UNIVERSITY OF CALIFORNIA, SAN DIEGO

# Learning structure and concepts in data through data clustering

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science and Engineering

by

Gregory James Hamerly

Committee in charge:

Professor Charles P. Elkan, Chairperson
Professor Serge Belongie
Professor Garrison Cottrell
Professor Sanjoy Dasgupta
Professor Virginia de Sa
Professor Kenneth Kreutz-Delgado

2003

Copyright
Gregory James Hamerly, 2003
All rights reserved.

The dissertation of Gregory Hamerly is approved, and it is acceptable in quality and form for publication on microfilm:

---

---

---

---

---

---

Chair

University of California, San Diego

2003

TABLE OF CONTENTS


Signature Page . . . iii

Table of Contents . . . iv

Acknowledgements . . . vii

List of Figures . . . x

List of Tables . . . xiii

Vita and Publications . . . xiv

Abstract . . . xv

I Introduction . . . 1
A. Data clustering motivation . . . 1
1. A small example: customer database . . . 1
2. Types of applications . . . 3
3. Specific applications . . . 5
B. Terms and definitions . . . 6
C. Clustering algorithms . . . 6
1. Iterative optimization clustering algorithms . . . 6
2. Hierarchical clustering . . . 10
3. Spectral clustering . . . 12
4. Other clustering algorithms . . . 13
D. Object similarity and distance metrics . . . 14
E. Opportunities in data clustering . . . 17
F. Outline of the dissertation . . . 18

II Finding high-quality clustering solutions . . . 19
A. Finding high-quality solutions . . . 19
B. Center-based clustering . . . 23
1. General iterative clustering . . . 23
2. Initialization methods . . . 25
3. K-means . . . 27
4. Gaussian expectation-maximization . . . 28
5. Fuzzy k-means . . . 28
6. K-harmonic means . . . 29
C. New clustering algorithms . . . 31
1. Hybrid 1: hard membership, varying weights . . . 32

2. Hybrid 2: soft membership, constant weights . . . 33
D. Experimental setup . . . 33
1. Experiment 1: BIRCH . . . 35
2. Experiment 2: Pelleg and Moore data . . . 36
E. Experimental results . . . 36
1. Experiment 1: BIRCH . . . 36
2. Experiment 2: Pelleg and Moore data . . . 42
F. Conclusion . . . 43
G. Acknowledgements . . . 44

III Estimating the number of clusters . . . 47
A. Introduction and related work . . . 47
B. The Gaussian-means (G-means) algorithm . . . 49
C. Testing clusters for Gaussian fit . . . 51
1. Constructing the test . . . 53
2. Performing the test for normality . . . 54
3. Adjusting significance for multiple tests . . . 56
4. Finding the critical values of the test . . . 56
5. Splitting initialization . . . 59
D. Experiments . . . 60
1. A small example . . . 60
2. Synthetic datasets . . . 61
3. Architecture datasets . . . 71
4. Discovering true clusters in labeled data . . . 72
5. Learning true dimension from clustering . . . 73
E. Statistical power and dimensional effects . . . 77
1. Statistical power . . . 78
2. High-dimensional effects . . . 80
F. Conclusion . . . 81
G. Acknowledgements . . . 83

IV Learning and predicting program behavior through clustering . . . 89
A. Introduction . . . 89
1. Choosing the best $k$ can be application-specific . . . 90
2. Background on processor simulation . . . 90
3. Capturing program structure . . . 93
B. Basic block vectors . . . 95
1. Basic block vectors . . . 95
2. Basic block vector difference . . . 99
3. Basic block difference graph . . . 100

4. Basic block similarity matrix . . . 102
C. Clustering . . . 109
1. Previous method for clustering basic block vectors . . . 109
2. Clusters and phase behavior . . . 110
3. Phase-finding algorithm . . . 111
D. Automatically finding simulation points . . . 117
1. Single simulation points . . . 119
2. Multiple simulation points . . . 121
E. Related work . . . 124
1. Time varying behavior of programs . . . 124
2. Training inputs and finding smaller representative inputs . . . 125
3. Fast forwarding and check-pointing . . . 126
4. Automatically finding where to simulate . . . 127
5. Statistical sampling . . . 127
6. Statistical simulation . . . 128
F. Summary . . . 128
G. Acknowledgements . . . 130

V Improvements and validation of program behavior prediction . . . 131
A. Introduction . . . 131
B. Methodology . . . 134
C. Early SimPoints . . . 135
1. Need for improvements over the original SimPoint algorithm . . . 136
2. Early SimPoint algorithm . . . 138
3. Early SimPoint results . . . 140
D. Statistical validation . . . 145
1. Statistical validation of simulation points . . . 145
2. Statistical validation of systematic sampling . . . 147
3. Discussion . . . 148
E. Using error bounds to choose clusterings . . . 149
1. Structured sampling using clustering . . . 150
2. Experiments . . . 151
3. Discussion . . . 154
F. Architecture independence . . . 155
G. Summary . . . 157
H. Acknowledgements . . . 158

VI Conclusion . . . 159

A Proofs pertaining to the G-means statistic . . . . . 162
A. G-means statistic distribution . . . . . 162
B. A distribution with higher expected distortion . . . . . 169

Bibliography . . . . . 172

# ACKNOWLEDGEMENTS

First I want to thank Charles Elkan, with whom I have enjoyed working and who is a great advisor. Thank you for your curiosity, patience, and encouragement. I have been very grateful to have you as a counselor and advisor. Serge Belongie, Brad Calder, Gary Cottrell, Sanjoy Dasgupta, Virginia de Sa, and Ken Kreutz-Delgado have all provided guidance and inspiration for interesting research. Thanks to my fellow students along the way: Bianca Zadrozny for encouragement and interesting discussions, Sameer Agarwal for fun, math, and eclecticism, Kristin Branson for encouragement and discussions on life goals, Dana Dahlstrom for sanity and shared hours of grading, Dave Kauchak for his ever-present smile, Eric Wiewiora for origami, Victor Gidofalvi for discussions on cycling and running, Ari Frank for his humor and hospitality, and Matt Dailey for welcoming me into the lab and showing the way through. Thanks to Erez Perelman, Tim Sherwood, Eric Tune and the rest of the Architecture lab for very interesting conversations and collaborations. Thanks to John Bellardo for friendship and keeping me sane my first year. I have derived a lot of pleasure from the ACM programming contest with fellow contestants and judges: John Rapp, Don Yang, Jeremy Lau, Stef Schoenmackers, Ben Ashbaugh, Mark Baysinger, and Geoff Voelker. Mr. J.L. Moore and his nephew Kevin Cavanagh deserve thanks for generously funding my education through the J.L. Moore Fellowship.

I have spent a great deal of time with friends from the UCSD Graduate Christian Fellowship. Thank you to Liam Palmer for kindness and great discussions, John Seng for teaching me more about juggling and for music, Laurie Rice for determination and for not succumbing to life's onslaughts, Christian Thomas and Kyle Beardsley for scaling large mountains with me, and to the rest of the people who have encouraged and prayed for me. Thanks to some friends who defy classification: Tycho Nightingale for discussions, encouragement, and helping me see more of San Diego than I have in 20 years, Ronnie Cheung for friendship and fun projects, and Mike Copenhafer for laughter, music, and counter-culture. To my too-numerous friends from San Luis Obispo and high school: Jim Shevlin, Jennifer Hellinger, Bret and Andrea Rooks, Jason Junkert, Jim and Mandy Richards, Josh and Amy Butz, Jason and Amber Ficht, Scott and Lisa Tyndall, Ryan Miller, Noelani Francis, Donna Fuller, Dan and Deb Cestone, Cecilia Ybarra, and everyone else, thanks for true friendship; I am beginning to learn how precious it is.

To my parents Jim and Peggy Hamerly: thank you so much for your constant encouragement and ridiculous amounts of patience. Dad, thank you for allowing me to discuss research ideas with you and for providing me with so many opportunities to improve in academics. Mom, thank you for calling me to let me know what's going on and that you were thinking of me, and for all your support. Mémé, Kim, Sue, and Tim, thank you for your love and interest in my life. I have lost both of my grandfathers during my tenure at UCSD; I'm thankful for having known them.

Finally, thank you to Ivy Orr (soon to be Hamerly), with whom I have shared the best and

worst of graduate school and with whom I am deeply in love. You are so fun, kind, and wise. I'm so glad that you have come into my life, and I look forward to the next 70 years with you. Steve and Pattie, thank you for entrusting me with your daughter, and for support throughout our relationship.

I have found in life that one should be careful with the word "never". Life, in ironic and awesome ways, has changed many of the things I thought I would "never" do into things that I now have done or will do, as a result of graduate school. I once thought I would never marry a Texan, much less move to Texas. I thought I would never be able to juggle five balls. I thought I would never go to the world finals in the programming contest. I thought I would never study artificial intelligence or do research in computer architecture. I am very grateful that I have been given the chance to do all these "nevers" through graduate school.

Psalm 111:10
The fear of the Lord is the beginning of wisdom;
all who follow his precepts have good understanding.
To him belongs eternal praise.

A mathematician is a device for turning coffee into theorems.
Paul Erdös

A computer scientist is a device for turning mountain dew into programs.
Ivy Orr

LIST OF FIGURES


I.1 Relationship between salary and average purchase price of a company. . . . . 2
I.2 Three projections of 10-dimensional data . . . . . 4

II.1 An illustration of a locally optimal, poor-quality clustering solution. . . . . 21
II.2 The $k$-harmonic means membership function. . . . . 31
II.3 The $k$-harmonic means data weight function. . . . . 32
II.4 Random sampling and random partition initializations . . . . . 35
II.5 Locally optimal, poor-quality $k$-means clustering solution. . . . 37
II.6 Convergence on the BIRCH dataset for Gaussian EM and $k$-means. . . . . 38
II.7 Convergence on the BIRCH dataset for $k$-harmonic means and fuzzy $k$-means. . . . . 39
II.8 Convergence on the BIRCH dataset for the Hybrid 1 and Hybrid 2 algorithms. . . . . 40
II.9 Convergence curves starting from random sampling and random partition initializations on 2-d synthetic data. . . . . 41

III.1 Two clusterings where $k$ was improperly chosen for the dataset being clustered. . . . . 48
III.2 The projection to one dimension preserves relevant information for performing a hypothesis test for normality. . . . . 55
III.3 Anderson-Darling statistic CDF. . . . . 58
III.4 An example of running G-means for three iterations on a 2-dimensional dataset with two true clusters and 1000 points. . . 60
III.5 An example where BIC overfits, while G-means selects the correct $k$. . . . . 71
III.6 Correspondence between clusters and true labels on the NIST handwritten digit recognition dataset. . . . . 74
III.7 Correspondence between clusters and true labels on the Pendigits handwritten digit recognition dataset. . . . . 75
III.8 Results from learning the dimension on data without noisy features. . . . . 77
III.9 Results from learning the dimension on data with noisy features. 78
III.10 Results from learning the dimension on the high-dimensional architecture dataset `gzip`. . . . . 79
III.11 Results from learning the dimension on the high-dimensional NIST dataset. . . . . 80

III.12 The power of the two hypothesis tests versus the BIC on spherical data. . . . . 85
III.13 The power of the two hypothesis tests versus the BIC on elliptical data. . . . . 86
III.14 Distribution of points in the one-dimensional random projection of data drawn from one spherical Gaussian. . . . . 87
III.15 Distribution of points in the one-dimensional random projection of data drawn from one spherical Gaussian. . . . . 88

IV.1 Basic blocks in assembly code . . . . . 96
IV.2 Two types of normalization for basic block vectors . . . . . 98
IV.3 A sample of basic block vector structure . . . . . 98
IV.4 Basic block difference graphs for the programs `wave`, `gzip-graphic`, `bzip2-graphic`, and `gcc-166` . . . . . 101
IV.5 Basic block similarity matrix for the programs `gzip-graphic` and `bzip-graphic`. . . . . 105
IV.6 Time varying IPC for `gzip-graphic`. . . . . 106
IV.7 Cluster graph for `gzip-graphic`. . . . . 106
IV.8 Basic block similarity matrices for original and projected data for `gcc`. . . . . 107
IV.9 Time varying IPC graph for `gcc-166`. . . . . 108
IV.10 Cluster graph for `gcc-166`. . . . . 108
IV.11 Motivation for projecting basic block vectors to 15 dimensions. 114
IV.12 Two-dimensional projection and clustering of `gzip`. . . . . 115
IV.13 Average IPC variance and max IPC variance versus the BIC score. 118
IV.14 Results for single simulation points. . . . . 119
IV.15 Multiple simulation point results. . . . . 122
IV.16 Average error results for the SPEC 2000 floating point and integer benchmarks. . . . . 125

V.1 CPI relative error for SimPoint, Early SimPoint, Systematic Sampling with 100 samples and 1000 samples. . . . . 143
V.2 Number of simulation point found for each program under the Original and Early SimPoint methods. . . . . 143
V.3 Fast-forwarding needed for simulating Original and Early SimPoint. . . . . 144
V.4 Time required to complete a simulation for each program using Checkpointing, Original and Early SimPoint, and Sampling. . . 144
V.5 True CPI error and estimated error bounds for Early SimPoint, 100 systematic samples, and 1000 systematic samples. . . . . 147

V.6 The percent error bound for one estimate of CPI, averaged over 18 programs. . . . . 152
V.7 The percent error bound for one estimate of CPI for `gzip-log`. 153
V.8 The true and estimated IPC for the program `gcc` for 20 different architecture configurations. . . . . 157

A.1 The function showing the minimum-distortion splitting point for Gaussian data. . . . . 168

LIST OF TABLES


I.1 Terms and definitions used in this dissertation. . . . . 7

II.1 Quality of solutions for one run on the BIRCH dataset. . . . . 41
II.2 Competition matrix for 2-d data starting from random sampling and random partition initializations. . . . . 45
II.3 The ratio of the $k$-means quality (versus the optimum) over 100 datasets in two dimensions. . . . . 46

III.1 Critical values of the Anderson-Darling statistic. . . . . 59
III.2 Synthetic spherical data, separation = $1.5\sigma$ . . . . . 62
III.3 Synthetic elliptical data, separation = $1.5\sigma$ . . . . . 63
III.4 Synthetic spherical data, separation = $3\sigma$ . . . . . 64
III.5 Synthetic elliptical data, separation = $3\sigma$ . . . . . 65
III.6 Synthetic spherical data, separation = $6\sigma$ . . . . . 66
III.7 Synthetic elliptical data, separation = $6\sigma$ . . . . . 67
III.8 Results of estimating the number of clusters on computer architecture datasets. . . . . 84
III.9 Anderson-Darling statistics for 1-dimensional projected Gaussian data . . . . . 86

## VITA

| | |
|---|---|
| November 16, 1977 | Born, Rochester, New York, USA |
| Summer, 1985 | Moved to San Diego, California |
| June, 1999 | B.S., California Polytechnic State University, San Luis Obispo |
| June, 2001 | M.S., University of California, San Diego |
| June, 2003 | Doctor of Philosophy,<br>University of California, San Diego |

## PUBLICATIONS

"Bayesian approaches to failure prediction for disk drives." In proceedings of the eighteenth international conference on machine learning (ICML 2001), pp. 202–209, June, 2001.

"Alternatives to the k-means algorithm that find better clusterings." In proceedings of the eleventh international conference on information and knowledge management (CIKM 2002), pp. 600–607, November, 2002.

"Automatically characterizing large scale program behavior." In the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), October, 2002.

"Learning the $k$ in $k$-means." Submitted for publication.

"Early Simulation Points with Statistical Validation." Submitted for publication.

ABSTRACT OF THE DISSERTATION

Learning structure and concepts in data using data clustering

by

Gregory James Hamerly
Doctor of Philosophy in Computer Science and Engineering
University of California, San Diego, 2003
Professor Charles P. Elkan, Chair

Data clustering is an important and applications-oriented branch of machine learning. Its goal is to estimate the structure or density of a set of data without a training signal. There are many approaches to data clustering that vary in their complexity and effectiveness, due to the wide number of applications that these algorithms have. Due to the explosive growth of the amount of data that humans want to analyze, fast (e.g. linear-time) algorithms are necessary, but they can often give poor quality results.

While maintaining the runtime characteristics of the fast algorithms, we show modifications that improve clustering algorithms in two ways. The first focus is on finding better solutions for a fixed number of clusters. We decompose the algorithms into fundamental parts, and analyze how the parts affect the quality of clustering solutions. The second focus is on estimating the number of clusters efficiently using statistical hypothesis tests, and how that may be applied in novel ways.

We also discuss the application of data clustering to the task of learning the structure of computer programs. We show how clustering may be used to improve the accuracy of computer processor simulations while simultaneously improving their efficiency.

# I

# Introduction

Data clustering is the task of grouping objects which are similar, while not grouping objects that are not similar. This is a very broad definition, for we have not defined what we mean by "object", "grouping", or "similar". However, this shall be the overarching definition we will use to begin our discussion.

## I.A Data clustering motivation

Let us begin this thesis by motivating data clustering with some examples of why we have a need to do it.

### I.A.1 A small example: customer database

A fictitious company has a large customer database which has thousands of customers with data on the purchasing history of each customer, as well as demographic data describing each customer (e.g. zip code, annual salary, and age). The company wants to better understand the types of customers it has, so as to improve its services and products. One task the company wishes to do is partition the customer database into several groups, where customers in each group have similar purchasing history and demographics, and thus discover which

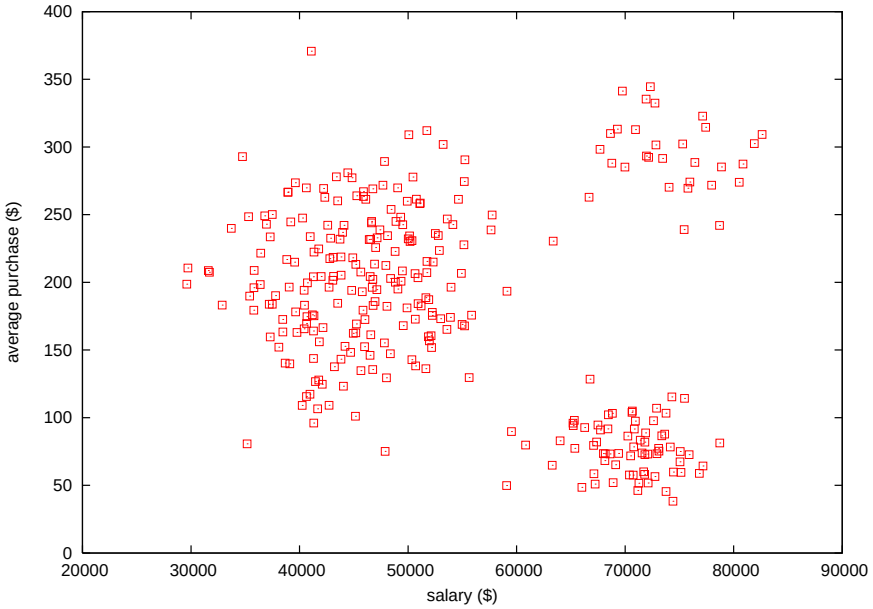


Figure I.1: This plot shows the relationship between salary and average purchase of customers of a company. It would be valuable information for the company to discover that there are three groups of people here: those with moderate incomes, those with higher incomes who spend little, and those with higher incomes that spend more.

types of customers they attract, and which types they may be missing out on. For example, one interesting group may be of customers with high income who, on average, do not spend much money with the company. Another interesting group may be customers who live far away from the company but who purchase more frequently than the average customer. Finding these groups may be easy for humans to accomplish if the data consists of only 1 to 3 values per customer. Two such relevant attributes about a customer may be (for example) the average amount spent per purchase with the company and the salary. Figure I.1 shows a plot that a human may examine to determine different clusters in a plot of average purchase versus salary.

However, if there are more than three attributes it becomes very difficult for humans to manually find groups in the data. One heuristic approach that a dedicated analyst may take is to plot each pair of attributes, and examine each plot, looking for separations among the groups. As dimensionality grows, the number of such plots grows by the square of the number of attributes. In [34], Hubert explains how this technique would prevent most analysts from having the patience to examine more than 10 attributes in this way, which would require the examination of $\binom{10}{2} = 45$ individual plots. In addition, this type of examination would only allow the analyst to search for relationships that exist between pairs or triples of variables; he or she could not effectively search for relationships that exist between more than three variables because of the difficulty of visualizing beyond three dimensions. Further, this technique of examining pairwise attributes is not as general as looking for general projections which show interesting relationships, and there are infinitely many projections in multivariate data. See Figure I.2 for an illustration of the difficulty of visualizing data in higher than two dimensions.

Such a customer database as we have been discussing may have hundreds of attributes stored for each customer. The problem becomes intractable for manual analysis, so we need an automatic way of finding interesting groups. Data clustering provides an answer.

### I.A.2 Types of applications

Data clustering is a central task in artificial intelligence. Humans are very good at finding clusters in low dimensions with small amounts of data, but it is very difficult to instruct a computer to find such relationships. Because it is difficult for computers, yet easy for humans in small amounts, it is an interesting problem to solve from an AI perspective. Here are some classes of data clustering

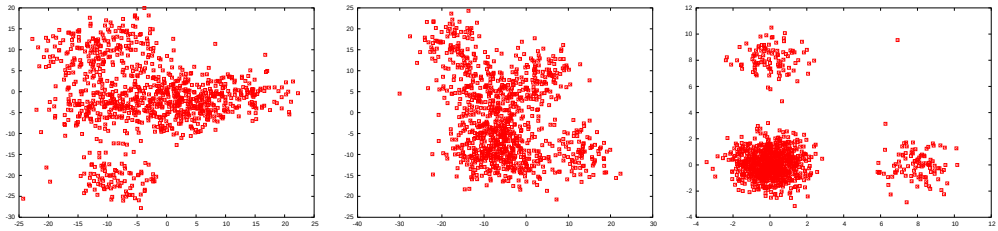

Figure I.2: These three plots show three views (two-dimensional projections) of the same 10-dimensional data. There are 10 well-separated clusters in this data, each lying 8 units away from the origin along its own dimension. No two-dimensional projection will illustrate all the clusters well. The left and middle plots show the data projected onto two different randomly chosen planes. The right plot shows a better projection, in which we are able to make out only three of the 10 clusters. This example illustrates how difficult it is for humans to find interesting structure in even a moderate number of dimensions.

applications.

**Data compression**: In data clustering, each cluster has some set of objects that belong to that cluster. If we do not need to store all of the objects but can afford to represent the set with some description about the whole set of objects, then each cluster's objects may be replaced with a set of descriptions (e.g. the number of objects and the boundary of the set of objects).

**Reduction of nearest-neighbor search space**: A common type of query in databases is searching for the database object nearest to some query object. If we cluster the data in the database before any queries, then we can do a two-level search which can be faster; searching for first the nearest cluster, and then doing a local search for the nearest object in that cluster.

**Unsupervised classification**: In contrast to the task of supervised classification, the task of unsupervised classification is to give labels to objects without knowing the true labels. A clustering algorithm that partitions a dataset

into several groups is naturally performing a classification task, where the labels are unknown.

**Learning the probability density of data**: If some data is assumed to be drawn from a mixture of distributions, we can use clustering to estimate the probability density of the data. The typical model for this is that each partition of the data is represented by a unimodal probability density model. Summing of all the cluster models gives a multimodal density estimator for the entire dataset.

### I.A.3 Specific applications

Underneath these general application topics just discussed, we would like to elaborate on some specific and important tasks in which data clustering plays an important role.

**Image color quantization**: Image compression can come in several forms, one of which is reducing the number of colors used to encode the image. With data clustering, we can find the $k$ best colors for representing the entire image, quantizing each original color in the image into one of $k$ code colors.

**Discerning between cancer populations** One of the largest sources of interesting data that is difficult to analyze comes from the field of genomics. Genomics is interested in answering the questions of what can be learned and predicted about living organisms based on their gene expressions. Clustering can find complex relationships within populations of gene expression [3].

**Image segmentation for computer vision**: One of the most important issues in building intelligent, autonomous systems is that of image understanding. In order to understand an image, the first thing a computer must do is segment the image into several parts. In a satellite image, we may want to divide the image automatically into buildings, water, forest, and agriculture. Data clustering plays an integral role in image segmentation algorithms.

**Anomaly detection**: In quality control applications, rare events are the interesting ones. Computer manufacturers have an interest in predicting hard disk drive failure before it happens, so that its data may be backed up and the drive replaced smoothly. In this application, we may use clustering to learn a probability model of the data of how hard drives normally operate, and then can use this probability model to predict how "normally" other hard disk drives of the same type are operating [29].

## I.B Terms and definitions

In Table I.1 we define the terminology we will use to describe and analyze the algorithms and mathematical constructs used in this dissertation.

## I.C Clustering algorithms

This section will describe and give examples of three important categories of data clustering algorithms: iterative optimization, hierarchical, and spectral. After these three categories, I will review several other important algorithms which fall outside these common categories.

### I.C.1 Iterative optimization clustering algorithms

A large number of clustering algorithms are based on iterative optimization, such as the popular $k$-means and Gaussian Expectation-Maximization algorithms. These algorithms begin with a solution, and then repeatedly improve the solution until no further (local) improvements can be made. The initialization of these algorithms is a very important component, though it is often overlooked by practitioners. We will look at various issues related to initialization later.

Table I.1: Terms and definitions used in this dissertation.

| | |
|---|---|
| object, datapoint | the atomic element of clustering, multiples of which are grouped or clustered together |
| $n$ | number of objects in a dataset |
| $k$ | number of clusters |
| $d$ | dimension |
| $\mathbb{R}$ | set of all real numbers |
| $X$ | dataset to be clustered |
| $x_i$ | datapoint belonging to $X$ |
| $D$ | pairwise distance matrix |
| $\in$ | set membership |
| $\bigcup$ | set union |
| $\langle \cdot, \cdot \rangle$ | vector dot product |
| $\|\| \cdot \|\|_\alpha$ | vector norm ($L_\alpha$ norm); if unspecified, $\alpha = 2$. |
| $\Sigma$ | covariance matrix |
| $\sigma$ | standard deviation or kernel width |
| $C$ | set of $k$-means centers |
| $c_j$ | $k$-means center belonging to $C$ |
| $\mu_j$ | Gaussian EM center |
| $I(\alpha)$ | the indicator function on predicate $\alpha$ |
| $\delta(\cdot)$ | the delta function |
| $\Phi(\cdot)$ | the cumulative distribution function of the normal distribution |

In general these algorithms do not give any guarantees of the global optimality of their solutions. They claim that their answers will be a *locally* optimum solution. Quite often this locally optimum solution is not the globally optimum solution, which leaves room for the research into algorithmic improvements which I give in this dissertation.

The iterative optimization clustering algorithms we will consider here have time complexity between $O(nkd)$ and $O(nkd^2)$ per iteration (except for $k$-medoids, which is $O(n^2k)$ per iteration). The number of iterations required can vary a lot depending on the initialization and the structure of the data, but a rule of thumb is that it is very roughly logarithmic in $n$, the number of points.

### The $k$-means algorithm

The $k$-means algorithm [47] is a very popular algorithm for data clustering, since it is very simple to implement, it is fast, and it is fairly easy to understand. Originally developed for and applied to the task of vector quantization, $k$-means has been used in a wide assortment of applications. Much of the work in this dissertation is based on the $k$-means clustering algorithm. Algorithm 1 shows a pseudocode description of the $k$-means algorithm.

### The Expectation-Maximization algorithm

The expectation-maximization algorithm is a very general algorithm for parameter estimation in the case where certain data are unknown. The original unifying paper [21] by Dempster, Laird, and Rubin formed the basis for a wide array of new research into algorithms in parameter estimation, closely related to machine learning.

The two basic ideas of expectation-maximization, or EM, are given in the name. EM underlies a class of algorithms in which there are two steps:

**Algorithm 1** Pseudocode for the $k$-means algorithm. Inputs to the algorithm are $k$ (the number of centers), $X$ (the $n$ datapoints in $d$ dimensions), and the initial locations of the centers $C = \{c_j\}$.

**kmeans**$(X \in \mathbb{R}^{n \times d}, k, C)$

1: **while** the any $c_j$ change location **do**
2: **for** $i \in \{1 \dots n\}$ **do**
3: $class(x_i) \leftarrow \arg\min_j ||x_i - c_j||$
4: **end for**
5: **for** $j \in \{1 \dots k\}$ **do**
6: $c_j \leftarrow \sum_i I(class(x_i) = j) x_i / \sum_i I(class(x_i) = j)$
7: **end for**
8: **end while**
9: return $C$

- An *expectation* step, in which the values of the known data are used to compute the expected values of the unknown data (also called an E step).
- A *maximization* step, in which the known and expected values of the known and unknown data are used to estimate the parameters of the model which will maximize the likelihood of the model given the data (also called an M step).

These two steps are preceded by an initialization of the model (similar to $k$-means). The algorithm then repeats an E step followed by an M step until no further improvement may be made (either the likelihood of the model does not change, or the expected values of the unknown data do not change). Pseudocode for the Gaussian EM algorithm is given in Algorithm 2.

In the context of clustering, we may ask what the unknown data are. They are the cluster memberships. If we believe that there are $k$ clusters in

dataset $X$, then the EM model assumes that each $x_i \in X$ was generated by one of the $k$ clusters (but we don't know which one, or where any of the clusters truly exists). Therefore the cluster memberships, which define which cluster generated each point, are the unknown data. In addition to the cluster memberships, the missing data may also include missing measurements in some $x_i$, but that is not as central to the topic of finding good-quality clusters.

#### The $k$-medoids algorithm

This iterative algorithm is similar in approach to $k$-means, but it imposes an additional constraint: that the centers that are used to represent the data are taken from the dataset itself. Thus a "medoid" is a datapoint that best represents a set of data. Because of this constraint, $k$-medoids can operate on data which do not live in a metric space, as long as the data can be described in terms of pairwise distances for the datapoints being clustered. However, the initial construction of the pairwise distance matrix $D$ requires time $O(dn^2)$ and the search for a new medoid (each iteration) takes time $O(n^2)$. Therefore, it is not a linear-time algorithm for clustering, and is of little practical interest for moderately-sized and large datasets. Pseudocode for $k$-medoids is given in Algorithm 3.

### I.C.2 Hierarchical clustering

Another approach to data clustering is hierarchical clustering. Algorithms in this class use heuristic splitting and merging functions to either build a cluster tree (or *dendrogram*) from the top down or the bottom up. Algorithms which build from the top down are called divisive, while the more common bottom-up algorithms are called merge-based. Hierarchical clustering algorithms are popular for applications where it is beneficial to see the structure of the data at many levels of granularity, as in clustering gene expression data.

**Algorithm 2** Pseudocode for the Expectation-Maximization algorithm for a mixture of Gaussian distributions. Inputs to the algorithm are $X$, $k$, and the initial values of $p(j), \{\mu_j\}$, and $\{\Sigma_j\}$. The function $p(x_i|\mu_j, \Sigma_j)$ is the Gaussian probability density function, and the term $p(x_i)$ can be obtained from summing over $j$ the values $p(x_i|\mu_j, \Sigma_j)p(j)$.

**Gaussian-EM(**$X \in \mathbb{R}^{n \times d}, k, p(j), \{\mu_j\}, \{\Sigma_j\}$**)**

**while** the likelihood $\mathcal{L}(\{\mu_j\}, \{Sigma_j\}|X)$ changes **do**
  *// Expectation step*
  **for** $i \in \{1, \ldots, n\}$ **do**
    **for** $j \in \{1, \ldots, k\}$ **do**
      $p(j|x_i) \leftarrow p(x_i|\mu_j, \Sigma_j)p(j)/p(x_i)$ *// Bayes' rule*
    **end for**
  **end for**
  *// Maximization step*
  **for** $j \in \{1, \ldots, k\}$ **do**
    $\mu_j \leftarrow \sum_i^n p(j|x_i)x_i / \sum_i^n p(j|x_i)$
    $p(j) = \sum_i p(j|x_i)/n$
    **for** $l, m \in \{1, \ldots, d\}$ **do**
      $\Sigma_{jlm} \leftarrow 1/n \sum_i^n p(j|x_i)(x_{il} - \mu_{jl})^T(x_{im} - \mu_{jm})$
    **end for**
  **end for**
**end while**
return $(\{\mu_j\}, \{\Sigma_j\})$

**Algorithm 3** Pseudocode for the $k$-medoids algorithm. Inputs to this algorithm are $D$, the $n \times n$ matrix of distances between all pairs of datapoints, the number of medoids desired $k$, and $M$, the initial medoids.

**kmedoids**($D \in \mathbb{R}^{n \times n}, k, M$)

```
while any medoid m_j changes do
    for i ∈ {1...n} do
        class(i) ← arg min_j D_ij
    end for
    for j ∈ {1...|M|} do
        m_j ← arg min_i I(class(i) = j) D_ij
    end for
end while
return M
```

In general, since hierarchical clustering algorithms operate on a pairwise distance matrix of size $n \times n$, they require time $O(n^2 d)$ per merge or split, and when constructing the full cluster tree, will require $n$ merges or splits, for a total time of $O(n^3 d)$.

### I.C.3 Spectral clustering

A significant shift in clustering techniques has recently been developing that comes from the area of spectral graph theory. New spectral clustering algorithms search for globally optimal cuts in a graph representing the data to be clustered. The graph $G$ can be thought of as analogous to a pairwise distance matrix, though the edges in the graph need not be metric distances.

Spectral clustering algorithms have been developed by computer vision researchers, as well as by graph theorists. They have been motivated from several

perspectives, including optimal graph cuts and random graph walks. As such, there are several different algorithms.

A large advantage to spectral clustering is the ability to form clusters of arbitrary shapes. This is because the graphs are fully connected, and do not assume an underlying model for any cluster. Therefore any datapoint may be considered connected with any other point.

While spectral clustering algorithms have some nice properties, they are difficult to use for two primary reasons: their running time is $O(n^3 + n^2 d)$ for clustering $n$ datapoints in $d$ dimensions, which is rather expensive and unacceptable for even moderately large $n$. The Nyström method [26] enables faster approximations to the full algorithm of finding a graph cut. The Nyström method basically subsamples the data so that the spectral clustering algorithm may be run on the smaller dataset (with the same running time), and then infers the cluster memberships for the culled datapoints based on the distance matrix and the cluster memberships of the sample that was clustered.

A second difficulty with spectral clustering is the requirement of a kernel width parameter, called $\sigma$. This parameter must be set beforehand, and it determines how local will be the algorithms' focus. If $\sigma$ is small, then the algorithm will only group datapoints which are actually very close to one another in the input space. If $\sigma$ is large, then the algorithm will find larger structure in the data. It is not clear how to set $\sigma$ *a priori* for a given dataset, nor is it clear how to learn the correct $\sigma$ efficiently, since running the algorithm would require $O(n^3)$ time for each value of $\sigma$.

### I.C.4 Other clustering algorithms

Several other types of clustering algorithms which do not fit into the above categories deserve mention. The mean shift algorithm [15] is an interesting

iterative algorithm which finds arbitrarily shaped clusters, and has been successfully used to analyze image features (performing tasks like image segmentation and edge-preserving blurring). The main idea behind mean shift is to begin with a number of kernel density estimators in the input space, and repeatedly move the estimators toward areas of higher density. This is somewhat similar to the maximization step of $k$-means and Gaussian EM, but in mean shift, there is no notion of kernel density estimators competing for the ownership of the input datapoints. Once the kernels have all reached stable points, the algorithm groups those kernels which have end up near one another, and segments the data based on where each kernel started. The difficulties of this method come in defining the post-processing heuristics used to group kernels (and data points), and the determination of the width of the kernel, which is a similar problem to that of determining $\sigma$ in spectral clustering.

Barbará *et al.* have tackled clustering from unique perspectives, such as using the fractal dimension [54] to determine which portions of data look self-similar, and they achieve linear run-time with this algorithm [4]. They have also done work in categorical clustering, which is a very important class of clustering algorithms where the points being clustered do not have a natural metric, so there is not a notion of distance between datapoints. Instead, they use a measure of entropy to develop an iterative clustering algorithm [5].

## I.D Object similarity and distance metrics

One concept which must be formalized when discussing clustering is the notion of object similarity, which is usually given in the form of a distance metric. Both clustering algorithms and instance-based learners (such as $k$-nearest neighbors) require a measure of object similarity in order to function. This underlying distance measure can even be considered an input to a clustering algorithm. Even

algorithms which do not directly compute distances (such as $k$-medoids and normalized cut) still use pairwise distance matrices, which must be computed in advance using some distance metric.

Most commonly each object being clustered is a vector of real numbers. In this case a natural distance metric is the Euclidean distance, or $L_2$ distance. This straight-line distance is:

$$||x_i||_2 = \sqrt{\sum_{m=1}^{d} x_{im}^2}$$

More generally, we can create any number of distance metrics by replacing the squared and square-root terms in the Euclidean distance metric with $\alpha$ and $1/\alpha$:

$$||x_i||_\alpha = \left( \sum_{m=1}^{d} |x_{im}|^\alpha \right)^{1/\alpha}$$

When $\alpha = 1$, this metric is the sum of the differences in each dimension. This is also called the Manhattan distance, and related to the Hamming distance in which case all the attributes are binary.

