sum by multiplying it by the size of dataset \( n = |D| \). It is easy to keep track of the sum

\[
  n\mu = \sum_{x_i \in D} \frac{1}{L} \sum_{j=1}^{L} A_j[H_j(x_i)].
\]

Observe that, if the new \( x \) goes to location \( H_j(x) \) in array \( A_j \) for any \( j \). The count of location \( H_j \) will be increment by 1. This will also lead to an increment in the scores of all the elements which maps to \( H_j(x) \) in \( j^{th} \) array by exactly \( \frac{1}{L} \). Since we already know the count value of \( A_j[H_j(x_i)] \), the total increment to the sum would be \( \frac{A_j[H_j(x_i)]}{L} \). In addition, the new data \( x \) will add an extra \( \frac{A_j[H_j(x_i)]+1}{L} \) for its own count. Thus, we can precisely compute the increment in the sum. The new mean \( \mu' \), for an
addition of data $x$, can be computed as

$$
\mu' = \frac{1}{n+1}\left( n\mu + \sum_{j=1}^{L} \frac{2A_j[H_j(x)] + 1}{L} \right). 
$$

(3.9)

### 3.5 Discussions: Privacy Preserving Anomaly Detection

Privacy is becoming one of the sought after directions in data mining and machine learning. Privacy preserving anomaly detection is of broad interest in the big-data and IoT (Internet of Things) community [VC04]. In many setting, we do want to detect anomalies in the data. However, it also desirable that the attribute information remains private and secure. It turns out that our proposed ACE algorithm has ideal properties for privacy preserving anomaly detection.

ACE does not require storing any data attributes, and the complete algorithm works only over aggregated counts generated from hashed data. If the hashes are not invertible, then the algorithm is safe. We can exploit advances in the secure computation to design protocols which hide the hashing mechanism [Gol98].

Obtaining differential privacy [BSS11; Mac+08] with ACE is quite appealing and neat. Since ACE algorithm relies on random projections to compute hashes, instead of original data, we can make ACE algorithm differentially private by adding only Gaussian noise instead of heavy-tailed Laplacian noise. [Ken+12] shows a way to release user information in a privacy-preserving way for near-neighbor search. The paper showed that adding Gaussian noise $N(0, \sigma^2)$ after the random projection preserves differential privacy. Any function of differentially private object it also differentially private. Thus, to compute a private variant of SRP (Signed random projection), we
used the sign of the differentially private random projections (generated by adding Gaussian noise to usual projection) as suggested in [Ken+12].

The final algorithm is very simple. The data is never revealed to anyone. At the source itself, the sign of differentially private random projections of data is used instead of usual SRP. All other process remains the same. Now since, we are only perturbing our algorithm with Gaussian noise, instead of Laplacian, we can expect a minimal loss in utility (or change in output).

Note, that privacy is significantly harder with other state-of-the-art anomaly detection algorithms that store the actual data or even samples. Making such algorithms private requires perturbing the system with heavy-tailed Laplacian noise, which can significantly hurt the outcome of the algorithm.

3.6 Experimental Evaluations

3.6.1 Datasets

We choose three real-world benchmark datasets for anomaly detection: 1) Statlog Shuttle, 2) Object Images (ALOI), and 3) KDD-Cup99 HTTP. These datasets are labeled and hence can be used for quantifying the effectiveness of anomaly detection measure. These three datasets also cover a broad spectrum of applications of unsupervised anomaly detection.

The first dataset we use is the shuttle dataset *. This dataset describes radiator positions in a NASA space shuttle with 9 attributes. It was designed for supervised anomaly detection. In the original datasets, about 20% of the data regarded as

*https://archive.ics.uci.edu/ml/datasets/Statlog+(Shuttle)
anomaly. By following the preprocessing steps in [AZL06; Rei+08], we reduce the number of anomalies by selecting the class 1 as normal and apply a stratified sampling for the classes 2, 3, 5, 6 and 7. The entire datasets contains 34,987 instances with 879 anomalies.

The second dataset is Object Images (ALOI) datasets†. The aloi dataset is derived from the “Amsterdam Library of Object Images” collection [GBS05]. It contains about 110 images of 1000 small objects taken under different light conditions and viewing angles. From the original images, a 27 dimensional feature vector was extracted using HSB color histograms [Sch+12]. Some objects were chosen as anomalies, and the data was down-sampled such that the resulting dataset contains 50,000 instances including 1508 anomalies.

The third dataset is KDD-Cup99 HTTP. ‡ KDD-Cup99 HTTP dataset [LL05] is the largest benchmark for unsupervised anomaly detection evaluation. It contains simulated normal and attack traffic on an IP level in a computer network environment in order to test intrusion detection systems. Following the preprocessing steps in [LL05; Car10], we use HTTP traffic only and also limit DoS traffic from the dataset. Furthermore, the features of “protocol" and “port" information were removed. The remaining binary categorical features represented as 0 or 1 resulting in a total of 36 dimensions. The dataset contains 596,853 instances with 1055 labeled anomalies.

The statistics of these datasets are shown in Table. 3.1.
Table 3.1: The statistics of the three datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Instances</th>
<th>Outliers</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statlog Shuttle</td>
<td>34,987</td>
<td>879</td>
<td>9</td>
</tr>
<tr>
<td>Object Images (ALOI)</td>
<td>50,000</td>
<td>1508</td>
<td>27</td>
</tr>
<tr>
<td>KDD-Cup99</td>
<td>596,853</td>
<td>1055</td>
<td>36</td>
</tr>
</tbody>
</table>

3.6.2 Baselines

We use 11 different baselines methodologies to compare with ACE. These methodologies cover the whole spectrum of unsupervised anomaly detection techniques with all sorts of variations developed over the years. Our baselines cover scoring mechanisms based on simple to sophisticated strategies which include near-neighbor, kernel density estimation, graph connectedness, etc. First, we briefly describe all the competing algorithms:

1. **ACE**: This is the proposed Algorithm 1. We implement this algorithm in C++.

2. **LOF (Local Outlier Factor) [Bre+00]**: This is the most popular and the state-of-the-art unsupervised anomaly detection algorithm which use density-based local outlier factors in a database. LOF uses a score based on the difference between the local density of a point with that of its near-neighbors as the outlier score. LOF-ELKI is the fastest implementation of this algorithm.

3. **FastVOA (Fast Variance of Angles) [PP12]**: FastVOA is our benchmark sampling based methodology. FastVOA is a randomized algorithm which uses

---

random sampling and random projections to estimate the variance of angle outlier measure. We use the C++ implementation§ provided by the authors.

4. **kNN (KNNOutlier) [RRS00]**: This is the simplest method that uses the distance of an object to its $k$ nearest neighbor. We use the optimized implementation provided by the ELKI package.

5. **KNNW (KNNWeightOutlier) [AP05]**: This is an improved version of the kNN algorithm where instead of the distance, we use the sum (accumulated) distance of a point to its $k$ nearest neighbors. Sum reduces the variance of the scores and it more stable. We again use the ELKI implementation.

6. **LoOP (Local Outlier probability) [Kri+09]**: This is a more advanced version of LOF. LoOP uses distance/density based algorithm similar to LOF to detect outliers, but with statistical methods to achieve better result stability. We use the implementation provided by ELKI.

7. **LDOF (Local Distance based Outlier Factor) [ZHJ09]**: LDOF defines outliersness as the ratio of the average scores between the target point to all the other points in the kNN set, and of the sum all the pair-wise distance over all instances in the kNN set. Again we use the ELKI package.

8. **ODIN (Outlier Detection using Indegree Number) [HKF04]**: ODIN defines outliersness as a low number of in-adjacent edges in the kNN graph. For more detail about this method, please refer [HKF04]. ODIN is available in ELKI.

§http://www.itu.dk/people/ndap/FastVOA.zip
9. **LDF (Local density factor) [LLP07]**: LDF replaces LOF’s density function by a variance-width Kernel density estimation (KDE). In the KDE, the original (Euclidean) distance is replaced with the reachability distance of LOF. For more details of LDF, please refer [LLP07]. LDF is available in ELKI.

10. **KDEOS (Kernel Density Estimation Outlier Score)**: KDEOS also uses Kernel Density Estimation (KDE) in the LOF framework. KDEOS keeps the mathematical kernel density estimation intact for comparison with neighbor densities. The KDE densities are standardized per point as z-scores with respect to the KDE densities of the kNN set and averaged over different neighborhood sizes \( k_{min}... k_{max} \). For more details of KDEOS, please refer to [SZK14].

11. **COF (Connectivity-based Outlier Factor) [Tan+02]**: COF modifies the density estimation of LOF to account for the “connectedness” of a neighborhood via a minimum spanning tree (MST) rooted at the point under study. For details of COF please refer [Tan+02].

12. **INFLO (Influenced Outlierness) [Jin+06]**: INFLO compares the local model of LOF with the same density estimate applied to the reference set of the union of kNN and RkNN sets. INFLO is thus an example of a local outlier detection strategy for which different definitions of the neighborhood are used for the context set and reference set. For more details of INFLO, please refer [Jin+06].

We use the highly optimized recent ELKI (Environment for Developing KDD-
Applications Supported by Index-Structures) package\(^4\) which is the most advanced set of anomaly detection algorithms noted for its efficient Java implementations. 10 of our baselines methodologies are implemented in this package. For FastVOA, a state-of-the-art randomized algorithm for variance of angle computation, we use the C++ package provided by the authors.

It should be noted that ACE and FastVOA are implemented in C++, while ELKI is a java package. A direct wall clock comparison is not fair. However, given the simplicity of our algorithm (Algorithm 1) which only requires simple hashing, use of primitive arrays, and simple summations. We do need any complex object other than arrays of short integers (primitives only). All other operations are primitive multiplications and summations. Thus, we expect that the difference between Java and C++ implementation would not be any significant for ACE. Furthermore, our results indicate a very significant speedup which cannot be explained by the difference in platforms.

**Parameter Settings:** Almost all of our baseline algorithms need hyper-parameters. We use most of the default settings of the parameters as implemented. For the baseline algorithms, the *ELKI* package has the recommended settings of parameters for these benchmark datasets. To avoid complications, we directly use those recommended settings. For the sake of reproducibility, we provide the precise recommended settings of the parameters for different methods and datasets used in the work in Table 3.2. It should be noted that for ACE we use the fixed value of \(K = 15\) and \(L = 50\) for all the datasets. ACE does not need the near neighbor parameter \(k\) (small). Variations in parameter \(K\) and \(L\) are discussed in Section 3.7.

\(^4\)https://elki-project.github.io/
System and Platform Details: Experiments were conducted on a 3.50 GHz core Xeon Windows platform with 16GB of RAM. We use g++ (version 5.4.0) as the C++ compiler for ACE and fastVOA. For ELKI package, we use OpenJDK 64bits version 1.8.0.

3.6.3 Methodology and Results

All of these 12 algorithms associate a score with every element in the data. After association, a significantly lower score from the mean indicates an anomaly. In order to convert these scores into an anomaly detections algorithm, there are many reasonable strategies. We can rank each candidate, based on scores, and report bottom-$k$ as the anomalies, but such rankings are not realistic. In real-time applications, ranking all the seen records is artificial. A more practical approach is to use a threshold strategy to report anomalies. We compute the mean $\mu$ and the standard deviations $\sigma$ of the scores on the dataset of interest and report any element with the associated score less than $\mu - \sigma$ as an anomaly.

With the above introduced anomaly detection strategies, we run all the 12 algorithms on the three datasets. We report seven different numbers separately for each of the three datasets: (1) Number of outliers reported; (2) Number of outliers correctly reported; (3) Number of Outliers missed; (4) F1-score; (5) Ranking of F1-score; (6) the CPU execution time for the different methods, and (7) Relative speed with ACE. The CPU executing time is the end to end time of the complete run of the algorithm, which includes data reading, preprocessing (if any), scoring every data instance, and reporting outliers. Relative speedup reports the ratio of the time required by a given algorithm to the time required by ACE algorithm. The results of each datasets are
shown in Table 3.5.