In high dimensions, clustering data using $L_\alpha$ norms becomes difficult because notions of near and far distances become weaker. Since the metric is based on a sum over all dimensions, as the dimensionality increases the distance between two objects tends to increase. Thus all objects become "far apart", making it more difficult to build algorithms which use distances to determine similarities. Aggarwal *et al.* have shown [2] that it can be beneficial to use the non-intuitive $L_\alpha$ norms for $0 < \alpha << 1$. While their results are somewhat promising, they introduce other problematic issues of numeric stability for taking the $\alpha^{th}$ root of a number for very small $\alpha$.

The Mahalanobis distance allows the allows the user to specify a distance

that is weighted by a covariance matrix.

$$||x_i||_{mahalanobis} = \left((2\pi)^d|\Sigma|\right)^{-1/2} \exp(x_i^T \Sigma^{-1} x_i/2)$$

This is the distance measure that is often used in Gaussian expectation-maximization clustering.

Another type of metric is the vector dot product, also known as the cosine distance. This is not a distance but actually a similarity metric, so that more positive values indicate similarity. The vector dot product is the sum of the *product* of each attribute from two vectors being compared. Here we present a normalized version of the metric so that the dot product is always a value between -1 and 1:

$$\langle x_i, x_j \rangle = \frac{\sum_{m=1}^{d} x_{im} x_{jm}}{||x_i|| ||x_j||}$$

The vector dot product does not measure the difference in magnitude between two vectors; only the difference in the angle between the vectors. If the normalized $\langle x_i, x_j \rangle = 1$, the two vectors are collinear and in pointed in the same direction, if the normalized $\langle x_i, x_j \rangle = -1$, the two vectors are collinear and pointed in opposite directions, and if $\langle x_i, x_j \rangle = 0$, then the two vectors are orthogonal. We can convert this similarity measure to a "distance" by subtracting it from one.

Vector dot products are more useful in high-dimensional data where the presence or absence of an attribute is more important than the magnitude of each attribute. For example, the vector dot product is highly useful in measuring document similarity when the vector describing a document is a list of boolean attributes, each attribute indicating the presence/absence of a certain word. If two documents are similar in this very simple model, then they will have many words in common, and a vector dot product that is close to 1. However, if they share very few words in common, then the documents will have a vector dot product close to zero (we could consider them orthogonal documents).

## I.E Opportunities in data clustering

The class of iterative optimization algorithms for data clustering have been very popular both in research and application. They are simple to implement, fast to execute (normally requiring time that is linear in the size of the dataset), and can provide very meaningful and interpretable results. Even in the company of more sophisticated algorithms (such as spectral clustering algorithms), iterative optimization clustering algorithms play integral roles, and are still important. Spectral clustering algorithms actually use iterative clustering as the final step of estimating the cluster memberships. Therefore, any improvements which we can make to iterative optimization algorithms will have benefits in many other areas as well.

There are five issues we will address which give opportunities for further research in iterative optimization clustering algorithms:

1. Iterative optimization clustering algorithms require initializations. With two different initializations, $k$-means may give two very different solutions. This sensitivity to initial conditions is something we would like to remove or reduce.

2. Clustering algorithms which do local search to optimize a function often find solutions which are far from the global optimum. This is a very common problem which practitioners often answer by simply restarting with different random initializations. If we can construct algorithms which are better able to bypass local optima in search of the global optimum, this will be a significant step.

3. Clustering algorithms require the user to choose the number of clusters before running the algorithm, and the number of clusters is often unknown. Ideally we would like a clustering algorithm which finds the number of clus-

ters automatically, in a way that is justified by the data.

4. The "curse of dimensionality" makes it difficult to do clustering or instance-based learning in high dimensions. The problem of improving clustering algorithm behavior on high-dimensional data is a very interesting one. This problem is important in most areas of machine learning.

5. Clustering can be applied to a wide array of of research. It is our desire find research areas in which the use of clustering can aid in scientific analysis.

## I.F Outline of the dissertation

In this dissertation I will explain the contributions I have made toward the fields of data clustering and machine learning. In this chapter I have already covered many of the basic algorithms and the opportunities for further research. Chapter II will address the problem of poor-quality local optima. My research into this topic includes identifying the algorithmic components that lead to finding better-quality cluster solutions.

Chapter III will discuss the work I have done on the issue of finding the number of true clusters in a dataset. This research addresses the task of estimating $k$, but as we will see, it also addresses the task of finding good-quality clusters.

Chapters IV and V will demonstrate my work in the application of data clustering to the important task of improving computer architecture simulation. This is a very interesting problem from the perspectives of both the computer architecture and machine learning communities. This application deals with the problem of clustering with extremely high dimensional data, and it is a good venue for using external criteria for determining the goodness of fit for a cluster model.

# II

# Finding high-quality clustering solutions

In this chapter we discuss the topic of what makes clustering algorithms good (or bad) at finding high-quality solutions. Specifically, we investigate the internals of several popular, efficient iterative optimization clustering algorithms, and investigate what makes certain algorithms perform better than others, and why.

## II.A Finding high-quality solutions

Typically in data clustering there is no one perfect clustering solution of a dataset, but algorithms do seek to minimize a certain mathematical criterion (which varies between algorithms). Minimizing such criteria is known to be NP-hard for the general problem of partitioning $d$-dimensional data into $k$ sets [23]. Because of this, algorithms like $k$-means seek local rather than the global minimum solutions. However, these iterative optimization algorithms can get stuck at poor solutions. In these cases we consider that a solution which better minimizes the mathematical criterion (for the same number of centers) to

be a better-quality clustering. For example, Figure II.1 shows two clusterings of the same data, with one that better minimizes the $k$-means metric, and has a higher-quality solution.

In this work we consider the iterative algorithms that use centers to model clusters, such as $k$-means and Gaussian expectation-maximization. Each of these algorithms uses a number of centers to represent and/or partition the input data. Each center defines a cluster with a central point and perhaps a covariance matrix. These center-based clustering algorithms begin with a guess about the solution, and then refine the positions of centers until reaching a local optimum. These methods can work well, but they can also converge to local minima that are far from the global minimum, i.e. the clustering that has the highest quality according to the criterion in use. The tendency for an algorithm to converge to a poor local optimum is related to its sensitivity to initialization, and is a primary difficulty in data clustering. Figure II.1 shows a example where the $k$-means algorithm has converged to a both a poor-quality local optimum and a high-quality, globally optimum solution. The goal of the work in this chapter is to understand and extend these iterative optimization clustering algorithms, so that the algorithms find good-quality clusterings in spatial data.

Recently, many wrapper methods have been proposed to improve clustering solutions. A wrapper method is one that transforms the input or output of the clustering algorithm, and/or uses the algorithm multiple times. One commonly used wrapper method is simply running the clustering algorithm several times from different random starting points (often called random restart), and taking the best solution. Prevailing conventional wisdom says that the clustering practitioner should run a clustering algorithm with as many random restarts as time allows, in order to find the best solution. This brute-force approach can be effective, but it operates on the symptoms of the problems of data clustering, and

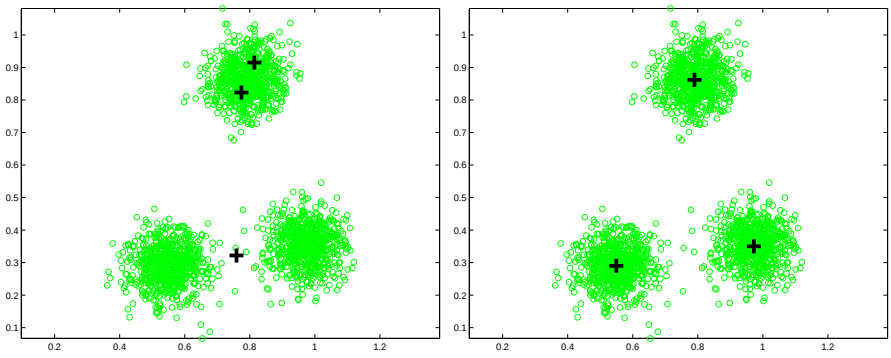

Figure II.1: On the left, $k$-means has converged to a poor solution; a local optimum. It is clear that the three centers (black crosses) have been incorrectly placed by the algorithm, because they don't have a 1-1 correspondence with the true clusters (datapoints in light color). Indeed, the $k$-means quality, a metric for measuring the clustering quality, says that the left solution has a score of 72.73. On the right, starting from a different initialization, $k$-means has successfully found a high-quality clustering, where each true cluster has a corresponding center chosen by the algorithm. This clustering has an improved score of 14.58 (this lower score means that the centers are, on average, closer to the datapoints they represent).

does not effectively address the core difficulty of sensitivity to initialization. In a similar vein to random restart, algorithms such as that used in [46] push this technique to its extreme, at the cost of computation, by essentially trying every possible reasonable initialization for the $k$ centers in an incremental fashion. Another wrapper method is searching for the best initializations possible; this has been looked at in [57, 48, 12]. This is fruitful research, as many clustering algorithms are sensitive to their initializations.

Other research [56, 58] has been looking at finding the appropriate number of clusters, and analyzing the difference between the cluster solution and the

dataset. This is useful when the appropriate number of centers is unknown, or the algorithm is stuck at a sub-optimal solution. In the next chapter we present our own contribution in this area, the G-means algorithm.

These approaches are beneficial, but they are attempting to fix the problems of clustering algorithms externally, rather than to improve the clustering algorithms themselves. We are interested in improving clustering algorithms directly to make them less sensitive to poor initializations and give better solutions. Of course, any clustering algorithm developed could benefit from wrapper methods.

Zhang *et al.* introduced a new clustering algorithm called $k$-harmonic means (KHM; also called harmonic $k$-means) that arises from an optimization criterion based on the harmonic mean [72, 71]. This algorithm shows promise in finding good clustering solutions quickly, and previous research has shown that it can outperform the very popular $k$-means (KM) and Gaussian expectation-maximization (GEM) in many tests, in terms of finding better-quality clustering solutions. The KHM algorithm also has a novel feature that gives more influence to data points that are not well-modeled by the clustering solution, but it is unknown how important this feature is. Our work is a first answer to this question.

In this chapter we present a unified framework for looking at iterative optimization clustering algorithms that use centers to partition data, and then derive two new algorithms that are based on properties of KM and KHM. We compare all of the algorithms both analytically and empirically. We find that the KHM algorithm does well primarily because of its soft membership function, which shares datapoints among different centers, making the algorithm less discrete than $k$-means. However, we also find that soft membership functions are not always beneficial in general, and show that the Gaussian EM algorithm's soft

membership function in particular gives poor results. We also demonstrate that the novel idea of giving extra weight to points that are far away from any cluster center has merit, and leads to a new $k$-means-type algorithm with re-weighting.

## II.B Center-based clustering

The algorithms $k$-means, Gaussian expectation-maximization, fuzzy $k$-means, and $k$-harmonic means are in the family of center-based clustering algorithms. They each have their own mathematical objective function, which defines how well a given clustering solution (i.e. locations and perhaps covariances of $k$ centers) fits a given dataset. Each algorithm is designed to minimize its objective function. Since these objective functions cannot be minimized directly, the algorithms use iterative updates which converge on local minima.

### II.B.1 General iterative clustering

We can formulate a general model for the family of clustering algorithms that use iterative optimization, following [37]. By expressing algorithms using a unified framework, we can directly compare the algorithms, which is very important for our goal of learning which parts of the algorithms contribute to good-quality clusterings. We define a $d$-dimensional set of $n$ data points $X = \{x_1, \ldots, x_n\}$ as the data to be clustered. Define a $d$-dimensional set of $k$ centers $C = \{c_1, \ldots, c_k\}$ as the clustering solution that an iterative algorithm refines.

This framework for iterative clustering requires two functions to be defined: the membership function and the weight function. We will look primarily at these two functions and how different definitions of them affect clustering algorithms.

A membership function $m(c_j|x_i)$ defines the proportion of data point $x_i$

that belongs to center $c_j$, with constraints:

$$\begin{aligned} m(c_j|x_i) &\geq 0 \\ \sum_{j=1}^{k} m(c_j|x_i) &= 1 \end{aligned}$$

In a generative context, the membership function can be thought of as the strength of belief that datapoint $x_i$ was generated by the process centered at $c_j$. Some algorithms use a *hard* membership function, meaning:

$$m(c_j|x_i) \in \{0, 1\}$$

while others use a *soft* membership function, meaning

$$0 \leq m(c_j|x_i) \leq 1$$

Kearns and colleagues have analyzed the differences between hard and soft membership from an information-theoretic standpoint [40]. One of the reasons that $k$-means can converge to poor solutions is due to its hard membership function. However, the hard membership function makes possible many computational optimizations that do not affect accuracy of the algorithm, such as using $kd$-trees [55].

The second important function is the data weight function. A weight function $w(x_i)$ defines how much influence data point $x_i$ has in recomputing the center parameters in the next iteration, with the constraint that $w(x_i) > 0$. The concept of giving points different weights is an important one, especially in the closely related field of supervised learning. However, data weighting has not been widely explored in unsupervised learning. Further, those that have considered data weighting have generally considered only those which are static, or unchanging, so that the weight of a point $x_i$ may be different than of point $x_j$, but those weights are fixed. Until recently with the work of Zhang with

generalized $k$-harmonic means [71], the idea of using a dynamic, or changing weighting function with clustering had not been considered. In this chapter we will consider it carefully, as it is an important development in clustering, which has philosophical connections to the idea of "boosting" in supervised learning [27]. Both boosting and KHM give more weight to data points that are not "well-explained" by the current solution. Unlike boosting, KHM's reweighting scheme does not create an ensemble of solutions.

Now we can define a general model of iterative, center-based clustering. In Algorithm 4 we see the approach outlined. Having a unified framework to describe these algorithms allows us to compare these algorithms based on differences in their membership and weight functions. The computational complexity of each clustering algorithm we consider in this chapter is linear in the size of the data and the number of clusters. Specifically, it is $O(nkd)$ for each iteration of the while loop in Algorithm 4. The algorithms vary by constant factors but have the same order complexity.

### II.B.2 Initialization methods

There are two popular initialization heuristics which we consider in this work, both based on the idea of a randomized start. The first, called a random sample, chooses $k$ datapoints at random from the dataset. These can be chosen according to a uniform distribution, or another distribution can be used if the practitioner has some knowledge of the structure of the dataset. This approach tends to place the initial centers throughout the space spanned by the dataset.

The second heuristic, called random partition, requires two steps. First, assign one of $k$ labels to each datapoint. Then calculate the location of each center $c_j$ as the average of the datapoints with label $j$. This is equivalent to performing one update (maximization) step in the generalized clustering algorithm

**Algorithm 4** Pseudocode for the generalized form of the clustering algorithms we will consider in this chapter. Inputs to the algorithm are the dataset $X$, the number of centers $k$ and their initial locations (and parameters) $C$, and functions $m$ and $w$. The functions $m$ and $w$ will completely define this algorithm. This generalized algorithm can take the form of many popular clustering algorithms depending on the supplied functions. The regions marked "expectation step" and "maximization step" are labeled to correspond to those steps in the standard expectation-maximization algorithm.

**generalized-cluster**($X \in \mathbb{R}^{n \times d}, k, m, w, C \in \mathbb{R}^{k \times d}$)

**while** the set of centers $C$ has not converged **do**
  *// expectation step*
  **for** $i \in \{1 \ldots n\}$ **do**
    **for** $j \in \{1 \ldots k\}$ **do**
      compute membership $m(c_j|x_i)$
      compute weight $w(x_i)$
    **end for**
  **end for**
  *// maximization step*
  **for** $j \in \{1 \ldots k\}$ **do**
    $c_j = \sum_{i=1}^{n} m(c_j|x_i)w(x_i)x_i / \sum_{i=1}^{n} m(c_j|x_i)w(x_i)$
  **end for**
**end while**

after having made a random expectation step. This method tends to place the centers near the middle of the dataset, since any random sample will have an average that is near the true middle of the dataset.

Figure II.4 gives an illustration of the two initializations. Of these two, random partitioning appears to be the poorer choice, since it places the centers near the middle of the dataset. However, as we will see, this initialization can be better than random sampling for certain clustering algorithms.

### II.B.3 K-means

The $k$-means algorithm (KM) [47] partitions data into $k$ sets. The solution is then a set of $k$ centers, each of which is located at the centroid of the data for which it is the closest center. For the membership function, each data point belongs to its nearest center, forming a Voronoi partition of the data. The objective function that the KM algorithm optimizes is

$$KM(X,C) = \sum_{i=1}^{n} \min_{j \in \{1...k\}} ||x_i - c_j||^2 \quad \text{(II.1)}$$

This objective function gives an algorithm which minimizes the within-cluster variance (the squared distance between each center and its assigned data points).

The membership and weight functions for KM are:

$$m_{KM}(c_l|x_i) = \begin{cases} 1 \ ; \ \text{if } l = \arg\min_j ||x_i - c_j||^2 \\ 0 \ ; \ \text{otherwise} \end{cases}$$

$$w_{KM}(x_i) = 1$$

KM has a hard membership function, and a constant weight function that gives all data points equal importance. KM is easy to understand and implement, making it a popular algorithm for clustering.

### II.B.4 Gaussian expectation-maximization

The Gaussian expectation-maximization (GEM) algorithm for clustering uses a linear combination of $d$-dimensional Gaussian distributions as the centers. It minimizes the objective function

$$GEM(X,C) = -\sum_{i=1}^{n} \log\left(\sum_{j=1}^{k} p(x_i|c_j)p(c_j)\right)$$

where $p(x_i|c_j)$ is the probability of $x_i$ given that it is generated by the Gaussian distribution with center $c_j$, and $p(c_j)$ is the prior probability of center $c_j$. We use a logarithm to make the math easier (while not changing the solution), and we negate the value so that we can minimize the quantity (as we do with the other algorithms we investigate). This is the typical expression for the log-likelihood. See [9, pages 59–73] for more about this algorithm. The membership and weight functions of GEM are

$$m_{GEM}(c_j|x_i) = \frac{p(x_i|c_j)p(c_j)}{p(x_i)} \quad \text{(II.2)}$$

$$w_{GEM}(x_i) = 1 \quad \text{(II.3)}$$

Bayes' rule is used to compute the soft membership, and $m_{GEM}$ is a probability since the factors in Equation II.2 are probabilities. GEM has a constant weight function that gives all data points equally importance, like KM. Note that $w_{GEM}(x_i)$ is not the same as $p(x_i)$.

### II.B.5 Fuzzy k-means

The fuzzy $k$-means algorithm (FKM; also called fuzzy $c$-means) [7] is an adaptation of the KM algorithm that uses a soft membership function. Unlike KM which assigns each data point to its closest center, the FKM algorithm allows a data point to belong partly to all centers, like GEM.

$$FKM(X,C) = \sum_{i=1}^{n}\sum_{j=1}^{k} u_{ij}^{r}||x_i - c_j||^2$$

The parameter $u_{ij}$ denotes the proportion of data point $x_i$ that is assigned to center $c_j$, and is under the constraints $\sum_{j=1}^{k} u_{ij} = 1$ for all $i$ and $u_{ij} \geq 0$. The parameter $r$ has the constraint $r \geq 1$. A larger value for $r$ makes the method "more fuzzy."

Bezdek and others give separate update functions for $u_{ij}$ and $c_j$. The $u_{ij}$ update equation depends only on $C$ and $X$, so we incorporate its update function into the update for $c_j$. Then we can represent FKM in the form of the general iterative update. The membership and weight functions for FKM are:

$$\begin{aligned} m_{FKM}(c_j|x_i) &= \frac{||x_i - c_j||^{-2/(r-1)}}{\sum_{j=1}^{k} ||x_i - c_j||^{-2/(r-1)}} \\ w_{FKM}(x_i) &= 1 \end{aligned}$$

FKM has a soft membership function, and a constant weight function. As $r$ tends toward 1 from above, the algorithm behaves more like standard $k$-means, and the centers share the data points less.

### II.B.6 K-harmonic means

The $k$-harmonic means algorithm (KHM) is a method similar to KM that arises from a different objective function [71]. The KHM objective function uses the harmonic mean of the distance from each data point to all centers.

$$KHM(X,C) = \sum_{i=1}^{n} \frac{k}{\sum_{j=1}^{k} \frac{1}{||x_i - c_j||^p}}$$

Here $p$ is an input parameter, and typically $p \geq 2$. The harmonic mean gives a good (low) score for each data point when that data point is close to any one center. This is a property of the harmonic mean; it is similar to the minimum function used by KM, but it is a smooth differentiable function.

The membership and weight functions for KHM are:

$$m_{KHM}(c_j|x_i) = \frac{||x_i - c_j||^{-p-2}}{\sum_{j=1}^{k} ||x_i - c_j||^{-p-2}} \qquad \text{(II.4)}$$

$$w_{KHM}(x_i) = \frac{\sum_{j=1}^{k} ||x_i - c_j||^{-p-2}}{\left(\sum_{j=1}^{k} ||x_i - c_j||^{-p}\right)^2} \quad \text{(II.5)}$$

Note that KHM has a soft membership function, and also a varying weight function. This weight function gives higher weight to points that are far away from every center, which aids the centers in spreading to cover the data. These functions are illustrated in Figures II.2 and II.3.

The membership function $m_{KHM}$ allows each datapoint to be shared by all the cluster centers. Figure II.2 shows this to be the case. Notice at the edges of the figure the tails are continuing to grow closer together. What the figure does not show is that for a point that is very far away from all the cluster centers, the membership function will place it equally in all centers. Therefore if datapoint $x$ is infinitely far from centers $c_{1...k}$, then $m_{KHM}(c_j|x) = 1/k$. This is very different from many other membership functions; even other soft membership functions. Conceptually, we can think of this as similar to the following question: if two people are standing together on the Earth at a certain location and time, which is closer to the planet Mars? The question has an answer, but the answer is not very significant, since both people are very far away; the difference in their distances from Mars is relatively insignificant. Returning to the membership function, this behavior is philosophically reasonable but does pose a problem for clustering in high dimensions, where every point tends to be far from every other point. That is okay, because the problem of finding high-quality clustering solutions in low dimensions remains open.

The implementation of KHM needs to deal with the case where $x_i = c_j$, in which case the objective function would divide by zero. We follow Zhang using $\max(||x_i - c_j||, \epsilon)$ and use a small positive value of $\epsilon$. We also apply this technique for FKM and the algorithms discussed in Section 3. We have not encountered any numerical problems in any of our tests.

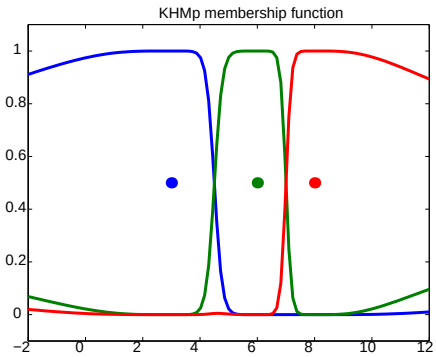


Figure II.2: The $k$-harmonic means membership function, as defined in Equation II.4. In this plot, the horizontal axis indicates the cluster space, and the vertical axis indicates the proportion of membership (between 0 and 1). The three cluster centers are indicated by the three dots in the plot (at values 3, 6, and 8). The three overlapping curves indicate the proportion of membership in each of the three clusters at each point along the horizontal axis (at each point, the sum of the three curves is 1). This is a soft membership function, since the curves have a smooth transition of membership for data that lies between two centers.

## II.C New clustering algorithms

We are interested in the properties of the KHM algorithm. It has a soft membership function and a varying weight function, which makes it unique among the algorithms we have encountered. KHM has been shown to be less sensitive to initialization on synthetic data [71].

We will analyze two aspects of KHM: the membership function and the weight function. To do this, we construct two new algorithms we call Hybrid 1 and Hybrid 2. They are named for the fact that they are hybrid algorithms that combine features of KM and KHM. The purpose for creating these algorithms is to find out what effects the membership function and weight function of KHM have by themselves.

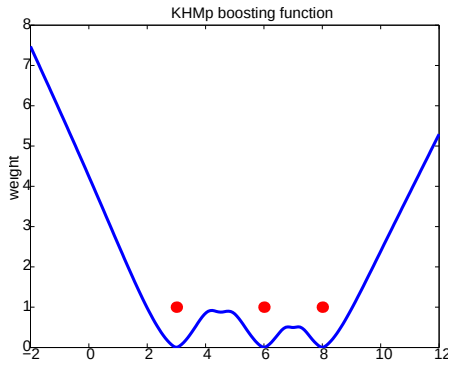


Figure II.3: The $k$-harmonic means data weight function, as defined in Equation II.5. In this plot, as in Figure II.2, the horizontal axis indicates the cluster space. The vertical axis indicates the weight that $k$-harmonic means assigns to each point along the horizontal axis, with respect to the three cluster centers at 3, 6, and 8. Points that are far away from all centers are given a large weight by this weight function, while points that are close to any center are given a small weight.

### II.C.1 Hybrid 1: hard membership, varying weights

Hybrid 1 (H1) uses the hard membership function of KM. Thus, every point belongs only to its closest center. However, H1 uses the KHM weight function, which gives more weight to points that are far from every center. We expect that this algorithm should converge more quickly than KM due to the weights, but will still have problems related to the hard membership function. As far as we know, adding weights in this manner to KM is a new idea.

The definitions of the membership and weight functions for H1 are:

$$m_{H1}(c_l|x_i) = \begin{cases} 1 \ ; \ \text{if } l = \arg\min_j ||x_i - c_j||^2 \\ 0 \ ; \ \text{otherwise} \end{cases}$$

$$w_{H1}(x_i) = \frac{\sum_{j=1}^{k} ||x_i - c_j||^{-p-2}}{\left(\sum_{j=1}^{k} ||x_i - c_j||^{-p}\right)^2}$$

### II.C.2 Hybrid 2: soft membership, constant weights

Hybrid 2 (H2) uses the soft membership function of KHM, and the constant weight function of KM. The definitions of the membership and weight functions for H2 are:

$$\begin{aligned} m_{H2}(c_j|x_i) &= \frac{||x_i - c_j||^{-p-2}}{\sum_{j=1}^{k} ||x_i - c_j||^{-p-2}} \\ w_{H2}(x_i) &= 1 \end{aligned}$$

Note that H2 resembles FKM. In fact, for certain values of $r$ and $p$ they are mathematically equivalent. It is interesting to note, then, that the membership function of KHM (from which we get H2) and FKM are also very similar. We investigate H2 and FKM as separate entities to keep clear the fact that we are investigating the membership and weight functions of KHM separately.

## II.D Experimental setup

We perform two sets of experiments to demonstrate the properties of the algorithms described in Sections 3 and 4. We want to answer several questions: how do different initializations affect each algorithm, what is the influence of soft versus hard membership, and what is the benefit of using varying versus constant weights.

Though each algorithm minimizes a different objective function, we measure the quality of each clustering solution by the square-root of the $k$-means objective function in Equation II.1. It is a reasonable metric by which to judge cluster quality, and by using a single metric we can compare different algorithms. We use the square root because the squared distance term can exaggerate the severity of poor solutions. We considered running KM on the output of each

algorithm, so that the KM objective function could be better minimized. We found that this did not help significantly, so we do not do this here.

Our experiments use two datasets already used in recent empirical work on clustering algorithms [73, 55], and a photograph of a hand from [15]. The algorithms we test are KM, KHM, FKM, H1, H2, and GEM. The code for each of these algorithms is our own (written in Matlab), except for GEM (FastMix code provided by [59]). We need to supply the the KHM, H1, and H2 with the parameter $p$, and FKM with $r$. We set $p = 3.5$ for all tests, as that was the best value found by Zhang. We set $r = 1.3$, as that is the best value we found based on our preliminary tests.

The two initializations we use are the random sampling and random partition methods [57]. The random sampling method chooses $k$ data points from the dataset at random and uses them as the initial centers. The random partition method assigns each data point to a random center, then computes the initial location of each center as the centroid of its assigned points. The random sampling method tends to spread centers out in the data, while the random partition method tends to place the centers in a small area near the middle of the dataset. Random Partition was found to be a preferable initialization method for its simplicity and quality in [57, 48]. For GEM, we also initialize $p(c_j) = 1/k$ and initialize the covariance to be $0.2I$, where $I$ is the identity matrix.

Before clustering, all datasets used in both experiments are shifted and re-scaled to give each dimension zero mean and unit variance. This is the standard z-score transformation. This can be a good idea before using algorithms based on distance metrics, as it gives the same influence to each dimension.

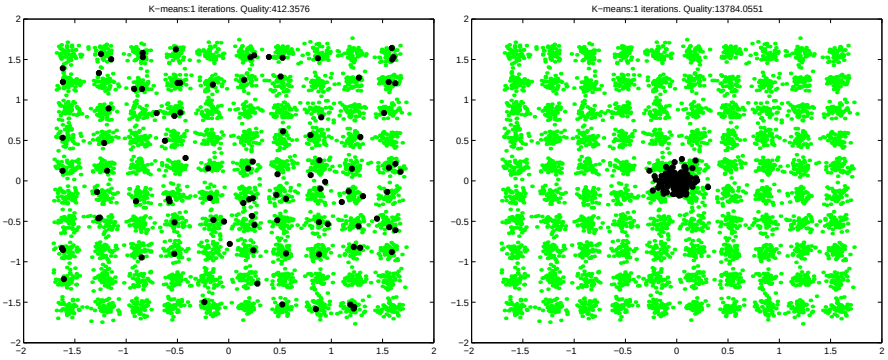


Figure II.4: Random sampling (left) and random partition (right) initializations for the BIRCH dataset. Centers are shown in the dark color, data points in the light color. This dataset has a grid of 10x10 natural clusters.

### II.D.1 Experiment 1: BIRCH

The purpose of our first experiment is to illustrate the convergence properties of the different algorithms, and to show the need to improve clustering algorithms. We use a randomly generated synthetic dataset we call BIRCH, as defined by [73]. This dataset has $k = 100$ true clusters arranged in a 10x10 grid in $d = 2$ dimensions. Each cluster generates 100 data points from its own Gaussian distribution, for a total of $n = 10,000$ data points. The distance between two adjacent cluster means is $4\sqrt{2}$ with cluster radius of $\sqrt{2}$ (meaning the variance in each dimension is 1). We run each algorithm twice for illustration purposes, once with the random sampling initialization, and once with the Random Partition initialization. Figure II.4 shows the two initializations. We use the same randomly chosen initializations for all algorithms. Our results are similar for other random initializations.

### II.D.2 Experiment 2: Pelleg and Moore data

The second experiment uses a synthetic dataset based on work by Pelleg and Moore [55]. Here we run many tests to determine the average-case behavior of the algorithms. We test datasets of dimensions 2, 4, and 6 to show that all these algorithms work well in low dimensions. Each dataset has $k = 50$ true natural clusters which generate $n = 2500$ total data points. The true cluster centers are chosen at random in the unit hypercube, then 2500 data points are generated by choosing a cluster randomly, and generating a data point according to a Gaussian distribution with standard deviation $s = d \times 0.012$ and mean at the true cluster center. We generate data that is more separated (clusters have less overlap) than the work by Pelleg and Moore (who used $s = 0.025d$), because this presents a more difficult task to the clustering algorithms. This is because it is harder for centers to move freely through the whole dataset due to gaps between natural clusters. See figure II.5 for a simple example.

For each $d \in \{2, 4, 6\}$ we generate 100 datasets, and two initializations (random partition and random sampling) for each dataset. Then we test each algorithm from both of these initializations. For each algorithm we allow it to run for 100 iterations, which is plenty for the algorithms to converge.

## II.E Experimental results

### II.E.1 Experiment 1: BIRCH

Running each algorithm on the BIRCH dataset once gives an intuition for how each behaves. Figure II.4 shows the two initializations we use. The results of KM, GEM, and KHM's runs are shown in Figures II.6, II.7, and II.8, and the cluster qualities for all are shown in Table II.1. FKM, H2, and KHM all found good clusterings for both types of initializations, and they are all soft

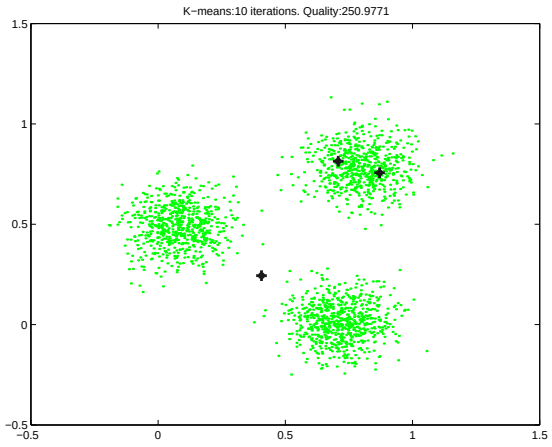


Figure II.5: Here $k$-means has converged to a local optimum which it would be able to escape if the clusters had more overlap. Without overlap, one of the "trapped" centers on the top-right cannot move to a cluster on the bottom or left because of the hard assignment function. Thus, a clustering problem is harder when clusters have less overlap, because the probability of finding bad local optima is higher.

membership algorithms.

The two hard membership algorithms, KM and H1, have distinctly different behavior for the two initializations. Starting from random sampling initialization these two algorithms perform reasonably well, but starting from the random partition these algorithms converge with many centers remaining in the middle of the dataset, "trapped" there by hard assignment. This is because the hard membership function prevents centers from moving if they do not own enough points. In Table II.1 we show the number of true clusters found, which is the number of true clusters (out of 100) that received a center by the algorithm.

Although GEM has a soft membership function, it does poorly on this dataset due to some centers having variance that is too large and taking over several clusters. The output of GEM initialized by random partition appears to

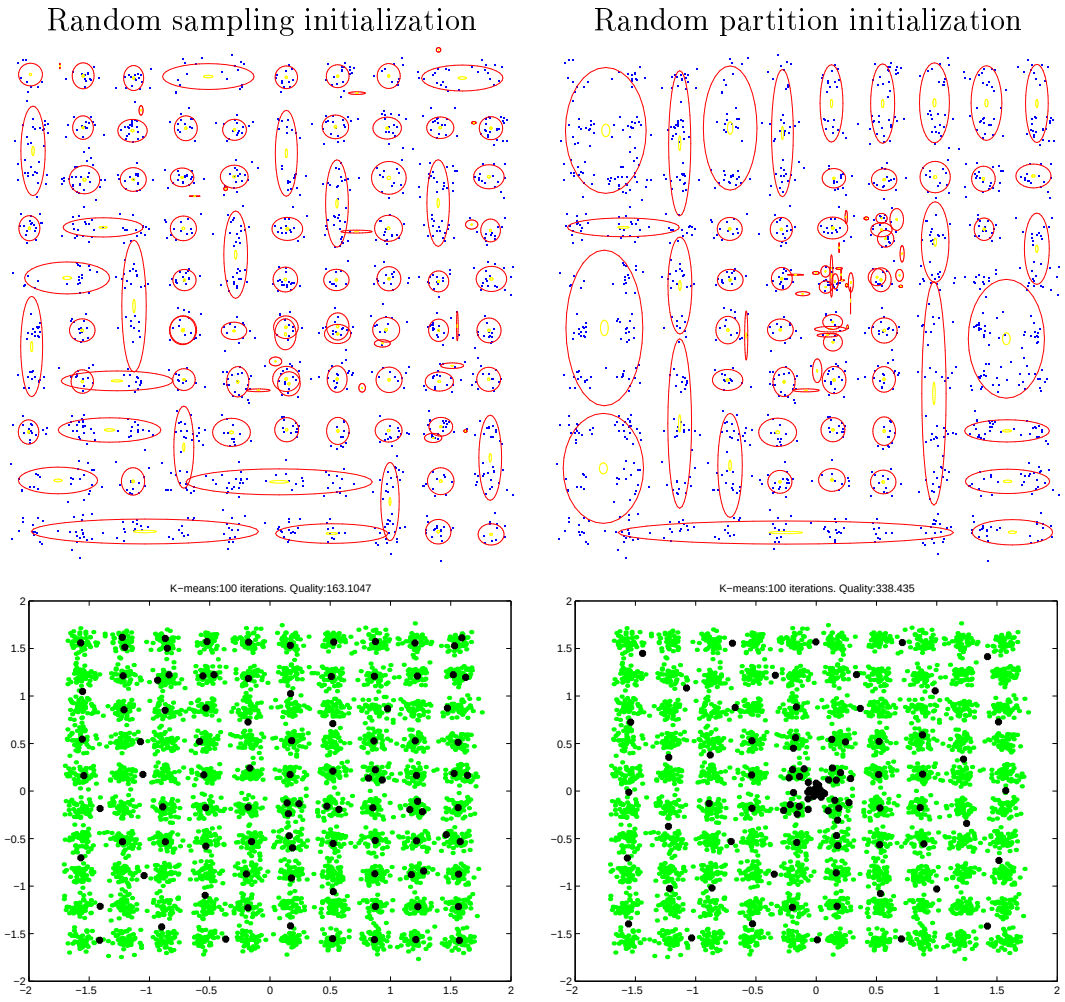


Figure II.6: Convergence on the BIRCH dataset for Gaussian EM (top row) and $k$-means (bottom row). Both converge to very low-quality optima. FastMix software generated the plots for Gaussian EM, showing the 1-sigma contours.