3.6.4 Accuracy Comparison.

We report the F1-scores [AY01] of each method. F1-score is a widely used method for evaluating the performance of anomaly detection methods, for the detailed definition of F1-score please refer [AY01]. Based on the F1-scores of each method, we rank the different methods. From the results, we can see LOF seems to be consistently more accurate than others. ACE is ranked consistently among the top-5 ranked methods on all the datasets. The number of anomalies reported correctly (true positives) with ACE is similar to other algorithms. ACE, however, seems to report slightly more anomalies (high false positives) than other algorithms. This is not a major concern though. Few extra false positives are easy to deal with because we can always further filter them using a more sophisticated algorithm, so long as they are small. Overall, our proposed new scoring scheme $S(q, D)$ and the corresponding estimator performs very competitively, in terms of accuracy, in comparison with many successful algorithms.

3.6.5 Running Time Comparison.

The most exciting part is the computational savings with ACE. From the result, we observe that ACE is significantly faster than any other alternatives irrespective of the choice of dataset. ACE algorithm is at least around 15x, 45x and 60x faster than the best competitor on Statlog Shuttle, Object Images (ALOI), and KDD-Cup99 HTTP datasets respectively. Most of the algorithms, based on near-neighbors except FastVOA, have similar speeds. This could be because almost all of them
requires computation of the order of the data. FastVOA is consistently very slow, which we suspect is because the estimators used in FastVOA is computationally very expensive. FastVOA estimators require multiple sorting and frequently computing costly medians. See [PP12] for details. ACE is around 150-300x faster than FastVOA.

3.6.6 Memory Analysis.

The results are even more exciting if we start considering the memory requirements. With $K = 15$ and $L = 50$, our methodology requires less than $4MB$ of operating memory for the complete run of the algorithm. Since we use the same $K$ and $L$ across all datasets, this $4MB$ requirement is unaltered. We never keep any data in the memory. On the other hand, all other methods except FastVOA require storing complete data in the memory. In our case, the KDD-Cup99 HTTP dataset itself is around 165MB to store. Although KDD-Cup99 HTTP dataset is the largest labeled benchmark, it is still tiny from big-data perspective. The disruptive performance of ACE is not surprising given the simplicity of the process. However, as argued, the process is a statistically sound procedure for estimating the proposed score $S(q, D)$.

3.7 Discussion: Effects of $K$ and $L$

Parameters $K$ and $L$ determine the memory and also the running time of the ACE algorithm. Note $L$ is also the number of independent samples used for averages. Therefore, a reasonably large $L$ is good enough, after which increasing $L$ does not give significant accuracy but hurts the performance. $K$ cannot be too small because locations in arrays should distinguish anomalies with everything else. However, too large $K$ is not needed either. Ideally, if $K$ is log $n$, then under random assignments
all data will go to the single bucket. Beyond this $K$, the performance is lost for no gain in accuracy.

To stress test, we ran ACE for with different values of $K = \{2, 5, ..., 20\}$ and $L = \{10, 20, ..., 100\}$. For the Image dataset, the minimum reasonable result appears at $K=8$, $L=30$. For the shuttle dataset, $K=11$, $L=10$ is fine, and for the KDDCUP dataset, the minimum fair result appears at $K=9$, $L=30$. These parameters give similar results as shown with fixed $K = 15$ and $L = 50$. With these parameters, the ACE took mere 0.5, 0.2 and 11.5 seconds on the Image, shuttle and KDDCUP dataset for a negligible loss in accuracy compared to what is shown in Table 3.5. The results degrade if we decrease $K$ and $L$ beyond these numbers. Increasing $K$ and $L$ values, significantly beyond $K = 15$ and $L = 50$, does not increase the accuracies significantly but, as expected, hurts the performance.

### 3.8 Discussion

Statistical measures for popular learning and data mining problems, such as anomaly detection, were designed without taking into account the computational complexity of the estimation process. When faced with current big-data challenges, most of these estimation process fail to address tight resources constraints. in this thesis, we showed that for the problem of unsupervised anomaly detection, we could leverage advances in probabilistic indexing and redesign a significantly efficient statistical measure.

We proposed ACE algorithm, for unsupervised anomaly detection, which is 60-300x faster than existing approaches with competing accuracy. Our algorithm requires mere $4MB$ of memory which can utilize L3 caches of modern processors leading to
fast-lookups. We believe ACE will replace existing unsupervised anomaly detection algorithms deployed in resource-frugal environments.
Table 3.2: Comparison Algorithms and Their Parameter Values Recommended for These Benchmark Datasets.

<table>
<thead>
<tr>
<th>Method</th>
<th>Shuttle</th>
<th>Image Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE</td>
<td>$k = 5$, $L = 50$</td>
<td>$k = 5$, $L = 50$</td>
</tr>
<tr>
<td>LOF</td>
<td>$k = 5$</td>
<td>$k = 5$</td>
</tr>
<tr>
<td>kNN</td>
<td>$k = 5$</td>
<td>$k = 5$</td>
</tr>
<tr>
<td>kNNW</td>
<td>$k = 5$, $k_r = k_c = 5$</td>
<td>$k = 5$, $k_r = k_c = 5$</td>
</tr>
<tr>
<td>LLOP</td>
<td>$k = 5$, $k_r = k_c = 5$, $\lambda = 0.2$</td>
<td>$k = 5$, $k_r = k_c = 5$, $\lambda = 0.2$</td>
</tr>
<tr>
<td>LDOF</td>
<td>$k = 5$</td>
<td>$k = 5$</td>
</tr>
<tr>
<td>ODIN</td>
<td>$k = 5$</td>
<td>$k = 5$</td>
</tr>
<tr>
<td>KDEOS</td>
<td>$k = 5$</td>
<td>$k = 5$</td>
</tr>
<tr>
<td>COF</td>
<td>$k = 5$, $h = 1$, $c = 0.1$, $</td>
<td>S_2</td>
</tr>
<tr>
<td>LDF</td>
<td>$k = 5$</td>
<td>$k = 5$</td>
</tr>
<tr>
<td>INFLO</td>
<td>$k = 5$, $</td>
<td>S_2</td>
</tr>
<tr>
<td>FastVOA</td>
<td>$k = 5$, $</td>
<td>S_2</td>
</tr>
<tr>
<td>Method</td>
<td>Reported</td>
<td>Correct</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>ACE</td>
<td>6763</td>
<td>273</td>
</tr>
<tr>
<td>LOF</td>
<td>4356</td>
<td>381</td>
</tr>
<tr>
<td>kNN</td>
<td>4897</td>
<td>493</td>
</tr>
<tr>
<td>kNNW</td>
<td>5264</td>
<td>610</td>
</tr>
<tr>
<td>LoOP</td>
<td>6145</td>
<td>201</td>
</tr>
<tr>
<td>LDOF</td>
<td>6433</td>
<td>330</td>
</tr>
<tr>
<td>ODIN</td>
<td>9775</td>
<td>375</td>
</tr>
<tr>
<td>KDEOS</td>
<td>12630</td>
<td>314</td>
</tr>
<tr>
<td>COF</td>
<td>9133</td>
<td>280</td>
</tr>
<tr>
<td>LDF</td>
<td>9809</td>
<td>375</td>
</tr>
<tr>
<td>INFLO</td>
<td>4488</td>
<td>183</td>
</tr>
<tr>
<td>FastVOA</td>
<td>8532</td>
<td>271</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 3.4: Comparison Results on Image Object Dataset

<table>
<thead>
<tr>
<th>Method</th>
<th>Reported</th>
<th>Correct</th>
<th>Missed</th>
<th>F1-score</th>
<th>F1-Rank</th>
<th>Time (s)</th>
<th>Speed-up with ACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE</td>
<td>7216</td>
<td>340</td>
<td>1168</td>
<td>0.078</td>
<td>5</td>
<td>1.26s</td>
<td>1x</td>
</tr>
<tr>
<td>LOF</td>
<td>4476</td>
<td>519</td>
<td>989</td>
<td>0.1735</td>
<td>1</td>
<td>72.31s</td>
<td>57.4x</td>
</tr>
<tr>
<td>kNN</td>
<td>5428</td>
<td>447</td>
<td>1061</td>
<td>0.1289</td>
<td>2</td>
<td>63.27s</td>
<td>50.2x</td>
</tr>
<tr>
<td>kNNW</td>
<td>5558</td>
<td>329</td>
<td>1508</td>
<td>0.089</td>
<td>4</td>
<td>89.96s</td>
<td>71.4x</td>
</tr>
<tr>
<td>LoOP</td>
<td>5121</td>
<td>253</td>
<td>1179</td>
<td>0.077</td>
<td>6</td>
<td>59.97s</td>
<td>47.6x</td>
</tr>
<tr>
<td>LDOF</td>
<td>7501</td>
<td>470</td>
<td>1038</td>
<td>0.1043</td>
<td>3</td>
<td>60.39s</td>
<td>47.9x</td>
</tr>
<tr>
<td>ODIN</td>
<td>10110</td>
<td>162</td>
<td>1346</td>
<td>0.028</td>
<td>12</td>
<td>72.69s</td>
<td>57.6x</td>
</tr>
<tr>
<td>KDEOS</td>
<td>9515</td>
<td>404</td>
<td>1104</td>
<td>0.073</td>
<td>7</td>
<td>55.89s</td>
<td>44.36x</td>
</tr>
<tr>
<td>COF</td>
<td>8746</td>
<td>284</td>
<td>1224</td>
<td>0.055</td>
<td>11</td>
<td>81.74s</td>
<td>64.9x</td>
</tr>
<tr>
<td>LDF</td>
<td>9133</td>
<td>301</td>
<td>1207</td>
<td>0.056</td>
<td>10</td>
<td>60.51s</td>
<td>48.0x</td>
</tr>
<tr>
<td>INFLO</td>
<td>10328</td>
<td>420</td>
<td>1088</td>
<td>0.071</td>
<td>8</td>
<td>72.13s</td>
<td>57.2x</td>
</tr>
<tr>
<td>FastVOA</td>
<td>8931</td>
<td>319</td>
<td>1189</td>
<td>0.061</td>
<td>9</td>
<td>291.10s</td>
<td>231.0x</td>
</tr>
</tbody>
</table>
Table 3.5: Comparison Results on KDD-CUP 99 Dataset

<table>
<thead>
<tr>
<th>Method</th>
<th>Reported</th>
<th>Correct</th>
<th>Missed</th>
<th>F1-score</th>
<th>F1-Rank</th>
<th>Time (s)</th>
<th>Speed-up with ACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE</td>
<td>15160</td>
<td>406</td>
<td>649</td>
<td>0.051</td>
<td>5</td>
<td>23.33s</td>
<td>1x</td>
</tr>
<tr>
<td>LOF</td>
<td>13260</td>
<td>523</td>
<td>532</td>
<td>0.073</td>
<td>1</td>
<td>1813.63s</td>
<td>77.7x</td>
</tr>
<tr>
<td>kNN</td>
<td>15432</td>
<td>365</td>
<td>690</td>
<td>0.044</td>
<td>7</td>
<td>1483.54s</td>
<td>63.5x</td>
</tr>
<tr>
<td>kNNW</td>
<td>14328</td>
<td>460</td>
<td>595</td>
<td>0.059</td>
<td>2</td>
<td>2125.43s</td>
<td>91.1x</td>
</tr>
<tr>
<td>LoOP</td>
<td>16578</td>
<td>396</td>
<td>659</td>
<td>0.045</td>
<td>6</td>
<td>1594.54s</td>
<td>68.3x</td>
</tr>
<tr>
<td>LDOF</td>
<td>16579</td>
<td>496</td>
<td>559</td>
<td>0.056</td>
<td>3</td>
<td>1674.43s</td>
<td>71.7x</td>
</tr>
<tr>
<td>ODIN</td>
<td>18054</td>
<td>365</td>
<td>690</td>
<td>0.038</td>
<td>10</td>
<td>1918.34s</td>
<td>82.2x</td>
</tr>
<tr>
<td>KDEOS</td>
<td>21095</td>
<td>469</td>
<td>586</td>
<td>0.042</td>
<td>8</td>
<td>1428.32s</td>
<td>61.2x</td>
</tr>
<tr>
<td>COF</td>
<td>20658</td>
<td>584</td>
<td>471</td>
<td>0.054</td>
<td>4</td>
<td>2043.43s</td>
<td>87.5x</td>
</tr>
<tr>
<td>LDF</td>
<td>19574</td>
<td>368</td>
<td>687</td>
<td>0.036</td>
<td>11</td>
<td>1485.85s</td>
<td>63.7x</td>
</tr>
<tr>
<td>INFLO</td>
<td>25704</td>
<td>565</td>
<td>490</td>
<td>0.042</td>
<td>9</td>
<td>1684.47s</td>
<td>72.2x</td>
</tr>
<tr>
<td>FastVOA</td>
<td>29316</td>
<td>354</td>
<td>701</td>
<td>0.023</td>
<td>12</td>
<td>3510.26s</td>
<td>150.4x</td>
</tr>
</tbody>
</table>
In this chapter, we introduce a second work of using locality sensitive hashing for scaling up the bayesian inference.