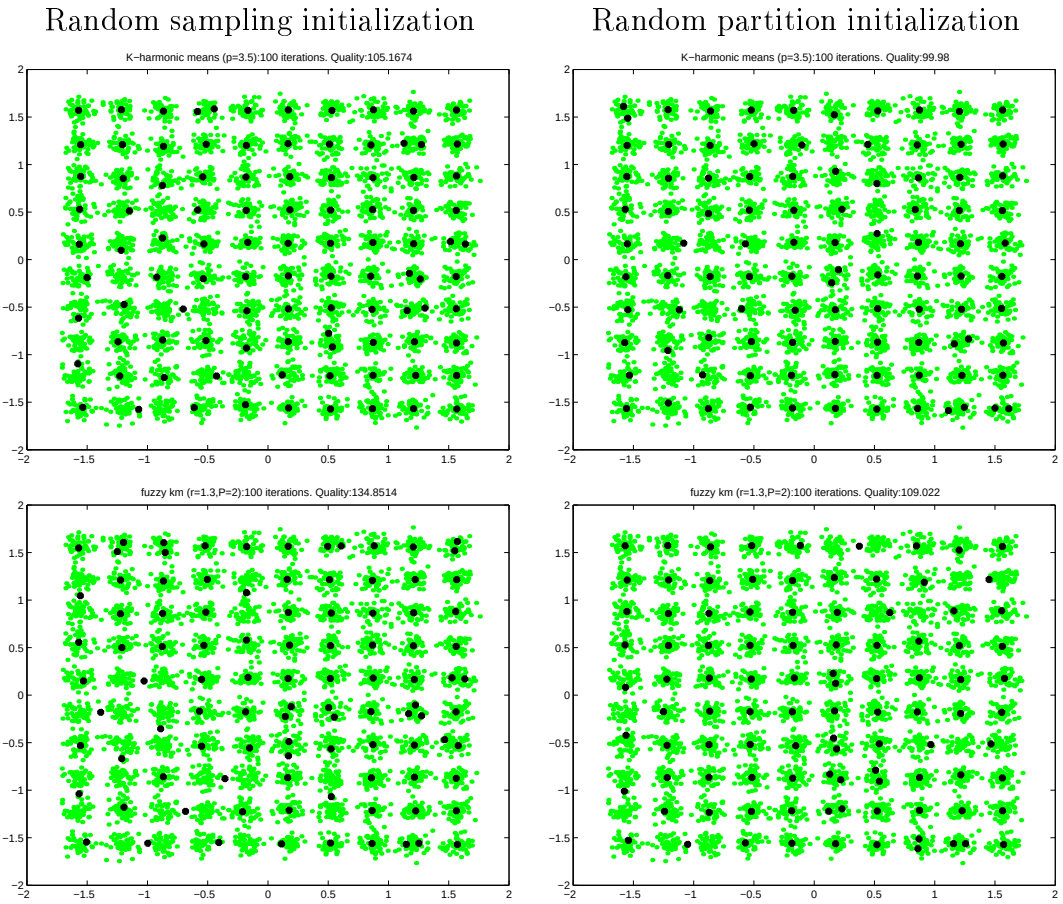


Figure II.7: Convergence on the BIRCH dataset for $k$-harmonic means (top row) and fuzzy $k$-means (bottom row). The $k$-harmonic means algorithm finds a higher-quality solution than all the other algorithms, both in terms of the number of true clusters found, and the $k$-means quality. Note that the output of these two algorithms are resistant to poor initializations.

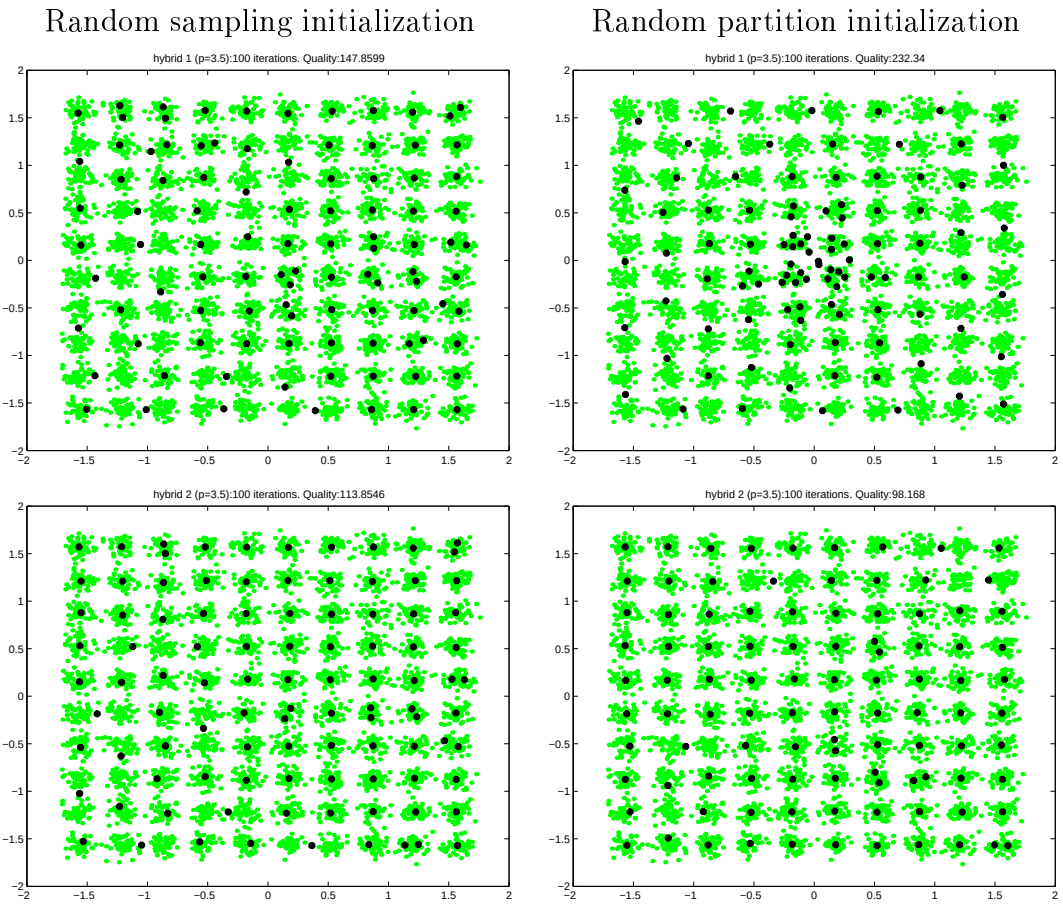


Figure II.8: Convergence on the BIRCH dataset for the Hybrid 1 and Hybrid 2 algorithms. Hybrid 2, with its soft membership function, does a great job of finding a good clustering solution regardless of initialization, while Hybrid 1 does somewhat better than $k$-means due to its dynamic weighting function

Table II.1: Quality of solutions for one run on the BIRCH dataset, using random sampling and random partition initializations. Lower quality scores are better. "Clusters found" is the number of true clusters (maximum 100) in which the algorithm placed at least one center.

| | $\sqrt{KM}$ **quality** | | **Clusters found (of 100)** | |
|---|---|---|---|---|
| | Random sampling | Random partition | Random sampling | Random partition |
| GEM | 15.530 | 24.399 | 77 | 49 |
| KM | 12.771 | 18.396 | 83 | 60 |
| H1 | 12.159 | 15.242 | 86 | 72 |
| FKM | 11.612 | 10.441 | 89 | 93 |
| H2 | 10.670 | 9.908 | 92 | 95 |
| KHM | 10.255 | 9.999 | 94 | 95 |

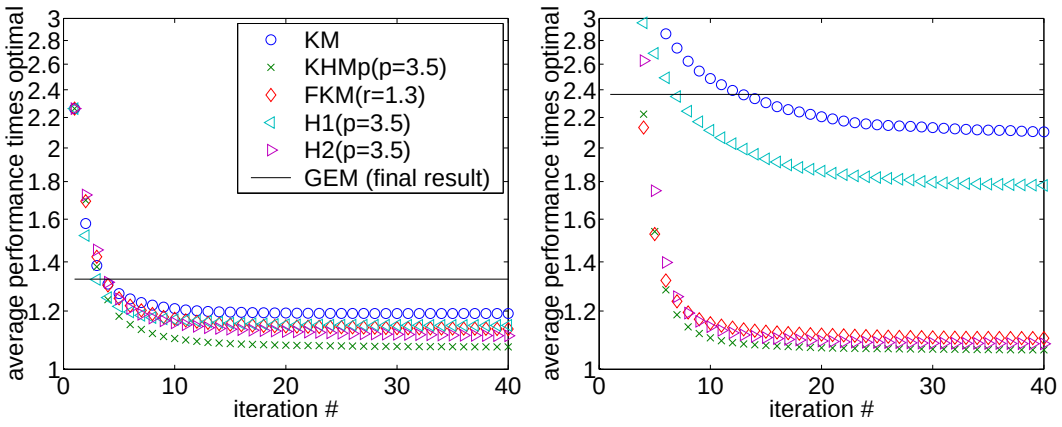


Figure II.9: Convergence curves starting from random sampling (left) and random partition (right) initializations on 2-d synthetic data. The x-axis shows the number of iterations, and the y-axis (log scale) shows average clustering quality score, where lower values are better. Only the final results for GEM are shown. Note that KM and GEM perform worse than every other algorithm.

have more centers concentrated in the middle of the dataset, where the centers began. This is similar to the hard membership results. The FastMix implementation we used for GEM started with 100 centers and removes centers whose prior became too small. For this reason, it ended with 98 (random sampling) and 81 (random partition) centers depending on the initialization. FastMix has the ability to search for the number of centers using density estimation. We tried starting FastMix without a pre-defined number of centers, and it found 23.

### II.E.2 Experiment 2: Pelleg and Moore data

Our second experiment shows the average performance of the algorithms compared over many randomly generated data sets in several dimensions. For each dataset $X_{d,i}$ where $d \in \{2, 4, 6\}$ and $1 \leq i \leq 100$ we compute the optimal KM partition $O_{d,i}$ by running KM to convergence starting with the centers that generated the data sets. Then we compute the score of a clustering $C_{d,i}$ as the ratio

$$R_{d,i} = \sqrt{\frac{KM(X_{d,i}, C_{d,i})}{KM(X_{d,i}, O_{d,i})}}$$

Table II.3 shows the mean and standard deviation of $R_{d,i}$ for each algorithm, computed using 100 datasets in 2 dimensions. Table II.2 shows the point-wise comparison of each algorithm for the same experiment. It is clear from this as well that soft membership algorithms (KHM, FKM, H2) perform better than hard membership algorithms (KM, H1) in both average performance and variance. The results for 4 and 6 dimensional datasets are very similar, so we do not report them here.

Figure II.9 shows the speed of convergence of each algorithm for $d = 2$ dimensions. The x-axis shows the iteration number, and the y-axis shows the average $k$-means quality ratio at that iteration, computed using the 100 datasets. We can see that GEM and KM are uniformly inferior to every other algorithm,

and that the soft membership algorithms KHM, H2, and FKM move quickly to find good solutions. Only the final result for GEM is plotted as we cannot capture clustering progress before the FastMix software terminates.

FastMix has the ability to add and remove centers to better fit its data. FastMix adds a center if the model underpredicts the data and removes a center if its prior probability is too low. We expect that FastMix's ability to add centers would be helpful in a dataset in which clusters are well-separated. For experiment 2, FastMix began with 50 centers and only removed centers. FastMix converged with an average of 48.39 centers (random sampling) and 40.13 centers (Random Partition) in the 2-dimension test. This shows that GEM is also sensitive to poor initializations.

## II.F Conclusion

Our experiments clearly show the superiority of the $k$-harmonic means algorithm (KHM) for finding clusterings of high quality in low dimensions. Our algorithms H1 and H2 let us study the effects of the KHM weight and membership separately. They show that soft membership is essential for finding good clusterings, as H2 performs nearly as well as KHM, but that varying weights are beneficial with a hard membership function, since H1 performs better than KM. Varying weights are intuitively similar to the weights applied to training examples by boosting [27]. It remains to be seen whether this analogy can be made precise.

Previous work in initialization methods has concluded that the random partition method is good for GEM and for KM, but our experiments do not confirm this conclusion. The random sampling method of initialization (choosing random points as initial centers) works best for GEM, KM, and H1. Overall, our results suggest that the best algorithms available today are FKM, H2, and KHM,

initialized by the random partition method.

Clustering in high dimensions has been an open problem for many years. However, recent research has shown that it may be preferable to use dimensionality reduction techniques before clustering, and then use a low-dimensional clustering algorithm such as $k$-harmonic means, rather than clustering in the high dimension directly. In [18] the author shows that using a simple, inexpensive linear projection preserves much of the properties of data (such as cluster distances), while making it easier to find the clusters. Thus there is a need for good-quality, fast clustering algorithms for low-dimensional data, such as $k$-harmonic means.

## II.G Acknowledgements

This chapter, in part, is a reprint of the material as it appears in [30]. The dissertation author was the primary researcher and author of this paper.

Table II.2: Competition matrix for 2-d data starting from random sampling (top) and random partition (bottom) initializations. Each entry shows the number of times (maximum 100) that the algorithm in the column had a better-quality clustering than the algorithm in the row. The results for 4 and 6 dimensions are similar, so we do not report them here. In particular, KHM is better than KM for 99 to 100 of 100 times in each dimension tested.

| | GEM | KM | KHM | FKM | H1 | H2 |
|---|---|---|---|---|---|---|
| GEM | | 97 | 100 | 100 | 98 | 100 |
| KM | 3 | | 99 | 91 | 73 | 91 |
| KHM | 0 | 1 | | 15 | 7 | 24 |
| FKM | 0 | 9 | 85 | | 19 | 62 |
| H1 | 2 | 27 | 93 | 81 | | 90 |
| H2 | 0 | 9 | 76 | 38 | 10 | |
| sum: | 5 | 143 | 453 | 325 | 207 | 367 |

| | GEM | KM | KHM | FKM | H1 | H2 |
|---|---|---|---|---|---|---|
| GEM | | 100 | 100 | 100 | 100 | 100 |
| KM | 0 | | 100 | 100 | 100 | 100 |
| KHM | 0 | 0 | | 21 | 0 | 34 |
| FKM | 0 | 0 | 79 | | 0 | 70 |
| H1 | 0 | 0 | 100 | 100 | | 100 |
| H2 | 0 | 0 | 66 | 30 | 0 | |
| sum: | 0 | 100 | 445 | 351 | 200 | 404 |

Table II.3: The mean and standard deviation of $R_{2,i}$, the ratio between the $k$-means quality and the optimum, over 100 datasets, in 2 dimensions. Lower values are better. KHM again performs the best, followed by the Hybrid 2 algorithm which uses KHM membership and KM weights. Results for 4 and 6 dimensions are similar, and have the same ranking.

| | Random sampling | Random partition |
|---|---|---|
| GEM | 1.3262 +/- 0.1342 | 2.3653 +/- 0.4497 |
| KM | 1.1909 +/- 0.0953 | 2.0905 +/- 0.2616 |
| H1 | 1.1473 +/- 0.0650 | 1.7644 +/- 0.2403 |
| FKM | 1.1281 +/- 0.0637 | 1.0989 +/- 0.0499 |
| H2 | 1.1077 +/- 0.0536 | 1.0788 +/- 0.0416 |
| KHM | 1.0705 +/- 0.0310 | 1.0605 +/- 0.0294 |

# III

# Estimating the number of clusters

## III.A Introduction and related work

Clustering algorithms are useful tools for data mining, compression, unsupervised learning, probability estimation, and other tasks in machine learning and statistics. However, most clustering algorithms require the user to provide the number of clusters (called $k$), and it is not always clear what is the best value for $k$. Figure III.1 shows examples where $k$ has been improperly chosen. In general, choosing $k$ is often an *ad hoc* decision based on prior knowledge, assumptions, and practical experience.

Center-based clustering algorithms (in particular $k$-means and Gaussian expectation-maximization) assume that each cluster adheres to a usually unimodal distribution (such as Gaussian). With these methods, only one center should be used to describe each subset of data that follows a unimodal distribution. If multiple centers are used to describe data drawn from one mode, the centers are a needlessly complex description of the data, and in fact the multiple centers capture the truth about the subset less well than one center.

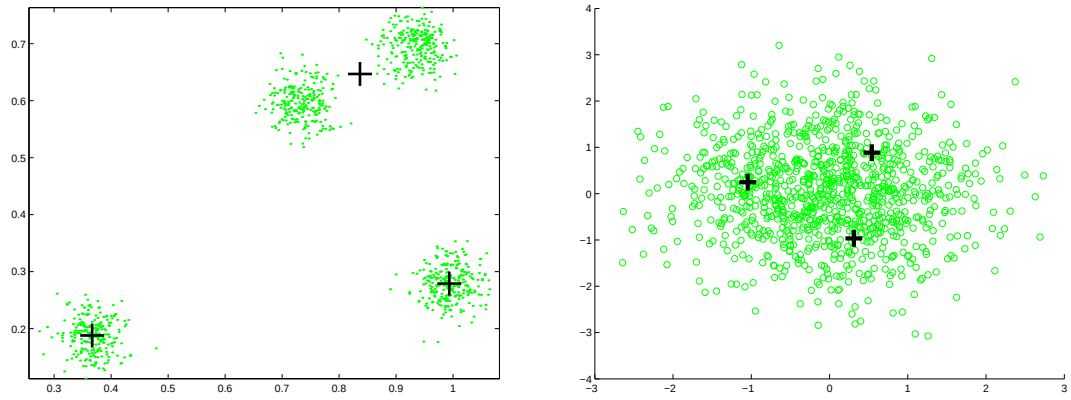


Figure III.1: Two clusterings where $k$ was improperly chosen for the dataset being clustered. Dark crosses are $k$-means centers. On the left, there are too few centers; four should be used. On the right, too many centers are used; one center is sufficient for representing the data. In general, one center should be used to represent one Gaussian cluster.

In this chapter we present a simple algorithm called G-means that discovers an appropriate $k$ using a statistical test for deciding whether to split a $k$-means center into two centers. We also present a new statistic for determining whether data are sampled from a Gaussian distribution, which we call the G-means statistic. We describe examples and present experimental results that show that the new algorithm is successful. This technique is useful and applicable for many clustering algorithms other than $k$-means, but here we consider only the $k$-means algorithm for simplicity.

Several algorithms have been proposed previously to determine $k$ automatically. Like our method, most previous methods are wrappers around $k$-means or some other clustering algorithm for fixed $k$. Wrapper methods use splitting and/or merging rules for centers to increase or decrease $k$ as the algorithm proceeds. Other research on agglomerative clustering suggests choosing $k$ based on the "stability" of the merging tree (dendrogram) of distances between points.

Pelleg and Moore [56] proposed a regularization framework for learning $k$, which they call $X$-means. The algorithm scores each clustering model using the so-called Bayesian Information Criterion

$$BIC(C|X) = \mathcal{L}(X|C) - \frac{p}{2}\log n$$

where $\mathcal{L}(X|C)$ is the log-likelihood of the dataset $X$ according to model $C$, $p = k(d+1)$ is the number of parameters in the model $C$ with dimensionality $d$ and $k$ cluster centers, and $n$ is the number of points in the dataset [39]. Then they choose the model with the best BIC score. Aside from the BIC, other scoring functions are also available.

Bischof *et al.* [8] use a minimum description length (MDL) framework, where the description length is a measure of how well the data are fit by the model. Their algorithm starts with a large value for $k$ and removes centers (reduces $k$) whenever that choice reduces the description length. Between steps of reducing $k$, they use the $k$-means algorithm to optimize the model fit to the data.

In hierarchical clustering algorithms, "cluster analysis" is used to determine the best number of clusters for a dataset. One heuristic is to build a merging tree of the data based on a cluster distance metric, and to search for areas of the tree that are relatively stable with respect to inter-cluster and intra-cluster distances [36, Section 5.1]. However, this method of finding the number of clusters is best applied with domain-specific knowledge and human intuition.

## III.B The Gaussian-means (G-means) algorithm

The G-means algorithm starts with a small number of $k$-means centers, and grows the number of centers. Each iteration of the algorithm splits into two those centers whose data appear not to come from a Gaussian distribution. Between each round of splitting, we run $k$-means on the entire dataset and all

the centers to refine the current solution. We can initialize with just $k = 1$, or we can choose some larger value of $k$ if we have some prior knowledge about the range of $k$. Algorithm 5 gives a pseudocode description.

---

**Algorithm 5** The G-means algorithm for learning $k$ while clustering. Inputs to the algorithm are the dataset $X$ and a confidence level $\alpha$, which is used in the statistical test.

---

**G-means($X$, $\alpha$)**

1: Let $C$ be the initial set of centers (usually $C \leftarrow \{\bar{x}\}$).
2: $C \leftarrow kmeans(C, X)$.
3: Let $\{x_i | \text{class}(x_i) = j\}$ be the set of datapoints assigned to center $c_j$.
4: Use a statistical test to detect if each $\{x_i | \text{class}(x_i) = j\}$ follow a Gaussian distribution (at confidence level $\alpha$).
5: If the data look Gaussian, keep $c_j$. Otherwise replace $c_j$ with two centers split from $c_j$.
6: Repeat from step 2 until no more centers are added.

---

G-means repeatedly makes decisions based on a statistical test for the data assigned to each center. If the data currently assigned to a $k$-means center appear to be Gaussian, then we want to represent that data with only one center. However, if the same data does not appear to be Gaussian, then we want to use multiple centers to model it properly. The algorithm will run $k$-means multiple times (up to $k$ times when finding $k$ centers), so the time complexity is at most $O(k)$ times that of $k$-means.

An optimization we make in the G-means algorithm is that once we have decided not to split a center $c_j$, we do not test the data belonging to that center again. This enables us to make $k$ statistical tests when finding $k$ centers, rather than up to $O(k^2)$ tests if every center is tested at every iteration of G-means (in

the worst case scenario).

The $k$-means algorithm implicitly assumes that the data points in each cluster are spherically distributed around the center. This is because the $k$-means quality metric described in prior chapters is based on the distance between datapoints and the centers to which they belong, where all directions are weighted equally. Less restrictively, the Gaussian expectation-maximization algorithm assumes that the data points in each cluster have a multidimensional Gaussian distribution with a covariance matrix that may or may not be fixed, or shared. The Gaussian distribution tests that we present below are valid for either type of covariance matrix. The tests also account for the number of data points involved (see Equations III.2 and III.1), which prevents the G-means algorithm from making bad decisions about clusters with few data points.

## III.C Testing clusters for Gaussian fit

To specify the G-means algorithm fully we need a test to detect whether the data assigned to a center are sampled from a Gaussian. The alternative hypotheses are:

- $H_0$: The data around the center are sampled from a Gaussian.
- $H_1$: The data around the center are not sampled from a Gaussian.

If we accept the null hypothesis $H_0$, then we believe that the one center is sufficient to model its data, and we should not split the cluster into two subclusters. If we reject $H_0$ and accept $H_1$, then we want to split the cluster.

In this work we have utilized two tests for normality. Both are one-dimensional tests which assume that the data has been z-scored; that is, converted to mean 0 and variance 1. The first is based on a new statistic we call the G-means statistic. This statistic comes from the distortion of the data, defined

as

$$r(X, C) = \sum_{i=1}^{|X|} \min_j ||x_i - c_j||^2$$

where $C = \{c_1, \ldots, c_k\}$ is the set of $k$ centers. The G-means statistic uses a specific formulation of $r(X, C)$, under constraints of univariate data with $k = 2$ centers. Specifically, given a one-dimensional set of data and two $k$-means clusters $\{c_1, c_2\}$, the G-means statistic is

$$r_{gm}(X) = \min_{c_1, c_2} \sum_{i=1}^{|X|} \min((x_i - c_1)^2, (x_i - c_2)^2)$$

This is the minimum of the $k$-means objective function for two centers in Gaussian data (when the null hypothesis is true).

The second test is based on the Anderson-Darling statistic. This one-dimensional test has been shown empirically to be the most powerful normality test that is based on the empirical cumulative distribution function (ECDF). Given a list of values $x_i$ that have been converted to mean 0 and variance 1, let $x_{(i)}$ be the $i$th ordered value. Let $z_i = F(x_{(i)})$, where $F$ is the $N(0, 1)$ cumulative distribution function. Then the statistic is

$$A^2(Z) = -\frac{1}{n} \sum_{i=1}^{n} (2i - 1) \left[\log(z_i) + \log(1 - z_{n+1-i})\right] - n$$

Stephens [67] showed that for the case where $\mu$ and $\sigma$ are estimated from the data (as in clustering), we must correct the statistic according to

$$A^2_*(Z) = A^2(Z)(1 + 4/n - 25/(n^2)) \quad \text{(III.1)}$$

With these two statistics, and their respective distributions, we will construct statistical tests for normality which will be used in the G-means algorithm.

### III.C.1 Constructing the test

Given a subset of data $X$ in $d$ dimensions that belongs to center $c$, the hypothesis test proceeds as follows:

1. Choose a significance level $\alpha$ for the test.

2. Initialize two $d$-dimensional centers in $X$, called "children" of $c$. See the text for good ways to do this.

3. Run $k$-means on these two centers in $X$. This can be run to completion, or to some early stopping point if desired. Let $c_1, c_2$ now represent the two final child centers chosen by $k$-means.

4. Let $v = c_1 - c_2$ be a $d$-dimensional vector that represents the separation axis that connects the two centers. This is the direction that $k$-means believes to be important for clustering. Then project $X, c_1, c_2$ onto $v$ using vector dot product $\langle \cdot, \cdot \rangle$ and $L_2$ norm $|| \cdot ||$:

$$\begin{aligned} x'_i &= \langle x_i, v \rangle / ||v||^2 \\ c'_1 &= \langle c_1, v \rangle / ||v||^2 \\ c'_2 &= \langle c_2, v \rangle / ||v||^2 \end{aligned}$$

   These $X', c'_1, c'_2$ are one-dimensional representations of the data and two child centers, projected onto $v$.

5. Transform $X'$ so that it has mean 0 and variance 1, and apply the same transform to $c'_1, c'_2$.

6. Compute the test statistic $T(X')$. If $X'$ appears normal according to the test we are using, at confidence level $\alpha$, then accept $H_0$, keep the original center $c$, and discard the two child centers $\{c_1, c_2\}$. Otherwise, reject $H_0$ and keep the $\{c_1, c_2\}$ in place of $c$.

For the final step, if we are using the G-means statistic, then we calculate $T(X') = r_{gm}(X')$ and compare it against the critical values for the G-means statistic distribution. If we are using the Anderson-Darling statistic, then we compare $T(X') = A^2_*(Z)$ against the critical values of its distribution, where $z_i = F(x'_{(i)})$.

Since both test statistics operate on univariate data, we have developed our test to facilitate this, since most data we cluster is multivariate. We project the data onto the vector $v$ that connects $c_1, c_2$. For the G-means statistic, this one-dimensional representation of the data allows us to consider only the relevant information, which is the distortion in the direction of the two centers. Along any other orthogonal direction, the distortion does not provide additional information (see Figure III.2 for an illustration). Our method is related to the problem of projection pursuit [34], where here $k$-means searches for a direction in which the data appears non-Gaussian.

This use of dimension reduction to perform a statistical test for learning $k$ is a primary contribution of our work. We simplify the test for Gaussian fit by projecting the data to one dimension, where powerful tests are easy to apply. [22] also used a similar approach for online dimensionality reduction during clustering, but not for learning the number of clusters.

### III.C.2 Performing the test for normality

Under $H_0$, the transform of $X'$ has the effect of giving the data a $N(0, 1)$ distribution. Then to perform the test we must compute the test statistic $T(X')$ (whether G-means or Anderson-Darling) and compare it to the critical value(s) of the statistic's distribution. Then we can conduct the test as follows:

- Reject $H_0$ if either $T(X')$ falls outside the range of acceptable values for its distribution.

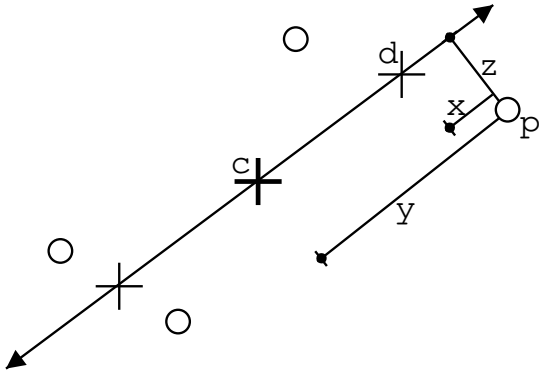


Figure III.2: This plot shows two alternative clusterings of the four circles: using one center $c$ (thick cross), and using two centers (thin crosses). The squared distance $||\cdot||^2$ between two points is the sum of the squared distances along each orthogonal axis. Thus $||p-d||^2 = x^2 + z^2$ is smaller than $||p-c||^2 = y^2 + z^2$ only along the vector connecting the two centers ($x < y$). Along orthogonal directions (i.e. $z$), all point-center distances remain the same.

- Otherwise accept $H_0$.

The G-means statistic has two rejection regions: $T(X') < V_{\alpha/2}$ and $T(X') > V_{1-\alpha/2}$, where $V$ represents a critical value of the distribution at the given significance level. If $T(X') < V_{\alpha/2}$ (the common case in this work when $H_0$ is not true), then this means that $c_1'$ and $c_2'$ have found areas of higher density than a single Gaussian, indicating that there is more than one cluster. In rare cases, it will happen that $T(X') > V_{1-\alpha/2}$; this can happen (for instance) when there are three clusters that are well-separated and have a certain distribution, in which case we still want to split into more than one cluster. See Appendix A for a proof and example.

The Anderson-Darling test is exclusively a one-sided test, since the test involves squared differences from the theoretical distribution. The statistic $A^2$ has

a lower bound of zero, and the larger the value is, the less likely the sample being tested has come from a Gaussian distribution. Therefore we will only discuss the upper-tail test on the Anderson-Darling statistic, and some critical value $V_{1-\alpha/2}$.

The analytical forms of the distributions of both statistics are unknown; therefore we may use simulations (or previously-published work for the Anderson-Darling test) to obtain the critical values for these statistics. We will discuss practical issues with obtaining these critical values shortly.

### III.C.3 Adjusting significance for multiple tests

We must choose the significance level of the test, $\alpha$, which is the probability with which we will make a Type 1 error (i.e. reject $H_0$ when we should not reject it). Since the algorithm performs many statistical tests while learning $k$, it is appropriate to use a Bonferroni adjustment to reduce the chance of making Type 2 errors over multiple tests. For example, if we want a 0.01 chance of making a Type 2 error in 100 tests, we should apply a Bonferroni adjustment to make each test use $\alpha = 0.01/100 = 0.0001$. To find $k$ final centers the G-means algorithm makes between $2k - 1$ and $k(k + 1)/2$ statistical tests. If we use the heuristic of not re-testing a center once we have accepted $H_0$ for that center, then we reduce the number of tests to $k$ for finding $k$ centers. In either case, this is not a large number of tests if $k$ is small, so the Bonferroni correction does not need to be extreme. In our tests, we use $\alpha = 0.0001$ except where indicated.

### III.C.4 Finding the critical values of the test

Both the G-means and Anderson-Darling statistics require a set of critical values in order to construct a test based on them. Since these statistics do not have known analytical distributions (which is not uncommon for many test statistics), we can choose between two options: to use computer simulation to

make a table of critical values, or to use another similar function to approximate the distribution.

Computer simulation for determining tables of critical values is a well-established technique, though it can be difficult for several reasons: it is difficult to generate high-quality, truly random numbers (even for a uniform distribution), it is even more difficult to generate high-quality Gaussian random numbers (one popular way is using the Box-Muller transformation from a uniform distribution [11]). Indeed, researching the topic of Anderson-Darling critical values which have been published in several sources has revealed several different sets of values (see [67, 17, 68], as well as several sources on the world wide web). Further, the critical values differ significantly depending on whether the parameters $\mu$ and $\sigma$ are both estimated from the data (which Stephens calls Case 3, for estimation of both parameters), or are fixed values (which Stephens calls Case 0). In this work, naturally, we are interested primarily in the case where all parameters are estimated from the data.

A contribution of our work is the calculation of a table of critical values for the Anderson-Darling statistic, which may be compared with earlier published work. We have done this using Matlab software and the built-in Matlab functions for generating normal random numbers and calculating the CDF values for the normal distribution. We have computed the table based on 100,000 experiments of the following:

1. generate 1,000 univariate datapoints from a $N(0,1)$ distribution, called $X$
2. convert $X$ to have mean 0 and variance 1 (making it Stephens Case 3)
3. let $z_i = F(x'_{(i)})$
4. compute the statistic $A^2(Z)$

The CDF of this experiment is shown in Figure III.3, and the critical values are

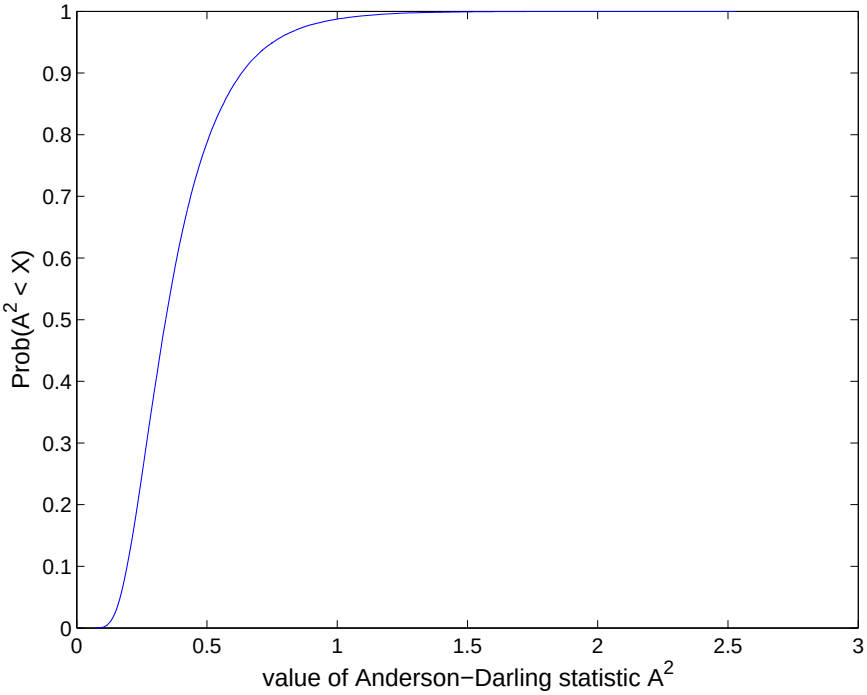


Figure III.3: Anderson-Darling statistic CDF for normally-distributed data where the mean and variance are estimated from the data.

given in Table III.1. Our table is more detailed than earlier published tables. We have used our table of critical values for implementing the Anderson-Darling test.

For the G-means statistic, we are able to compute tables of critical values, but in addition we have found that the distribution of $r_{gm}(X')$ for data drawn from an $N(0,1)$ distribution is approximated well by a Gamma distribution. This makes sense, since the $\chi^2$ distribution is a special case of the Gamma distribution, and the $\chi^2$ is a sum of squared $N(0,1)$ values, as is $r_{gm}(X')$. We found that the Gamma parameters

$$\gamma = n(\pi - 2) - 4 \tag{III.2}$$

$$\beta = 1/\pi \tag{III.3}$$

Table III.1: Critical values of the Anderson-Darling statistic for various levels of statistical confidence, computed by computer simulation of 100,000 experiments.

| $\alpha$ | 0.5 | 0.6 | 0.7 | 0.75 | 0.8 |
|---|---|---|---|---|---|
| critical value | 0.3416 | 0.3843 | 0.4373 | 0.4706 | 0.5110 |
| $\alpha$ | 0.85 | 0.9 | 0.95 | 0.975 | 0.98 |
| critical value | 0.5632 | 0.6329 | 0.7544 | 0.8746 | 0.9163 |
| $\alpha$ | 0.99 | 0.995 | 0.999 | 0.9995 | 0.9999 |
| critical value | 1.0412 | 1.1625 | 1.4607 | 1.5943 | 1.8195 |

fit the best. The Kolmogorov-Smirnov test reveals that the empirical distribution of $r$ differs from the gamma distribution at a significance level of only 0.01 as $n$ grows. We use the Gamma to approximate the true distribution since it makes the algorithm faster and easier to implement.

### III.C.5 Splitting initialization

Two ways to initialize the two child centers prior to the hypothesis test each give a different algorithm. The first way is to randomly choose a $d$-dimensional vector $m$, and let the two child centers be $c \pm m$, where $c$ is the parent center (and the mean of the data). To be effective, $m$ should have a small norm with respect to the standard deviation of the data belonging to $c$. This method is very fast (the computational complexity of the entire algorithm remains linear in time/space).

A second way to initialize the two child centers is to find the first principal component $s$ (the eigenvector of the covariance matrix with the largest eigenvalue $\lambda$), and set them initially to $c \pm s\sqrt{2\lambda/\pi}$. This deterministic method places the two centers in their expected locations if the data truly are Gaus-

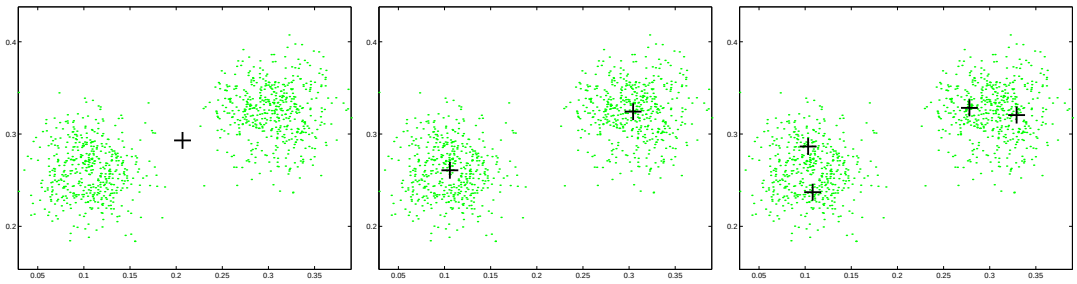

Figure III.4: An example of running G-means for three iterations on a 2-dimensional dataset with two true clusters and 1000 points. Starting with one center (left plot), G-means splits into two centers (middle). The test for normality is significant, so G-means rejects $H_0$ and keeps the split. After splitting each center again (right), the test values are *not* significant, so G-means accepts $H_0$ for both tests and does not accept these splits. The middle plot is the G-means answer. See the text for further details.

sian (see Appendix A), potentially reducing the number of iterations required for $k$-means. Calculating the covariance matrix requires time $O(nd^2)$ and space $O(d^2)$. Finding the eigendecomposition naïvely takes time $O(d^3)$, but since we only want the largest eigenvalue/vector pair, we can use faster methods like the power method, which takes time that is at most linear in the ratio of the two largest eigenvectors [20]. There are also other methods for speeding the convergence of the power method. All the results reported in this chapter use the eigenvector-based splitting heuristic.

## III.D Experiments

### III.D.1 A small example

Figure III.4 shows a run of the G-means algorithm on a synthetic dataset with two true clusters and 1000 points, using $\alpha = 0.0001$. The critical value for

the Anderson-Darling test is 1.8195 for this confidence level. Starting with one center, after one iteration of G-means, we have 2 centers and the $A_*^2$ statistic is 38.103. This is much larger than the critical value, so we reject $H_0$ and accept this split. On the next iteration, we split each new center and repeat the statistical test. The $A_*^2$ values for the two splits are 0.386 and 0.496, both of which are well below the critical value. Therefore we accept $H_0$ for both tests, and discard these splits. Thus G-means with the Anderson-Darling test gives a final answer of $k = 2$.

For the same example, we now look at the relevant test using the G-means statistic instead of the Anderson-Darling statistic. For the first split,

| | two centers |
|---|---|
| $n$ | 1000 |
| $r_{gm}(X')$ | 76.4 |
| $\Gamma_{\alpha/2}$ | 321.8 |
| $\Gamma_{1-\alpha/2}$ | 405.3 |

Here $r_{gm}$ is the value of the distortion statistic, which is smaller than the critical value $\Gamma_{\alpha/2}$, so we reject $H_0$ and accept the split. Next we split each of the previous two centers, giving $k = 4$ total centers. Now the statistics are:

| | centers 1 & 2 | centers 3 & 4 |
|---|---|---|
| $n$ | 509 | 491 |
| $r_{gm}(X')$ | 164.2 | 167.0 |
| $\Gamma_{\alpha/2}$ | 155.4 | 149.4 |
| $\Gamma_{1-\alpha/2}$ | 214.9 | 207.8 |

The observed values of $r_{gm}$ are both not significant compared to the $\Gamma$ critical values. We accept $H_0$ for both, rejecting each split. Thus the final answer is $k = 2$.

### III.D.2 Synthetic datasets

Tables III.2 through III.6 show the results from running $k$-means, G-means (with both statistical tests) and $X$-means on many large synthetic datasets

Table III.2: Synthetic spherical data, separation = $1.5\sigma$

| Dataset | Method | $k$ found | Distortion × opt. | BIC $\times 10^3$ | Time × $k$-means |
|---|---|---|---|---|---|
| $d$=2, $k$=5 | $k$-means | | 1.1±0.3 | -38.0±1.3 | |
| | G-means(GM) | 5.9±3.0 | 1.8±3.4 | -33.1±8.7 | 6.3 |
| | G-means(AD) | 11.1±11.6 | 0.8±0.3 | **-20.8±31.7** | 24.1 |
| | $X$-means | **5.0±0.3** | **1.0±0.0** | -37.6±1.4 | **4.9** |
| $d$=2, $k$=20 | $k$-means | | 4.3±2.1 | 7.3±2.1 | |
| | G-means(GM) | 19.5±1.2 | 1.6±1.7 | 11.7±5.4 | 2.1 |
| | G-means(AD) | 20.5±0.9 | **1.0±0.0** | **15.2±2.4** | 2.4 |
| | $X$-means | **20.2±1.8** | 1.0±0.1 | 14.4±4.9 | **1.8** |
| $d$=2, $k$=80 | $k$-means | | 18.6±10.2 | 163.4±2.1 | |
| | G-means(GM) | 73.8±4.2 | 11.0±10.1 | 151.9±14.7 | 2.9 |
| | G-means(AD) | **78.8±1.0** | **1.1±0.0** | 174.1±3.0 | 2.4 |
| | $X$-means | 81.6±10.2 | 2.4±3.6 | **179.4±27.5** | **1.6** |
| $d$=8, $k$=5 | $k$-means | | 1.1±0.2 | -103.3±4.5 | |
| | G-means(GM) | 5.1±0.3 | **1.0±0.0** | -100.7±2.4 | **4.7** |
| | G-means(AD) | 5.4±0.6 | 1.0±0.0 | **-100.1±2.8** | 6.1 |
| | $X$-means | **5.1±0.2** | **1.0±0.0** | -100.9±2.2 | 34.4 |
| $d$=8, $k$=20 | $k$-means | | 1.4±0.1 | -52.0±2.4 | |
| | G-means(GM) | **20.3±0.7** | **1.0±0.0** | -45.2±2.4 | **2.9** |
| | G-means(AD) | 20.7±0.8 | **1.0±0.0** | **-44.3±2.8** | 3.2 |
| | $X$-means | 20.5±0.8 | **1.0±0.0** | -44.7±2.2 | 13.2 |
| $d$=8, $k$=80 | $k$-means | | 1.6±0.1 | 109.2±2.1 | |
| | G-means(GM) | **80.6±1.7** | 1.0±0.1 | 118.8±5.6 | 3.5 |
| | G-means(AD) | 81.1±1.4 | **1.0±0.0** | **120.7±4.3** | **3.5** |
| | $X$-means | 80.8±0.9 | **1.0±0.0** | 119.9±3.1 | 23.5 |
| $d$=32, $k$=5 | $k$-means | | 1.2±0.2 | -457.8±10.8 | |
| | G-means(GM) | 11.1±4.6 | 1.0±0.0 | **-430.5±12.4** | 15.0 |
| | G-means(AD) | 5.6±0.7 | **1.0±0.0** | -445.1±3.2 | **6.3** |
| | $X$-means | **5.0±0.0** | **1.0±0.0** | -446.8±2.9 | 70.3 |
| $d$=32, $k$=20 | $k$-means | | 1.2±0.1 | -399.8±6.4 | |
| | G-means(GM) | 32.4±4.4 | 1.0±0.0 | **-354.1±10.7** | 5.9 |
| | G-means(AD) | 20.7±0.7 | **1.0±0.0** | -383.0±3.0 | **3.8** |
| | $X$-means | **20.0±0.0** | **1.0±0.0** | -384.7±2.3 | 46.6 |
| $d$=32, $k$=80 | $k$-means | | 1.2±0.0 | -239.9±3.1 | |
| | G-means(GM) | 101.8±6.2 | 1.0±0.0 | **-169.8±14.8** | 7.0 |
| | G-means(AD) | 80.8±1.1 | **1.0±0.0** | -220.2±3.1 | **5.9** |
| | $X$-means | **80.0±0.2** | **1.0±0.0** | -222.0±2.0 | 81.1 |

Table III.3: Synthetic elliptical data, separation = $1.5\sigma$

| Dataset | Method | $k$ found | Distortion × opt. | BIC $\times 10^3$ | Time × $k$-means |
|---|---|---|---|---|---|
| $d$=2, $k$=5 | $k$-means | | 1.1±0.3 | -37.9±1.3 | |
| | G-means(GM) | **15.1±9.9** | **1.1±2.6** | -8.0±26.4 | 28.1 |
| | G-means(AD) | 28.9±11.2 | 0.3±0.3 | **28.8±31.0** | 60.8 |
| | $X$-means | 16.4±4.2 | 0.3±0.2 | -3.3±12.3 | **3.7** |
| $d$=2, $k$=20 | $k$-means | | 4.3±2.2 | 7.5±2.4 | |
| | G-means(GM) | **19.2±2.6** | 3.8±6.2 | 9.4±10.6 | 2.7 |
| | G-means(AD) | 22.1±3.2 | **1.0±0.1** | 19.5±8.2 | 3.8 |
| | $X$-means | 63.0±13.1 | 1.3±3.8 | **123.9±34.2** | **1.5** |
| $d$=2, $k$=80 | $k$-means | | 19.2±9.5 | 163.2±1.9 | |
| | G-means(GM) | 73.4±4.2 | 12.2±10.5 | 150.7±15.0 | 2.8 |
| | G-means(AD) | **79.4±1.4** | **1.0±0.0** | 175.9±3.7 | 2.4 |
| | $X$-means | 179.0±30.2 | 4.1±7.9 | **421.9±74.4** | **1.6** |
| $d$=8, $k$=5 | $k$-means | | 1.2±0.2 | -103.7±3.9 | |
| | G-means(GM) | **5.2±0.6** | **1.0±0.0** | -100.1±3.1 | **3.6** |
| | G-means(AD) | 6.9±2.7 | 0.9±0.1 | -94.4±9.0 | 8.4 |
| | $X$-means | 20.0±0.0 | 0.5±0.0 | **-49.2±2.1** | 13.9 |
| $d$=8, $k$=20 | $k$-means | | 1.3±0.1 | -51.2±2.4 | |
| | G-means(GM) | **20.4±0.7** | **1.0±0.0** | -44.7±2.5 | **3.2** |
| | G-means(AD) | 21.1±1.9 | 1.0±0.0 | -42.8±5.2 | 3.9 |
| | $X$-means | 80.0±0.0 | 0.5±0.0 | **117.0±1.7** | 8.1 |
| $d$=8, $k$=80 | $k$-means | | 1.5±0.1 | 109.6±2.1 | |
| | G-means(GM) | **79.8±1.4** | 1.0±0.1 | 116.9±5.1 | 3.6 |
| | G-means(AD) | 80.3±0.8 | **1.0±0.0** | 118.6±2.9 | **3.4** |
| | $X$-means | 205.0±15.7 | 0.6±0.0 | **435.6±40.1** | 9.5 |
| $d$=32, $k$=5 | $k$-means | | 1.2±0.1 | -460.3±9.8 | |
| | G-means(GM) | **5.0±0.0** | **1.0±0.0** | -446.8±3.0 | **4.2** |
| | G-means(AD) | **5.0±0.0** | **1.0±0.0** | -446.8±3.0 | 4.5 |
| | $X$-means | 20.0±0.0 | 0.8±0.0 | **-390.0±3.1** | 49.4 |
| $d$=32, $k$=20 | $k$-means | | 1.2±0.1 | -402.7±4.9 | |
| | G-means(GM) | **20.0±0.0** | **1.0±0.0** | -384.8±2.3 | **3.5** |
| | G-means(AD) | **20.0±0.0** | **1.0±0.0** | -384.8±2.3 | 3.6 |
| | $X$-means | 80.0±0.0 | 0.8±0.0 | **-221.8±2.6** | 39.8 |
| $d$=32, $k$=80 | $k$-means | | 1.3±0.1 | -240.8±4.1 | |
| | G-means(GM) | 80.6±0.8 | **1.0±0.0** | -220.5±3.7 | **4.5** |
| | G-means(AD) | **80.0±0.2** | **1.0±0.0** | -221.9±2.0 | 4.5 |
| | $X$-means | 159.1±12.9 | 0.9±0.0 | **-25.0±32.5** | 125.7 |

Table III.4: Synthetic spherical data, separation = $3\sigma$

| Dataset | Method | $k$ found | Distortion × opt. | BIC $\times 10^3$ | Time × $k$-means |
|---|---|---|---|---|---|
| $d$=2, $k$=5 | $k$-means | | 4.7±4.9 | -36.6±4.3 | |
| | G-means(GM) | **5.0±0.2** | **1.0±0.0** | -30.9±1.1 | **3.4** |
| | G-means(AD) | 5.3±0.5 | 1.0±0.0 | **-30.2±1.6** | 4.4 |
| | $X$-means | **5.0±0.2** | **1.0±0.0** | -30.9±1.1 | 4.1 |
| $d$=2, $k$=20 | $k$-means | | 15.7±15.4 | 8.8±3.5 | |
| | G-means(GM) | 19.2±3.5 | 36.4±176.7 | 17.4±9.4 | 1.8 |
| | G-means(AD) | **20.1±0.3** | **1.0±0.0** | 21.5±1.4 | 2.0 |
| | $X$-means | 21.4±11.3 | 6.3±18.0 | **22.6±30.4** | **1.5** |
| $d$=2, $k$=80 | $k$-means | | 74.5±39.7 | 163.5±1.9 | |
| | G-means(GM) | 68.7±18.9 | 656.8±2275.7 | 137.1±55.9 | 2.5 |
| | G-means(AD) | **80.1±0.2** | **1.0±0.0** | 184.7±1.3 | **2.3** |
| | $X$-means | 87.2±17.2 | 22.8±31.7 | **194.0±46.8** | 2.5 |
| $d$=8, $k$=5 | $k$-means | | 2.0±0.9 | -85.4±10.0 | |
| | G-means(GM) | 5.0±0.2 | **1.0±0.0** | **-73.8±1.6** | **4.3** |
| | G-means(AD) | **5.0±0.0** | **1.0±0.0** | -73.9±1.6 | 5.0 |
| | $X$-means | **5.0±0.0** | **1.0±0.0** | -73.9±1.6 | 19.5 |
| $d$=8, $k$=20 | $k$-means | | 2.9±0.8 | -39.6±5.0 | |
| | G-means(GM) | 20.3±0.7 | **1.0±0.0** | **-18.0±2.3** | **1.9** |
| | G-means(AD) | 20.2±0.5 | **1.0±0.0** | -18.4±1.9 | 2.1 |
| | $X$-means | **20.1±0.3** | **1.0±0.0** | -18.5±1.7 | 5.8 |
| $d$=8, $k$=80 | $k$-means | | 4.1±0.6 | 117.1±2.7 | |
| | G-means(GM) | 80.3±0.9 | 1.0±0.1 | 145.3±3.3 | 3.0 |
| | G-means(AD) | 80.3±0.6 | **1.0±0.0** | **145.8±2.3** | **3.0** |
| | $X$-means | **80.2±0.4** | **1.0±0.0** | 145.7±1.9 | 10.8 |
| $d$=32, $k$=5 | $k$-means | | 1.7±0.7 | -372.3±32.8 | |
| | G-means(GM) | 10.6±4.7 | 1.0±0.0 | **-321.2±11.9** | 18.9 |
| | G-means(AD) | 5.6±1.2 | **1.0±0.0** | -334.4±4.1 | **9.4** |
| | $X$-means | **5.0±0.0** | **1.0±0.0** | -336.2±2.5 | 80.3 |
| $d$=32, $k$=20 | $k$-means | | 2.2±0.4 | -333.5±16.6 | |
| | G-means(GM) | 32.5±5.7 | 1.0±0.0 | **-242.0±13.7** | 5.1 |
| | G-means(AD) | 20.7±1.1 | **1.0±0.0** | -271.1±3.5 | **2.9** |
| | $X$-means | **20.0±0.0** | **1.0±0.0** | -272.9±2.4 | 39.9 |
| $d$=32, $k$=80 | $k$-means | | 2.4±0.2 | -179.4±6.5 | |
| | G-means(GM) | 101.9±6.7 | 1.0±0.0 | **-57.5±16.3** | 5.9 |
| | G-means(AD) | 80.9±0.9 | **1.0±0.0** | -107.8±2.9 | **4.7** |
| | $X$-means | **80.0±0.0** | **1.0±0.0** | -110.0±2.0 | 136.2 |

Table III.5: Synthetic elliptical data, separation = $3\sigma$

| Dataset | Method | $k$ found | Distortion × opt. | BIC $\times 10^3$ | Time × $k$-means |
|---|---|---|---|---|---|
| $d$=2, $k$=5 | $k$-means | | 2.6±2.4 | -34.4±3.7 | |
| | G-means(GM) | **5.0±0.0** | **1.0±0.0** | -30.9±0.9 | 4.0 |
| | G-means(AD) | 9.2±10.0 | 0.9±0.2 | -19.7±27.1 | 11.9 |
| | $X$-means | 18.2±3.2 | 0.4±0.1 | **7.2±9.1** | **3.3** |
| $d$=2, $k$=20 | $k$-means | | 18.1±10.4 | 7.9±3.2 | |
| | G-means(GM) | 19.0±2.6 | 19.9±49.8 | 14.1±14.5 | 2.1 |
| | G-means(AD) | **20.2±0.7** | **1.0±0.0** | 21.8±1.9 | 2.3 |
| | $X$-means | 70.5±11.4 | 6.8±16.2 | **147.7±34.7** | **1.6** |
| $d$=2, $k$=80 | $k$-means | | 77.6±19.0 | 163.3±1.3 | |
| | G-means(GM) | 71.4±4.3 | 105.8±101.0 | 142.8±16.1 | 2.3 |
| | G-means(AD) | **80.0±0.3** | **1.0±0.0** | 184.9±1.2 | 2.3 |
| | $X$-means | 180.9±32.8 | 45.5±91.3 | **427.1±90.2** | **1.0** |
| $d$=8, $k$=5 | $k$-means | | 2.2±1.1 | -87.9±9.3 | |
| | G-means(GM) | **5.0±0.0** | **1.0±0.0** | -74.6±1.6 | **3.5** |
| | G-means(AD) | **5.0±0.0** | **1.0±0.0** | -74.6±1.6 | 4.2 |
| | $X$-means | 20.0±0.0 | 0.5±0.0 | **-22.7±2.1** | 10.8 |
| $d$=8, $k$=20 | $k$-means | | 3.1±1.0 | -40.1±6.5 | |
| | G-means(GM) | **20.0±0.2** | **1.0±0.0** | -18.6±1.8 | **2.6** |
| | G-means(AD) | **20.0±0.2** | **1.0±0.0** | -18.6±1.8 | 2.8 |
| | $X$-means | 80.0±0.0 | 0.5±0.0 | **143.7±1.7** | 4.9 |
| $d$=8, $k$=80 | $k$-means | | 4.4±0.6 | 115.5±2.8 | |
| | G-means(GM) | 79.5±1.5 | 1.1±0.4 | 141.5±8.6 | **3.0** |
| | G-means(AD) | **80.2±0.6** | **1.0±0.0** | 145.3±2.1 | 3.0 |
| | $X$-means | 218.0±26.3 | 0.6±0.1 | **494.6±66.1** | 7.4 |
| $d$=32, $k$=5 | $k$-means | | 1.8±0.7 | -377.1±32.5 | |
| | G-means(GM) | **5.0±0.0** | **1.0±0.0** | -336.3±2.1 | **5.4** |
| | G-means(AD) | **5.0±0.0** | **1.0±0.0** | -336.3±2.1 | 5.9 |
| | $X$-means | 20.0±0.0 | 0.8±0.0 | **-279.2±2.2** | 47.1 |
| $d$=32, $k$=20 | $k$-means | | 2.1±0.4 | -333.0±15.1 | |
| | G-means(GM) | **20.0±0.0** | **1.0±0.0** | -273.9±2.2 | **2.5** |
| | G-means(AD) | **20.0±0.0** | **1.0±0.0** | -273.9±2.2 | 2.6 |
| | $X$-means | 80.0±0.0 | 0.8±0.1 | **-113.2±8.1** | 30.7 |
| $d$=32, $k$=80 | $k$-means | | 2.4±0.2 | -180.1±7.5 | |
| | G-means(GM) | 80.5±0.6 | **1.0±0.0** | -109.0±2.0 | 3.2 |
| | G-means(AD) | **80.0±0.0** | **1.0±0.0** | -110.4±1.7 | **3.2** |
| | $X$-means | 165.9±14.4 | 0.8±0.0 | **103.8±35.3** | 136.8 |

Table III.6: Synthetic spherical data, separation = $6\sigma$

| Dataset | Method | $k$ found | Distortion × opt. | BIC $\times 10^3$ | Time × $k$-means |
|---|---|---|---|---|---|
| $d$=2, $k$=5 | $k$-means | | 22.1±32.8 | -35.0±7.6 | |
| | G-means(GM) | 5.0±0.2 | **1.0±0.0** | **-24.2±1.3** | **3.5** |
| | G-means(AD) | **5.0±0.0** | **1.0±0.0** | -24.3±1.2 | 4.2 |
| | $X$-means | **5.0±0.0** | **1.0±0.0** | -24.3±1.2 | 3.9 |
| $d$=2, $k$=20 | $k$-means | | 59.0±33.3 | 8.0±2.8 | |
| | G-means(GM) | 18.2±4.6 | 162.1±591.8 | 18.0±19.5 | 1.5 |
| | G-means(AD) | **20.0±0.0** | **1.0±0.0** | **27.5±0.9** | 1.8 |
| | $X$-means | 19.8±2.8 | 16.1±35.9 | 22.4±14.7 | **1.1** |
| $d$=2, $k$=80 | $k$-means | | 297.3±111.7 | 163.1±1.5 | |
| | G-means(GM) | 69.2±6.6 | 480.7±375.2 | 135.6±21.2 | 2.2 |
| | G-means(AD) | **80.0±0.0** | **1.0±0.0** | 191.3±1.1 | **2.1** |
| | $X$-means | 86.4±19.2 | 161.0±229.0 | **192.4±55.8** | 2.5 |
| $d$=8, $k$=5 | $k$-means | | 6.4±4.2 | -77.0±18.9 | |
| | G-means(GM) | 5.1±0.3 | **1.0±0.0** | **-46.3±2.0** | **4.2** |
| | G-means(AD) | 5.0±0.2 | **1.0±0.0** | -46.5±1.7 | 5.0 |
| | $X$-means | **5.0±0.0** | **1.0±0.0** | -46.6±1.7 | 13.9 |
| $d$=8, $k$=20 | $k$-means | | 11.9±4.5 | -38.9±7.2 | |
| | G-means(GM) | 20.4±0.6 | **1.0±0.0** | **10.4±2.4** | **1.7** |
| | G-means(AD) | 20.1±0.2 | **1.0±0.0** | 9.5±1.9 | 1.9 |
| | $X$-means | **20.0±0.0** | **1.0±0.0** | 9.3±1.7 | 3.2 |
| $d$=8, $k$=80 | $k$-means | | 16.5±2.8 | 117.6±3.1 | |
| | G-means(GM) | 79.4±2.6 | 2.8±3.4 | 161.5±24.2 | **2.5** |
| | G-means(AD) | **80.0±0.2** | **1.0±0.0** | **173.5±1.4** | 2.5 |
| | $X$-means | **80.0±0.6** | 1.1±0.6 | 172.4±7.2 | 10.3 |
| $d$=32, $k$=5 | $k$-means | | 4.8±3.1 | -329.3±67.4 | |
| | G-means(GM) | 12.0±6.3 | 1.0±0.0 | **-206.4±16.8** | 18.3 |
| | G-means(AD) | 5.5±0.9 | **1.0±0.0** | -223.4±4.3 | **6.4** |
| | $X$-means | **5.0±0.0** | **1.0±0.0** | -224.9±3.3 | 30.6 |
| $d$=32, $k$=20 | $k$-means | | 6.6±1.9 | -310.5±24.1 | |
| | G-means(GM) | 31.0±5.0 | 1.0±0.0 | **-135.7±11.9** | 4.6 |
| | G-means(AD) | 20.8±0.9 | **1.0±0.0** | -160.7±2.9 | **2.7** |
| | $X$-means | **20.0±0.0** | **1.0±0.0** | -162.8±2.1 | 23.0 |
| $d$=32, $k$=80 | $k$-means | | 7.7±1.1 | -162.1±11.4 | |
| | G-means(GM) | 101.9±5.3 | 1.0±0.0 | **52.7±13.1** | 5.5 |
| | G-means(AD) | 80.8±0.8 | **1.0±0.0** | 1.8±2.6 | **4.2** |
| | $X$-means | **80.0±0.2** | **1.0±0.0** | 0.1±2.0 | 75.0 |

Table III.7: Synthetic elliptical data, separation = $6\sigma$

| Dataset | Method | $k$ found | Distortion × opt. | BIC $\times 10^3$ | Time × $k$-means |
|---|---|---|---|---|---|
| $d$=2, $k$=5 | $k$-means | | 11.4±13.9 | -32.6±6.3 | |
| | G-means(GM) | **5.0±0.2** | **1.0±0.0** | -23.9±1.2 | 4.0 |
| | G-means(AD) | 5.2±1.1 | 1.0±0.0 | -23.5±3.1 | 5.4 |
| | $X$-means | 18.2±3.1 | 0.4±0.2 | **13.7±9.4** | **3.3** |
| $d$=2, $k$=20 | $k$-means | | 71.7±42.5 | 7.6±3.6 | |
| | G-means(GM) | 17.5±4.1 | 171.2±433.0 | 10.0±23.1 | 1.8 |
| | G-means(AD) | **20.2±0.9** | **1.0±0.0** | 28.5±2.4 | 2.2 |
| | $X$-means | 62.9±15.4 | 70.3±144.2 | **129.8±49.4** | **1.2** |
| $d$=2, $k$=80 | $k$-means | | 308.9±116.7 | 162.9±1.3 | |
| | G-means(GM) | 69.7±5.2 | 373.8±285.1 | 137.5±16.4 | 2.4 |
| | G-means(AD) | **80.0±0.0** | **1.0±0.0** | 191.2±1.1 | 2.2 |
| | $X$-means | 176.8±25.8 | 270.9±318.4 | **410.8±69.0** | **1.9** |
| $d$=8, $k$=5 | $k$-means | | 5.1±4.6 | -68.6±21.5 | |
| | G-means(GM) | **5.0±0.0** | **1.0±0.0** | -45.7±2.0 | **5.1** |
| | G-means(AD) | **5.0±0.0** | **1.0±0.0** | -45.7±2.0 | 6.0 |
| | $X$-means | 20.0±0.0 | 0.5±0.0 | **6.0±2.2** | 13.6 |
| $d$=8, $k$=20 | $k$-means | | 11.7±4.4 | -38.3±7.7 | |
| | G-means(GM) | 19.7±1.4 | 2.6±6.7 | 4.4±19.0 | **2.2** |
| | G-means(AD) | **20.0±0.0** | **1.0±0.0** | 9.4±1.8 | 2.4 |
| | $X$-means | 80.0±0.0 | 0.5±0.0 | **171.8±1.8** | 3.2 |
| $d$=8, $k$=80 | $k$-means | | 17.8±3.4 | 116.1±3.2 | |
| | G-means(GM) | 78.5±4.0 | 3.1±5.7 | 160.8±28.8 | 2.5 |
| | G-means(AD) | **80.0±0.2** | **1.0±0.0** | 173.5±1.7 | **2.4** |
| | $X$-means | 240.9±43.4 | 0.6±0.1 | **580.3±109.0** | 4.5 |
| $d$=32, $k$=5 | $k$-means | | 4.8±3.0 | -328.1±70.2 | |
| | G-means(GM) | **5.0±0.0** | **1.0±0.0** | -225.0±3.0 | **4.9** |
| | G-means(AD) | **5.0±0.0** | **1.0±0.0** | -225.0±3.0 | 5.3 |
| | $X$-means | 20.0±0.0 | 0.8±0.0 | **-168.3±3.2** | 25.5 |
| $d$=32, $k$=20 | $k$-means | | 6.7±1.7 | -312.4±20.9 | |
| | G-means(GM) | **20.0±0.0** | **1.0±0.0** | -163.0±2.3 | **2.4** |
| | G-means(AD) | **20.0±0.0** | **1.0±0.0** | -163.0±2.3 | 2.4 |
| | $X$-means | 80.0±0.0 | 0.8±0.0 | **-0.8±2.3** | 19.3 |
| $d$=32, $k$=80 | $k$-means | | 7.5±1.0 | -160.4±10.3 | |
| | G-means(GM) | 80.7±0.6 | **1.0±0.0** | 1.5±3.2 | **2.6** |
| | G-means(AD) | **80.0±0.0** | **1.0±0.0** | -0.2±2.0 | 2.6 |
| | $X$-means | 169.3±10.5 | 0.8±0.0 | **222.4±26.2** | 112.6 |

with different parameters. The parameters we investigate are: cluster shape (spherical versus elliptical), cluster overlap, dimension, and number of true clusters. Half of the experiments are on datasets with spherically-distributed clusters, and the other half are on data with slightly elliptical clusters. For both spherical and elliptical clusters, we test for data that is allowed to overlap very much (allowing clusters within $1.5\sigma$ of each other), somewhat ($3\sigma$), and hardly at all ($6\sigma$). Here $\sigma$ is the standard deviation of $||x_i - c_j||$, the distance between each point and its assigned center.

The synthetic datasets used here each have 5000 datapoints in $d = 2/8/32$ dimensions. The true $k$s are 5, 20, and 80, which give average cluster sizes of 1000, 250, and 62.5 points per cluster, respectively. For each synthetic dataset type, we generate 30 datasets with the true center means chosen uniformly randomly from the unit hypercube, and choosing $\sigma$ so that no two clusters are closer than $1.5\sigma$, $3\sigma$, or $6\sigma$ apart, depending on the experiment. For three of the experiments, each cluster is spherical, while for the other three experiments each cluster is also given a transformation to make it non-spherical, by multiplying the data by randomly chosen scaling and rotation matrices. We run $k$-means with the correct $k$, starting from a random sampling initialization. We run G-means starting with one center. We allow $X$-means to search between 2 and $4k$ centers (where here $k$ is the true number of clusters).

When reading these datasets, the best values are in boldface. The best results are indicated by three things in conjunction: finding the correct $k$, having a distortion ratio that is close to 1, and having a fast run-time. Thus when an algorithm consistently estimates the correct $k$ and has a distortion ratio that is close to 1.0, the algorithm is finding the original distribution that generated the dataset. The BIC measurements are given for reference to the $X$-means metric of quality.

On spherical data that overlaps heavily (seen in Table III.2, $X$-means has a slight edge on the simplest dataset ($k$=5, $d$=2). Here G-means with the Anderson-Darling test is overfitting the data on rare occasions (hence the high standard deviation in the estimated $k$), while the same algorithm with the G-means statistical test is doing well in terms of estimating $k$, but the distortion measure shows that it is not actually finding good clusterings on average. However, outside of this one dataset, both $X$-means and G-means with the Anderson-Darling test do well at finding the correct clusterings on spherical data. When the number of dimensions is high, the Anderson-Darling test performs better than the G-means test; see Section III.E.2 for more on how dimension affects these tests.

In terms of distortion, $k$-means does well when clusters overlap, as discussed in the previous chapter. However, when clusters do not overlap, the $k$-means algorithm does poorly, since the hard assignment of datapoints to cluster centers prevents the centers from moving freely through the input space. This illustrates a problematic duality: it is easier to estimate the number of clusters when the clusters are well-separated (intuitively, and as we show here empirically), but it is easier to use an algorithm like $k$-means on data which overlaps, since the landscape of clustering solutions is smoothly connected. Even in the best conditions and given the correct $k$, however, $k$-means still does not do as well as G-means in finding correct clusterings.

For other experiments with spherical data (see Tables III.4 and III.6), G-means and $X$-means are equally good at finding the correct clusterings. For non-spherical data, however, the results are very different. G-means is consistently better than $X$-means at finding both the correct number of clusters, and finding the correct clustering according to the distortion. In most cases, $X$-means chooses at or near the maximum number of centers we allow it to search with (we bound

the search by 4 times the true $k$). Most real-world data is not spherical, and in such cases these results indicate that $X$-means will choose too many clusters. Figure III.5 shows an example clustering by G-means with the Anderson-Darling test, and the same data clustered by $X$-means. The dataset is from $k = 5$, $d = 2$, and separation $= 3\sigma$. The BIC causes $X$-means to choose too many clusters (20), and moreover it does not distribute them evenly.

Across the experiments, there is a general correlation that choosing more centers leads to a larger BIC score, even in the spherical data for which the BIC is tuned. $X$-means chooses a clustering with the largest BIC score, which leads it to overfit the data as a result. This indicates that the BIC is not putting significant enough penalty on the model's complexity. In the few cases where G-means chose too many centers, it also often got the highest BIC score.

On all datasets where $d > 2$, G-means is faster than $X$-means. Each value in the time column is a multiple of the time it takes our implementation of $k$-means. All our code is written in Matlab; $X$-means is written in C. $X$-means is designed to be fast by using $kd$-tree data structures to partition the input space. However, these structures are known to be slow in even moderate numbers of dimensions (e.g. greater than around 10 dimensions), since they have run time that is exponential in $d$. There have been many papers that work on reducing the computational cost of clustering and instance learning through input space partitioning with such structures as $kd$-trees [55, 38], $R$-trees [28], $X$-trees [6], ball-trees [49], and approximate nearest neighbor searching [50, 42]. Outside of randomized methods, it remains a very difficult problem to do spatial search in high dimensions, and is often fastest to use brute-force search through all $O(n)$ comparisons with $O(d)$ cost per comparison for Euclidean distances. If the distance metric is very costly, methods such of reducing the number of comparisons required [24] can be effective. In the $X$-means paper, Pelleg and Moore claim

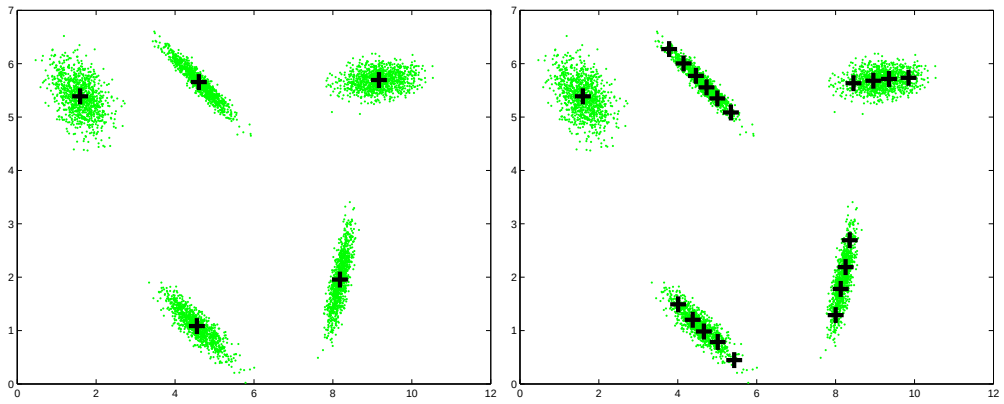

Figure III.5: 2-$d$ synthetic dataset with 5 true clusters. On the left, G-means correctly chooses 5 centers and deals well with non-spherical data. On the right, the BIC causes $X$-means to overfit the data, choosing 20 unevenly distributed clusters.

that their algorithm is much faster than $k$-means for all their experiments, especially as the number of datapoints grows. Our experiments show that for a moderate number of datapoints, in 2 dimensions $X$-means does not perform as well as $k$-means.

### III.D.3 Architecture datasets

We test G-means on several real-world datasets from computer architecture. These datasets are program traces used to estimate processor performance through simulation [64]. Clustering can identify where the programs are spending most of their time so we can simulate just these areas in detail, rather than the whole program. This saves simulation time while drastically improving prediction of program behavior. The datasets clustered are 15-dimensional random projections of the original datasets (which had thousands of dimensions). For further information on work in this area and these datasets, please see the following chapters.

There is no "correct" number of clusters, but we want an automated way to estimate $k$ and cluster the data. For G-means we again use $\alpha = 0.0001$, while for $X$-means we allowed it to run from $k = 2$ to 1000. Again, we see that the BIC statistic is estimating far more clusters in most cases than G-means, since the data is not spherical. For example, $X$-means chooses 827 centers for the dataset `ss-applu-ref`, but the dataset has only 2239 datapoints – an average of only about 2.7 points per cluster. On average, $X$-means chooses only 7.87 points per cluster. The method which chooses a larger number of clusters will usually have a lower distortion, and a higher BIC score, making these statistics less meaningful for choosing $k$. The BIC does not place enough penalty on each center, as illustrated here.

### III.D.4 Discovering true clusters in labeled data

We tested these algorithms on two real-world datasets for handwritten digit recognition: the NIST dataset [45] and the Pendigits dataset [10]. The goal is to cluster the data without knowledge of the labels and measure how well the clustering captures the true labels. Both datasets have 10 true classes (digits 0-9). NIST has 60000 training examples and 784 dimensions ($28 \times 28$ pixels). We use 6000 randomly chosen examples and we reduce the dimension to 50 by random projection (following [18]). The Pendigits dataset has 7984 examples and 16 dimensions; we did not change the data in any way.

We cluster each dataset with G-means and $X$-means, and measure performance by comparing the cluster labels $L_c$ with the true labels $L_t$. We define the *partition quality* as

$$pq \;=\; \sum_{i=1}^{k_t}\sum_{j=1}^{k_c} p(i,j)^2 \Bigg/ \sum_{i=1}^{k_t} p(i)^2$$

where $k_t$ is the true number of classes, and $k_c$ is the number of clusters found by the algorithm. This metric is maximized when $L_c$ induces the same partition of

the data as $L_t$; in other words, when all points in each cluster have the same true label, and the estimated $k$ is the true $k$. The $p(i, j)$ term is the frequency-based probability that a datapoint will be labeled $i$ by $L_t$ and $j$ by $L_c$. This quality is normalized by the sum of true probabilities, squared. This statistic is related to the Rand statistic for comparing partitions [35].