4.1 Introduction and Background

Bayesian mixture models are of great interest due to their flexibility in fitting a countably infinite number of components which can grow with the data [MYB04]. The growth of model complexity with the data is also in agreement with modern progress in machine learning over massive datasets. However, the appealing properties of Bayesian modeling come with hard computational challenges. Even with simple mixture models, the mathematical problems associated with training and inference are intractable. As a result, recent research focuses on developing tractable computational techniques. In particular, the use of Markov chain Monte Carlo (MCMC) methods, to sample from the posterior distribution [And+03; Nas07; WB12] is widely prevalent. The practical utility of these methods is illustrated in several applications including haplotype reconstruction [EGT03], nucleotide substitutions [HR01], gene expression [SA15], etc.

Metropolis-Hastings (MH) [And+03] is a favorite class of MCMC methods, which includes several state-of-the-art algorithms that have proven useful in practice. MH
is associated with a transition kernel which provides a proposal step. This step is followed by appropriate stochastic acceptance process that ensures detailed balance. A notable example of MH is the Split-Merge MCMC algorithm [JN04; WR15] which is particularly useful for problems where an MCMC state can be thought of as consisting of a number of components (or clusters). Here as the name suggests, the proposal step comprises of either a split or a merge. A split move partitions an existing mixture component (or cluster) into two, while a merge move combines two mixture components into one.

In the seminal work of [JN04], split-merge MCMC procedure was proposed. To illustrate the process, the authors first introduce a random split-merge MCMC, where the split and the merge decision were taken uniformly at random. However, it was also pointed out, in the same work, that due to the random nature of the proposal it was unlikely to lead to a new state $x'$ with higher likelihood $\mathcal{L}(x')$ leading to low acceptance. To mitigate the slow progress, the authors then propose the restricted Gibbs split-merge (RGSM). In RGSM, the idea was to use restricted Gibbs sampling to generate proposals with a higher likelihood of acceptance, instead of a random proposal. Thus, a less number of MCMC iterations were sufficient for convergence due to fewer rejections. However, the cost of restricted Gibbs is very high. As a result, even though the iterations are less, each iteration is costly making the overall algorithm slow, especially for large datasets. Our experiments confirm this slow convergence of RGSM.

An essential and surprising observation about space asymmetry with smart proposals in split-merge MCMC was made in [WR15]. The authors show the necessity to mix smart and dumb (random) proposals for faster progress. They proposed a
Smart-Dumb/Dumb-Smart Algorithm (SDDS) as an alternative to RGSM. Instead of relying on Gibbs sampling, the SDDS algorithm uses the likelihood of the model itself as a guiding strategy for smart proposals. In other words, the SDDS method evaluates a large number of possible proposals $x'$ based on the likelihood of each $x'$ and choose the best ones. This strategy, as expected, ensures a higher chance of improving the state $x$ with every proposal. However, from a computational perspective, it is not difficult to see that smart proposal $x'$ obtained after evaluation of a large number of proposal states, based on the likelihood, is equivalent to evaluating all these states for acceptance/rejection as part of MH [WR15]. As a result, the reduction in the number of iteration is not helpful in obtaining an efficient algorithm. Our experiments show that SDDS also has poor convergence.

Unfortunately, most MCMC methodologies ignore the tradeoff between the number of iteration and computations associated with each iteration. They instead only focus on reducing the number of rejections, which is often achieved by informative proposals with increased per iteration cost. In this thesis, we are interested in efficient split-merge MCMC algorithm which leads to overall fast convergence. Thus, reducing both is the aim of this work.

Parallelization is Complementary: Due to the significance of the problem there are several works which try to scale up MCMC by using parallelism. Parallelism is often achieved by running parallel MCMC chains on subsets of data and later merging them [CF13]. Since our proposal reduces the overall cost of split-merge MCMC algorithm in general, it will reduce the cost of each of the parallel chains thereby increasing the effectiveness of these parallelisms on MCMC. Thus, existing advances in parallelizing MCMC is complementary to our proposal.
**Our Contributions:** In this chapter, we show that it is possible to construct informative proposals without sacrificing the per-iteration cost. We leverage a simple observation that while designing proposals we can favor configurations where entities similar are likely to be in the same component. We use standard notions of vector similarity such as cosine or Weighted jaccard. To perform such sampling efficiently, we capitalize on the recent advances in LSH sampler [LS18; SS17a; CS17a] that can perform adaptive sampling based on similarity. This forms our first proposal.

Our first proposal leads to around 3x improvements over state-of-the-art methods. However, with similarity driven sampling, computing the Metropolis-Hastings (MH) ratio requires quadratic cost in the size of the cluster being split or merged. This is because while computing the state transition probability, we need to evaluate all possible ways that can lead to the desired split configuration. All these configurations have different probabilities due to similarity-based adaptive sampling and hence the probability computation is expensive. It appears at first that this cost is unavoidable. Surprisingly, it turns out that there is a rare sweet spot. With Weighted MinHash, we can design a split-merge proposal where the total cost of MH update is only linear in the size of the cluster being split or merged. The possibility is unique to MinHash due to its $k$-way generalized collision probability [SL13]. Our proposal and novel extension of MinHash collision probability could be of independent interest in itself.

Overall, our proposed algorithms obtain a sweet tradeoff between the number of iteration and computational cost per iteration. As a result, we reduce the overall convergence in time, not just in iterations. On two large public datasets, our proposal MinHash Split-Merge (MinSM) significantly outperforms other state-of-the-art split-merge MCMC algorithms in convergence speed as measured on wall clock time on
the same machine. Our proposed algorithm is around 6x faster than the second best baseline on synthetic datasets as well as realworld datasets without loss in accuracy.

4.2 Literature Review

4.2.1 Weighted (or Generalized) MinHash

Minwise hashing [LRU14] is the LSH for resemblance similarity. The minwise hashing family applies a random permutation \( \pi : \Omega \rightarrow \Omega \), on the given set \( W \), and stores only the minimum value after the permutation mapping. Formally MinHash is defined as \( h_{\pi}^{\min}(W) = \min(\pi(W)) \). Given sets \( W_1 \) and \( W_2 \), it can be shown by elementary probability argument that:

\[
\Pr(h(x) = h(y)) = \frac{\sum_{i=1}^{D} \min\{x_i, y_i\}}{\sum_{i=1}^{D} \max\{x_i, y_i\}},
\]

Weighted Minwise Hashing is a known LSH for the Weighted Jaccard similarity [LRU14]. Given two positive vectors \( x, y \in \mathbb{R}^D \), \( x, y > 0 \), the (generalized) Weighted Jaccard similarity is defined as \( \mathbb{J}(x, y) = \frac{\sum_{i=1}^{D} \min\{x_i, y_i\}}{\sum_{i=1}^{D} \max\{x_i, y_i\}} \), where \( \mathbb{J}(x, y) \) is a frequently used measure for comparing web-documents [LRU14], histograms (specially images), gene sequences, etc.

Weighted Minwise Hashing (WMH) (or Minwise Sampling) generates randomized hash (or fingerprint) \( h(x) \), of the given data vector \( x \geq 0 \), such that for any pair of vectors \( x \) and \( y \), the probability of hash collision (or agreement of hash values) is given by \( \Pr(h(x) = h(y)) = \frac{\sum_{i=1}^{D} \min\{x_i, y_i\}}{\sum_{i=1}^{D} \max\{x_i, y_i\}} \).

A unique property of Minwise Hashing is that there is a natural extension of \( k \)-way collision [SL13]. In particular, given vectors \( x^{(1)}, x^{(2)}, ..., x^{(s)} \), the simultaneous
collision probability is given by:

\[
Pr(h(x^{(1)}) = h(x^{(2)}) = \ldots = h(x^{(s)})) = \frac{\sum_D \min\{x_{j}^{(1)}, x_{j}^{(2)}, \ldots, x_{j}^{(s)}\}}{\sum_D \max\{x_{j}^{(1)}, x_{j}^{(2)}, \ldots, x_{j}^{(s)}\}}
\]  
(4.1)

Minwise hashing can be extended to negative elements using simple feature transforms [Li17], which essentially doubles the dimensions to 2D. In this thesis, MinHash and Weighted MinHash denote the same thing.

4.2.2 Split-Merge MCMC

Split-Merge MCMC [HFS12] is useful for dealing with the tasks such as clustering or topic modeling where the number of clusters or components are not known in advance. Split-Merge MCMC is a Metropolis-Hastings algorithm with two main transitions: Split and Merge. During a split, a cluster is partitioned into two components. On the contrary, a merge takes two components and makes them to one.

During the MCMC inference process, split and merge moves simultaneously change the number of components and change the assignments of entities to different components. [JN04] proposes the first non-trivial Restricted Gibbs Split-Merge (RGSM) algorithm, which was later utilized for efficient topic modeling over large datasets in [WB12].

In [WR15], the authors presented a surprising argument about information asymmetry. It was shown that both informative split and merge leads to poor acceptance ratio. The author proposed a combination of the smart split with dumb (random) merge and dumb split with smart merge as a remedy. The algorithm was named
as Smart-Dumb/Dumb-Smart Split Merge algorithm (SDDS), which was superior to RGSM. To obtain non-trivial smart split (or merge), the authors propose to evaluate a large number of dumb proposals based on the likelihood and select the best. This search process made the proposal very expensive. It is not difficult to see that finding a smart split is computationally not very different from running a chain with several sequences of dumb (random) splits [WR15].

4.3 LSS based Split-Merge MCMC

**Utilizing Similarity Information:** in this chapter, we make an argument that similarity information, such as cosine similarity, between different entities is almost always available. For example, in the clustering task, the vector representation of the data is usually easy to get for computing the likelihood. Even in an application where we deal with complex entities such as trees, it is not uncommon to have approximate embeddings [BWG10].

It is natural to believe that similar entities, in terms of cosine similarity or Jaccard distance, of the underlying vector representation, are more likely to go to the same cluster than non-similar ones. Thus, designing proposals which favor similar entities in the same cluster and dissimilar entities in different clusters is more likely to lead to acceptance than random proposals.