For the NIST dataset, G-means finds 31 clusters in 30 seconds with a partition quality score of 0.177. $X$-means finds 715 clusters in 4149 seconds, and 369 of these clusters contain only one point, indicating an overestimation problem with the BIC criterion. $X$-means receives a partition quality score of 0.024. For the Pendigits dataset, G-means finds 69 clusters in 30 seconds, with a partition quality score of 0.196; $X$-means finds 235 clusters in 287 seconds, with a partition quality score of 0.057. Figure III.6 shows Hinton diagrams of the G-means clusterings of both datasets, showing that G-means is successful at identifying the true clusters concisely, without aid from the labels. The confusions between different digits in the NIST dataset (seen in the off-diagonal elements) are common for other researchers using more sophisticated techniques, see [18].

### III.D.5 Learning true dimension from clustering

We now turn to the general problem of dimension reduction. Many real-world datasets have a high number of dimensions, and in order to work with them it is often beneficial to reduce the dimension of the data prior to using learning algorithms. This is effective because often the structure of the data may be described in far fewer dimensions, and most learning algorithms perform best when the dimension is low. What we would like is an automated way of learning the underlying dimension of high-dimensional data. This is a well-researched area; we approach it from a slightly different perspective. We suppose that we have a black-box algorithm which can tell us how much "structure" exists in

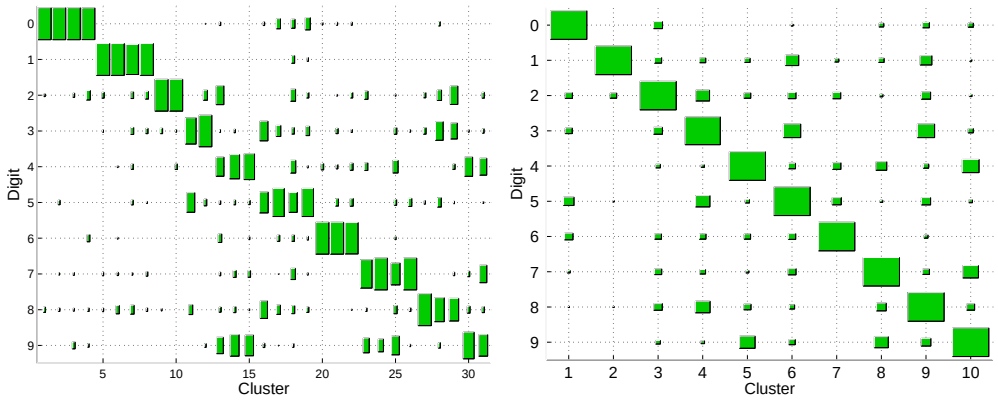


Figure III.6: Hinton diagrams of the correspondence between each true digit (rows) and each cluster (columns) found by G-means (left), and for the optimal clustering with 10 clusters (right). The data is 10% of the NIST digit classification set, projected from 784 down to 50 dimensions (see [18] for why we chose 50). G-means was run without information about the labels, yet it finds meaningful clusters that correspond with the true labels, and even improves the prediction on some digits.

a dataset. We will then use this black box in a generate-and-test fashion to repeatedly determine how much structure exists in various reduced-dimension datasets. Starting with a small number of dimensions and increasing, we will look for the point at which no more structure can be discovered in the dataset by increasing the dimension. This critical point will be the true dimension of the data.

The intuition behind this algorithm is that when data is reduced to some dimension that is too small, structure that can be found in the original data must necessarily be collapsed, and unavailable. However, as the number of dimensions increases, more structure will unfold to be discovered. If there is some lower-dimensional space in which the full structure can be represented, then we can identify that space using our black box. This is related to the reconstruction of

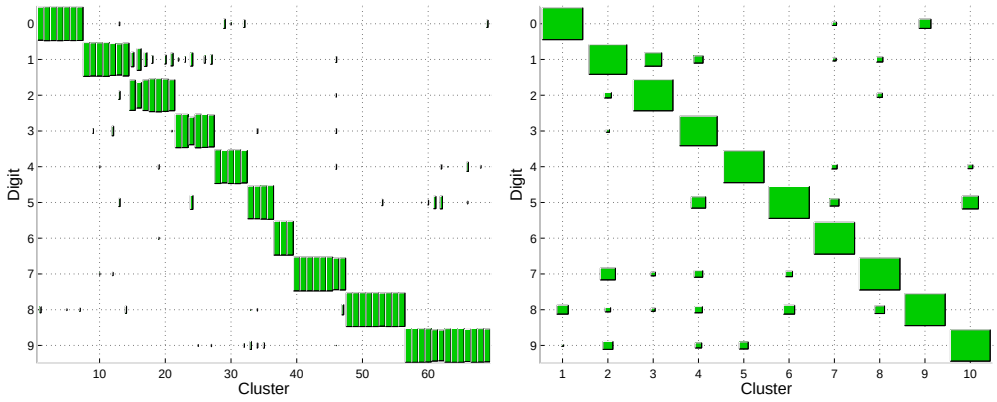


Figure III.7: Hinton diagram of the correspondence between each true digit (rows) and each cluster (columns) found by G-means (left), and for the optimal clustering with 10 clusters (right). The data is the unmodified Pendigits digit classification set. G-means was run without information about the labels, yet it finds meaningful clusters that correspond with the true labels.

dimension in chaotic systems by identifying false nearest neighbors [1] (in chaotic systems parlance, the dimension is actually the number of time steps into the past to observe). Our algorithm for learning dimension is given in Algorithm 6.

To completely define this algorithm, we must fill in a dimension reduction method as well as the black box for measuring structure. To experiment with this technique, we will use two dimension reduction techniques: principal components analysis (PCA), and random linear projection. To measure the structure of the data, we will use the G-means algorithm to find clusters in the data. The number of clusters that it estimates are in the data will be the metric of structure.

We performed several synthetic and real-world experiments to test our dimension reduction algorithm. We generate two datasets: a random dataset with 20 true spherical clusters in 20 dimensions, and a second dataset in the same way, but we add 20 dimensions (for a total of 40) which have uniform noise. We then apply our algorithm, using PCA or random linear projection to reduce the

**Algorithm 6** An algorithm for learning the underlying dimension of data. The input, $X$, is a set of datapoints with dimension $d$. The *reduce-dimension* function returns a version of $X$ that has been reduced to $c$ dimensions. The *compute-structure* function returns a ranked value for the amount of structure that exists in the dataset $Y$, such as the number of clusters. The algorithm returns the minimum dimension for which the maximum structure has been seen.

**Learn-dimension**($X$)

1: let $d$ be the original dimension of $X$
2: **for** $c \in \{1, \ldots, d\}$ **do**
3: $\quad Y \leftarrow$ reduce-dimension$(X, c)$
4: $\quad s_c \leftarrow$ compute-structure$(Y)$
5: **end for**
6: return $\min\{c | s_c = \max(s)\}$

dimension, and using G-means to rank the structure of each reduced-dimension dataset. We use the Anderson-Darling test for normality. The left plot in Figure III.8 shows the ranked eigenvalues for the first dataset; it is not clear from this plot how many dimensions are significant. However, the right plot in the same figure shows that the data can be sufficiently represented in only 8 dimensions using PCA; much smaller than the 20 original dimensions. Figure III.9 shows that the second dataset with 20 extra noise dimensions can also be captured in a small number of dimensions, about 10. Random linear projection also works, but not as effectively as PCA in these cases.

Figure III.10 shows the results from learning the true dimension of the 8163-dimensional dataset `gzip`. This dataset shows that it only requires 12 to 15 dimensions to exhibit all of its structure, which is a radical reduction in the dimension. For this test the original data has too many dimensions to compute the principal components quickly (requiring $O(d^3)$ operations), so random pro-

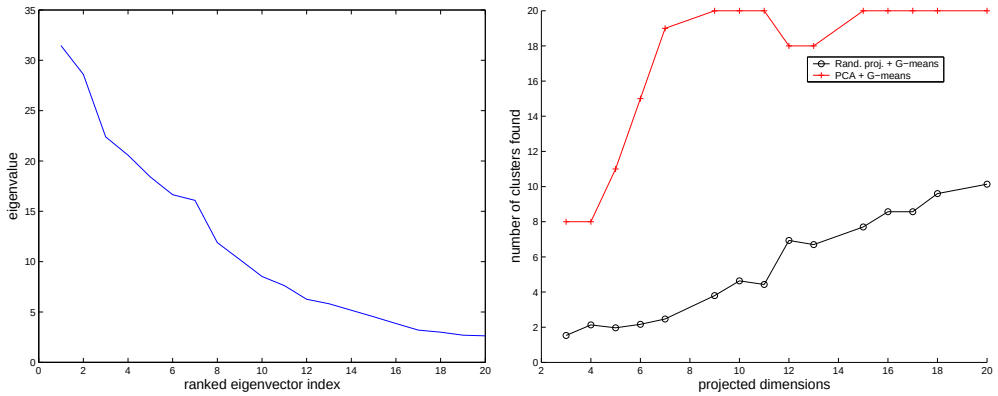


Figure III.8: Results from learning the dimension on data without noisy features. The data is 500 datapoints in 20 dimensions that form 20 clusters. The left plot shows the ranked eigenvalues for the original data; they do not provide much help in determining the appropriate number of dimensions. However, the plot on the right shows our dimension-learning algorithm for PCA and random projection, which show that the data may be reduced to about 8 dimensions using PCA.

jection (which is linear) is the preferred alternative, and it works well for such high-dimensional applications.

Figure III.11 shows the results from learning the true dimension of the 784-dimensional NIST dataset. For this experiment we use random projection and PCA. Using PCA we are able to find more structure in the data, but the faster random projection also does well at unlocking structure. These results support the 50-dimension random projection proposed by Dasgupta for this data [18].

## III.E Statistical power and dimensional effects

Here we will discuss a few aspects of hypothesis testing for normality.

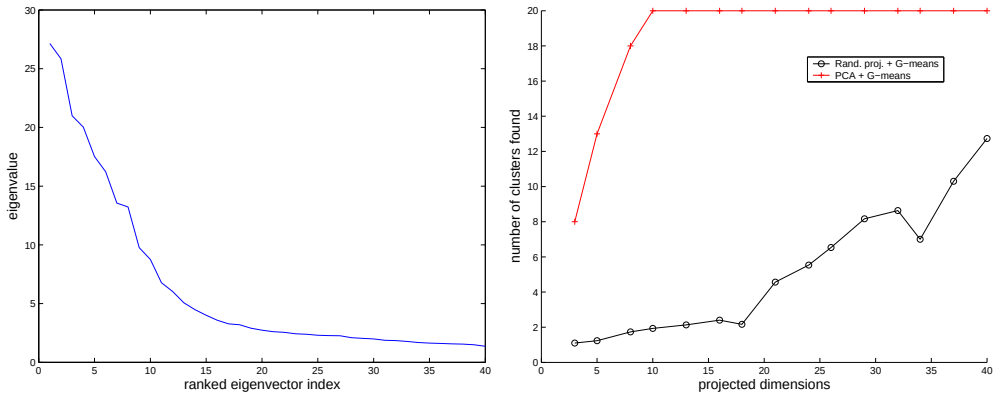


Figure III.9: Results from learning the dimension on data with noisy features. The data is 500 datapoints in 40 dimensions that form 20 clusters. Of the 40 dimensions, 20 are purely noise. The left plot shows the ranked eigenvalues for the original data; they provide only a little help in determining the appropriate number of dimensions. However, the plot on the right shows our dimension-learning algorithm for PCA and random projection, which show that the data may be reduced to about 10 dimensions using PCA.

### III.E.1 Statistical power

Figures III.12 and III.13 show the power of the Anderson-Darling test, G-means test, and BIC. In Figure III.12, we generate 1000 experiments of data from one true normal cluster in 1 or 2 dimensions for the test when $H_0$ is true (left plots), and from 2 normal clusters separated by $5\sigma$ when $H_1$ is true (right plots). These show the frequency of making Type I/II errors for each test. For the G-means and Anderson-Darling tests, we fix the significance level to be 0.001. It is clear from these plots that the BIC tends to choose to split clusters when the number of datapoints is small, while the other two tests do not. This is a benefit of our fixed-significance tests, which will not overfit when the number of datapoints is small.

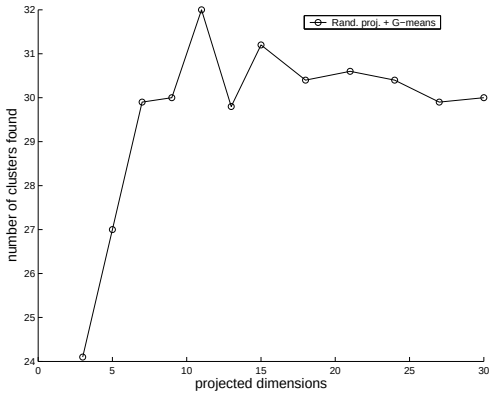


Figure III.10: Results from learning the dimension on high-dimensional architecture dataset `gzip`. The data is 1038 datapoints (basic block vectors) from a trace of the program `gzip`, with 8163 dimensions. In such a high dimensional space, it would be very hard to use a distance-based learning algorithm, such as $k$-means. We use random projection and G-means to learn the underlying dimension, and from this we find that 12 to 15 dimensions are appropriate for learning in this data, which is a radical reduction from 8163.

Figure III.13 illustrates how the BIC will overfit when clusters are not spherical, as is common in real-world datasets. This dataset has one true cluster in two dimensions, elongated in one dimension being twice as much as the other. This forms a central problem for the BIC, and illustrates why it is difficult to use complexity-penalty models for general-purpose tasks. Often, it is necessary to tune complexity models to the task at hand.

The BIC can be considered a likelihood ratio test, but with a significance level that cannot be fixed. The significance level instead varies depending on $n$ and $\Delta k$, the difference in the number of model parameters between two compared models. As $n$ or $\Delta k$ decrease, the significance level increases (becomes weaker as a statistical test) [39]. In [41] the authors show that complexity-penalty based methods require tuning for problem-specific penalties and don't generalize as well

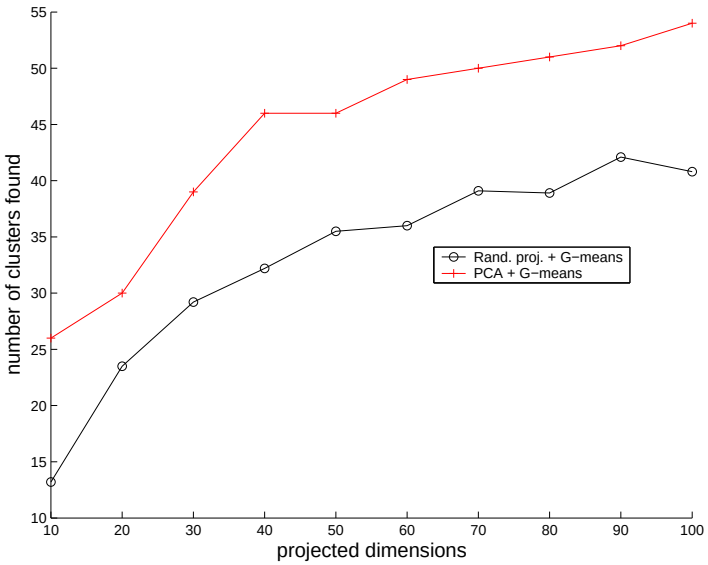


Figure III.11: Results from learning the dimension on high-dimensional NIST dataset for recognizing handwritten digits. The data is 6000 images of handwritten digits, at $28 \times 28 = 784$ pixel resolution (dimensions). We use random projection and PCA combined with G-means to learn the underlying dimension. From this result we can see that 50 dimensions are reasonable, when using random projection, to capture most of the structure in the data. This result supports the best small dimension for supervised classification error found by Dasgupta [18].

as other methods, such as cross validation.

### III.E.2 High-dimensional effects

Consider data drawn from a high-dimensional Gaussian distribution. The one-dimensional projection that we use makes it simple to test the hypothesis that the data is sampled from a Gaussian. Under normal conditions, it is expected that the linear projection of a Gaussian into a lower-dimensional space also appears Gaussian (even "more Gaussian", due to the central limit theorem).

However, when the number of datapoints $n$ is small, it is the case that there will exist many projections where the projected data will look bi-modal. This is a non-intuitive result which was also discovered by Day [19], and attributed to bias in the estimates using maximum-likelihood when $n$ is small. As $n$ grows, however, the data look more Gaussian.

As it applies to our work, we have found that $k$-means often finds projections of high-dimensional Gaussian data which look very non-Gaussian, especially when the number of samples is small. This is a double-edged sword: it means that $k$-means is truly good at finding "interesting projections" in high dimensions, as in the goal of projection pursuit. However, it also means that as the dimension grows, our algorithm will have a tendency to choose more centers than it should. Figure III.15 shows examples of a one-dimensional projection of data sampled from a high-dimensional Gaussian, where the projections that are found by $k$-means give results that look very non-Gaussian. Contrast this with Figure III.14, where the projections chosen are random, and the data retains its Gaussian properties. Table III.9 shows the Anderson-Darling test statistics for these plots. As the dimension grows, the statistic detects that the data looks non-Gaussian for projections chosen by $k$-means. Again, this indicates that $k$-means is doing a great job of finding projections that are interesting in the data, but it also means that we will have to employ even more sophisticated tests to avoid overfitting the data when clustering high-dimensional data. Of course, using random projection prior to clustering (as in [18]) would aid in this.

## III.F Conclusion

We have introduced a new statistical test for determining whether data points are a random sample from a Gaussian distribution with arbitrary dimension and covariance matrix. The new test uses a distortion statistic that is well-

approximated by a Gamma distribution, making the test easy to implement. Our G-means algorithm uses this statistical test as a wrapper around $k$-means to discover the number of clusters $k$ automatically. The only parameter supplied to the algorithm is $\alpha$, the significance level of the statistical test, which can easily be set in a standard way. We have also utilized the Anderson-Darling test for normality in our G-means algorithm, and found that it also performs very well in our framework.

Empirically, the G-means algorithm works well at finding the correct number of clusters and the locations of genuine cluster centers, and we have shown it works well in moderately high dimensions. We have also shown that the BIC complexity penalty, as it is formulated for $X$-means, tends to overfit (choose too many clusters) when looking at small numbers of points, or when data have a non-spherical distribution. The statistical test used by G-means could also be used as a test to *merge* centers rather than split them; this is room for future work. The G-means algorithm takes linear time and space (plus the cost of the splitting heuristic) in the number of data points $n$ and the dimension $d$, since $k$-means is itself linear in time and space.

Clustering in high dimensions has been an open problem for many years. Recent research has shown that it may be preferable to use dimensionality reduction techniques before clustering, and then use a low-dimensional clustering algorithm such as $k$-means, rather than clustering in the high dimension directly. In [18] the author shows that using a simple, inexpensive linear projection preserves much of the properties of data (such as cluster distances), while making it easier to find the clusters. Thus there is a need for good-quality, fast clustering algorithms for low-dimensional data. This work is a step in this direction.

We have introduced a novel method for learning the underlying dimension of data by ranking the structures that can be found in varying dimensions.

Our method uses simple dimension reduction techniques like principal components analysis or random linear projection. We have shown that this method works well at finding the true underlying number of dimensions in a dataset, both on synthetic and real-world data.

Additionally, recent image segmentation algorithms such as normalized cut [65, 51] are based on eigenvector computations on distance matrices. These "spectral" clustering algorithms still use $k$-means as a post-processing step to find the actual segmentation (usually in a lower-dimensional space than the original input) and they require $k$ to be specified externally. Thus we expect G-means will be useful in combination with spectral clustering.

## III.G Acknowledgements

This chapter, in part, is a reprint of material that has been submitted for publication. The dissertation author was the primary researcher and author with Charles Elkan.

Table III.8: Results of estimating the number of clusters on computer architecture datasets. Though there is no "true" number of clusters, $X$-means is choosing many more centers than G-means for each dataset. The average number of points per cluster is very low for $X$-means (averaging 7.87 points per cluster across all datasets), indicating that it is overfitting.

| Dataset | Method | $k$ found | $k/n$ | Raw distortion | BIC $\times 10^3$ | Time |
|---|---|---|---|---|---|---|
| `ss-applu-ref` | G-means(GM) | 99 | 22.6 | 159.5 | 24.2 | 16.3 |
| $n = 2239$ | G-means(AD) | 103 | 21.7 | 158.4 | 33.0 | 23.4 |
| $d = 15$ | $X$-means | 827 | 2.7 | 975.7 | 1.0 | 4.8 |
| `ss-apsi-ref` | G-means(GM) | 105 | 33.1 | 262.6 | 33.8 | 7.8 |
| $n = 3480$ | G-means(AD) | 168 | 20.7 | 377.5 | 23.2 | 27.7 |
| $d = 15$ | $X$-means | 563 | 6.1 | 1140.3 | 0.4 | 34.9 |
| `ss-bzip2-graphic-ref` | G-means(GM) | 36 | 39.8 | 49.9 | 38.6 | 1.5 |
| $n = 1436$ | G-means(AD) | 39 | 36.8 | 50.5 | 43.8 | 2.3 |
| $d = 15$ | $X$-means | 471 | 3.0 | 370.9 | 1.1 | 7.4 |
| `ss-galgel-ref` | G-means(GM) | 76 | 53.8 | 207.5 | 171.2 | 6.1 |
| $n = 4094$ | G-means(AD) | 301 | 13.6 | 688.5 | 51.0 | 37.0 |
| $d = 15$ | $X$-means | 118 | 34.6 | 341.1 | 32.7 | 14.0 |
| `ss-gcc-00-166-ref` | G-means(GM) | 13 | 36.1 | 5.4 | 71.0 | 0.1 |
| $n = 470$ | G-means(AD) | 26 | 18.0 | 11.8 | 22.1 | 0.3 |
| $d = 15$ | $X$-means | 147 | 3.1 | 38.5 | 4.7 | 0.7 |
| `ss-gcc-00-integrate-ref` | G-means(GM) | 12 | 11.0 | 1.9 | 9.8 | 0.0 |
| $n = 132$ | G-means(AD) | 16 | 8.2 | 2.1 | 8.5 | 0.1 |
| $d = 15$ | $X$-means | 44 | 3.0 | 3.1 | 5.1 | 0.0 |
| `ss-gzip-random-ref` | G-means(GM) | 12 | 68.5 | 28.3 | 5.5 | 0.1 |
| $n = 822$ | G-means(AD) | 55 | 14.9 | 47.3 | 2.9 | 1.1 |
| $d = 15$ | $X$-means | 172 | 4.7 | 110.4 | 0.0 | 1.8 |
| `ss-wupwise-ref` | G-means(GM) | 75 | 46.6 | 248.2 | 9.1 | 7.6 |
| $n = 3497$ | G-means(AD) | 151 | 23.1 | 380.9 | 7.4 | 14.6 |
| $d = 15$ | $X$-means | 600 | 5.8 | 1215.4 | 0.3 | 39.0 |

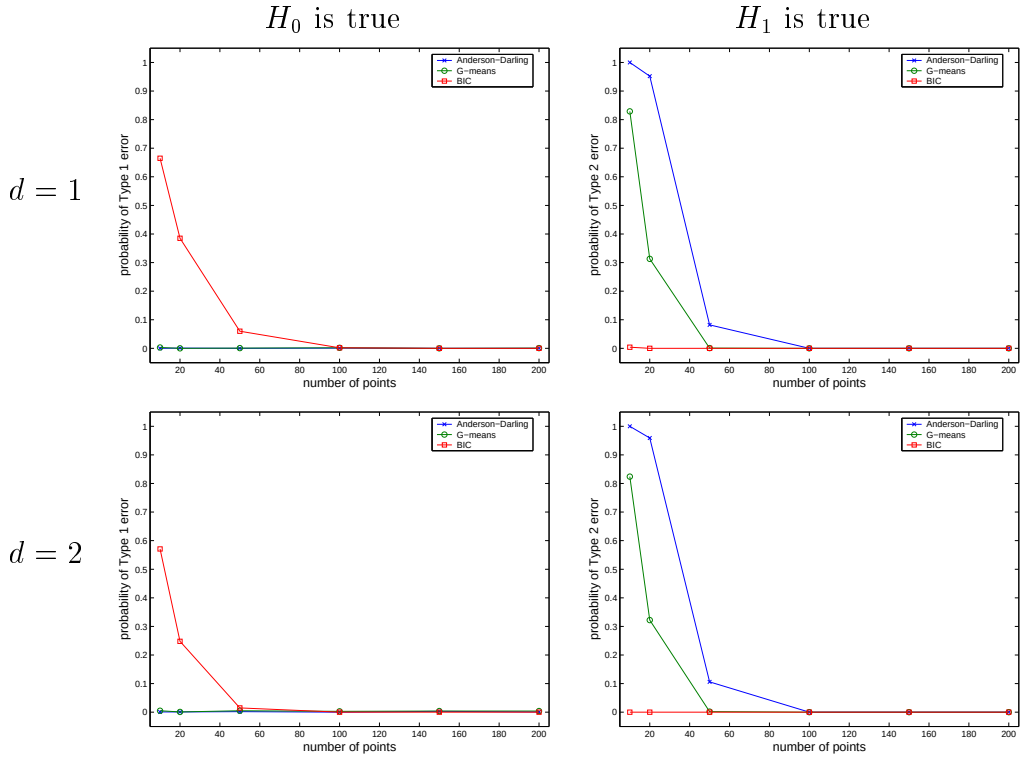


Figure III.12: A comparison of the power of the G-means and Anderson-Darling hypothesis tests versus the BIC. For the hypothesis tests we fixed the significance level ($\alpha = 0.001$), while the significance level of the BIC test depends on the number of points. The left plots shows the probability of incorrectly splitting (Type I error) one true spherical cluster, in 1 or 2 dimensions . The right plots shows the probability of incorrectly *not* splitting two true clusters separated by $5\sigma$ (Type II error). Both plots are functions of $n$. Both plots show that the BIC overfits (splits clusters) when $n$ is small.

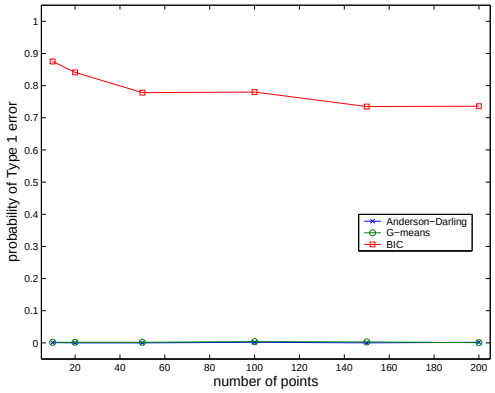


Figure III.13: A comparison of the power of the G-means and Anderson-Darling hypothesis tests versus the BIC. For the hypothesis tests we fixed the significance level ($\alpha = 0.001$), while the significance level of the BIC test depends on the number of points. The data here is one true elliptical cluster in two dimensions. The BIC almost always chooses to incorrectly split, a result of its tuning for spherical data.

Table III.9: Anderson-Darling statistics for 1-dimensional projected Gaussian data using a projection found by $k$-means (left table), and found by random projection (right table). We can see that the data looks pronouncedly non-Gaussian as the dimension grows, when using the $k$-means projection, since the statistic values are growing. See Figures III.14 and III.15 for the plots of this data.

| | $k$-means projection | | | random projection | | |
|---|---|---|---|---|---|---|
| $n \backslash d$ | 1 | 10 | 100 | 1 | 10 | 100 |
| 50 | 0.334 | 0.802 | 0.588 | 0.334 | 0.360 | 0.487 |
| 500 | 0.312 | 0.625 | 3.462 | 0.312 | 0.250 | 0.341 |
| 5000 | 0.509 | 0.962 | 4.765 | 0.509 | 0.384 | 0.445 |

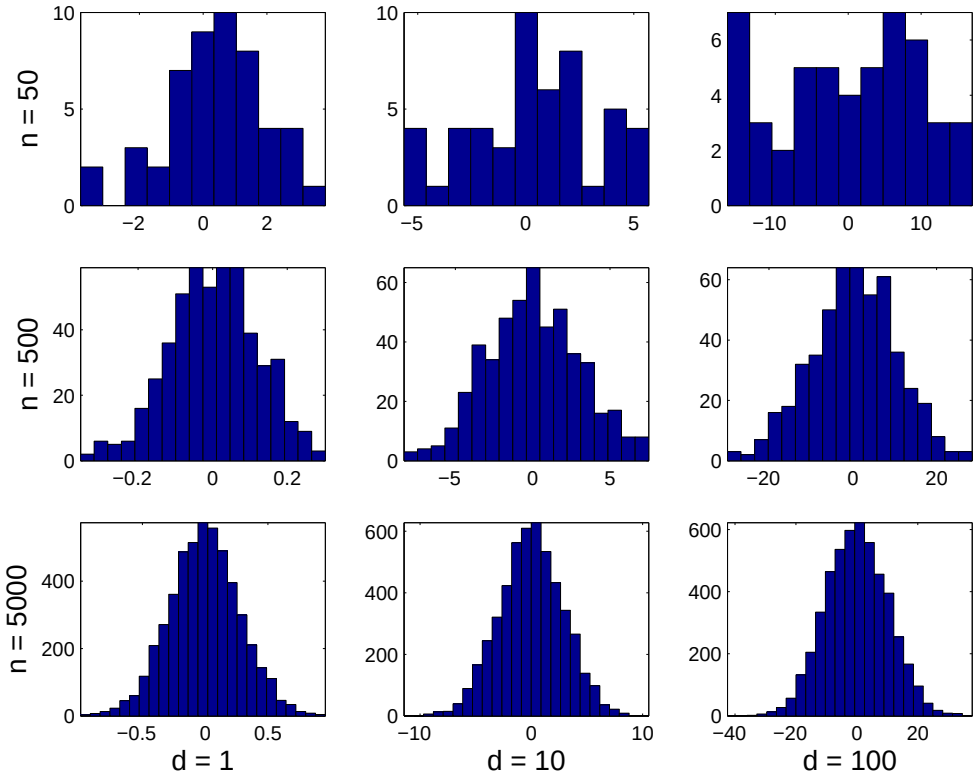


Figure III.14: These histograms show the distributions of data sampled from 1, 10, and 100 dimensions that has been projected down to one dimension using a randomly chosen projection. Note that the data retains its properties of looking Gaussian in this low-dimensional representation. The plots in each row correspond to 50, 500, or 5000 samples. Contrast this plot with Figure III.15.

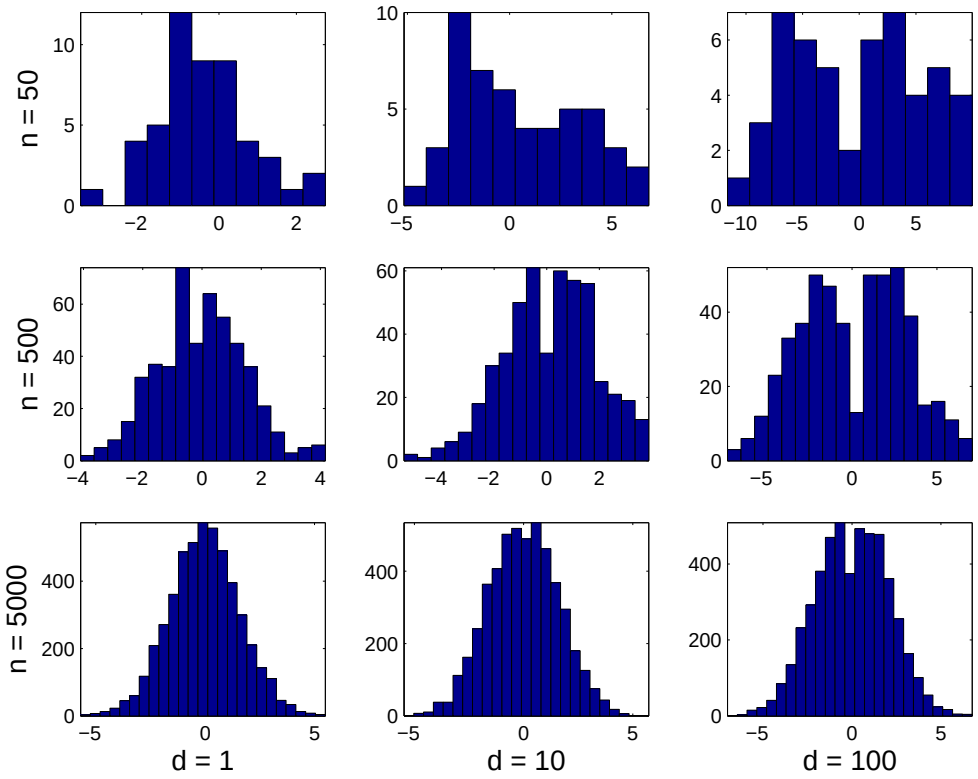


Figure III.15: These histograms show the distributions of data sampled from 1, 10, and 100 dimensions that has been projected down to one dimension using a randomly chosen projection. Note that the data retains its properties of looking Gaussian in this low-dimensional representation. The plots in each row correspond to 50, 500, or 5000 samples. Contrast this plot with Figure III.14.

# IV

# Learning and predicting program behavior through clustering

## IV.A Introduction

Until now we have been discussing the general problem of finding good-quality clusterings in data. We now turn to an application of clustering in the field of computer architecture. Computer architecture designers are interested in optimizing computer processors for many things: overall speed, cache efficiency, power consumption, etc. Modern designers test their planned optimizations virtually, in software simulation of a processor. These simulations are very computationally expensive, since processors are very complex. Therefore what would require a modern CPU (such as a Pentium 4-class chip) several minutes to do on the real hardware may take up to a month to do in simulation.

With this difficulty comes an rather large opportunity for improvement, which is what both this chapter and the next chapter are about. We are interested in applying clustering to the task of improving both the quality and the efficiency of processor simulation. We use clustering to discover the dynamic structure of computer programs, and exploit this information in the simulation process.

### IV.A.1 Choosing the best $k$ can be application-specific

At the heart of data clustering are two questions, as I have already outlined: how many clusters are there, and what are the clusters? In the prior two chapters of this dissertation, I have focused on work that assumes that the data supports some ideal number of clusters that is self-evident in the data and may be detected by clustering. But depending on who is answering the question of how many clusters exist in a given dataset, a researcher may receive many different answers. In this chapter and the next, we discuss the application of clustering to a particular problem of modeling data.

For this application we want the model to be as exact as possible, and in the limit we would not use the model at all if we had limitless computing resources; thus every datapoint would be a cluster unto itself. However, the practical limitations of computing are what enable this research, and therefore cluster models become very useful. However, the question "what is $k$ (the number of clusters)?" is not the same question as in previous chapters. Instead, in this application one question we want to answer is "what is the smallest $k$ I can use to get good results"? The crux of the difference lies in the definition of "good results". In this chapter and the next we will see a wholly different approach to choosing the number of clusters that is answered in ways that are specific to architecture simulation. We point out that is very important when doing research to understand the questions being asked, as illustrated by the question of choosing the best $k$.

### IV.A.2 Background on processor simulation

I will take a bit of space now to explain how computer processor simulation typically works (from my viewpoint as a machine learning researcher), for the benefit of those who do not have a background in the area. Similar to the

way that machine learning researchers develop new algorithms and test them on datasets as one way of seeing how well they work, computer architects develop optimizations for computer processors and test them on program workloads. These workloads are standardized in benchmarks.

For example, the SPEC2000 benchmarks have two sets of programs that are representative of various tasks that computers perform. The programs in one set primarily do integer computations (such as a program like `gzip` which does data compression). Integer programs, as they are called, spend most of their time using integer instructions, may have many jumps in the program, and typically have complex structure. Programs in the second set primarily do floating point computations, and are characterized by long sequences of similar computation and simpler structure than integer programs.

A program alone (such as `gzip`) does not typically define a workload. An input is also needed. For example, `gzip` operates on data, and that data must be specified as input to the program. Thus a program/input pair make up a workload, and the program/input pair is what computer architects simulate.

When making an optimization to a processor in simulation, running these benchmark workloads on the simulated processor will show how well the processor performs in terms of speed, branch prediction accuracy, cache misses, power consumption, etc. Speed is of course all-important, and is measured in terms of CPI (cycles per instruction) or IPC (the inverse of CPI). These statistics can be compared to previously published statistics for the same workloads. The goal of improving simulation is to obtain these statistics quickly (requiring less CPU time) and accurately.

There are two levels of computer simulation used in this work, and it is important to understand the distinction. One is called "detailed", which is most interesting to computer architects. It is the simulation that gives statistics about

CPI, data cache miss rate, etc, because it simulates the entire processor in detail (including registers, functional units, caches, etc.). Gathering these statistics is necessary, but practical limits prevent using it exclusively since it is so costly in time. The second type of simulation is much faster, and is accordingly called "fast-forwarding" simulation. Its purpose is to keep some of the state of the CPU intact while simulating (such as the program counter), but it keeps state on far fewer things than detailed simulation. There are two primary reasons for fast-forwarding:

1. To find the total sequence of instructions that a program/input pair executes, called its dynamic image. This is the list of instructions that are actually executed on the processor during the simulation of a program. This is different than the static image of the program, which is what a compiled program is. It too contains a list of instructions, but not in the order or proportion they will appear when executed on the processor. It is this fast-forward simulation that gives the data which will be clustered in order to learn program structure.