However, the problem is far from being solved. Any similarity based sampling requires computing all pairwise similarity as a prerequisite, which is a quadratic operation $O(n^2)$. Quadratic operations are near-infeasible for large datasets. One critical observation is that with the modern view of LSH as samplers, described
preview section, we can get around this quadratic cost and design cheaper non-trivial proposals.

4.3.1 Naive LSS based Proposal Design

This section discusses how LSH can be used for efficient similarity sampling which will lead to an informative proposal. In addition, we also want the cost of computing the transition probabilities $q(x'|x)$, which is an important component of the acceptance ratio $\alpha(x'|x)$ [JN04], to be small. Here, $x$ denotes the state before split/merge, and $x'$ denote the state after split/merge. For a good proposal design, it is imperative that $q(x'|x)$ is easy to calculate as well as the proposed state $x'$ is informative. Thus, designing the right MCMC proposal process is the key to speed up computation. Following the intuition described before, we introduce our LSS based proposal design in the rest of this section.

We first create the hash tables $T$ for sampling. We use Sign Random Projection as the LSH function, thus our notion of similarity is cosine [G+99]. It is pointed out that, we can also use other LSH functions when the similarity notion is different. We pay a one-time linear cost for this preprocessing. Note, we need significantly less $K$ and $L$ (both has value 10 in our experiments) compared to what is required for near-neighbor queries as we are only sampling. The sampling is informative (better than random) for any values of $K$ and $L$. For the details of analysis on $K$ and $L$, please refer [SS17a].

For our informative proposal, we will need capabilities to do both similarity sampling as well as dissimilarity sampling for merge and split respectively. The similarity sampling is the usual sampling algorithm discussed in previous sections, which en-
sures that given a query $u$, points similar to $u$ (cosine similarity) are more likely to be sampled. Analogously, we also need to sample points that are likely to be dissimilar. With cosine similarity, flipping the sign of the query, i.e., changing $u$ to $-u$ will automatically do dissimilarity sampling.

Inspired from [WR15], we also leverage the information asymmetry and mix smart and dumb moves for better convergence. However, this time our proposals will be efficient. At each iteration of MCMC, we start by choosing randomly between an LSH Smart-split/Dumb-merge or an LSH Smart-merge/Dumb-split operation. These two operations are defined below:

**Naive LSH Smart-split/Dumb-merge**

LSH based split begins by randomly selecting an element $u$ in the dataset. Then, we use LSS (Locality-sensitive Sampler) to sample points likely to be dissimilar to $u$. Thus, we query our data structure $T$ with $-u$ as the query to get another element $v$ which is likely far away from $u$. If $u$ and $v$ belong to the same cluster $C$, we split the cluster. During the split, we create two new clusters $C_u$ and $C_v$. We assign $u$ to $C_u$ and $v$ to $C_v$. For every element in $C$, we randomly assign them to either $C_u$ or $C_v$. Since we ensure that dissimilar points $u$ and $v$ are split, this is an informative or smart split. If we find $u$ and $v$ are already in a different cluster, we do a dumb merge: randomly select two components, and merge these two components into one component.

The most important part is that we can precisely compute the probability of the proposed split move $q(x'|x)$ and the corresponding inverse move probability $q(x|x')$ as follow:
\begin{align*}
q(x'|x) &= \left(\frac{1}{2}\right)^{|Cu|+|Cv|} \\
&= \sum_u \sum_v \left(\frac{1}{n} \left(1 - (1 - Pr(-u, v)K)^L\right) \frac{|C_v \cap S_{-u}|}{|S_{-u}|}\right) \\
&= \frac{\sum_u \sum_v \left(\frac{1}{n} \left(1 - (1 - Pr(-u, v)K)^L\right) \frac{|C_v \cap S_{-u}|}{|S_{-u}|}\right)}{2^{(|Cu|+|Cv|)-2}} \\
q(x|x') &= \frac{2}{M_{x'}(M_{x'} - 1)}.
\end{align*}

In the above, \(n\) is the number of data point. \(S_{-u}\) is the set of data points that returned by querying in \(T\) using \(-u\), and \(|S_{-u}|\) denotes the number of elements in \(S_{-u}\). \(M_{x'}\) denotes the number of clusters in state \(x'\). \(C\) denotes the original component, \(C_u\) and \(C_v\) are the two new components after split with elements \(u\) and \(v\) in them. \(K\) is the number of bits used for hashing, and \(L\) is the number of hash tables probed. \(Pr(-u, v)\) is the collision probability between \(-u\) and \(v\).

For the LSH Smart-split move, \(q(x'|x)\) denotes that: given the state \(x\), what is the probability that we can go from \(x\) to \(x'\). In this setting, state \(x'\) contains components \(C_u\) and \(C_v\), and state \(x\) contains the original component \(C\). So, under our LSH split proposal design, every combination of \(u \in C_u\) and \(v \in C_v\) can lead state \(x\) goes to state \(x'\). So, we sum over all possible combinations of \(u\) and \(v\) in our probability. Once we choose \(u\) and \(v\), then splitting their components \(C\) into \(C_u\) and \(C_v\) has the probability of \(\left(\frac{1}{2}\right)^{|Cu|+|Cv|} \cdot 2^{(|Cu|+|Cv|)-2}\). We multiply them all together and yield the desired expression.
Inside the summation, for each particular \( u \) and \( v \), we have the probability

\[
\left( \frac{1}{n} \left( 1 - (1 - Pr(-u, v)^K)^L \right) \frac{|C_v \cap S-u|}{|S-u|} \right)
\]

to sample them under our proposal. This expression is obtained by combining the fact that \( 1/n \) is the probability of choosing \( u \). \( (1 - (1 - Pr(-u, v)^K)^L) \) is the probability of having \( v \) in the buckets probed. \( \frac{|C_v \cap S-u|}{|S-u|} \) is the probability of getting \( v \in C_v \) by randomly sampling the bucket. Multiply these three expression together will lead to the desire expression:

\[
\left( \frac{1}{n} \left( 1 - (1 - Pr(-u, v)^K)^L \right) \frac{|C_v \cap S-u|}{|S-u|} \right)
\]

The corresponding inverse move (Dump Merge) probability \( q(x|x') \) is as follow:

\[
q(x|x') = \frac{2}{M_{x'}(M_{x'} - 1)}.
\]

The inverse move of the LSH Smart-split move is a dumb move: given the state \( x' \) which contains \( C_u \) and \( C_v \), what is the probability that we can combine \( C_u \) and \( C_v \) to generate \( C \) (state \( x \)). Under dumb merge, we have \( \frac{1}{M_{x'}} \) to choose one of \( C_u \) or \( C_v \), and probability \( \frac{1}{M_{x'}-1} \) to choose another. Then we add a factor of two and multiplier of these two probabilities to obtain the corresponding probability.
Naive LSH Smart-merge/Dumb-split

LSH based Merge begins by randomly selecting an element $u$ in the dataset. Then use LSS to sample from hash tables $T$ to get another element $v$ which is similar with $u$. Then, if the mixture component of $u$ and $v$ are different, then we do merge operation for the corresponding two mixture component. If $u$ and $v$ are in the same components, we do a dumb split: randomly select one cluster, and split this component into two separate components.

We provide the the probability of the merge move $q(x'|x)$ and the corresponding inverse probability $q(x|x')$:

$$q(x'|x) = \sum_u \sum_v \left( \frac{1}{n} \left( 1 - (1 - Pr(u, v)^K)^L \right) \frac{|C_v \cap S_u|}{|S_u|} \right),$$

$$q(x|x') = \frac{1}{M_{x'}} \left( \frac{1}{2} \right)^{|C_u|+|C_v|}.$$

In the above, $S_u$ is the set of data points that returned by query in $T$ using $u$. $|S_u|$ denotes the number of elements in $S^u$. All the other symbols have the same meaning as before. $Pr(u, v)$ is the collision probability between $u$ and $v$. The probability of the LSH Smart-merge move is given the state $x$, what is the probability that we can go from $x$ to $x'$. In our setting, state $x'$ contains the merged component $C$, and state $x$ contains the original components $C_u$ and $C_v$. So, under our LSH merge proposal design. Every combination of $u \in C_u$ and $v \in C_v$ can lead to the state $x'$ from state $x$. The analysis of this probability is the same of LSH merge proposal distribution that introduced before. Once we choose $u \in C_u$, and $v \in C_v$, we have probability 1 to merge $C_u$ and $C_v$ together to get $C$ under our proposal. So we multiply them all
together and yield the desired expression.

The corresponding inverse (Dump Split) probability \( q(x|x') \) is:

\[
q(x|x') = \frac{1}{M_{x'}} \left( \frac{1}{2} \right)^{|C_u| + |C_v|}.
\]

The inverse move of the LSH Smart-merge move is a dumb split: given the state \( x' \) which contains the combined component \( C \), what is the probability that we can split \( C \) to \( C_u \) and \( C_v \) (state \( x \)). Under dumb split, we have \( \frac{1}{M_{x'}} \) to choose component \( C \). Then we do a random split, so we have probability \( \left( \frac{1}{2} \right)^{|C_u| + |C_v|} \) to split \( C \) into \( C_u \) and \( C_v \). We multiply these two expressions together and yield the desired expression.

Notice that, to calculate the transition probabilities in Eq. 4.2 and Eq. 4.3, we need to sum over all possible \( u \) and \( v \) in the two components \( C_u \) and \( C_v \). This could be expensive when the cluster size is large. In other words, this complexity of this proposal is quadratic to the size of the cluster.

The quadratic cost seems unavoidable. LSH does similarity based sampling. Thus, we can sample pairs \( u \) and \( v \) in adaptive fashion efficiently. A split of cluster \( C \) into \( C_u \) and \( C_v \) can happen because of any two elements \( x \in C_u \) and \( y \in C_v \) being samples. As a result, the transition probability requires accumulating non-uniform probabilities of all possible combinations, making it quadratic to compute. On the other hand if every pair has same probability then the proposal is random. Overall, it seems hopeless to split the cluster adaptively and at the same time get the probability of split linear in the size of cluster.

It turns out, surprisingly, that a very unique design of proposal that satisfies our wishlist. It is the unique mathematical properties of MinHash and a novel general-
ization of its $k$-way collision probability that makes this possible. In the next section, we will introduce the method of use $k$-way minhash for scaling up MCMC [SL13].

### 4.3.2 MinSM: MinHash based Split-Merge MCMC

Ideally, after identifying $u$ we should split so that all the elements similar to $u$ goes to $C_u$ and rest goes to $C_v$. This will be a significantly more informative proposal than random assignments to $C_u$ and $C_v$. However, evaluating the transition probability of configuration under LSH would be computationally intractable, as LSH sampling is correlated and the expressions are contrived as we introduced in previous sections.

We next show that MinHash with a very specific design exactly achieves this otherwise impossible and ideal state with the cost of evaluating the transition probability linear in the size of the cluster. A unique property of MinHash is that we can compute, in closed form and linear cost, the probability of collision of a set of points of any size $\geq 2$. Such computation is not possible with any other know LSH including the popular random projections [G+99].

We provide a novel extension of the collision probability of MinHash to also include the probability of collision with a given set and no collision with another given set (See Equation 4.1). It is surprising that despite many non-trivial correlations, the final probability expression is very simple and only requires linear computations. As a result, we can directly get the split of a cluster into two sets (or clusters) and at the same time compute the transition probability. The novel design and analysis of Minhash, presented here, could be of independent interest in itself.

Minhash is a powerful tool for hashing sets [Li17]. Weighted Minhash, as a weighted version of the Minwise Hash, can deal with the dataset with weighted
weights. However, the vanilla version of weighted Minhash can only deal with the data points with positive weights.

In this chapter, the proposed GMM kernel is defined on general data types which can have both negative and positive entries. The basic idea is to first transform the original data into nonnegative data and then compute the Minhash on the transformed data.

Generalized Min-Max Kernel, [Li17] is designed for dealing with the data types which can have both negative and positive entries. Consider the data vector $u_i$, which is the $i$-th feature of data point $u$. The GMM model uses the following transformation to deal with the negative features:

\[
\begin{cases}
\tilde{u}_{2i-1} = u_i, u_{2i} = 0 & \text{if } u_i > 0 \\
\tilde{u}_{2i-1} = 0, u_{2i} = -u_i & \text{if } u_i \leq 0
\end{cases}
\]

For the details of this transformation, please refer to [Li17].

After transfer all the data point with $D$ dimension into the $2D$ dimension by using GMM model, we can directly use Minhash to do hash the data point. In the rest of the section, we will introduce the details of our proposal design by using Minhash.

**MinHash Smart-split/Dumb-merge**

MinHash based split begins by flipping a coin to randomly choose from the action of Smart-split or Dumb merge.

The LSH smart-split begins by randomly selecting an element $u$ in the dataset. Then, we use LSS (Locality-sensitive Sampler) to sample a set of points that are likely to be similar to $u$ from $T$, i.e., query $T$ with $u$. Here we use Weighted MinHash as
the LSH and \( K = 1 \) is necessary. \( K \geq 1 \) makes the probability computations out of reach. Instead of sampling a point from the bucket, as we do with LSS, we just report the whole bucket as the set. Let us denote this sampled set as \( S_u \). We now split the component \( C_u \) into two components: \( C_u \cap S_u \), \( C_u - S_u \). If the action is a dumb merge, then we randomly select two components and merge these two components into one component.

Define \( p \) as the probability of agreement of weighted minhash of \( u \) with all of the data point in the queried set \( S_u \). The known theory \([G+99]\) says that the expression of \( p \) is given by Equation 4.1. However, we want something more, we want all elements of \( S_u \) to collide with \( u \) in the bucket and anything in \( C_u - S_u \) to not collide. Define \( Prob \) as the probability of agreement of weighted minhash of \( u \) with all of \( S_u \) and none of the data point in \( C_u - S_u \). It turns out that we can calculate this probability exactly as:

\[
Prob = \frac{\sum_j^{2D} \max\{0, (x_{\min}^j - x_{\max}^j)\}}{\sum_j^{2D} x_{all}^j}, \tag{4.4}
\]

where \( x_{\min}^j = \min_{x \in C_u \cap S_u}\{x_j\} \), \( x_{\max}^j = \max_{x \in C - S_u}\{x_j\} \) and \( x_{all}^j = \max_{x \in C_u}\{x_j\} \)

When we only use \( K = 1 \) Minhash, then the corresponding proposal distribution is shown as follow:

\[
q(x'|x) = \frac{|S_u|}{n} \times Prob. \tag{4.5}
\]

Here, we give an illustration of the proof. Consider Figure 4.1. Let’s start with
vanilla MinHash over sets and the arguments will naturally extend to weighted versions. Given $X_1$, $X_2$ and $X_3$. We want the probability that the MinHash of $X_1$ and $X_2$ collide but not of $X_3$. From the theory of consistent sampling [SL13; Shr16; MMT10]. This will happen if we sample from $b$ and the possibility is the union. Thus the probability is

$$\frac{b}{a + b + c + d + e + f + g} = \frac{|X_1 \cap X_2| - |X_3|}{|X_1 \cup X_2 \cup X_3|}$$

which is essentially we want the minimum of $|X_1 \cup X_2 \cup X_3|$ to be sampled from the intersection of $X_1$ and $X_2$ and not from $X_3$. That is the only way the MinHash of $X_1$ and $X_2$ will agree but not of $X_3$. This argument can be naturally extendend if we want $X_1, X_2, ..., X_h$ to have same minhash and not $Y_1, Y_2, ..., Y_g$, the probability can be written as:
\[
\frac{\max\{0, |X_1 \cap X_2 \cap \ldots \cap X_h| - |Y_1 \cup Y_2 \cup \ldots \cup Y_g|\}}{|X_1 \cup X_2 \cup \ldots \cup X_h \cup Y_1 \cup Y_2 \cup \ldots \cup Y_g|}.
\]

Now for weighted sets (non-binary), we can replace intersection with minimum and unions with max leading to the desired expression, which is due to the seminal works in consistent weighted sampling a strict generalization of MinHash. See [SL13; Shr16; MMT10] for details. Also using [LRU14] we can extend it to negative weights as well using simple feature transformation.

It should be noted that this expression only requires cost linear in the size of the cluster \(C_u\) being split. With this value of \(Prob\), the corresponding transition probability for the split move is:

\[
q(x'|x) = \frac{|S_u|}{n} \times Prob, \quad q(x|x') = \frac{2}{M_{x'}(M_{x'} - 1)}. \tag{4.6}
\]

In the above, \(n\) is the number of data point. \(S_u\) is the set of data points that returned by querying in \(T\) using \(u\), and \(|S_u|\) denotes the number of elements in \(S_u\). \(D\) is the dimension of the data. \(M_{x'}\) denotes the number of clusters in state \(x'\).

To be able to compute this expression and also get an informative split was the primary reason for many choices that we made. For example, \(K = 1\) as needed so that we can compute \(Prob\) in a simple closed form. As a result, we obtain a very unique proposal. The idea and design could be of independent interest in itself.

**Derivation:** For the smart-split move, \(q(x'|x)\) denotes that: given the state \(x\), what is the probability that we can go from \(x\) to \(x'\). In this setting, state \(x'\) contains
components $C_u \cap S_u$ and $C_u - S_u$. The state $x$ contains the original component $C_u$. So, under our LSH split proposal design, we have the probability $\frac{|S_u|}{n}$ to choose $u$. Then after pick $u$, we have the probability \[ \frac{\sum_{i=2}^{2D} \max \{ w_1^{(1)}, w_1^{(2)}, \ldots, w_1^{(|C_u \cap S_u|)} \}}{\sum_{i=2}^{2D} \max \{ w_1^{(1)}, w_1^{(2)}, \ldots, w_1^{(|C_u \cap S_u|)} \}} \], which is the collision probability, that all the data point in $C_u \cap S_u$ goes into the same bucket. We multiply them together and yield the desired expression.

The inverse move of the smart-split move is a dumb move: given the state $x'$ which contains $C_u \cap S_u$ and $C_u - S_u$, what is the probability that we can combine $C_u \cap S_u$ and $C_u - S_u$ to generate $C_u$ (state $x$). Under dumb merge, we have $\frac{1}{M_{x'}}$ to choose one of $C_u \cap S_u$ or $C_u - S_u$, and probability $\frac{1}{M_{x'} - 1}$ to choose the other one. So the corresponding probability is the multiplier of these two probabilities.

**Minhash Smart-merge/Dumb-split**

The proposed smart-merge begins by randomly selecting a center $u$ in the dataset. Then, we use LSS (Locality-sensitive Sampler) to sample a center $v$ that are likely to be similar to $u$. Then we merge the component $C_u$ and $C_v$ to one component.

If the action is a dumb split: randomly select one cluster, and split this component into two separate components uniformly.

Given a new state $x'$, and the corresponding old state $x$. We provide the probability of the merge move $q(x'|x)$ and the corresponding inverse probability $q(x|x')$ as follow:
In the above, \( S_u \) is the set of data points that returned by the query in hash table \( T \) using \( u \). \(|S_u|\) denotes the number of elements in \( S_u \). \( u_j \) denotes \( j \)-th feature of the data point \( u \). All the other symbols have the same meaning as before.

For the smart-merge move, \( q(x'|x) \) denotes that: given the state \( x \), what is the probability that we can go from \( x \) to \( x' \). In this setting, state \( x' \) contains the merged components \( C \). The state \( x \) contains the original two component \( C_v \) and \( C_u \).

So, under our LSH merge proposal design, we have the probability \( \frac{1}{M_x} \) to choose the component \( C_u \). Then, we have the probability

\[
\frac{\sum_{i}^{2D} \min\{u_i, v_i\} \cdot 1}{\sum_{i}^{2D} \max\{u_i, v_i\} \cdot |S_u|}
\]

to pick cluster center \( v \) in the hash table. Here,

\[
\frac{\sum_{i}^{2D} \min\{u_i, v_i\}}{\sum_{i}^{2D} \max\{u_i, v_i\}}
\]

is the probability that two centers \( u \) and \( v \) hashed into the same bucket, and \( \frac{1}{|S_u|} \) is the probability to pick \( v \) from the bucket. We multiply these two expressions together and yield the desired expression.

The inverse move of the LSH Smart-merge move is a dumb split: given the state \( x' \) which contains the combined component \( C \), what is the probability that we can
split $C$ to $C_u$ and $C_v$ (state $x$). Under dumb split, we have $\frac{1}{M_u}$ to choose component $C$. Then we do a random split, so we have probability $(\frac{1}{2})^{|C_u|+|C_v|}$ to split $C$ into $C_u$ and $C_v$. We multiply these two expressions together and yield the desired expression.

The overall procedure is then summarized as Algorithm 2. This algorithm implements the methods we have introduced in the above subsections.

As we introduced before, our proposed algorithm belongs to the general framework of metroplis-hastings algorithm [And+03]. After each split/merge move, we need to calculate the acceptance rate $\alpha(x'|x)$ for this move which is given by:

$$\alpha(x'|x) = \min\{1, \frac{L(x')q(x|x')}{{L(x)q(x'|x)}}\}$$

, where $x'$ is the proposed new state, $x$ is the previous state, $q(x'|x)$ here is the designed proposal distribution, and it can be calculated as introduced in previous sections. $L(x)$ is the likelihood value of the state $x$.

The likelihood of the data is generally in the form of

$$L(x) = \prod_D p_j(e_i)$$

, where $p_j(e_i)$ is the probability of $e_i \in D$ in it’s corresponding component $C_j$. $D$ denotes the total dataset. In the split merge MCMC, only the components that being split/merged will change of the likelihood value. So, that the ratio $\frac{L(x')}{L(x)}$ is cheap to compute, since all the probability of unchanged data will be canceled.
4.4 Empirical Study

In this section, we demonstrate the advantage of our proposed models by applying it to the Gaussian Mixture model inference and compare it with state-of-the-art sampling methods.

4.4.1 Gaussian Mixture Model

We briefly review the Gaussian Mixture Model. A Gaussian mixture density is a weighted sum of component densities. For a \( M \)-class clustering task, we could have a set of GMMs associated with each cluster. For a \( D \)-dimensional feature vector denoted as \( x \), the mixture density is defined as

\[
p(x) = \sum_{i=1}^{M} w_i p_i(x)
\]

, where \( w_i, i = 1, ..., M \) are the mixture weights which satisfy the constraint that \( \sum_{i=1}^{M} w_i = 1 \) and \( w_i \geq 0 \). The mixture density is a weighted linear combination of component \( M \) uni-model Gaussian density functions \( p_i(x), i = 1, ..., M \). The Gaussian mixture density is parameterized by the mixture weights, mean vectors, and covariance vectors from all components densities.

For a GMM-based clustering task, the goal of the model training is to estimate the parameters of the GMM so that the Gaussian mixture density can best match the distribution of the training feature vectors. Estimating the parameters of the GMM using the expectation-maximization (EM) algorithm [Nas07] is popular. However, in most of the real world applications, the number of clusters \( M \) is not known, which is required by the EM algorithm. On the other hand, Split-Merge based MCMC
algorithms are used for inference when $M$ is unknown, which is also the focus of this work. We therefore only compare our proposal LSHSM and other state-of-the-art split-merge algorithms on GMM clustering which does not require the prior knowledge of the number of clusters.