2. To keep the state of the program somewhat up-to-date between detailed simulations. To simulate a program in detail in two places of the program (say, the beginning and the middle), one could use fast-forwarding to get from the end of the first detailed simulation to the beginning of the second. This comes at the cost that the detailed architectural state will be lost during fast-forwarding, but this loss is usually amortized by performing longer periods of detailed simulation.

Fast-forwarding is used for both purposes in the work in this chapter.

It is quite common in processor optimization to try many different architectural configurations to find out which works the best – basically performing a parameter sweep over several attributes. This adds to the difficulty of simulating

workloads efficiently, since there may be many different configurations and many workloads, each combination of which requires significant simulation time.

### IV.A.3 Capturing program structure

Programs can have wildly different behavior over their run time, and these behaviors can be seen even on the largest of scales. Understanding these large scale program behaviors can unlock many new optimizations. These range from new thread scheduling algorithms that make use of information on when a thread's behavior changes, to feedback directed optimizations targeted at not only the aggregate performance of the code but individual phases of execution, to creating simulations that accurately model full program behavior. To enable these optimizations, we must first develop the analytical tools necessary to automatically and efficiently analyze program behavior over large sections of execution.

In order to perform such an analysis we need to develop a hardware independent metric that can concisely summarize the behavior of an arbitrary section of execution in a program. In [61], Sherwood *et al.* presented the use of *basic block vectors* (BBVs), which uses the structure of the program that is exercised during execution to determine where to simulate. A BBV represents the code blocks executed during a given interval of execution. The goal is to find a single continuous window of executed instructions that match the whole program's execution, so that this smaller window of execution can be used for simulation instead of executing the program to completion. Using the BBVs provides us with a hardware independent way of finding this small representative window.

In this chapter we examine the use of basic block vectors for analyzing large scale program behavior using data clustering. We use BBVs to explore the

large scale behavior of several programs and discover the ways in which common patterns, and code, repeat themselves over the course of execution. We quantify the effectiveness of basic block vectors in capturing this program behavior across several different architectural metrics (such as IPC, branch, and cache miss rates).

In addition to this, there is a need for a way of classifying these repeating patterns so that this information can be used for optimization. We show that this problem of classifying sections of execution is related to the problem of data clustering and we develop an algorithm to quickly and effectively find these sections based on clustering. Our techniques automatically break the full execution of the program up into several sets, where the elements of each set are very similar. Once this classification is completed, analysis and optimization can be performed on a per-set basis.

We demonstrate an application of this cluster-based behavior analysis to simulation methodology for computer architecture research. By making use of clustering information we are able to accurately capture the behavior of a whole program by taking simulation results from representatives of each cluster and weighing them appropriately. This results in finding a set of simulation points that when combined accurately represents the target application and input, which in turn allows the behavior of even very complicated programs such as `gcc` to be captured with a small amount of simulation time.

Section IV.B presents a brief review of basic block vectors and an in depth look into the proposed techniques and algorithms for identifying large scale program behaviors, and an analysis of their use on several programs. Section IV.C describes how clustering can be used to analyze program behavior, and describes the clustering methods used in detail. Section IV.D examines the use of the techniques presented in Sections IV.B and IV.C on an example problem: finding where to simulate in a program to achieve results representative of full program

behavior. Related work is discussed in Section IV.E, and the techniques presented are summarized in Section IV.F.

## IV.B Basic block vectors

A basic block is a section of code that is executed from start to finish with one entry and one exit. We use the frequencies with which basic blocks are executed as the metric to compare different sections of the application's execution to one another. The intuition behind this is that the behavior of the program at a given time is directly related to the code it is executing during that interval, and basic block distributions provide us with this information.

A program, when run for any interval of time, will execute each basic block a certain number of times. Knowing this information provides us with a fingerprint for that interval of execution, and tells us where in the code the application is spending its time. The basic idea is that knowing the basic block distribution for two different intervals gives us two separate fingerprints which we can then compare to find out how similar the intervals are to one another. If the fingerprints are similar, then the two intervals spend about the same amount of time in the same code, and the performance of those two intervals should be similar.

### IV.B.1 Basic block vectors

A basic block vector is a single dimensional array, where there is a single element in the array for each static basic block in the program. For the results in this chapter, the basic block vectors are collected in intervals of 100 million instructions throughout the execution of a program. At the end of each interval, the number of times each basic block is entered during the interval is recorded and a new count for each basic block begins for the next interval of 100 million

```
...
---------------- default_switch: -------------------------------
                    cli                 # no interrupts allowed !
Basic block A       movb    $0x80, %al  # disable NMI for bootup sequence
                    outb    %al, $0x70
                    lret
---------------- bootsect_helper: ------------------------------
Basic block B       cmpw    $0, %cs:bootsect_es
-------------------- jnz bootsect_second ------------------------
                    movb    $0x20, %cs:type_of_loader
                    movw    %es, %ax
                    shrw    $4, %ax
Basic block C       movb    %ah, %cs:bootsect_src_base+2
                    movw    %es, %ax
                    movw    %ax, %cs:bootsect_es
                    subw    $SYSSEG, %ax
                    lret                    # nothing else to do for now
---------------- bootsect_second: ------------------------------
                    pushw   %cx
Basic block D       pushw   %si
                    pushw   %bx
                    testw   %bx, %bx            # 64K full?
-------------------- jne bootsect_ex ----------------------------
...
```

Figure IV.1: Paraphrased assembly code from the 2.4.20 Linux kernel (`arch/i386/boot/setup.S`). The basic blocks in this source code are separated by "jump" commands, such as `jnz` and `jne`, and entry points that can be jumped to, shown by the labels (ending with ':').

instructions. Therefore, each element in the array is the count of how many times the corresponding basic block has been entered during an interval of execution, multiplied by the number of instructions in that basic block. Multiplying in the number of instructions in each basic block insures that instructions are weighted equally regardless of whether they reside in a large or small basic block. A basic block vector which was gathered by counting basic block executions over an interval of $n \times 100$ million instructions is a basic block vector of duration $n$.

Because we are not interested in the actual count of basic block executions for a given interval, but rather the proportions between time spent in basic blocks, a BBV is normalized by having each element divided by the sum of all the elements in the vector. This normalization ensures that the sum of all the elements in the BBV is 1, which in turn allows us to compare vectors of different durations. Such a normalization effectively scales each basic block vector to be on the hypersurface defined by

$$\sum_i |b_i| \quad = \quad 1$$

where $b_i$ is the $i$th entry in the basic block vector. Of course, basic block vectors never have attribute values which are negative. This normalization method and an alternative method are illustrated in Figure IV.2.

The basic block vector representation is convenient for describing programs in a platform-independent way, since programs may be broken down into basic blocks. Thus the dimensionality, or number of attributes, of the basic block vectors are the number of basic blocks in a program (which can be determined statically). This number can be quite large; ranging from several hundred for a small program to hundreds of thousands or even millions for a large modern program.

Describing programs using basic blocks is simple and convenient, yet other ways exist. For example, the procedure call could be the fundamental unit

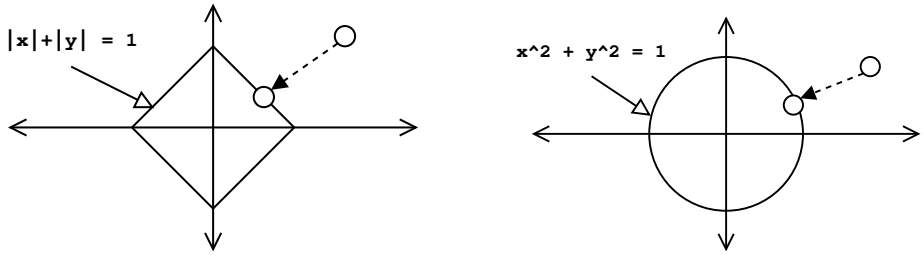


Figure IV.2: Two types of vector normalization that are considered for basic block vectors. These normalizations allow comparison of vectors of differing magnitudes. On the left is the normalization technique we employ, which projects each vector onto the hypersurface corresponding to $\sum_i |b_i| = 1$. On the right is an alternative normalization technique, commonly used in document clustering, which projects each vector onto the hypersphere corresponding to $\sum_i b_i^2 = 1$. Here $b_i$ refers to the $i$th entry in the basic block vector.

Figure IV.3: This shows the typical structure of a program represented as a matrix, where each row is a basic block vector which has been normalized. Rows sum to one (in this example not all the data is shown) since we are normalizing each value by its row sum.

| | **basic block** | | | | | |
|---|---|---|---|---|---|---|
| **instruction interval** | 1 | 2 | 3 | 4 | ... | $d$ |
| 0-99,999 | 0.3 | 0.7 | | | | |
| 100,000-199,999 | 0.1 | 0.8 | | 0.1 | | |
| 200,000-299,999 | | | 0.5 | | | |
| ⋮ | | | | | ⋱ | ⋮ |
| 5,100,000-5,199,999 | | | 0.1 | | ... | 0.2 |

of execution, rather than the basic block. This has the advantage that it is a higher level of abstraction which may generalize better across different compilers than does the basic block. Procedure calls can also give a better intuitive feel for what the program is doing in each interval (most high-level profilers operate on the procedure call level). However, current architecture simulators are better suited to provide detailed simulation results at the basic block level.

As long as the dynamic representation of the program uses time spent in each part of the program over a given time interval, the size required to represent a program with something like basic block vectors will be directly dependent upon both the static size of the program (which indicates the number of basic blocks for basic blocks), and the dynamic "size" of the program (the number of instructions executed). With these types of representations, the data tends to be very sparse since the number of basic blocks is so large compared with the time scale we are interested in simulating at.

The *target* BBV is defined as the basic block vector that contains the normalized basic block frequencies for the entire execution of the program. When trying to find representative simulation points in Section IV.D, we will be searching to find a basic block vector, or set of vectors, of small duration that are very similar to the target BBV. In finding this we will have found a section of code that is representative of the whole.

### IV.B.2 Basic block vector difference

In order to find a basic block vector that is similar to the target BBV, we must first have some way of comparing two basic block vectors. This operation should receive as input two basic block vectors, and it should return a number which tells us how close they are to each other. We call this the *basic block difference* (BBDiff). Given two basic block vectors $b$ and $c$, the BBDiff is

computed as:

$$\text{BBDiff}(b, c) = \sum_i |b_i - c_i| \tag{IV.1}$$

This is in fact the Manhattan distance, or $L_1$ distance. Since both vectors $b$ and $c$ are normalized such that $\sum_i b_i = 1$ and $\sum_i c_i = 1$, the value of BBDiff is a value between 0 and 2. This value gives a measure of how closely related the two BBVs are. The closer the BBDiff is to 0, the more similar the two vectors are. We call this the *difference* between the two BBVs. Now that we have a way of comparing two basic block vectors, we can begin to look into how the execution of a program changes over time.

### IV.B.3 Basic block difference graph

We now concentrate on understanding how the execution of a program changes over time. Sherwood *et al.* did this by creating a basic block difference graph [61]. The basic block difference graph is a plot of how well each individual sample in the program compares to the target basic block vector created for the entire run to completion.

Figure IV.4 shows the plot of all of these BBV differences across the entire execution creating a *basic block difference graph*. For each consecutive interval of 100 million instructions, we create a BBV and calculate its difference from the target BBV. The x-axis is the number of instructions in multiples of 100 million, and the y-axis is our measure of comparison, the BBDiff, between the interval in question and the target vector. A basic block difference of 2 means that the two vectors are completely orthogonal and share no basic blocks traversals in common, while a difference of 0 is the result of a perfect match between a BBV and the target vector.

As shown in [61], most of these programs start with a section of code which is very different from the full execution of the program. This section is

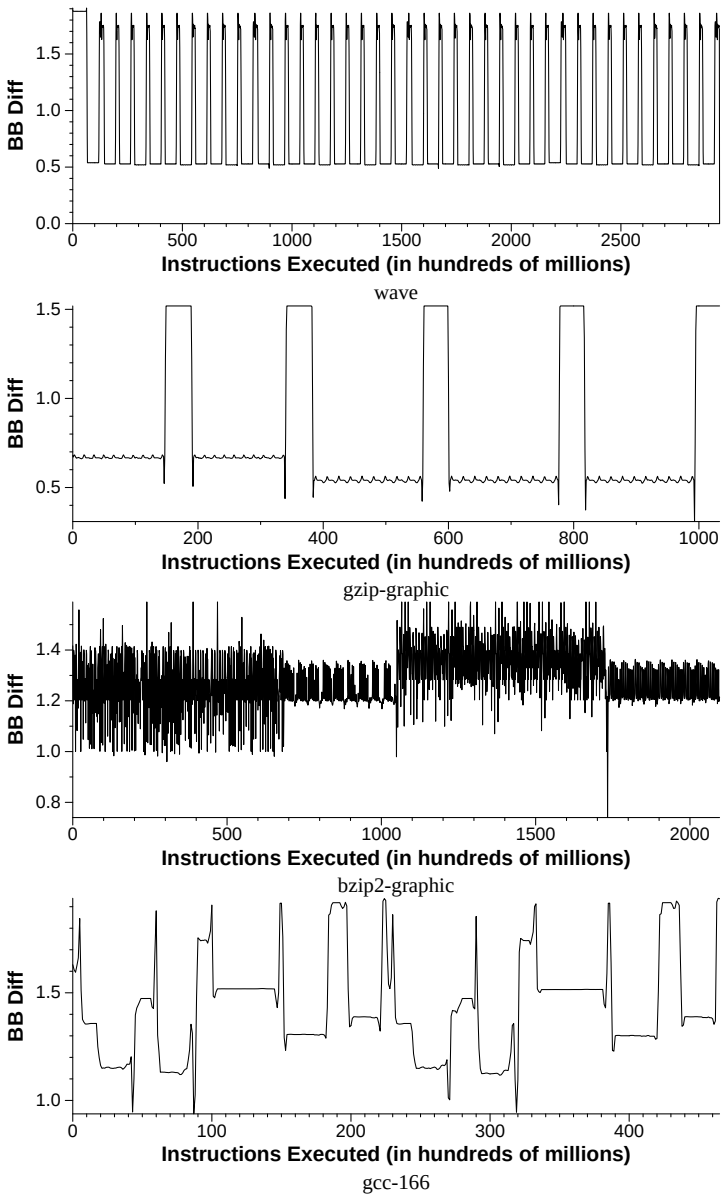


Figure IV.4: Basic block difference graphs for the programs `wave`, `gzip-graphic`, `bzip2-graphic`, and `gcc-166`. Each x-axis unit represents 100 million executed instructions. The graphs show the basic block vector difference on the y-axis, which is calculated by comparing the target BBV (the full execution of the program) with the BBV generated for each 100 million interval of executed instructions.

usually setting up and initializing structures for the compute intensive part. Then programs may begin a repetitive pattern through the rest of the execution of the program. But not all programs follow this trend as can be seen by `gcc` and `bzip`. Sherwood *et al.* used the basic block difference graph to try to find a single simulation point that was representative of the whole program's execution, which works well for programs like `wave`. But this does not work well for program's like `gcc` and `bzip`. This motivated us to explore new techniques for representing the execution of the program over time.

### IV.B.4 Basic block similarity matrix

Now that we have a method of comparing intervals of program execution to one another, we can now concentrate on finding phase-based behavior. A phase of program behavior can be defined in several ways. Past definitions are built around the idea of a phase being a contiguous interval of execution during which a measured program metric is relatively stable. We extend this notion of a phase to include all similar sections of execution regardless of temporal adjacency, which is a significant and powerful modification.

A key observation from this chapter is that the phase behavior seen in any program metric (such as CPI or branch prediction miss rate) can be seen as a function of the code being executed. Because of this we can use the comparison between the basic block vectors as an approximate bound on how closely related any other metrics will be between those two intervals.

To illustrate how intervals of execution relate to one another we create a *basic block similarity matrix*. The similarity matrix is an upper triangular $n \times n$ matrix, where $n$ is the number of intervals in the program's execution. An entry at $(x, y)$ in the matrix represents the basic block difference between the basic block vector at interval $x$ and the basic block vector at interval $y$.

Figures IV.5(left and right) and IV.8(left) shows the similarity matrices for `gzip`, `bzip`, and `gcc` using the basic block difference. The diagonal of the matrix represents the program's execution over time from start to completion. The darker the points, the more similar the intervals are (the basic block difference is closer to 0), and the lighter they are the more different they are (the basic block difference is closer to 2).

The top left corner of each graph is the start of program execution and is the origin of the graph, $(0, 0)$, and the bottom right of the graph is the point $(n-1, n-1)$ where $n$ is the number of intervals that the full program execution was divided up into. The way to interpret the graph is to start considering points along the diagonal axis drawn. Each point is perfectly similar to itself, so the points directly on the axis all are drawn dark. Starting from a given point on the diagonal axis of the graph, you can begin to compare how that point relates to it's neighbors forward and backward in execution by tracing horizontally or vertically. If you wish to compare a given interval $x$ with the interval at $x+n$, you simply start at the point $(x, x)$ on the graph and trace horizontally to the right until you reach $(x, x+n)$.

To examine the phase behavior of programs, let us first examine `gzip` because it has behavior on such a large scale that it is easy to see. If we examine an interval taken from 70 billion instructions into execution, in Figure IV.5 (left), this is directly in the middle of a large phase shown by the triangle block of dark color that surrounds this point. This means that this interval is very similar to it's neighbors both forward and backward in time. We can also see that the execution at 50 billion and 90 billion instructions is also very similar to the program behavior at 70 billion. We also note, while it may be hard to see in a printed version that the phase interval at 70 billion instructions is similar to the phases at interval 10 and 30 billion, but they are not as similar as to those

around 50 and 90 billion. Compare this with the IPC and data cache miss rates for `gzip` shown in Figure IV.6. Overall, Figure IV.5(left) shows that the phase behavior seen in the similarity matrix lines up quite closely with the behavior of the program, with 5 large phases (the first 2 being different from the last 3) each divided by a small phase, where all of the small phases are very similar to each other.

The similarity matrix for `bzip` (shown on the right of Figure IV.5) is very interesting. `Bzip` has complicated behavior, with two large parts to it's execution, compression and decompression. This can readily be seen in the figure as the large dark triangular and square patches. The interesting thing about `bzip` is that even within each of these sections of execution there is complex behavior. This, as will be shown later, makes the behavior of `bzip` impossible to capture using a small contiguous section of execution.

A more complex case for finding phase behavior is `gcc`, which is shown on the left of Figure IV.8. This similarity matrix shows the results for `gcc` using the basic block difference. The similarity matrix on the right will be explained in more detail in Section IV.C.3. This figure shows that `gcc` does have some regular behavior. It shows that, even here, there is common code shared between sections of execution, such as the intervals around 13 billion and 36 billion. In fact the strong dark diagonal line cutting through the matrix indicates that there is good amount of repetition between offset segments of execution. By analyzing the graph we can see that interval $x$ is very similar to interval $(x + 23.6B)$ for all $x$.

Figures IV.6 and IV.9 show the time varying behavior of `gzip` and `gcc`. The average IPC and data cache miss rate is shown for each 100 million interval of execution over the complete execution of the program. The time varying results graphically show the same phase behavior seen by looking at only the

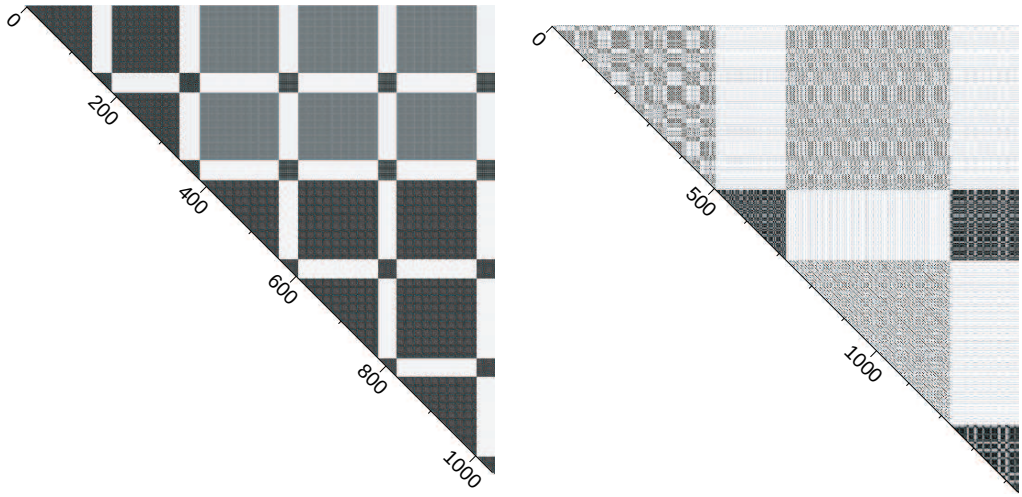


Figure IV.5: Basic block similarity matrix for the programs `gzip-graphic` (shown left) and `bzip-graphic` (shown right). The diagonal of the matrix represents the program's execution to completion with units in billions of instructions. The darker the points, the more similar the intervals are (the basic block difference is closer to 0), and the lighter the points the more different they are (the basic block difference is closer to 2).

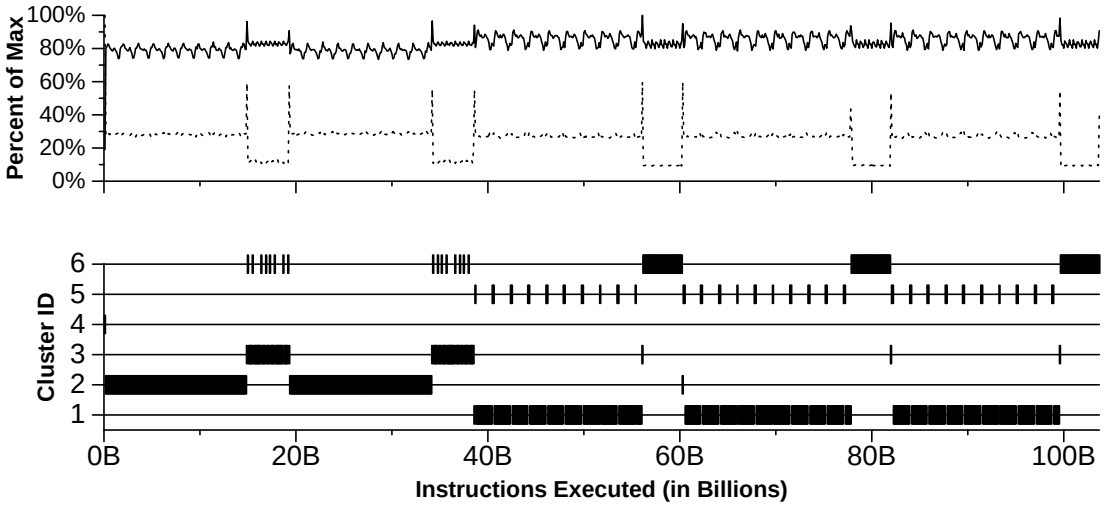


Figure IV.6: (top graph) Time varying graph for `gzip-graphic`. The average IPC (drawn with solid line) and L1 data cache miss rate (drawn with dotted line) are plotted for every interval (100 million instructions of execution) showing how these metrics vary over the program's execution. The x-axis represents the execution of the program over time, and the y-axis the percent of max value the metric had during execution. The results are non-accumulative.

Figure IV.7: (bottom graph) Cluster graph for `gzip-graphic`. The full run of the execution is partitioned into a set of 6 clusters. The x-axis is in instructions executed, and the graph shows for each interval of execution (every 100 million instructions), which cluster the interval was placed into.

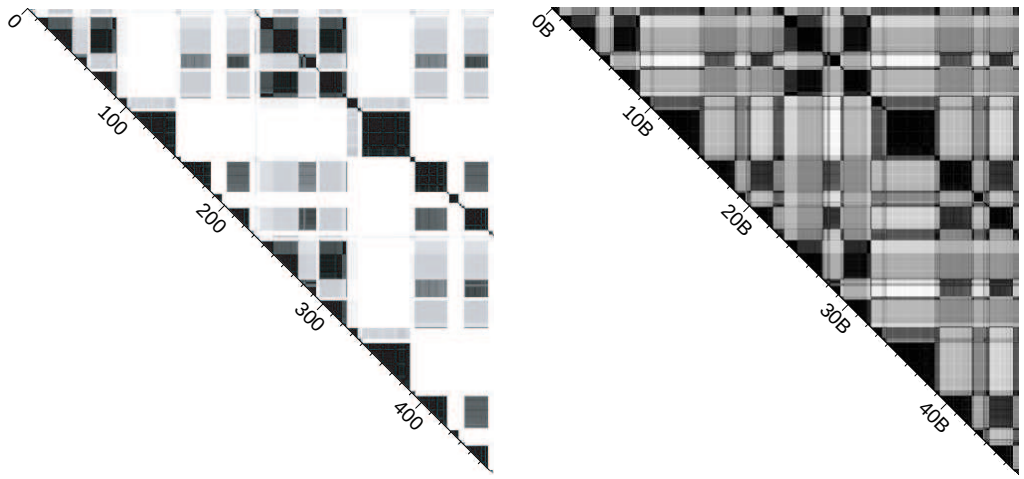


Figure IV.8: The original basic block similarity matrix for the program `gcc` (shown left), and the similarity matrix for `gcc-166` drawn from projected data (on right). The figure on the left use the original basic block vectors (each of which has over 100,000 dimensions) and uses the Manhattan distance as a method of measuring difference. The figure on the right uses projected data (down to 15 dimensions) and uses the Euclidean distance for measuring difference.

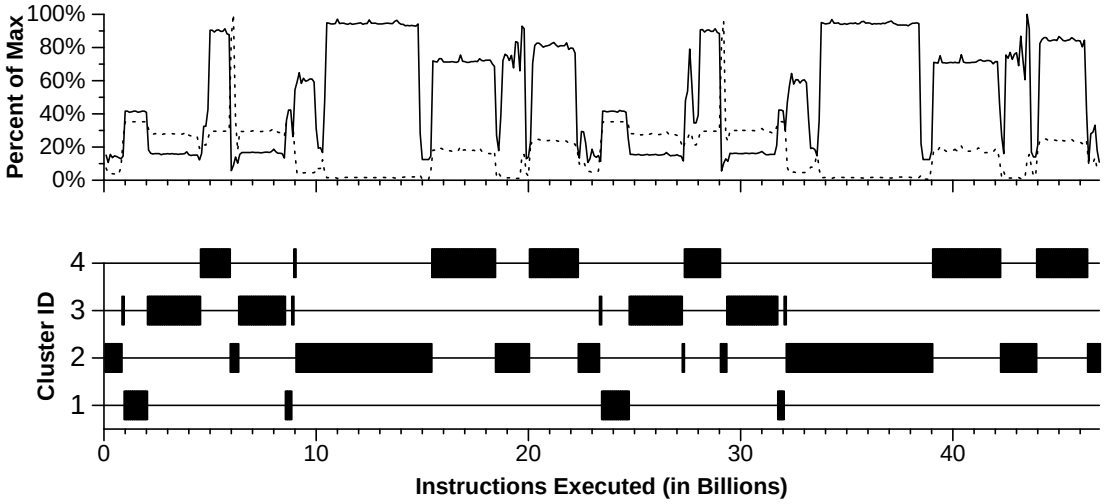


Figure IV.9: (top graph) Time varying graph for `gcc-166`. The average IPC (drawn with solid line) and L1 data cache miss rate (drawn with dotted line) are plotted for every interval (100 million instructions of execution) showing how these metrics vary over the program's execution. The x-axis represents the execution of the program over time, and the y-axis the percent of max value the metric had during execution. The results are not cumulative.

Figure IV.10: (bottom graph) Cluster graph for `gcc-166`. The full run of the execution is partitioned into a set of 4 clusters. The x-axis is in instructions executed, and the graph shows for each interval of execution (every 100 million instructions), which cluster the interval was placed into.

code executed. For example, the two phases for `gcc` at 13 billion and 36 billion, shown to be very similar in Figure IV.8, are shown to have the same IPC and data cache miss rate in Figure IV.9.

## IV.C Clustering

The basic block vectors provide a compact and representative summary of the program's behavior for intervals of execution. By examining the similarity between them, it is clear that there exists a high level pattern to each program's execution. In order to make use of this behavior we need to start by delineating a method of finding and representing the information. Because there are so many intervals of execution that are similar to one another, one efficient representation is to group the intervals together that have similar behavior. This problem is analogous to a *clustering* problem. Later, in Section IV.D, we demonstrate how to use the clusters to find multiple simulation points for irregular programs or inputs like `gcc`. By simulating only a single representative from each cluster, we can accurately represent the whole program's execution.

### IV.C.1 Previous method for clustering basic block vectors

In [61], Sherwood *et al.* use a heuristic clustering algorithm based on the recursive partitioning of a graph to cluster the basic block vectors. The fully-connected weighted graph has $n$ nodes. The weight of each edge in the graph is the basic block difference between the two intervals that those nodes represent. So this graph is another form of the basic block difference graph. Recall that each weight in the graph is a value between 0 and 2. Their algorithm groups nodes that are all connected by less than some small weight threshold (a small edge weight means high similarity between nodes). Equivalently, this operation is removes all edges that are above the threshold weight. This action produces a set

of clusters. If this set of clusters is satisfactory, then the clustering is complete. If it not satisfactory, then the threshold is increased, and the process is repeated. The term "satisfactory" means two things: the number of clusters is small, and at least 90% of the data is included in clusters with other data. The initial threshold they use is on the order of 0.1.

### IV.C.2 Clusters and phase behavior

Figures IV.7 and IV.10 show the 6 clusters formed for `gzip` and the 4 clusters formed for `gcc`. The X-axis corresponds to the execution of the program in billions of instructions, and each interval (each of 100 million instructions) is tagged to be in one of the $n$ clusters (labeled on the Y-axis). These figures, just as for Figures IV.5 and IV.8, show the execution of the programs to completion.

For `gzip`, the full run of the execution is partitioned into a set of 6 clusters. Figure IV.5(left) for comparison shows that the cluster behavior captured by SimPoint lines up quite closely with the behavior of the program. The majority of the points are contained by clusters 1,2,3 and 6. Clusters 1 and 2 represent the large sections of execution which are similar to one another. Clusters 3 and 6 capture the smaller phases which lie in between these large phases, while cluster 5 contains a small subset of the larger phases, and cluster 4 represents the initialization phase.

In the cluster graph for `gcc`, shown in Figure IV.10, the run is now partitioned into 4 different clusters. Looking to Figure IV.8 for comparison, even the more complicated behavior of `gcc` is captured correctly by SimPoint. Clusters 2 and 4 correspond to the dark boxes shown parallel to the diagonal axis. It should also be noted that the projection does introduce some degree of error into the clustering. For example, the first group of points in cluster 2 are not really that similar to the other points in the cluster. Comparing the two similarity matrices

in Figure IV.8, shows the introduction of a dark band at (0,30) on the graph which was not in the original (un-projected) data. Despite these small errors, the clustering is still very good, and the impact of any such errors will be minimized in the next section.

### IV.C.3 Phase-finding algorithm

For our algorithm, we use random linear projection followed by $k$-means. We choose to use the $k$-means clustering algorithm, since it is a very fast and simple algorithm that yields good results for this application. In this chapter, to choose the value of $k$, we use the Bayesian Information Criterion (BIC) score [39, 56]. The following steps summarize our algorithm, and then several of the steps are explained in more detail:

1. Profile the basic blocks executed in each program (using fast-forward simulation) to generate the basic block vectors for every interval 100 million instructions of execution.

2. Reduce the dimension of the basic block vector data to 15 dimensions using random linear projection.

3. Try the $k$-means clustering algorithm on the low-dimensional data for $k$ values 1 to 10. Each run of $k$-means produces a clustering, which is a partition of the data into $k$ different clusters.

4. For each clustering ($k = 1 \ldots 10$), score the fit of the clustering using the BIC. Choose the clustering with the smallest $k$, such that it's score is at least 90% as good as the best score.

### Random projection

For this application, we have to address the problem of dimensionality. All clustering algorithms suffer from the so-called "curse of dimensionality", which refers to the fact that it becomes extremely hard to cluster data as the number of dimensions increases. For the basic block vectors, the number of dimensions is the number of executed basic blocks in the program, which ranges from 2,756 to 102,038 for our experimental data, and could grow into the millions for very large programs. Another practical problem is that the running time of our clustering algorithm depends on the dimension of the data, making it slow if the dimension grows too large.

Two ways of reducing the dimension of data are dimension selection and dimension reduction. Dimension selection simply removes all but a small number of the dimensions of the data, based on a measure of goodness of each dimension for describing the data. However, this throws away a lot of data in the dimensions which are ignored. Dimension reduction reduces the number of dimensions by creating a new lower-dimensional space and then projecting each data point into the new space (where the new space's dimensions are not directly related to the old space's dimensions). This is analogous to taking a picture of 3 dimensional data at a random angle and projecting it onto a screen of 2 dimensions.

For this work we choose to use *random linear projection* [18] to create a new low-dimensional space into which we project the data. This is a simple and fast technique that is very effective at reducing the number of dimensions while retaining the properties of the data such as inter-cluster separation. There are two steps to reducing a dataset $X$ (which is a matrix of basic block vectors and is of size $n \times d_{numbb}$, where $n$ is the number of intervals and $d_{numbb}$ is the number of basic blocks in the program) down to $d_{new}$ dimensions using random linear projection:

- Create a $d_{numbb} \times d_{new}$ projection matrix $M$ by choosing a random Gaussian value for each matrix entry.
- Multiply $X$ times $M$ to obtain the new lower-dimensional dataset $X'$ which will be of size $n \times d_{new}$.

For clustering programs traces, we found that using $d_{new} = 15$ dimensions is sufficient to still differentiate the different phases of execution. Figure IV.11 shows why we chose to project the data down to 15 dimensions. The graph shows the number of dimensions on the x-axis. The y-axis represents the $k$ value found to be best on average, when the programs were projected down to the number of dimensions indicated by the x-axis. The best $k$ is determined by the $k$ with the highest BIC score, which is discussed in Section IV.C.3. The y-axis is shown as a percent of the maximum $k$ seen for each program so that the curve can be examined independent of the actual number of clusters found for each program. The results show that for 15 dimensions the number of clusters found begins to stabilize and only climbs slightly. This is a direct application of the dimension-learning method we described in the previous chapter, using the BIC as the metric of structure, and reinforces our findings using the G-means algorithm.

There are two primary advantages of using random linear projection for dimension reduction. First, creating new vectors with a low dimension of 15 is extremely fast and can even be done at simulation time. Secondly, using only 15 dimensions speeds up the $k$-means algorithm significantly, and reduces the memory requirements by several orders of magnitude over using the original basic block vectors.

Figure IV.8 shows the similarity matrix for `gcc` on the left using original BBVs, whereas the similarity matrix on the right shows the same matrix but on the data that has been projected down to 15 dimensions. For the reduced

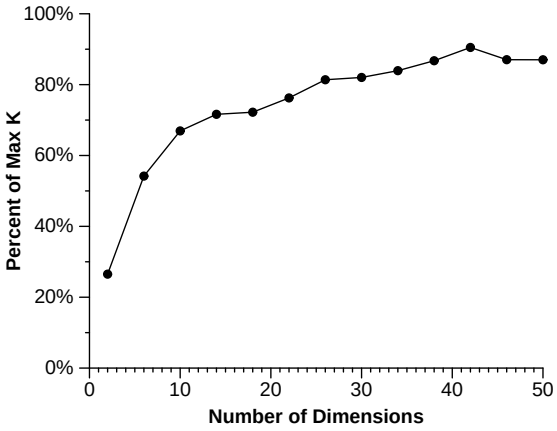


Figure IV.11: Motivation for random projection down to 15 dimensions ($d$=15). The x-axis is the number of dimensions of the projection, and the y-axis is the percent of the max number of clusters found for each program averaged over all spec programs. The results show that as you decrease the number of dimensions too far (the lowest point is two dimensions) the true clusters become collapsed on one another, and the algorithm cannot find as many clusters. By $d$=15 most of this effect has gone.

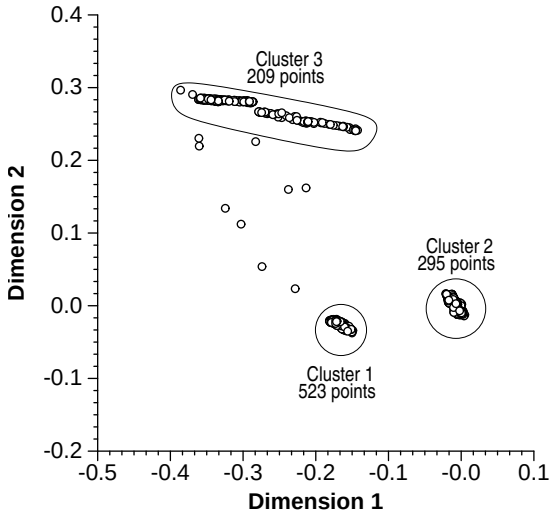


Figure IV.12: The program trace `gzip` has been reduced to 2 dimensions using random projection, and three clusters have been identified as regions which execute similar code. The goal of the SimPoint tool is to automatically identify such clustered regions and exploit this knowledge of the structure. This dataset has 1038 datapoints and 8163 original dimensions.

dimension data we use the Euclidean distance to measure differences, rather than the Manhattan distance used in the BBDiff metric. After the projection, some information will be blurred, but overall the phases of execution that are very similar with full dimensions can still be seen to have a strong similarity with only 15 dimensions.

### Bayesian information criterion

To compare and evaluate the different clusters formed for different $k$, we use the *Bayesian Information Criterion* (BIC) as a measure of the "goodness of fit" of a clustering to a dataset. More formally, the BIC is an approximation to the probability of the clustering given the data that has been clustered. Thus, the larger the BIC score, the higher the probability that the clustering being

scored is a "good fit" to the data being clustered. We use the BIC formulation given in [56] for clustering with $k$-means, however other formulations of the BIC could also be used.

Despite the fact that we have shown in the previous chapter that the BIC overfits data when looking for the optimal $k$, we use it here in a constrained way. The number of clusters that the algorithm finds is linearly related to the amount of simulation our method will require. Therefore, we limit the number of clusters to be within a certain range. Even with this restriction, the BIC's overfitting causes it to typically choose a large number of clusters. Therefore the algorithm searches for the clustering that 90% of the best BIC score seen within the constrained range. Finally, we use BIC scoring with regular $k$-means; we do not use G-means or $X$-means to cluster the data since $k$-means is faster than both, and we know the range of $k$ in which we want to search. In the next chapter, we will discuss a technique of choosing the $k$ based on structured sampling, which is more directly related to the desired results than is the BIC.

The BIC score is a penalized likelihood. There are two terms in the BIC: the likelihood and the penalty. The likelihood is a measure of how well the clustering models the data. To get the likelihood, each cluster is considered to be produced by a spherical Gaussian distribution, and the likelihood of the data in a cluster is the product of the probabilities of each point in the cluster given by the Gaussian. The likelihood for the whole dataset is just the product of the likelihoods for all clusters. However, the likelihood tends to increase without bound as more clusters are added. Therefore the second term is a penalty that offsets the likelihood growth based on the number of clusters. The BIC is formulated as

$$BIC(d,k) = \mathcal{L}(d|k) - \frac{p_j}{2}\log(n)$$

where $\mathcal{L}(d|k)$ is the likelihood, $n$ is the number of points in the data, and $p_j$ is the number of parameters to estimate, which is $(k-1)+dk+1 = k(d+1)$ for $(k-1)$

cluster probabilities, $k$ cluster center estimates which each require $d$ dimensions, and 1 variance estimate. To compute $\mathcal{L}(d|k)$ we use

$$\begin{aligned}\mathcal{L}(d|k) &= \sum_{i=1}^{k} -\frac{n_i}{2}\log(2\pi) - \frac{n_i d}{2}\log(\sigma^2) - \frac{n_i - 1}{2} \\ &\quad +n_i \log(n_i/n)\end{aligned}$$

where $n_i$ is the number of points in the $i$th cluster, and $\sigma^2$ is the average variance of the Euclidean distance from each point to its cluster center.

For a given program and inputs, the BIC score is calculated for each $k$-means clustering, for $k$ from 1 to some limit. We then choose the clustering that achieves a BIC score that is at least 90% of the spread between the largest and smallest BIC score that the algorithm has seen. Figure IV.13 shows the benefit of choosing a BIC with a high value and its relationship with the variance in IPC seen for that cluster. The y-axis shows the percent of IPC variance seen for a given clustering, and the corresponding BIC score the clustering received. Each point on the graph represents the average or max IPC variance for all points in the range of $\pm 5\%$ of the BIC score shown. The results show that picking clusterings that represent greater than 80% of the BIC score resulted in an IPC variance of less than 20% on average. The IPC variance was computed as the weighted sum of the IPC variance for each cluster, where the weight for a cluster is the number of points in that cluster. The IPC variance for each cluster is simply the variance of the IPC for all the points in that cluster.

## IV.D Automatically finding simulation points

Modern computer architecture research relies heavily on cycle accurate simulation to help evaluate new architectural features. While the performance of processors continues to grow exponentially, the amount of complexity within a processor continues to grow at an even a faster rate. With each generation of

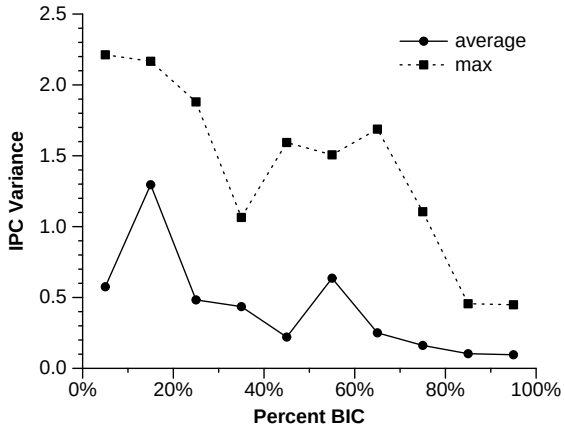


Figure IV.13: Plot of average IPC variance and max IPC variance versus the BIC. These results indicate that for our data, a clustering found to have a BIC score greater than 80% will have, on average, and IPC variance of less than 0.2.

processor more transistors are added, and more things are done in parallel on chip in a given cycle while at the same time cycle times continue to decrease. This growing gap between speed and complexity means that the time to simulate a constant amount of processor time is growing. It is already to the point that executing programs fully to completion in a detailed simulator is no longer feasible for architectural studies. Since detailed simulation takes a great deal of processing power, only a small subset of a whole program can be simulated.

SimpleScalar [13], one of the faster cycle-level simulators, can simulate around 400 million instructions per hour. Unfortunately many of the new SPEC 2000 programs execute for 300 billion instructions or more. At 400 million instructions per hour this will take approximately 1 month of CPU time.

Modern computer architecture research relies heavily on cycle-accurate simulation to help evaluate new architectural features. These models capture the complex behaviors of the pipelines, caches, busses, and queuing delays. Detailed simulation is an important step in architectural design and evaluation, but be-

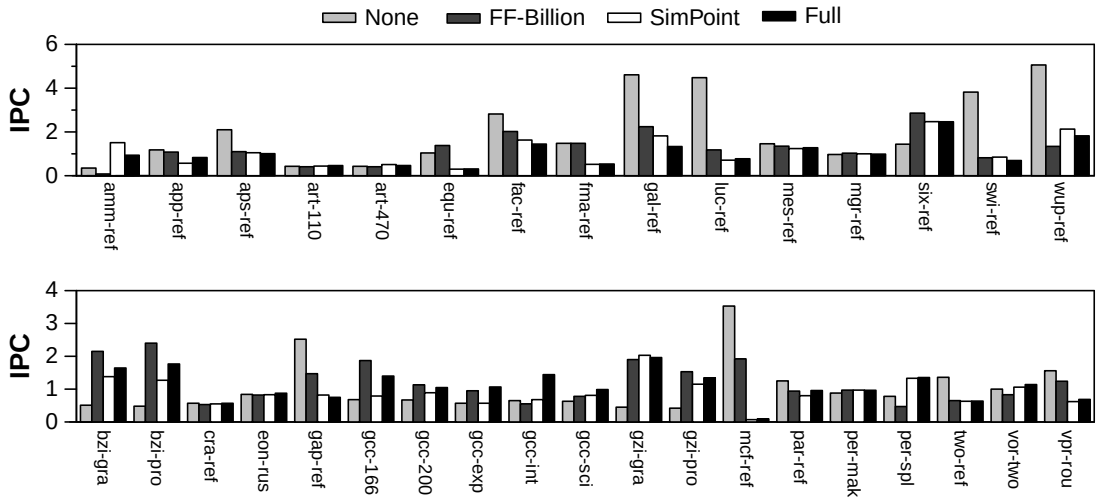


Figure IV.14: Simulation results starting simulation at the start of the program (none), blindly fast forwarding 1 billion instructions, using a single simulation point, and the IPC of the full execution of the program.

cause it takes a great deal of processing power only a small subset of a whole program is often simulated.

Because it is only feasible to execute a small portion of the program, it is very important that the section simulated is an accurate representation of the program's behavior as a whole. The basic block vector and cluster analysis presented in Sections IV.B and IV.C will allow us to make sure that this is the case.

### IV.D.1 Single simulation points

In [61], Sherwood *et al.* used basic block vectors to automatically find a single simulation point to potentially represent the complete execution of a program. A *Simulation Point* is a starting simulation place (in number of instructions executed from the start of execution) in a program's execution. That algorithm creates a target basic block vector, which is a BBV that represents

the complete execution of the program. The basic block difference between each interval BBV and the target BBV is computed. The BBV with the lowest basic block difference represents the single simulation point that executes the code closest to the complete execution of the program. This approach is used to calculate the long single simulation points (LongSP) described below.

In comparison, the single simulation point results in this chapter are calculated by choosing the BBV that has the smallest Euclidean distance from the centroid of the whole dataset in the 15-dimensional space, a method which we find superior to the original method. The 15-dimensional centroid is formed by taking the average of each dimension over all intervals in the cluster.

Figure IV.14 shows the IPC estimated by executing only a single interval, all 100 million instructions but chosen by different methods, for all SPEC 2000 programs. This is shown in comparison to the IPC found by executing the program to completion. The results are from SimpleScalar using the architecture model described in Section V.B, and all fast forwarding is done so that all of the architecture structures are completely warmed up when starting simulation (no cold-start effect).

The first bar, labeled `none`, is the IPC found when executing only the first 100 million instructions from the start of execution (without any fast forwarding). The second bar, `FF-Billion` shows the results after blindly fast forwarding 1 billion instructions before starting simulation. The third bar, `SimPoint` shows the IPC using the single simulation point analysis described above, and the last bar shows the IPC of simulating the program to completion (labeled `Full`). Because these are actual IPC values, values which are closer to the `Full` bar are better.

The results in Figure IV.14 shows that the single simulation points are very close to the actual full execution of the program, especially when compared

against the ad-hoc techniques. Starting simulation at the start of the program results in an average error of 210%, whereas blindly fast forwarding results in an average 80% IPC error. Using the single simulation point analysis, the average IPC error is reduced to 18%. These results show that it is possible to reasonably capture the behavior of the most programs using a very small slice of execution.

These results show that a single simulation point can be accurate for many programs, but there is still a significant amount of error for programs like `bzip`, `gzip` and `gcc`. This occurs because there are many different phases of execution in these programs, and a single simulation point will not accurately represent all of the different phases. To address this, we used our clustering analysis to find multiple simulation points to accurately capture these programs behavior, which we describe next.

### IV.D.2 Multiple simulation points

To support multiple simulation points, the simulator can be run from start to stop, only performing detailed simulation on the selected intervals. Or the simulation can be broken down into $k$ simulations, where $k$ is the number of clusters found via analysis, and each simulation is run separately. This has the further benefit of breaking the simulation down into parallel components that can be distributed across many processors. This is the methodology we use in our simulator. For both cases results from the separate simulation points need to be weighed and combined to arrive at overall performance for the program [16]. Care must be taken to combine statistics correctly (simply averaging will give incorrect results for statistics such as rates).

Knowing the clustering alone is not sufficient to enable multiple point simulation because the $k$-means cluster centers do not correspond to actual intervals of execution. Instead, we must first pick a representative for each cluster

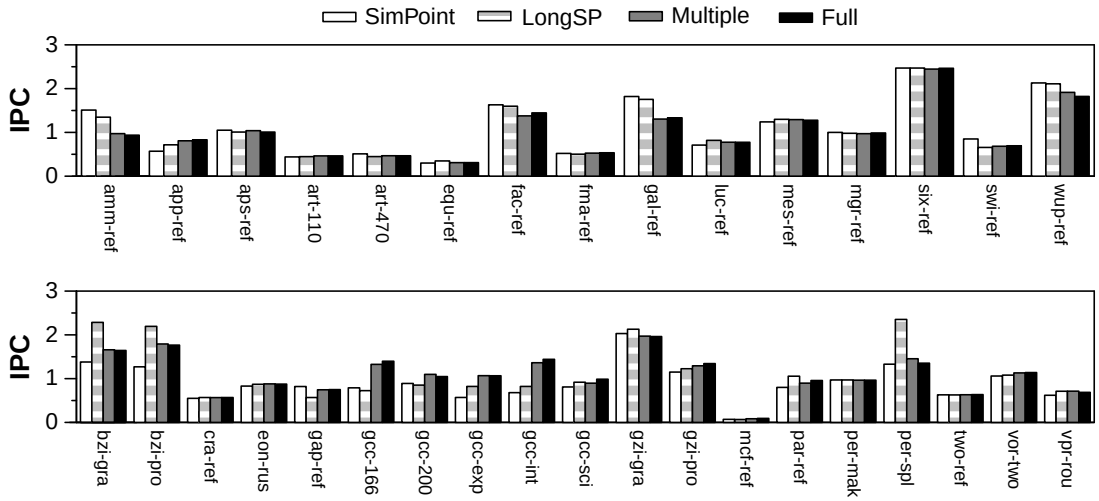


Figure IV.15: Multiple simulation point results. Simulation results are shown for using a single simulation point simulating for 100 million instructions, LongSP chooses a single simulation point simulating for the same length of execution as the multiple point simulation, simulation using multiple simulation points, and the full execution of the program.

that will be used to approximate the behavior of the the full cluster. As such, the result of our clustering algorithm is more similar to the result of $k$-medoids, which also chooses a datapoint as the representative for each cluster. However, $k$-medoids has quadratic time and space complexity, while $k$-means is linear. In order to pick this representative, we choose for each cluster the actual interval that is closest to the center (centroid) of the cluster. In addition to this, we weigh any use of this representative by the size of the cluster it is representing. If a cluster has only one point, it's representative will only have a small impact on the overall outcome of the program.

Figure IV.15 shows the IPC results for multiple simulation points. The first bar shows the results for our single simulation points simulating for 100 million instructions. The second bar `LongSP` chooses a single simulation point, but the length of simulation is identical to the length used for multiple simulation points (which may go up to 1 billion instructions). This is to provide a fair comparison between the single simulation points and multiple. The `Multiple` bar shows results using the multiple simulation points, and the final bar is IPC for full simulation. As in Figure IV.14, the closer the bar is to `Full`, the better.

The results show that the average IPC error rate is reduced to 3% using multiple simulation points, which is down from 17% using the long single simulation point. This is significantly lower than the average 80% error seen for blindly fast forwarding. The benefits can be most clearly seen in the programs `bzip`, `gcc`, `ammp`, and `galgel`. The reason that the long contiguous simulation points do not do much better is that they are constrained to only sample at one place in the program. For many programs this is sufficient, but for those with interesting long term behavior, such as `bzip`, it is impossible to approximate the full behavior.

Figure IV.16 is the average over all of the floating point programs (top

graph) and integer programs (bottom graph). Errors for IPC, branch miss rate, instruction and data cache miss rates, and the unified L2 cache miss rate for the architecture presented in Section V.B are shown. The errors are with respect to these metrics for the full length of simulation using SimpleScalar. Results are shown for starting simulation at the start of the program `None`, blindly fast forwarding a billion instructions `FF-Billion`, single simulation points of duration 1 (`SimPoint`) and $k$ (`LongSP`), and multiple simulation points (`Multiple`).

The first thing to note is that using the just a single small simulation point performs quite well on average across all of the metrics when compared to blindly fast-forwarding. Even though a single SimPoint does well, it is clearly beaten by using the clustering based scheme presented in this chapter across all of the metrics examined. One thing that stands out on the graphs is that the error rate of the instruction cache and L2 cache appear to be high (especially for the integer programs) despite the fact that our technique is doing quite well in terms of overall performance. This is due to the fact that we present here an arithmetic mean of the errors, and there are several programs that have high error rates due to the very small number of cache misses. If there are 10 misses in the whole program, and we estimate there to be 100, that will result in a error of 10X. We point to the overall IPC as the most important metric for evaluation as it implicitly weighs each of the metrics by it's relative importance.

## IV.E Related work

### IV.E.1 Time varying behavior of programs

In [60], Sherwood *et al.* provided a first attempt at showing the periodic patterns for all of the SPEC 95 programs, and how these vary over time for cache behavior, branch prediction, value prediction, address prediction, instructions per

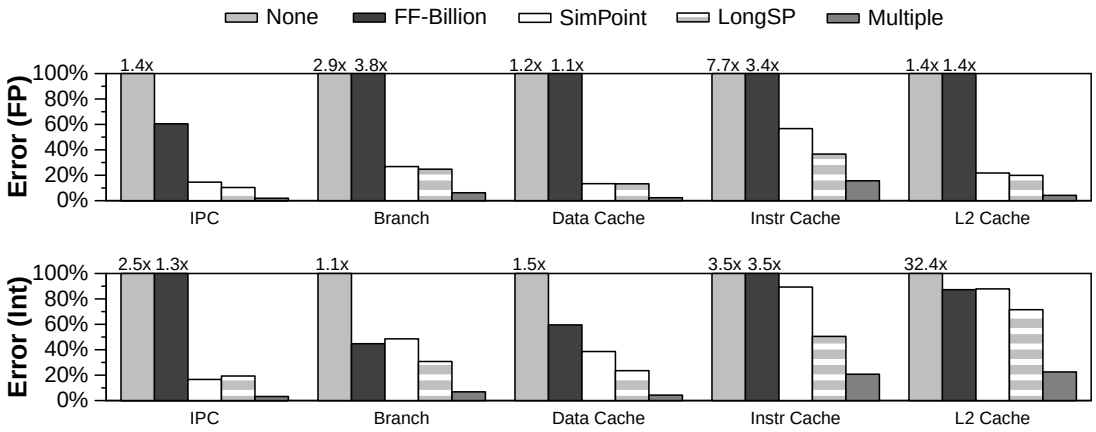


Figure IV.16: Average error results for the SPEC 2000 floating point (top) and integer (bottom) benchmarks for IPC, branch misprediction, instruction, data and unified L2 cache miss rates.

cycle (IPC) and register usage unit (RUU) occupancy.

### IV.E.2 Training inputs and finding smaller representative inputs

One approach for reducing the simulation time is to use the training or test inputs from the SPEC benchmark suite. For many of the benchmarks, these inputs are either (1) still too long to fully simulate, or (2) too short and place too much emphasis on the startup and shutdown parts of the program's execution, or (3) inaccurately estimate behavior such as cache accesses do to decreased working set size.

KleinOsowski *et al.* [43], have developed a technique where they manually reduce the input sets of programs. The input sets were developed using a range of approaches from truncating of the input files to modification of source code to reduce the number of times frequent loops were traversed. For these input sets they develop, they make sure that they have similar results in terms of IPC, cache, and instruction mix.

### IV.E.3 Fast forwarding and check-pointing

Historically researchers have simulated from the start of the application, but this usually does not represent the majority of the program's behavior because it is still in the initialization phase. Recently researchers have started to *fast-forward* to a given point in execution, and then start their simulation from there, ideally skipping over the initialization code to an area of code representative of the whole. During fast-forward the simulator simply needs to act as a functional simulator, and may take full advantage of optimizations like direct execution on native hardware. After the fast-forward point has been reached, the simulator switches to full cycle level simulation.

After fast-forwarding, the architecture state to be simulated is still cold, and a warmup time is needed in order to start collecting representative results. Efficiently warming up execution only requires references immediately proceeding the start of simulation. Haskins and Skadron [32] examined probabilistically determining the minimum set of fast-forward transactions that must be executed for warm up to accurately produce state as it would have appeared had the entire fast-forward interval been used for warm up [32]. They recently examined using reuse analysis to determine how far before full simulation warmup needs to occur [33].

An alternative to fast forwarding is to use check-pointing to start the simulation of a program at a specific point. With check-pointing, code is executed to a given point in the program and the state is saved, or checkpointed, so that other simulation runs can start there. In this way the initialization section can be run just one time, and there is no need to fast forward past it each time. The architectural state (e.g., caches, register file, branch prediction, etc) can either be stored in the trace (if they are not going to change across simulation runs) or can be warmed up in a manner similar to described above.

### IV.E.4 Automatically finding where to simulate

Our work is based upon the basic block distribution analysis in [61] as described in prior sections. Recent work on finding simulation points for data cache simulations is presented by Lafage and Seznec [44]. They proposed a technique to gather statistics over the complete execution of the program and use them to choose a representative slice of the program. They evaluate two metrics, one which captures memory spatial locality and one which captures memory temporal locality. They further propose to create specialized metrics such as instruction mix, control transfer, instruction characterization, and distribution of data dependency distances to further quantify the behavior of the both the program's full execution and the execution of samples.

### IV.E.5 Statistical sampling

Several different techniques have been proposed for sampling to estimate the behavior of the program as a whole. These techniques take a number of contiguous execution samples, referred to as clusters in [16], across the whole execution of the program. These clusters are spread out throughout the execution of the program in an attempt to provide a representative section of the application being simulated. Conte *et al.* [16] formed multiple simulation points by randomly picking intervals of execution, and then examining how these fit to the overall execution of the program for several architecture metrics (IPC and branch and data cache statistics). Our work is complementary to this, where we provide a fast and metric independent approach for picking multiple simulation points based just on basic block vector similarity. When an architect gets a new binary to examine they can use our approach to quickly find the simulation points, and then validate these with detailed simulation in parallel with using the binary.

### IV.E.6 Statistical simulation

Another technique to improve simulation time is to use statistical simulation [53]. Using statistical simulation, the application is run once and a synthetic trace is generated that attempts to capture the whole program behavior. The trace captures such characteristics as basic block size, typical register dependencies and cache misses. This trace is then run for sometimes as little as 50-100,000 cycles on a much faster simulator. Nussbaum and Smith [52] also examined generating synthetic traces and using these for simulation and was proposed for fast design space exploration. We believe the techniques presented here are complementary to the techniques of Oskin et al. and Nussbaum and Smith in that more accurate profiles can be determined using our techniques, and instead of attempting to characterize the program as a whole it can be characterized on a per-phase basis.

## IV.F Summary

At the heart of computer architecture and program optimization is the need for understanding program behavior. As we have shown, many programs have wildly different behavior on even the very largest of scales (over the full lifetime of the program). While these changes in behavior are drastic, they are not without order, even in very complex applications such as `gcc`. In order to help future compiler and architecture researchers in exploiting this large scale behavior, we have developed a set of analytical tools that are capable of automatically and efficiently analyzing program behavior over large sections of execution.

The development of the analysis is founded on a hardware independent metric, *basic block vectors*, that can concisely summarize the behavior of an arbitrary section of execution in a program. We showed that by using basic block

vectors one can capture the behavior of programs as defined by several architectural metrics (such as IPC, and branch and cache miss rates).

Using this framework, we examine the large scale behavior of several complex programs like `gzip`, `bzip`, and `gcc`, and find interesting patterns in their execution over time. The behavior that we find shows that code and program behavior repeat over time. For example, in the input we examined in detail for `gcc` we see that program behavior repeats itself every 23.6 billion instructions. Developing techniques that automatically capture behavior on this scale is useful for architectural, system level, and runtime optimizations. We present an algorithm based on the identification of clusters of basic block vectors that can find these repeating program behaviors and group them into sets for further analysis. For two of the programs `gzip` and `gcc` we show how the clustering algorithm results line up nicely with the similarity matrix and correlate with the time varying IPC and data cache miss rates.

It is increasingly common for computer architects and compiler designers to use a small section of a benchmark to represent the whole program during the design and evaluation of a system. This leads to the problem of finding sections of the program's execution that will accurately represent the behavior of the full program. We show how our clustering analysis can be used to automatically find multiple simulation points to reduce simulation time and to accurately model full program behavior. We call this clustering tool to find single and multiple simulation points *SimPoint*. SimPoint along with additional simulation point data can be found at: `http://www.cs.ucsd.edu/~calder/simpoint/`. For the SPEC 2000 programs, we found that starting simulation at the start of the program results in an average error of 210% when compared to the full simulation of the program, whereas blindly fast forwarding resulted in an average 80% IPC error. Using a single simulation point found, using our basic block vector analy-

sis, resulted in an average 17% IPC error. When using the clustering algorithm to create multiple simulation points we saw an average IPC error of 3%.

Automatically identifying the phase behavior using clustering is beneficial for architecture, compiler, and operating system optimizations. To this end, my colleagues have used the notion of basic block vectors and a random projection to create an efficient technique for identifying phases on-the-fly [63], which can be efficiently implemented in hardware or software. Besides identifying phases, this approach can predict not only when a phase change is about to occur, but to which phase it is about to transition. Using phase information can lead to new compiler optimizations with code tailored to different phases of execution, multi-threaded architecture scheduling, power management, and other resource distribution problems controlled by software, hardware or the operating system.

## IV.G Acknowledgements

This chapter, in part, is a reprint of the material as it appears in [62]. The dissertation author was a secondary researcher and author of this paper.

# V

# Improvements and validation of program behavior prediction

## V.A Introduction

Understanding the cycle level behavior of a processor running an application is crucial to modern computer architecture research. To gain this understanding, detailed cycle level simulators are typically employed. Unfortunately, this level of detail comes at the cost of speed, and simulating the full execution of an industry standard benchmark on even the fastest simulator can take weeks to months to complete. This fact has not gone unnoticed in the academic community, and several researchers have started to develop techniques aimed at reducing simulation time.

Typically in the course of doing architecture research it is necessary to take one instance of a program with a given input, and simulate its performance over many different configurations for an architecture feature, searching the design space for Pareto optimal points in terms of performance, area, and power. The same program binary with the input may be run hundreds or thousands of times to examine how, for example, the effectiveness of a given architecture

changes with changes to its cache size. Our goal in creating SimPoint [61, 62] is to (1) significantly reduce simulation time, (2) provide an accurate characterization of the full program, and (3) to automatically perform the analysis to accomplish the first two goals in a matter of minutes. These goals are met by simulating only a handful of *intelligently* chosen sections of the full program. When these sections (simulation points) are carefully chosen, it provides an accurate picture of the complete execution of the program and results in highly accurate estimations of performance

The key to our approach is that for a given binary and input, the simulation points only need to be chosen once. This is because we select them using a method that is independent of any particular architecture configuration. The simulation points are selected using a metric that is only based on the code that is executed over time for a program/input pair. Once the simulation points are chosen they can be used for the hundreds or thousands of independent simulations that may be needed, significantly reducing simulation time.

To pick the simulation points in [61, 62], we introduced the concept of profiling basic block vectors as a way capturing the important behaviors of the program over time [61], as described in the previous chapter. A basic block vector captures the relative frequency of the code blocks executed during a given portion of execution. After profiling a program with a particular input, we compare the basic block vectors to see how similar they are to one another. Intervals of execution that execute the same code blocks with the same frequency are grouped together into clusters using $k$-means clustering. We found that sections of execution (represented by basic block vectors) that are grouped into the same cluster have very similar behavior across all the architecture metrics we have examined. Once we break the program into clusters, we pick a single point from each cluster (appropriately weighted) to serve as its representative. The set of

representative sections are where detailed simulation is performed. Simulating only these simulation points provides an accurate and efficient representation of the complete execution of the program.

The initial SimPoint approach covered in the last chapter focused on presenting a new clustering technique for identifying phase information, where the purpose is on the approach for picking simulation points. The work therefore did not address several important issues for using the simulation points for an accurate and efficient simulation infrastructure. In this chapter, we address and provide solutions to these issues. This includes providing efficient simulation for environments that have support for fast-forwarding, but not for checkpoints. To achieve this, we provide an algorithm for picking simulation points that are earlier in the program to significantly reduce the fast-forward time needed. In addition, we examine using significantly smaller sized simulation points. Another important aspect not addressed in the prior work [61, 62] is that no method of statistically validating the simulation points was provided, and we provide such analysis in this chapter using parametric bootstrapping. In summary, this chapter makes the following contributions:

1. Early Simulation Points. Our prior SimPoint algorithm focused on picking the most representative points from a given cluster. If the simulation points are at the end of execution, and our simulation environment only supports fast-forwarding, then significant time is spent fast-forwarding to obtain our results. Therefore, we have created an algorithm for picking simulation points from a cluster that are early in execution, but at the same time representative of the cluster. This algorithm finds simulation points more friendly to fast-forwarding, which we call *Early Simulation Points* (EarlySP).

2. Statistical Validation. We provide statistical validation of the Simulation Points chosen to represent a program/input execution. The statistical anal-

ysis provides an error bound and confidence for a given set of simulation points, and is used to show that the clustering our algorithm chooses does an accurate job at breaking the program's execution into homogeneously-performing clusters of execution.

3. Architecture Independence. Our approach breaks a program into clusters based upon the similarities and differences in the executed code regions. This is done completely independent of the underlying architecture, since we only examine the executed code. We have found that samples that are put into the same cluster have the same behavior even as the underlying architecture changes, and we provide results showing this.

4. Smaller Simulation Point Samples. In our prior work [62], we focused on simulating for samples that were 100 million instructions in length. This was so that we did not have to deal with any warmup issues. In this chapter, we present results for 1 million instructions per interval when choosing simulation points.

5. Comparison to Statistical Sampling. Statistical Sampling has been shown to achieve very accurate results [16, 69], and we provide a comparison to statistical sampling with statistical validation.

## V.B Methodology

In performing this research we made use of three different tools, ATOM [66], SimpleScalar3.0c [13], and SimPoint [62]. These tools are all designed or configured to work with the Alpha AXP ISA or are platform agnostic.

ATOM, a binary modification tool, is used to quickly gather profiling information in the form of basic block vectors which can be fed as input to the SimPoint tool. The SimPoint tool then takes these vectors and performs $k$-means

clustering analysis and BIC scoring and determines the set of SimPoints to use for that program-input pair.

SimpleScalar is used to collect the full detailed results, collect random samples, and to validate the phase behavior we found when clustering our basic block profiles showing that this corresponds to the phase behavior in the programs performance and architecture metrics. We modified SimpleScalar to enable fast-forwarding between an arbitrary number of non-contiguous detailed simulation intervals, in one serial run. This modification is necessary for random sampling, since it involves hundreds of non-contiguous interval simulations from each program.

We analyzed and simulated the SPEC 2000 benchmarks compiled for the Alpha ISA for multiple inputs. We provide detailed results for the programs that we have found to have the more complex phase behavior, which are the most challenging for our SimPoint analysis, and average results for the rest of the programs. The binaries we used in this study and how they were compiled can be found at `http://www.simplescalar.com/`.

## V.C Early SimPoints

In this section we examine reducing simulation time over our prior SimPoint [62] algorithm by picking simulation points earlier in execution and using a significantly smaller sample length. The SimPoint algorithm was described in detail in the previous chapter.

In our previous work, the goal was to pick a single simulation point from each cluster that best represents all the intervals of execution in that phase. For simulation environments that do not support checkpointing, it can require up to several days to fast-forward to the latter part of the execution if that is where the simulation point is located. Our goal is to find early simulation

points to significantly reduce time spent fast-forwarding while still accurately representing the overall execution of the program. Researchers at Intel using SimPoint expressed a desire for such an optimization due to the time required to fast-forward to simulation points in the latter part of execution.

If we consider a simulation environment where multiple points can be simulated one after the other (run through the program once, interleaving fast-forwarding and detailed simulation), then the *last* simulation point in program execution order will determine the total simulation time. Our Early SimPoint algorithm focuses on choosing a clustering that is both representative of the program's execution and has some feasible simulation points early in the program for all clusters (if possible). This might not be achievable for all programs, since an important phase of execution may only appear at the end of the program's execution. We therefore give priority in our algorithm toward ensuring that the clustering represents the overall execution of the program. Once the early clustering is performed, the algorithm chooses a representative simulation point early in the execution from each cluster.

### V.C.1 Need for improvements over the original SimPoint algorithm

One potential issue with both the original SimPoint algorithm and statistical sampling is that the points that are selected for detailed simulation can be anywhere within the program. For a fast-forwarding simulation environment, you may have to fast-forward all the way to the end of the program to get through all the simulation points, and even fast-forwarding takes significant time when it has to be done for hundreds of billions of instructions. For example, the program `sixtrack` takes 50 CPU hours to just fast-forward from start to end using SimpleScalar 3.0c.

The two main issues with the previously published SimPoint algorithm

with respect to fast forwarding (needing to functionally emulate the complete program) are:

- The original algorithm chose the best overall clustering, and did not take into account how the clusters are spread out over the execution of the program. For example, some intervals of execution toward the start of the program might be slightly different than those at the end of the program, and treating them as separate clusters may improve accuracy by some small percent when compared to treating them as the same cluster, but having them as separate clusters may significantly increase the simulation time, even though the extra error may be tolerable. To address this, we modify the clustering algorithm from the last chapter to not only take into consideration the formation of the clustering using the BIC score, but also give priority to clustering where all clusters have some points toward the start of the program's execution.

- The original SimPoint algorithm chose a representative simulation point for each given cluster by finding the point with behavior closest to that of the center of the cluster as a whole. The closest point was always used regardless of how far you would have to fast forward to get there, even if there was another point which was almost as good but far closer to the start of execution. We examine using a different method for picking the representative SimPoint that takes into account the overhead from fast forwarding.

By (1) giving weight to clusterings that have early representative points, and (2) picking early points in the cluster to represent the cluster, we can significantly reduce the total time of simulation when accounting for overhead from fast-forward.

### V.C.2 Early SimPoint algorithm

We present a new algorithm for finding early simulation points, which reduces the amount of fast-forwarding required. This algorithm is optimized to find simulation points that are both early in the execution and accurately represent the phases in which they reside. The new approach also uses substantially smaller simulation points than ones used in the prior method (1 million instructions per interval versus 100 million), reducing overall detailed simulation time significantly.

If we consider the combination of simulation points for a particular program as a serial set for simulation, with fast-forwarding in between, then we only care about the location of the last simulation point. The last simulation point will determine the minimum amount of fast-forwarding needed for the simulation. If we pick the earliest point from each phase, the last simulation point in the program is the location of the first encounter of the last phase. Since the last point establishes the required fast-forward duration regardless of the position of other earlier points, we can still optimally choose the earlier points with the added restriction that we do not pick points not past the last simulation point.

**Light-weight simulation points**: The SimPoint method described in the previous chapter used 100 million instructions simulation intervals, considering up to 10 phases per program. In that approach a program potentially requires simulating 1 billion instructions in detail if 10 phases are found. In an effort to minimize simulation time, we propose using smaller simulation intervals of 1 million instructions. With these light-weight simulation intervals a program can be clustered into more phases since the cost of additional simulation points will not substantially increase detailed simulation time. Clustering a program into more phases achieves higher accuracy since each simulation point represents a tighter and more homogeneous phase.

**Picking a clustering**: When picking a clustering, the location of the last phase found is crucial, since it will determine the minimum overhead due to fast-forwarding. In the original simulation point analysis, the phases are computed during the clustering stage of the algorithm. Clusterings are computed for each $k$ number of phases (for $k = 1 \ldots 10$), and then each clustering is scored with the BIC. For the results in this chapter, we gather our basic block vectors at intervals of 1 million instructions, since we use this as the size of our sample to further reduce simulation time as describe above.

Our new algorithm takes into account the position in time of phases produced during this clustering stage of the SimPoint analysis. We introduce a new metric, *EarlySP*, which is the BIC score weighted by the first encounter of the last phase (or cluster): $EarlySP$ = BIC $\times$ ( 1 - $StartLastPhase$), where $StartLastPhase$ is the fraction into execution of the program that the last phase is first encountered. The $StartLastPhase$ is critical for picking early simulation points since it will be the minimum distance required to fast-forward. The intuition behind *EarlySP* is that we reward the clusterings that have representatives from every cluster near the start of the program.

This new score provides us with a goodness of fit of the clustering weighted by how early the last phase starts. We then pick the clustering that achieves a *EarlySP* score that is at least 90% of the spread between the largest and smallest *EarlySP* scores. We set the number of phases we are willing to simulate higher than the limit used in [62] since we can afford picking a clustering with more simulation points since their impact on simulation time is minimal.

**Picking simulation points**: After we picked the clustering with the EarlySP score, we pick a cutoff point in the program. No points after this cutoff point will be under consideration to act as representative points. This ensures that no point after the cutoff will be picked as a simulation point and thus bounds

the amount of overhead due to fast-forwarding.

This cutoff point is determined by first picking an early simulation point for the phase that starts the latest in execution. This cutoff (the last) simulation point is picked by choosing the earliest simulation point in the cluster that is within 1% of the distance of the centroid of this late cluster in the space. Once this cutoff has been determined, the simulation points for the remaining clusters are selected from all the potential points, those from the start of the program up until the last cutoff simulation point, such that the simulation point from each cluster is the closet to the centroid of the cluster.

### V.C.3 Early SimPoint results

To efficiently simulate a collection of disjoint simulation points or random samples on a simulator capable of fast-forwarding, we find that a single run simulating all the intervals serially is more efficient than fast-forwarding them all separately. With this method, the total time spent fast-forwarding is bound by the time spent fast-forwarding to the last interval.

Figure V.1 shows the percent error in CPI using Systematic Sampling, SimPoints from [62], and the algorithm described here, Early SimPoint. The error is calculated by comparing the estimated error using the different sampling techniques to the CPI found doing detailed simulation of all the programs to completion. The SimPoint method from [62] uses up to 10 intervals, each 100 million instructions in length. Early SimPoint uses less than 100 intervals/samples for each program of length 1 million, while Systematic Sampling uses both 100 and 1000 distinct samples, each using intervals of length 1 million.