4.4.2 Competing Algorithms

**Competing Algorithms:** We compare following four split-merge MCMC sampling algorithm on GMM with an unknown number of clusters:

- **RGSM:** Restricted Gibbs split-merge MCMC algorithm [JN04] is considered as one of the state-of-the-art sampling algorithm.

- **SDDS:** Smart-Dumb/Dumb-Smart Split Merge algorithm [WR15]. SDDS combines “smart” split/merge move that proposes plausible splits of heterogeneous clusters with a “dumb” merge move that proposes merging random pairs of clusters.

- **LSHSM:** The Naive version of LSH based Split Merge algorithm by using Sign Random Projection. In the LSHSM method, we use fixed $K = 10$ and $L = 10$ for all the dataset. We fix the hashing scheme to be signed random projection.

- **MinSM:** LSH based split merge algorithm is the proposed method in this chapter. In the MinSM method, we use fixed $K = 1$ and $L = 1$ for all the dataset.

4.4.3 Dataset

We evaluate the effectiveness of our algorithm on both two large real-world datasets: **KDDCUP** and **PubMed**. **KDDCUP** data was used in the KDD Cup 2004 data mining competition. It contains 145751 data point. The dimensionality of the dataset
is 74. We have 2000 ground truth cluster labels for this dataset. * The PubMed abstraction dataset contains 8200000 abstractions that extracted from the PubMed †. All the documents represented as the bag-of-words representation. In the data set, we have 141043, different words. This data set is ideal for document clustering or topic modeling. The dataset is available from the UCI machine learning dataset Repository. ‡

Synthetic data is a standard way of testing GMM models [Nas07]. So, in this chapter, we also use synthetic datasets as a sanity check to evaluate the performance of different methods. The process of generating the synthetic dataset is as follow: Randomly generate $k$ different Gaussian distributions (with different corresponding mean and variance). We fix the $k = 10$ in our experiment. Then based on the randomly generated Gaussian distributions, we generate a set of data points for each Gaussian distribution. Here we fix the dimensionality of each data point to 25. In

---

* https://cs.joensuu.fi/sipu/datasets/
† www.pubmed.gov
‡ https://archive.ics.uci.edu/ml/index.php
Figure 4.3: The time and iteration wise comparison of the likelihood for different methods on the two real datasets. It is obvious that our proposed MinSM algorithm can be at least 6 times faster than the state of the art algorithms in the real large dataset.

Table 4.1: The statistics for the two real world dataset

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Samples</th>
<th>Dim</th>
<th>True #Clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>KDDCUP</td>
<td>145751</td>
<td>74</td>
<td>2000</td>
</tr>
<tr>
<td>PubMed</td>
<td>8200000</td>
<td>141043</td>
<td>10000</td>
</tr>
</tbody>
</table>

In this experiment, we generate three synthetic datasets with different sizes (e.g., 100, 1000, 10000). We name the three synthetic datasets as S1, S2, S3.

4.4.4 Speed Comparison and Analysis

We first plot the evolution of likelihood both as a function of iterations as well as the time of all the three competing methods. The evolution of likelihood and time with iterations on two real-world data is shown in Fig. 4.3. The result on three synthetic data set is shown in Fig. 4.4.

We can see a consistent trend in the evolution of likelihood, which holds true for
both simulated as well as real datasets. First of all, RGSM consistently performs poorly and requires both more iterations as well as time. This demonstrate that the need of combining smart and dumb moves for faster convergence made in \cite{WR15} is necessary. RGSM does not use it and hence leads to poor, even iteration wise, convergence.

SDDS seems to do quite well, compared to our proposed LSHSM when we look at iteration wise convergence. However, when we look at the time, the picture is completely changed. MinSM is significantly faster than SDDS, even if the convergence is slower iteration wise. This is not surprising because the per-iteration cost of MinSM is orders of magnitude less than SDDS. SDDS hides the computations inside the iteration by evaluating every possible state in each iteration, based on likelihood, is equivalent to several random iterations combined. Such costly evaluation per iteration can give a false impressing of less iteration.

It is clear from the plots that merely comparing iterations and acceptance ratio can give a false impression of superiority. Time wise comparison is a legitimate comparison of overall computational efficiency. Clearly, MinSM outperforms the other baselines by a large margin.

4.4.5 Clustering Accuracy Comparison

To evaluate the clustering performance of different algorithms, we use two widely used measures (Accuracy and NMI \cite{Nas07}). Normalized Mutual Information (NMI) \cite{Nas07} is widely used for measuring the performance of clustering algorithms. It can be calculated as \( \text{NMI}(C, C') = \frac{I(C; C')}{\sqrt{H(C)H(C')}} \), where \( H(C) \) and \( H(C') \) are the marginal entropies, \( I(C; C') \) is the mutual information between \( C' \) and \( C \). The Accuracy
Table 4.2: Clustering Accuracy for Different Methods

<table>
<thead>
<tr>
<th>Methods</th>
<th>Metric</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>KDD</th>
<th>Pub</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGSM</td>
<td>NMI</td>
<td>0.96</td>
<td>0.93</td>
<td>0.88</td>
<td>0.74</td>
<td>0.63</td>
</tr>
<tr>
<td></td>
<td>Accuracy</td>
<td>0.95</td>
<td>0.92</td>
<td>0.87</td>
<td>0.68</td>
<td>0.62</td>
</tr>
<tr>
<td>SDDS</td>
<td>NMI</td>
<td><strong>0.97</strong></td>
<td><strong>0.96</strong></td>
<td>0.95</td>
<td><strong>0.86</strong></td>
<td><strong>0.80</strong></td>
</tr>
<tr>
<td></td>
<td>Accuracy</td>
<td><strong>0.98</strong></td>
<td><strong>0.97</strong></td>
<td>0.94</td>
<td><strong>0.85</strong></td>
<td><strong>0.77</strong></td>
</tr>
<tr>
<td>LSHSM</td>
<td>NMI</td>
<td>0.96</td>
<td>0.95</td>
<td><strong>0.96</strong></td>
<td>0.84</td>
<td>0.77</td>
</tr>
<tr>
<td></td>
<td>Accuracy</td>
<td>0.97</td>
<td>0.94</td>
<td>0.96</td>
<td>0.83</td>
<td>0.75</td>
</tr>
<tr>
<td>MinSM</td>
<td>NMI</td>
<td>0.96</td>
<td>0.94</td>
<td><strong>0.96</strong></td>
<td>0.83</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>Accuracy</td>
<td>0.97</td>
<td>0.94</td>
<td><strong>0.97</strong></td>
<td>0.84</td>
<td>0.74</td>
</tr>
</tbody>
</table>

measure, which is calculated as the percentage of target objects going to the correct cluster, is defined as \(\text{Accuracy} = \frac{\sum_{i=1}^{k} a_i}{n}\), where \(a_i\) is the number of data objects clustered to its corresponding true cluster, \(k\) is the number of cluster and \(n\) is the number of data objects in the dataset.

Table 4.2 shows the clustering accuracy of different competing methods. We can see that the MinSM, LSHSM and SDDS are much more accurate than RGSM. This observation is in agreement with the likelihood plots. On the other hand, the accuracy difference between MinSM, LSHSM and SDDS is negligible. This small difference is due to the mismatch between the likelihood value and clustering accuracy. It should be noted that the difference is small for SDSS and MinSM variants because both achieved the same likelihood value. For the Random Split merge with the worse likelihood, the difference is huge, indicating the clustering results does correlate with likelihood values except for minor variations.
4.5 Discussion

The Split-Merge MCMC (Monte Carlo Markov Chain) is one of the essential and popular variants of MCMC for problems with an unknown number of components. It is well known that the inference process of Split-Merge MCMC is computational expensive which is not applicable for the large-scale dataset. Existing approaches that try to speed up the split-merge MCMC are stuck in a computational chicken-and-egg loop problem.

In this chapter, we proposed MinSM, accelerating Split Merge MCMC via weighted Minhash. The new splitmerge MCMC has constant time update, and at the same time the proposal is informative and needs significantly fewer iterations than random split-merge. Overall, we obtain a sweet tradeoff between convergence and per update cost. Experiments with Gaussian Mixture Model on two real-world datasets demonstrate much faster convergence and better scaling to large datasets.
Algorithm 2 MinSM Algorithm

**Input:** Dataset $D$, Parameter $K$, $L$

**Output:** The Result Mixture Components.

Transfer all the data using Min-Max Kernel [Li17]. Hash all the data point in $D$ into LSH Data structure $T$. Randomly assign data to different component. Hash all the centers into LSH Data structure $T_c$.

**while** Convergence **do**

Choose a move type randomly: TYPE = (split, merge).

**switch** TYPE **do**

**case** Split **do**

Choose a move type randomly: TYPE = (split, merge).

**switch** TYPE **do**

**case** Split **do**

Randomly sample one element $u$ in $D$.

Query $u$ in $T$ to get $S_u$.

Split the component $C_u$ to two components $C_u \cap S_u$, $C_u - S_u$.

**end**

**case** Merge **do**

Randomly Pick one component $C$. Randomly split $C$ into $C_u$ and $C_v$.

**end**

**end**

**case** Merge **do**

Choose a move type randomly: TYPE = (split, merge).

**switch** TYPE **do**

**case** Merge **do**

Randomly sample one component center $u_c$.

Query $u_c$ in $T_c$ to get another center $v_c$.

Merge $C_u$ and $C_v$.

**end**

**case** Split **do**

Randomly pick two components $C_u$ and $C_v$. Merge $C_u$ and $C_v$.

**end**

**end**

Calculate $\mathcal{L}(x)$, $\mathcal{L}(x')$, $q(x'|x)$, $q(x|x')$ using Equations introduced in 4.3.1.

Calculate the acceptance ratio $\alpha$.

Accept the proposal with probability $\alpha$.

**end**
Figure 4.4: The time and iteration wise comparison of the likelihood for difference methods on the Synthetic Dataset. MinSM outperforms the other baselines by a large margin. It is also clear that requiring less iteration does not mean faster convergence.
Chapter 5

CapSule: Image based Indoor navigation using LSH

In this chapter, we introduce a hashing system that used for indoor localization.

5.1 Introduction

Indoor localization technology is expected to be a 4 billion dollar industry by 2018 [Liu+07]. Increased demand for accurate indoor localization market is due to venue-based marketing, poor performance of GPS in indoor environments [Mau09], and government initiatives in developing positioning systems for public safety and urban security segments.

GPS signals are blocked indoors and therefore have poor accuracy. Therefore, there are a variety of algorithms utilizing other sensors, such as WiFi [FNI13], for estimating the location indoor. Such algorithms rely on aggregating information from multiple sensors to get good accuracy, which makes then expensive and complicated.

Very recently, it was found that an elegant way of localizing a mobile more accurately is by utilizing the device’s camera [Lia+13]. The idea is to match the current image from the camera with a database of geo-tagged images. Recent advances in vision have made image matching technology quite accurate, which makes camera based image positioning a very promising direction. However, current image matching algorithms are quite expensive from both latency and energy perspectives, and
therefore they cannot run locally on a mobile device. For instance, we show that current state-of-the-art image matching algorithm when run on the database of 719 images require more than 1000 seconds using around 2100 Joules of energy for getting the current location; entirely impractical for use in a mobile context. An alternate is to consider a cloud-based service to perform image matching.

There are three major concerns with cloud-based image matching: 1) Communication, 2) Energy Consumption and 3) Privacy.

1 Communication: Image matching requires transmitting the current image from the mobile device to the cloud, followed by the location, inferred in the cloud and transferred back. Communication often has unpredictable latency, as it requires WiFi, cellular networks, etc.

2 Energy Consumption: Image matching is an expensive operation. The cloud-based service even if very fast, is likely to consume a significant amount of energy. Thus cloud-based image matching service is not a sustainable solution.

3 Privacy: Transfer of data back and forth to the cloud compromises the privacy of user’s information. It opens the possibility of potential privacy breaches.

**Hope: Trading (insignificant) Quality for Energy.** The philosophy of trading (a small amount of) quality for significant gains in energy has recently gained significant attention [Pal14; Mar15; Pal03]. Capitalizing on this energy-quality tradeoff is deemed to be a future of SoC technology [8]. Here, we provide a concrete demonstration of this philosophy. Our proposed end-to-end system shows that by trading a small amount of accuracy we can get away with all the three shortcomings, as mentioned earlier, associated with the cloud-based image matching techniques.
**Our Contributions:** We propose CaPSuLe for image based device positioning, a first of its kind system, which is free from all the three problems of communication, energy consumption, and privacy. At the heart of CaPSuLe lies an approximate image matching algorithm, based on fast locality sensitive hashing, which is more than 500x times cheaper than state-of-the-art image matching algorithm. Such a significant gain in computation and energy cost is a result of careful choices of hash tables, hash functions, and related operations. This massive reduction allows us to perform end-to-end image matching on the mobile device itself. Our algorithm takes 1.92 seconds requiring 3.78 Joules energy on Samsung Galaxy S4 archiving 92.11% accuracy in estimating the location. Since all computations are local and are performed on the device, our algorithm is free from privacy infringements as no information is transmitted. We hope that our work will lead to many new energy efficient machine learning algorithms where the need for cloud computing can be eliminated.

5.2 Devise Positioning via Image Matching

Image based positioning system [Lia+13; Kaw+10] takes the current picture of the location and matches it with images in a pre-collected database of geo-tagged images of the area of the building such as a shopping mall. The location of the matched image is deemed to be the current position of the device. The key observation is that building a densely sampled dataset consisting of images, tagged with their geo-location at different places in the indoor environment is a relatively easy task with the surge in the number of images. The accuracy of the system is then directly dependent
on the ability to identify the right matching image in the database, which is a classical computer vision problem.

Formally, we are given a collection of geo-tagged images $C$ consisting of images from the given indoor environment, e.g., shopping mall, campus, etc., where the device needs to be positioned. By using its camera, we create a query image $q$. The goal of the image matching algorithm is the find an image $I \in C$ which maximizes the “similarity” with the query $q$. Formally,

$$I = \arg \min_{I \in C} \text{Sim}(q, I)$$

(5.1)

The critical vision component in Equation 5.1 is the design of the similarity function or $\text{Sim}(., .)$ which captures the notion of semantic similarity between different images. $\text{Sim}(., .)$ must tolerate variations in pose, resolutions, shifts, etc [BK08; Aga06]. For better demonstration of the challenges associated with the state of the art, we first describe our setting and the dataset in next subsection.

5.2.1 Dataset and Settings

We chose the LOTTE Department Store main branch, which is a major shopping mall in Seoul, Korea, for our positioning system. We collected a total of 871 images of different shops in the mall. Images were collected by using Naver Indoor Maps *. Besides, to get a good coverage of the mall, we also manually took pictures of stores by a cellphone camera. Overall, we covered 45 different location in the shopping mall. The images are taken with varying poses and lighting to ensure that the datasets

*http://map.naver.com/*
reflect the real settings. Also, two separate sources of images make the setting more real and free of common bias.

Each image is annotated with its corresponding locations. We further downscaled each image to $640 \times 360$ pixels to reduce the computational cost associated with matching without significant loss of accuracy. Such downsampling of images are commonly adopted in many real applications [You99]. For evaluation, we partition the data into two sets: 719 training and 152 query images. Figure 1 shows some query images and the matched training images for these query images using our CaPSuLe system. We can clearly see the complexity of the problem as the matching images can have varying poses.
Table 5.1: Target System Description

<table>
<thead>
<tr>
<th>Hardware</th>
<th>System on Chip</th>
<th>Exynos 5410 Octa</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>Quad-core 1.6 GHz Cortex-A15</td>
</tr>
<tr>
<td></td>
<td>Main Memory</td>
<td>2GB</td>
</tr>
<tr>
<td></td>
<td>Storage</td>
<td>16 GB NAND Flash</td>
</tr>
<tr>
<td>Software</td>
<td>OS Kernel</td>
<td>Linux kernel Version 3.4.5</td>
</tr>
<tr>
<td></td>
<td>Android Framework</td>
<td>Android 5.0.1</td>
</tr>
<tr>
<td></td>
<td>OpenCV</td>
<td>OpenCV 3.1 for Android</td>
</tr>
</tbody>
</table>

5.2.2 Device and Platform

We use a Samsung Galaxy S4 smartphone with Android 5.0.1 Lollipop running with a Linux kernel 3.4.5. The smartphone has an ARM processor that consists of four Cortex-A15 cores and 2 GB DRAM. We additionally used a Monsoon Power Meter to measure the power consumption of the smartphone. The detailed hardware and software configuration of the target system is shown in Table 5.1.

5.2.3 The Image Matching Problem and its Computational (Energy) Barrier

With advancements in vision technology, image matching is quite accurate. However, they are far from being cheap. Modern matching algorithms require costly similarity measure Sim for good accuracy. Such expensive computations cannot be performed on the device because of their significant computational requirements. We elaborate our baselines which is the state-of-the-art algorithm, as implemented in the widely used OpenCV package [10], [11], for computing Equation 1.

The similarity measure, Sim(., .), used in the OpenCV package leads to 93% accuracy on our dataset. Other similarity measures based on Euclidian distance over
Bag-of-Words (BoW) only yields 75% or less accuracy because our dataset contains many variations seen in the real environment which is not adequately captured by BoW methods.

OpenCV implementation for determining the similarity between the query $q$ and any given image $I \in C$ requires the following three steps:

1. **Extract Features from Images:** The first step is to extract a set of SURF [BTV06] features from both the $q$ and $I$. Each of these features is a 64-dimensional vector. In our system, we used 512 SURF features. We thus get 512 different 64-dimensional SURF features from each image. SURF features are the best-known features invariant to scale and other spurious transformations such as rotations. It is further known that SURF performs even better than traditional SIFT features [Low04]. Note, for every image $I$ in the given training collection $C$, feature extraction is done off-line. For the query, however, feature extraction needs to be done on the fly.

2. **Threshold all pairwise features:** The Euclidian distance between all possible feature combinations between $q$ and $I$ is then computed. This requires $512 \times 512$ Euclidian distance computations between 64 dimensional vector from $q$ and $I$, totalling $512 \times 512 \times 64$ multiplications.

3. **Compute the Similarity Values:** The final score is the number of distances out of $512 \times 512$, which are smaller than a threshold. Roughly, this similarity measure scores based on the number of cross matches from the 512 different SURF features between the query and the image $I$.

The bottleneck is step number 2 which requires $512 \times 512 \times 64 \geq 16$ million multiplications for computing the similarity between the query and one image $I \in C$. 
Thus, for 719 images in our datasets, a single query requires $512 \times 512 \times 64 \times 719$ more than 12 billion multiplications. If we plan to run this amount of computation on the mobile device, then to reiterate a single query takes more than 1030 seconds consuming more than 2100 Joules of energy. Step 2 is the primary reason why current image-based positioning algorithm needs the cloud to perform matching in reasonable time. However, as argued before, the cloud-based solution has many shortcomings. We will refer this as the Brute-force Method.

5.2.4 Clustering (Bag-of-Words (BoW), sparse coding, etc.) does not seem to help

It might seem that step 2, requiring $512 \times 512$ distance computation can be side-stepped. The other most popular feature representation which does not require $512 \times 512$ distance calculation is the BoW \cite{Csu+04} (or sparse coding \cite{Lee+07}) feature representation. BoW tries to eliminate the need for multiple comparisons by pre-clustering all the extracted 512 SURF features. After clustering, BoW calculates the distances between all feature vectors of the current image and the cluster centers. BoW then produces a histogram expressing the closeness between cluster centers and the training image’s SURF feature vectors. Image matching is finally performed by comparing the query image’s histogram and the stored training image’s histograms. This process is relatively cheap. However, it comes with a significant loss in accuracy. With our dataset, this approach barely reached 75% even with as many as 1000 clusters. Changing the cluster size of 5000 has no effect on accuracy.

Image matching is a harder task than object detection. For instance, two images may have the same categorical object (such as a chair), but they may not match with
each other. This is probably the main reason why BoW is more common for object detection rather than image matching and popular state-of-the-art package OpenCV [10] implements more costly matching algorithms described earlier.

5.3 Hope: Probablistic Hashing Algorithms

However, if we are willing to relax the need for accuracy by a small amount, then the picture changes completely. In particular, we will use the cheap Locality Sensitive Hashing algorithms combined with the careful choice of hash functions and estimation procedure to get more than 500x reduction in the computational and the energy cost.

Locality Sensitive Hashing (LSH) [IM98; Shr15] is popular for efficient sub-linear time matching. LSH generates a random hash map $h$ which takes the input (usually the data vector) and outputs a discrete (random) number. For two data vectors $x$ and $y$, the event $h(x) = h(y)$ is called the collision (or agreement) of hash values between $x$ and $y$. The hash map has the property that similar data vectors, in some desired notion, have a higher probability of collisions than non-similar data vectors. Informally, if $x$ and $y$ are similar, then $h(x) = h(y)$ is a more likely event, while if they are not similar then $h(x) \neq h(y)$ is more likely. The output of the hash functions is a noisy random fingerprint of the data vector [CW79; Rab81; KR87], which being discrete is used for indexing training data vectors into hash tables. These hash tables represent an efficient data structure for matching [IM98] and learning [SS17e; Li+11].

The fundamental observation is that in Step 2 of image matching (Section II-C), for every SURF feature of the query image $q$, we search for matching SURF features from image $I$. This matching can be made efficient using hashing. However, only
performing fast near neighbor search with hashing does not yield the desired benefit. It further requires many careful choices which we describe in the next section.

5.4 The CapSule System: Near-Cloud Performance with on-Device Computation.

Our CaPSuLe system is summarized in Figure 5.2. At the heart of our system lies a set of lightweight hash tables which, for a given SURF feature of a query image, finds all the potential matching SURF features from the training set. This search is done in near-constant time, by querying hash tables indexed by LSH, which saves a significant amount of computational (and hence energy) overhead without compromising the accuracy.

CaPSuLe uses two parameters K and L which trades accuracy for gains notably in energy and in computational time. The algorithm works in two main phases [And04] for device positioning. We first describe the two phases, and later we provide more details about the design choices.

1) Preprocessing Phase (Offline): In the offline phase, following step 1 in previous section, we extract 512 SURF features (64 dimensional) from each geo-tagged image I in the training collection C. We then create L different hash tables of size $2^K$, i.e., K-bit keys (or indices). For every 512 SURF feature of I, we map it to a K-bit signature $H_j(I)$, using some LSH scheme $H_j$, for $j \in 1, 2, \ldots, L$. Image I is then placed into hash table number j indexed by the K-bit signature $H_j(I)$ (as the key). Thus, every image is mapped to 512 keys (can be duplicate) in each of the L hash tables. The preprocessing step thus generates L independent hash tables.
Figure 5.2: Example Query and returned matches image by CaPSuLe system. The match is with varying pose and orientations showing the complexity of our dataset.
2) **Query Phase (Online):** Given a query image \( q \), we again extract 512 SURF features (64 dimensional each). For each of these 512 SURF features, we retrieve the bucket associated with the key \( H_j(q) \) in hash table \( j \). Overall, we get \( 512 \times L \) keys and probe the corresponding buckets (values) in the associated hash tables. Every image is then ranked based on the number of times it is observed in the \( 512 \times L \) buckets. The location of the top ranked image is returned as the current location as the final answer.

In this methodology, we made five novel and careful choices in CaPSuLe, all of which are critical. The system is prohibitively expensive if we remove any of the five choices. These choices are as follows:

1. **Reduce Hashing Cost:** The cost of computing \( L \) different \( K \)-bit hashes is expensive with popular LSH schemes such as signed random projections [26]. In particular, traditional LSH requires \( K \times L \times 512 \times 64 \) multiplications for computations of all the hashes (also the keys), which is very expensive. We instead used a cheap and sparse variant as described in [Ach01] which reduces the total hashing cost per query to \( \frac{1}{3}(K \times L \times 512 \times 64) \) additions/subtractions. This is a significant reduction also since multiplications are costlier than additions

2. **Buckets of Bit Arrays:** Our hash tables need to store multiple images for every key. Even if we store only integer image IDs, the cost is significant. Since we have 719 images, we store a 719-bit array indexed by the \( K \)-bit key (\( 2^K \) values). If image numbered \( n_i \) gets a particular key, we simply set the bit numbered \( n_i \) in the bit-array associated with the key as shown in Figure 2. This idea leads to around 32x reduction in the hash table size compared to the traditional scheme. Furthermore, we remove any memory associated with empty buckets during preprocessing to avoid
unnecessary memory usage.

3. **Cheap and Crude Ranking Estimation based on Bucket Matches:**
Hashing reports many images (sometimes multiples of a 100) as potential matches. For computing the best match, the recommended option in the literature is to rank candidates using the similarity function Sim. However, as argued in Section II, computing Sim is expensive. We utilize the property of LSH, and cheaply estimate the ranking by counting the number of times an image is hit by the query. Estimation using LSH signatures are significantly cheaper than similarity computation as reported in [SL12].

4. **Ignoring Noisy Buckets:** As our hash functions are cheap, there is a significant possibility that individual key values are likely due to bias in the LSH functions. Such bias will make some of the buckets unnecessarily crowded. Crowded buckets increase the computational time and energy since the algorithm retrieves unnecessary candidates. To eliminate this issue, we ignore buckets (treat it as empty) if they are overcrowded.

5. **Reducing Main Memory:** Although hash tables are significantly small (few hundred MBs), for mobile devices, loading all of them in main memory is still a concern. Our hash tables are organised into contiguous buckets, i.e., $2^K$ indices each of 719 bits (see Figure 2) We, therefore, store the hash tables in device memory and load the L buckets (719 bits for each bucket) on demand during runtime (using the fseek function) without noticeable I/O overhead. This ensures that our application needs low main memory.
5.4.1 Dynamic Updates

One of the unique characteristics of CaPSuLe is that it can be incrementally updated. In particular, adding/deleting images to/from the database with only amounts to flipping a few bits, to add the new image (with labels) in the corresponding buckets, into the appropriate hash table. Thus, increasing the number of images or locations can be handled with no modification to the algorithm and minimal change to the data structure.

5.5 Experimental Result

We implemented CaPSuLe on the platform described in Figure 1. We evaluate CaPSuLe on four metrics: 1) Response Time, 2) Energy Consumption, 3) Accuracy and 4)
Table 5.2: Evaluations for CaPSuLe and Bruteforce

<table>
<thead>
<tr>
<th></th>
<th>CaPSuLe</th>
<th>Bruteforce</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Accuracy</strong></td>
<td>92.11%</td>
<td>93.42%</td>
</tr>
<tr>
<td><strong>Energy Consumption</strong></td>
<td>3.78J</td>
<td>2103.22J</td>
</tr>
<tr>
<td><strong>Response Time</strong></td>
<td>1.92 sec</td>
<td>1030.43 sec</td>
</tr>
<tr>
<td><strong>Required Storage Space</strong></td>
<td>294.39 MB</td>
<td>363 MB</td>
</tr>
<tr>
<td><strong>Required Memory Space</strong></td>
<td>78.90 MB</td>
<td>171.41 MB</td>
</tr>
</tbody>
</table>

Main Memory. Response time and energy consumption are measured for the complete end-to-end process, i.e., including the feature extraction, retrieval, and final ranking. Accuracy is measured over the test set as the percentage of time the algorithm identifies the correct location. Main memory usage is the amount of DRAM used. It is imperative that all of these four metrics are properly balanced for the system to be practical.

Cost-Quality Tradeoff through K and L: There are two main parameters in the CaPSuLe system, K, and L. To reiterate, K determines the range of the hash table (K-bits), which is also its size. L specifies the number of hash tables. K and L are the knobs which give us finer control over the cost-quality tradeoff. If we increase K and L, the recall is better, but the space required grows significantly.

See Figure 3 for the plot of memory utilization with varying K. If we use K = 24 the amount of main memory needed by a single hash table easily grows to around 1GB which for L > 1 hash tables is infeasible. If we lower K, then the accuracy drops by around 10%. We found that other than memory, the computational, energy, and response time costs are not sensitive to variations in K and L. Memory-accuracy is the main tradeoff. We found K = 22 and L = 24 to be the sweet-spot that balances both accuracy and memory nicely.
Our system uses these values for K and L. Note, we have two parameters which can be tuned offline. Competing Solutions: Our primary goal is to approximate the accuracy of the brute force algorithm described in Section II-C. However, we want our solution to run with limited energy, memory, and latency range, which are crucial for a device positioning system. Our primary baseline is, therefore, the brute force algorithm in the state-of-the-art package OpenCV.

In addition to brute force and CaPSuLe, we tried two classical and cheaper baseline approaches:

1) BoW based image matching: As described in Section II-D, we used popular BoW based features which exploit clustering over SURF features to make matching efficient. However, there is a significant drop in the accuracy in this case. With 1000 Bag-of-words (or cluster centers), we could barely achieve 75% accuracy even after fine tuning. Increasing BoW to 5000 lead to no significant gains.

2) Supervised Learning. We tried another possibility of treating location identification as a multi-class classification problem. We treat each location as a class label and use training images labeled with the location as the standard supervised multi-class classification. However, supervised learning fails to achieve more than 80% of accuracy. We used VLFeat [VF10], an open source package for image classification in this experiment.

5.5.1 Performance Summary

We used K = 22 and L = 24 for our settings. The response time and energy consumption for brute force and our approach are evaluated.

1) Accuracy: For our dataset, the accuracy of brute force is 93%, Bow 75%, super-
Figure 5.4: The accuracies of bruteforce, BoW and CaPSuLe.

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervised</td>
<td>77</td>
</tr>
<tr>
<td>Bag of Words</td>
<td>92.11</td>
</tr>
<tr>
<td>CaPSuLe</td>
<td>92.11</td>
</tr>
</tbody>
</table>

2) Response Time: We estimated the response time of Bruteforce and CaPSuLe. The response time using the Bruteforce method is 537 times more than CaPSuLe on the target mobile device. CaPSuLe takes only 1.92 seconds in the positioning phase on the device. However, the response time using the Bruteforce method is 1030.43 seconds in the online phase, which is unacceptable.

3) Energy Consumption: The Bruteforce method requires $2103.22J$ on our mobile system in the online phase. This amount of energy consumption further makes...
current algorithms non-sustainable. However, CaPSuLe consumed mere $3.78J$ for localization, which is 557x smaller than Bruteforce method.

The energy-time comparisons are illustrated in Figure 5. The overall comparison between CaPSuLe and the state-of-the-art Bruteforce matching algorithm on our platform and dataset are summarized in Table II. By sacrificing only 0.89% of the accuracy, CaPSuLe is 537 times faster in the response time and 557 times cheaper in energy consumption.

5.6 Discussion

It is widely assumed that cloud-based Machine Learning Solutions are the future. However, cloud-based applications are not ideal for the societal problem of sustainability and privacy. We have shown that by trading a small (insignificant) amount
of quality, modern machine learning solutions can be made private and sustainable, thus eliminating the need for the cloud. We capitalize on the cost-quality control provided by randomized hashing algorithms and demonstrate an end-to-end indoor camera-based positioning system CaPSuLe which can localize a mobile device, with 92.11% accuracy, in 1.92 seconds of local (on-device) computations consuming 3.78 Joules of energy, using a Samsung Galaxy S4 platform. With the ever increasing computational power of mobile devices, we believe that such cloud-independent private and sustainable solutions are the future of SoCs. We hope many works will follow this line of thought.
Chapter 6

Conclusion

Locality Sensitive Hashing (LSH) is an algorithm for approximate nearest neighbor (ANN) search in high dimensional space. In this thesis, instead of using LSH as an ANN tool, we investigate the possibility of using LSH for addressing the computational and memory challenges in large scale machine learning tasks. We show some rare ‘gems’ of locality-sensitive hashing that can shed important lights on large scale learning system.

We first showed that we showed that for the problem of unsupervised anomaly detection, we could leverage advances in probabilistic indexing and redesign a significantly efficient statistical measure. We proposed ACE algorithm, for unsupervised anomaly detection, which is 60-300x faster than existing approaches with competing accuracy. Our algorithm requires mere $4MB$ of memory which can utilize L3 caches of modern processors leading to fast-lookups. We believe ACE will replace existing unsupervised anomaly detection algorithms deployed in resource-frugal environments.

Then, we proposed MinSM, accelerating Split Merge MCMC via weighted Min-hash. The new split merge MCMC has constant time update, and at the same time, the proposal is informative and needs significantly fewer iterations than random split-merge. Overall, we obtain a sweet tradeoff between convergence and per update cost. Experiments with Gaussian Mixture Model on two real-world datasets demonstrate much faster convergence and better scaling to large datasets.
In the end, we showed a practical system for using LSH for indoor navigation. It is widely assumed that cloud-based Machine Learning Solutions are the future. However, cloud-based applications are not ideal for the societal problem of sustainability and privacy. We have shown that by trading a small (insignificant) amount of quality, modern machine learning solutions can be made private and sustainable, thus eliminating the need for the cloud. We capitalize on the cost-quality control provided by randomized hashing algorithms and demonstrate an end-to-end indoor camera-based positioning system CaPSuLe which can localize a mobile device, with 92.11% accuracy, in 1.92 seconds of local (on-device) computations consuming 3.78 Joules of energy, using a Samsung Galaxy S4 platform. With the ever-increasing computational power of mobile devices, we believe that such cloud-independent private and sustainable solutions are the future of SoCs. We hope many works will follow this line of thought.

6.1 Moving Forward

This thesis introduced some Locality sensitive hashing based techniques that can be used for large scale machine learning. There are some further opportunities using hashing techniques to improve current machine learning models and systems. The following are some of the interesting future directions.

**Distributed Hashing System** Hashing systems are widely used in real-world applications for nearest neighbor search [Wan+18]. Most of the existing hashing systems [Wan+18] are based on the single machine (e.g. one node with multi GPUs and CPUs). However, in real-world applications, the data are usually huge and stored
in a distributed fashion. A single machine-based hashing system is not able to deal with all these large scale distributed datasets. So design a distributed hashing system is a very exciting topic.

**Memory Efficient Data mining using Hashing** In modern computational applications, traditional machine learning and data mining tasks are suffered by the extreme large-scale data sets. Designing memory-efficient machine learning techniques that can deal with this amount of data is paramount. Hashing is a perfect technique to handle large scale datasets. So use hashing to design energy-efficient machine learning models is a great research topic.

**Hashing for Natural Language Processing** Natural language understanding/processing is one of the emerging research topics in computer science. The features extracted for building up natural language processing models are usually very high dimensional and sparse. Hashing especially Minwise hashing is designed for dealing with very large scale sparse data. So using hashing techniques to enhance the performance of the NLP models is a promising and exiting research direction.
Bibliography


[LL05] Kingsly Leung and Christopher Leckie. “Unsupervised anomaly detection in network intrusion detection using clusters”. In: Proceedings of the


Yiqiu Wang et al. “Randomized algorithms accelerated over cpu-gpu for ultra-high dimensional similarity search”. In: Proceedings of the 2018 In-