All 3 techniques perform well. The accuracy of Systematic Sampling with 1000 samples performs the best with an average of 0.9% error. Early SimPoint has the next leading performance with an average of 2.3% error, while the

Original SimPoint method achieves an average of 2.5%. Systematic Sampling using 100 samples achieves an average error of 2.6%. There are some interesting cases to consider here:

1. One of the most complex programs, gcc, has better performance using Early SimPoint than Original SimPoint and Systematic Sampling with 100 samples. This can be explained by the ability to break down the program into more phases/samples then the original method, and the samples picked are better representatives than those in the Systematic Sampling set using 100 samples.

2. There are cases when Early SimPoint performs worse than the other two techniques, such as with art, gap, and gzip-program. These programs are not very complex, but they have phases that do not show up until later in the execution, and Early SimPoint gave preferences to early simulation points that were not as accurate.

In Figure V.2, we see the number of phases the Original and Early SimPoint found for each program. The smaller-sized intervals that Early SimPoint uses have a shorter detailed execution time than those used by Original SimPoint. This allows for a greater number of phases to be chosen, thus increasing the accuracy with which a program can be represented. Early SimPoint also captures the complexity of the programs, as evidenced by the fact that `gcc` requires the maximum number of simulation points out of all the programs, and it is in fact one of the most complex programs.

Figure V.3 shows the minimum number of instructions required to fast-forward for the two SimPoint techniques. Here the benefit of using Early SimPoint over the former SimPoint technique is clearly seen from the amount of fast-forwarding that can be avoided.

Figure V.4 shows the time required to complete a simulation based on following techniques: checkpointing, Original SimPoint, Early SimPoint, and Systematic Sampling. Checkpointing is a technique to load the perfect state of the machine at an arbitrary point in execution. The overhead for this operation is the time required to load the state for a given interval, times the number of intervals this will be done for. When checkpointing is used with the samples generated from Early SimPoint, it is by far the fastest technique for completing a simulation. However this technique is not readily available on all simulators.

The majority of simulators have the ability to fast-forward, and with some modification can be used to collect a number of detailed samples from a single run of the program. For the Original SimPoint method, we do not use any warmup since the interval size is large enough to overcome any cold-start effects. The time for this approach is based on a single run fast-forwarding between detailed simulation of the simulation points. The techniques involving smaller intervals of 1 million instructions, Early SimPoint and Statistical Sampling, cannot ignore cold-start effects. For these techniques we provide both an ideal time fast-forwarding with no warm-up between intervals of detailed simulation, and the time it would take to fast forward if *only* the memory structures are kept warm. This is equivalent to running a memory simulator (not cycle level), which is not as fast as the fast-forwarding but still considerably faster then detailed simulation.

From these results it is shown that Early SimPoint is 4 times faster then Original SimPoint and 5 times faster than Systematic Sampling for both using fast-forwarding and functionally warm simulations.

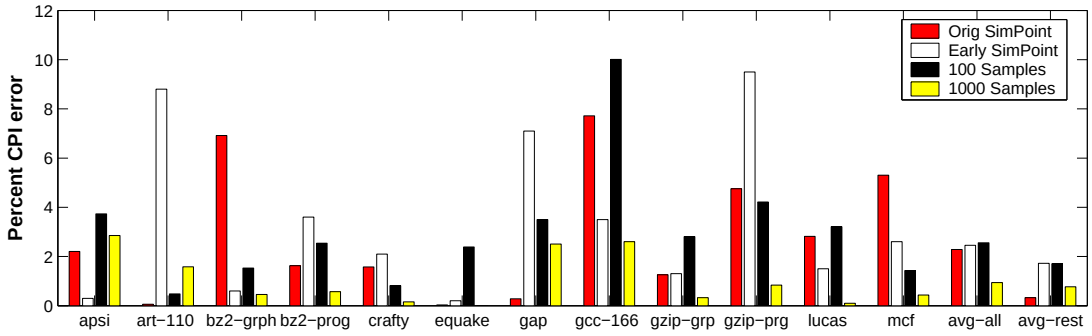


Figure V.1: CPI relative error for SimPoint, Early SimPoint, Systematic Sampling with 100 samples and 1000 samples. The Early SimPoint algorithm achieves comparable or better error rates for most programs than the other three methods, but it is able to do so much more quickly.

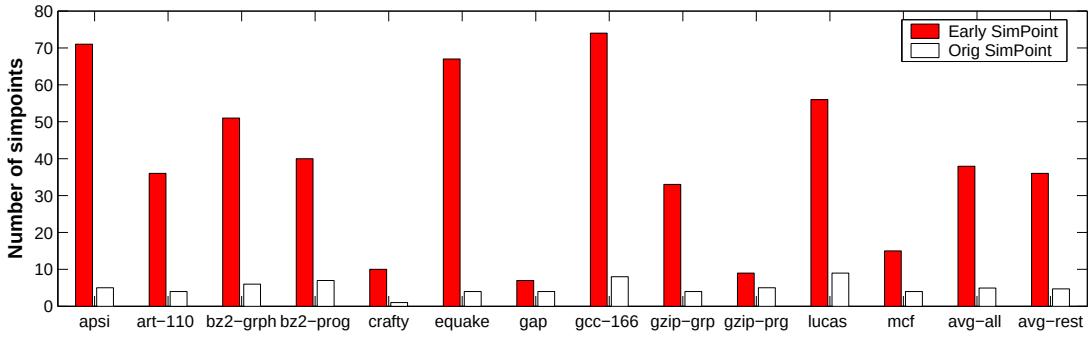


Figure V.2: Number of simulation point found for each program under the Original and Early SimPoint methods. The number of simulation points is equivalent to the number of phases found since from each phase a single simulation point is picked.

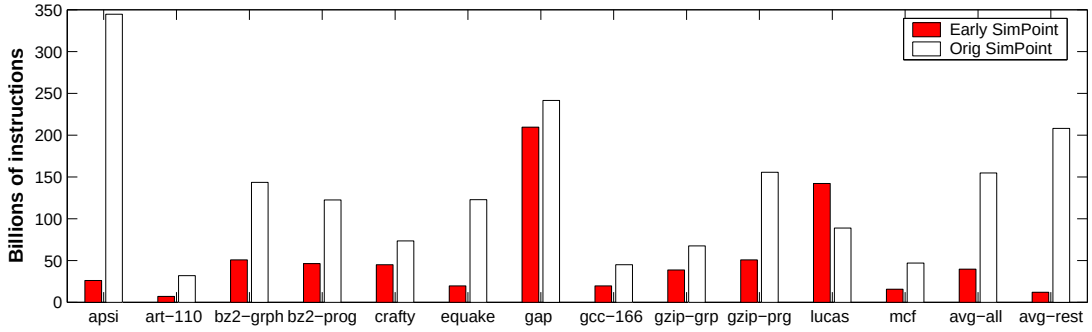


Figure V.3: Fast-forwarding needed for simulating Original and Early SimPoint. The Early SimPoint algorithm usually reduces the amount of fast-forwarding required dramatically over Original SimPoint.

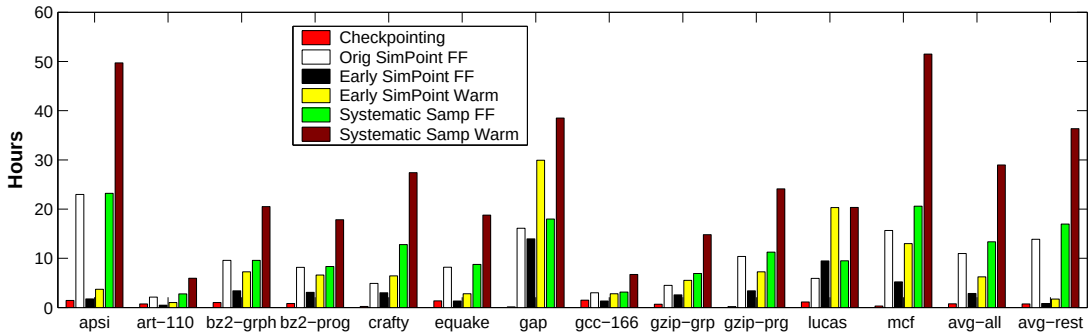


Figure V.4: Time required to complete a simulation for each program using Checkpointing, Original and Early SimPoint, and Sampling. FF is fast-forwarding between intervals of detailed simulation, and Warm is functional simulation keeping the major structures warm between intervals of detailed simulation. When checkpointing is not available, the Early SimPoint algorithm is much faster than both Original SimPoint and Systematic Sampling techniques.

## V.D Statistical validation

Our Early SimPoint algorithm chooses one interval to simulate from each phase of the program, and then combines the results from all phases, weighted by the size of each phase. This gives a single estimate of the desired metric, but we also desire to have some measure of how accurate that estimate is. In this section, we show how to find an error bound and confidence for using a single set of simulation points. The results show that our clustering is very effective at breaking a program's execution into phases of execution that have similar performance, and this is accomplished without looking at any hardware metrics since we only examine the code executed. It shows that picking one simulation point from each cluster provides good accuracy at a high level of statistical confidence.

### V.D.1 Statistical validation of simulation points

We have developed a technique for estimating what the expected error is at a given level of statistical confidence. To establish an error bound, we must find the variance of the estimator. A very simple and intuitive way to do this is to repeatedly take estimates from the program in the same way that we do for our Early SimPoint algorithm. This sampling technique is known as a parametric bootstrap [14, p. 480], where the parametric form is the clustering structure we have learned. The clustering prompts us to choose one simulation point per phase. To find the error bounds on one estimate, we do the following:

1. Find a clustering using the Early SimPoint algorithm.
2. Do the following $m$ times:
   (a) Choose one interval at random from each cluster/phase.

(b) Compute the CPI estimate from the CPI measurements from each chosen interval, weighted by the size of each phase.

3. The percentage relative error bound on *one* SimPoint estimate of the CPI is $100 * z\sigma/\mu$.

Here $z$ is the value such that the area under the Gaussian curve to the left of $z$ is $\alpha$, the desired confidence. In other words, $\alpha = \Phi(z)$. The $\sigma$ is the standard deviation of the computed estimates. The $\mu$ is the CPI estimate for one set of $k$ simulation points. The value of $m$ is the number of times to compute estimates. Larger $m$ gives a more accurate measurement of the standard deviation; $m = 10$ or larger is reasonable; for our measurements we have used $m = 100$. Note that gathering all the simulation points for the $m$ CPI estimates only requires one run through the program, fast-forwarding between the samples that we are gathering.

Figure V.5 shows the predicted error bars above the actual error for several program/input pairs, the average-all, and the average-rest. The error bars are for $\alpha = 0.95$, or 95% confidence. To obtain these error bounds we took $m = 100$ sets of $k$ randomly-chosen simulation points and gathered their CPI to establish the standard deviation $\sigma$ of the CPI estimator. This $\sigma$ is used in calculating the error bound at a chosen confidence when only one set of simulation points are used. For a choice of *one* set of simulation points, we expect the true CPI to be within $100 * z\sigma/\mu$ percent of the estimate $\mu$. The first bar in Figure V.5 shows the true error and bound when using *only* **one set** *of k early simulation points* for gathering the results. This graph shows that the true CPI errors of our Early SimPoint algorithm are within the bounds of the predicted errors for all the programs.

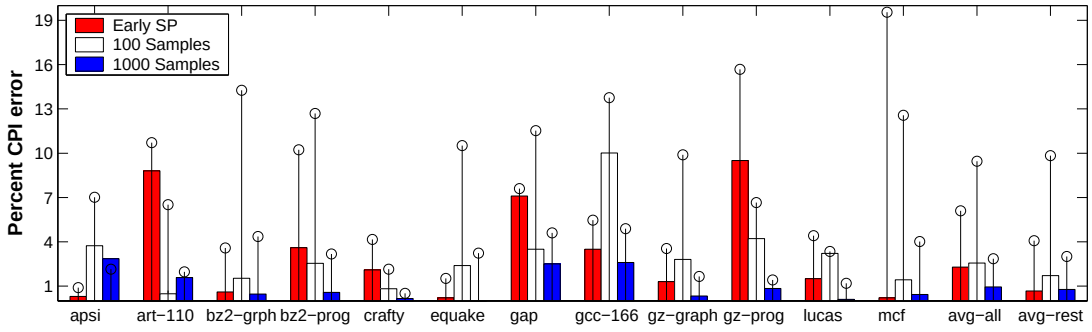


Figure V.5: True CPI error (bars) and estimated error bounds (stems) for Early SimPoint, 100 systematic samples, and 1000 systematic samples. The error bounds are for 95% confidence. For Early SimPoint, the error bounds represent the error for one set of chosen simulation points.

### V.D.2 Statistical validation of systematic sampling

Wunderlich *et al.* [69] used systematic statistical sampling to provide an error bound and confidence and to guide how many samples should be gathered. When performing their statistical validation they are gathering performance estimates to calculate the variance of their estimator over the complete execution of the program. This allows them to immediately tell the user what the expected error bound and confidence is, after every simulation run.

The second and third bars in Figure V.5 show results for comparing systematic sampling with 100 intervals and 1000 intervals. It is clear that usually the more samples taken, the lower the error bound and the true error. However, our Early SimPoint algorithm achieves similar confidence bounds for fewer simulation points than both systematic sampling approaches. This shows the benefits of partitioning the program's execution into similar phases, and that points within a given phase have similar behavior.

### V.D.3 Discussion

The confidence intervals/error bounds seen for SimPoint are looser than reported in [69],since we are placing an error bound on only one set of simulation points to represent the complete program's execution. If a tighter error bar is desired, for simulation one could sample repeatedly from the clustering, and take an average of the results. This will reduce the standard deviation of the estimate by a factor of $1/\sqrt{m}$ from the one-sample case if $m$ samples are taken. Then the error bar will be simply $100 * z\sigma/(\mu\sqrt{m})$.

Our method requires $m * k$ simulation points to establish the error bars on one program input pair ($m$ estimates of $k$ simulation points). This shows that only one set of $k$ simulation points is needed to arrive at reasonable error bounds with 95% confidence as shown in Figure V.5.

The analysis presented in this section can be used to arrive at error bounds and confidence intervals for a set of simulation points for a given $k$ clustering. Across all the programs we have examined, our results show that clustering does an excellent job at breaking the program's execution into phases that have similar behavior.

As we will show in Section V.F, even as the architecture changes a given set of simulation points maintains a consistent relative error, and the error is biased in the same direction. Intuitively this can be explained by the fact that the simulation points are chosen completely independent of the underlying architecture, and intervals of execution are only grouped together and predicted to have similar performance if they execute the same code with the same frequencies. Given this, it is expected that they have similar architecture metrics, even as the underlying architecture changes.

## V.E Using error bounds to choose clusterings

The goal of our work is to obtain good estimates of performance with minimal cost in simulation. In the previous section we have looked at how we can use sampling techniques to obtain measures of confidence in the error of estimates that our Early SimPoint algorithm produces. The procedure we explained assumes that a clustering has been chosen, and then uses sampling to measure the estimator variance. We may also look at the inverse problem of defining an error bound, and then using that error bound to guide the clustering algorithm to a choice of the number of simulation points.

We have previously used a modified BIC criterion in order to rank a set of different clusterings. The BIC and other scoring functions like it measure a goodness of fit of a clustering to a set of data, but does not actually take into account the behavior of the program, though we have have shown in the previous chapter that a high BIC score does correlate with low prediction error on the architecture metrics we are interested in. However, it is possible to make the choice of clustering more explicitly dependent upon the metrics we are ultimately interested in obtaining, such as CPI and cache miss rates, by utilizing the error bound confidence intervals outlineed in the previous section.

We desire a small number of simulation points (small $k$) because simulation time is linearly related to the number of simulation points we choose to use. As we saw in this chapter, simulation time is also related to the position of each simulation point, with later simulation points being more costly to simulate. However, we also desire to have a sufficient number of simulation points so that we can accurately capture the entire program's behavior. A practitioner may have in mind a desired error bound and confidence for the metrics he or she is measuring. For example, estimate the CPI to within 4% error from the true value, at 95% confidence. We can define a method that finds an appropriate set

of simulation points based on the specified parameters for error and confidence.

### V.E.1 Structured sampling using clustering

We propose the following algorithm for finding an appropriate set of simulation points, guided by a desired error bound and confidence level. The

**Algorithm 7** Structured sampling of architecture metrics to determine clustering complexity.

1. Use $k$-means clustering to cluster a program/input pair into phases. Do this for several values of $k$.
2. For each clustering, use a parametric bootstrap to establish the estimated error bound of the desired metric(s) for that clustering.
3. Choose the most efficient clustering (smallest $k$ and earliest-beginning last phase) that achieves the desired error estimate at the desired level of confidence.
4. If none of the clusterings achieve the desired results, restart the algorithm for larger values of $k$.

central idea behind this approach remains that breaking a program up into phases of similar code will aid in simulation, since the metrics we are interested in will be more homogenous when sampled within one phase compared to sampling across the entire program. More homogeneity will give lower variance when taking the bootstrap, which will give low error bounds at high confidence levels. This algorithm is naturally incremental; we may start with a low limit on $k$, and increase that limit until the algorithm achieves the desired results. For small $k$, the phases will be larger and less homogenous, leading to error bounds that may be larger than desired. As $k$ increases, the phases will necessarily become smaller

and thus more homogenous, and the variance of samples taken across all phases will become smaller. In the limit, every interval of execution is its own phase, at which point there is no variance, since the estimate is the true value.

Initially, this technique appears more costly than simply doing a random sampling, since it requires a set of random samples from every clustering. However, this can be mitigated by choosing random samples carefully with clustering in mind. The random samples to compute the estimator variances can be chosen two ways:

1. Perform all the clusterings, and then take random samples within each established phases.

2. Take random samples from the entire program, and then cluster the data.

The first method has the advantage that a clustering can be guaranteed to have a sufficient number of samples in each phase to perform the bootstrap. The second method has the advantage that the samples may be obtained before any clustering is performed (or in parallel with the clustering procedure). However, if the number of samples chosen is small, there may be phases which do not have an adequate number of samples to perform an accurate bootstrap.

### V.E.2 Experiments

Figure V.6 shows the results of our method in terms of reducing the error bounds of the estimated CPI for a given level of statistical confidence. As the number of phases and samples increase, the variance decreases, since we are sampling larger portions of the data, but the sampling that is driven by clustering underbounds the naive sampling method, since clusters are more homogeneous than choosing from anywhere in the data. This corresponds with standard statistical results, which say that the variance of an estimator for the

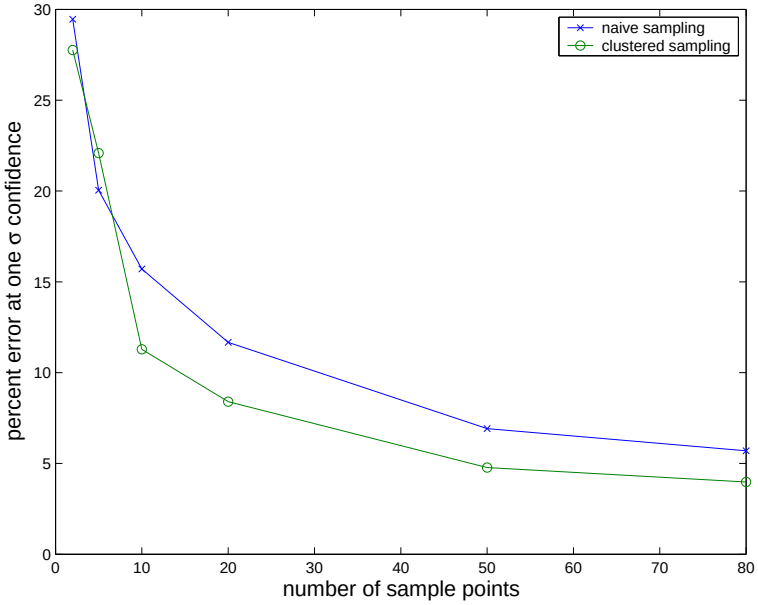


Figure V.6: The percent error bound for one estimate of CPI, averaged over 18 programs. The percent error reported here is based a normalized version of the standard deviation of the estimator, so these numbers show a one-$\sigma$ confidence bound on the percentage error. Higher confidence bounds will have the same shape, but will have larger scale. The x-axis shows the number of samples taken per estimate, which is the number of phases used for clustering for our method. For the clustering method, out of each phase we choose one interval as a representative, and compute a weighted average of the chosen intervals. For the naive bootstrap, we simply take $k$ samples, and use a bootstrap resampling procedure to compute the variance of the estimator.

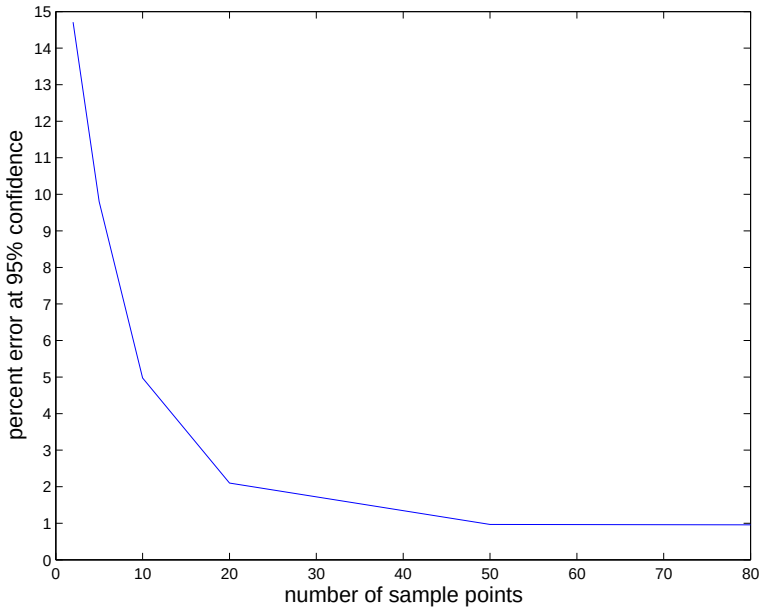


Figure V.7: The percent error bound for one estimate of CPI for `gzip-log`. These values differ from Figure V.6 in that these reflect a 95% confidence bound, which is 1.96 times greater than the one-$\sigma$ confidence bound shown in Figure V.6. The confidence level is simply a scaling of a term related to the standard deviation $\sigma$. For this program, a practitioner may set a desired percent error within which they would like to obtain the true value of the metric, and use the data from this graph to determine how many simulation points to use. For an error bound of 3%, we would choose 20 or more simulation points.

average of a sample is proportional to the size of the sample, or $\mathrm{Var}(\bar{x}) \propto 1/\sqrt{n}$ where $\bar{x}$ is the estimator of a sample of size $n$.

Using data like that which are displayed Figure V.6 will allow the computer architect to choose an appropriate number of phases independent of the BIC score, but rather on a directly related statistic, which is the variance of the metric estimator. For example, Figure V.7 shows the 95% confidence error bounds (in percentage error) for the program `gzip-log`. Supposing that a practitioner desires a 95% confidence bound on an estimate of the true error, and wants a true error of less than 3%, then he or she would choose $k = 20$ (or more) based on the results from this bootstrap.

### V.E.3 Discussion

When doing architectural simulations for many parameters on the same program/input data, we are willing to expend extra effort at the beginning of the study to find good phases to simulate. The cost of the extra effort will be amortized by using those phases on many other architectural configurations. Therefore, we have proposed a method for finding a good set of simulation points for a program that may require some additional effort in finding many phases (clustering the basic block vectors several times), and sampling from each phase (which requires detailed simulation). However, after these initial steps have been made, future simulations (using different architectural configurations) will just need to use the simulation points chosen initially by our algorithm.

The techniques described in this section allow an architecture-independent way to choose the number of simulation points to obtain accurate simulation. In addition, they give measure of statistical confidence for the expected error of the estimator. We believe that this further validates the ideas of applying clustering to the task of learning program structure for the purposes of efficient and accurate

simulation. We now turn to the issue of architectural independence.

## V.F Architecture independence

Typically in the course of doing architecture research it is necessary to take one instance of a program with a given input, and simulate its performance over many different configurations of the architecture. The same program binary with the input may be run hundreds or thousands of times to examine how, for example, the effectiveness of a given architecture changes with its cache size. Since these configurations may involve fine adjustments on the architectural parameters, it is necessary that the outcomes of the simulations are accurate and more importantly representative of the relative performance across the different configurations.

When examining overall performance, we have found that our simulation points provide representative results when varying the architecture parameters. The key insight for why this works is that the entire analysis for picking the simulation points is independent of the architecture configuration.

To examine the independence of our simulation points from the underlying architecture, we performed simulations where we greatly vary the memory hierarchy, and used the same set of simulation points for all the runs. We varied the configurations of the L1 and L2 caches. We chose L1 instruction and data cache sizes between 4 KBytes and 65 KBytes, varied the associativity between direct mapped and 4-way associative, and varied the latency from 1 to 3 cycles. In addition to this, at the same time we varied the size of the unified L2 from 500 KBytes to 2 MBytes, its associativity from 2-way to 4-way associative, and varied the latency from 10 to 25 cycles.

Figure V.8 shows the performance of the most complex SPEC benchmark `gcc` across 20 different configurations. The $y$-axis represents the perfor-

mance in $InstructionsPerCycle$ and the $x$-axis represents different memory configurations from the baseline architecture. Two lines are shown, one for the true IPC from the *complete* detailed simulation for every configuration, and the second for the estimated IPC using one set of early simulation points. Graphed for each configuration on the $x$-axis is the true IPC of the configuration and the estimated IPC from using SimPoint. The true IPC results show that significant change is seen in the performance of `gcc` as the memory hierarchy changes, where we see a range of IPC from 0.8 to 2.3. Using Early SimPoint we can see that the relative performance deviation across the configurations is consistent ranging from an error of 6% to 10%. More importantly, we can see that the error is always biased in the same direction.

These results show that the same set of simulation points provide accurate results across different architecture configurations. In [69], they say that when looking at `gcc` that "SimPoint exhibits large variations in their L2 miss rate" across different configurations they examined. We have shown and found that this is not the case when looking at important metrics like overall performance. We have found that certain *miss rates* can have a relatively large degree in error, but this occurs when the miss rates are so small, in terms of overall performance, that they do not have that large of an impact in the programs overall performance. In [69], they used an L2 cache size of 2 Megs 8-way associative for `gcc`. When using a similar L2 cache size, we have found that the true miss (local) rate for the L2 cache is 2.5%, and the estimated miss rate is 2.0%, which results in an 25% error. This may be considered to be a large variance, but this is in terms of the L2 local miss rate. In terms of the global execution of the program, only 0.02% of the executed instructions miss in the L2 cache. Therefore, this variance in L2 miss rate has very little effect on the program's performance.

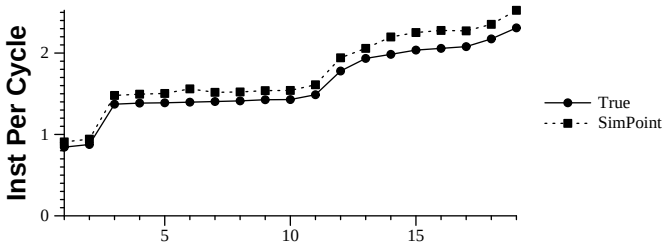


Figure V.8: This plot shows the true and estimated IPC for the program `gcc` for 20 different architecture configurations. The estimates come from our Early SimPoint algorithm. It is clear that Early SimPoint IPC estimate tracks the true IPC very well across many configurations, keeping approximately the same relative error.

## V.G Summary

To simulate a single second of execution of a 2 GHz processor running two instructions per cycle can take up to 10 hours. Consequently, researchers are forced to simulate only a subset of the full program.

In this chapter we presented an Early SimPoint algorithm that significantly reduces the time required to fast-forward through a program, while maintaining accuracy, if the simulation environment does not support checkpointing. In addition, we have presented a statistical validation method for providing an error bounds and confidence for a set of simulation points using parametric bootstrapping. For the results in this chapter, we used 100 early simulation point sets to provide the estimates to perform the statistical validation, which only required one run through the program to gather all of the desired samples, with fast-forwarding in between. In addition, we showed that the simulation points provided by Early SimPoint are representative, even as the underlying architecture changes significantly.

Statistical sampling is very accurate, as is shown in [69], but it can

take 5 times longer to gather the simulation results to reduce the CPI error from 2.3% down to 0.9%. Accurate simulation is a necessity for meaningful validation and performance estimation, but when performing design space exploration for an architecture study it is widely practiced that required accuracy can be reduced in order to efficiently search a wider design space. We envision architects wanting to have both SimPoint simulation points and statistical sampling in their tool box. Our simulation points can very quickly allow a researcher to search a design space looking for Pareto optimal design points over hundreds to thousands of configurations. If desired, these Pareto optimal points can then be statistically validated using the validation approach we present in Section V.D. If additional accuracy, is needed with a higher confidence threshold then statistical analysis can be applied over the set of configurations near the Pareto optimal points.

SimPoint is a useful technique in this methodology since there is a one time cost for picking simulation points for a program input-pair, which can then be used for accurate simulation across different architectural configurations. The one time cost of running the SimPoint tool is an attractive method for somebody that will run many simulations of the same program input pair, and because the simulation points themselves are optimized to minimize simulation time, it is an effective technique for testing a lot of configurations at minimal cost.

## V.H Acknowledgements

This chapter, in part, is a reprint of material that will appear in PACT 2003. The dissertation author was a secondary researcher and author of this paper. For a more refined version of this work, please refer to [?].

# VI

# Conclusion

The amount of data available to all kinds of scientists continues to grow at an astonishing rate. Astronomy surveys, computer architecture simulation, genome sequencing, text catalogues on the web, and recording atmospheric data are just some examples of activities which may produce up to terabytes of data in a single day. With all this data comes the potential for much knowledge, given the right tools. Aside from the challenge of storing all this data, there is the more significant problem of analyzing it automatically, as no one can hope to analyze even a small portion of the data being collected from these large projects.

Data clustering is a key technique for understanding and finding relationships in data which is either too time consuming or too difficult to analyze by hand. It also has the distinct advantage of being able to find relationships in more than three dimensions, which is a limitation of human analysis. There are many clustering algorithms for analyzing spatial data, and the $k$-means algorithm is particularly popular because it is fast (linear in time and space) and simple to implement. However, the $k$-means algorithm, and other similar algorithms, have problems because they tend to find poor-quality solutions.

In this dissertation, we have addressed this problem through several modifications of and wrappers around standard clustering algorithms. We have

shown how using alternative data weighting and cluster membership functions to create new clustering algorithms which find higher-quality solutions than $k$-means and Gaussian expectation-maximization, and how the key to finding high-quality clustering solutions with a fixed number of clusters is using a soft cluster membership function. We have also shown that the soft membership function of Gaussian expectation-maximization still leads to poor-quality solutions.

We have developed a wrapper that can operate around any existing center-based clustering algorithm that estimates the number of clusters efficiently, using statistical hypothesis testing. Our G-means algorithm performs very well at estimating the true number of clusters in data, and in finding high-quality clustering solutions. We have developed the G-means statistic, as well as shed light on using the existing Anderson-Darling statistic, as two bases for deciding whether data follows a Gaussian distribution.

G-means is able to find high-quality clusterings that correspond strongly with supervised training signals, such as handwritten digit classification. Our algorithm also lends itself to novel applications, such as estimating of the amount of structure that exists in data, so that it may be used to learn the true dimension of data. G-means is a step towards the ability to automatically analyze large amounts of spatial data.

We have shown how clustering is useful in the task of learning structure in computer architecture simulation traces. The goal of our work is to make it efficient to develop new computer architecture optimizations, and estimate quickly how well they work. Previous to our work, acceptable error rates were in the range of 80%. Our work has shown that it is possible to retain efficient simulations while reducing the error rates down to less than 10%, which is within the accuracy bounds of the simulators being used to collect statistics. We have done this through clustering and a code similarity metric called basic block vectors

to identify similar regions of program execution. We have developed this further by using structured sampling to give confidence bounds on the estimates that we predict, and to enable us to choose the appropriate number of clusters based on desired error bounds.

# Appendix A

# Proofs pertaining to the G-means statistic

Thanks to Sameer Agarwal for help with these proofs.

## A.A G-means statistic distribution

Here we prove the expected distortion and the optimal placement (in terms of minimum distortion) of two $k$-means centers in univariate Gaussian data. Let dataset $X = \{x_1, \ldots, x_n\}$ be an i.i.d. sample where each $x_i \in \mathbb{R}$. Define the minimum distortion of two $k$-means centers in $X$ as

$$r(X) \quad = \quad \frac{1}{n} \min_{c_1, c_2} \sum_i \min\{(x_i - c_1)^2, (x_i - c_2)^2\}$$

We wish to find the expected value of $r(X)$ and the expected optimal placements of $c_1, c_2$ for some distribution.

**Lemma 1 (Main results).** *Given a dataset $X = \{x_1, \ldots, x_n\}$ such that $x_i$ are i.i.d. and $x_i \sim \mathcal{N}(0, \sigma^2)$,*

$$r(X) \quad = \quad \sigma^2 \left(1 - \frac{2}{\pi}\right)$$

*and the optimal centers are located at:*

$$\begin{aligned} c_1^* &= -\frac{2\sigma}{\sqrt{2\pi}} \\ c_2^* &= \frac{2\sigma}{\sqrt{2\pi}} \end{aligned}$$

There are two ways to approach the problem of finding the expected values of $r(X)$ and $c_1, c_2 \in \mathbb{R}^d$. We may frame the problem as the minimization over $c_1, c_2$ of $r(X)$, which seems to be the obvious first step. A simpler way is to recognize that any two centers $c_1 \neq c_2$ define a unique splitting plane that bisects the data (a Voronoi tesselation). However, the converse is not true: a splitting plane does not define two unique centers. Therefore we first need to prove the following simple lemma.

**Lemma 2 (Minimum distortion is at the weighted mean).** *Given a non-negative function $f(x)$, the value of $c$ for which*

$$\int (x-c)^2 f(x) dx$$

*is minimized is given by*

$$c = \frac{\int x f(x) dx}{\int f(x) dx}$$

*Proof.*

$$\int (x-c)^2 f(x) dx \quad = \int (x^2 - 2xc + c^2) f(x) dx$$

Differentiating the above expression with respect to $c$ and setting to zero gives:

$$\begin{aligned} \frac{\partial}{\partial c} \int (x^2 - 2xc + c^2) f(x) dx &= 0 \\ \int \frac{\partial}{\partial c} (x^2 - 2xc + c^2) f(x) dx &= 0 \\ \int 2(c-x) f(x) dx &= 0 \\ c &= \frac{\int x f(x) dx}{\int f(x) dx} \end{aligned}$$

□

Now let $-\infty < a < \infty$ be the splitting point which splits the data and the centers, such that $c_1 < a < c_2$. Then we can re-define the optimal center positions $c_1^*$ and $c_2^*$ according to Lemma 2:

$$\begin{aligned} c_1^* &= \frac{\int_{-\infty}^{a} x f(x) dx}{\int_{-\infty}^{a} f(x) dx} \\ c_2^* &= \frac{\int_{a}^{\infty} x f(x) dx}{\int_{a}^{\infty} f(x) dx} \end{aligned}$$

Then we have the expected distortion (per point) as:

$$\begin{aligned} r(X) &= \min_{c_1,c_2} \int_{-\infty}^{\infty} \min\{(x-c_1)^2, (x-c_2)^2\} f(x) dx \\ &= \min_{a} \int_{-\infty}^{\infty} \min\{(x-c_1^*)^2, (x-c_2^*)^2\} f(x) dx \\ &= \min_{a} \left[ \int_{-\infty}^{a} (x-c_1^*)^2 f(x) dx + \int_{a}^{\infty} (x-c_2^*)^2 f(x) dx \right] \\ &= \min_{a} \left[ \int_{-\infty}^{a} (x^2 - 2xc_1^* + c_1^{*2}) f(x) dx + \int_{a}^{\infty} (x^2 - 2xc_2^* + c_2^{*2}) f(x) dx \right] \end{aligned}$$

Now we consider the problem of minimizing this quantity by finding the optimal $a$:

$$\begin{aligned} & \arg\min_{a} \left[ \int_{-\infty}^{a} (x^2 - 2xc_1^* + c_1^{*2}) f(x) dx + \int_{a}^{\infty} (x^2 - 2xc_2^* + c_2^{*2}) f(x) dx \right] \\ = & \int_{-\infty}^{\infty} x^2 f(x) dx \\ & + \arg\min_{a} \left[ \int_{-\infty}^{a} (-2xc_1^* + c_1^{*2}) f(x) dx + \int_{a}^{\infty} (-2xc_2^* + c_2^{*2}) f(x) dx \right] \end{aligned}$$

We can drop the quantity $\int_{-\infty}^{\infty} x^2 f(x) dx$, since it does not depend on $a$. Continuing, we obtain:

$$\begin{aligned} & \arg\min_{a} \left[ \int_{-\infty}^{a} (-2xc_1^* + c_1^{*2}) f(x) dx + \int_{a}^{\infty} (-2xc_2^* + c_2^{*2}) f(x) dx \right] \\ = & \arg\min_{a} \left[ -2c_1^* \int_{-\infty}^{a} x f(x) dx + c_1^{*2} \int_{-\infty}^{a} f(x) dx \right. \end{aligned}$$

$$-2c_2^* \int_a^\infty x f(x)dx + c_2^{*2} \int_a^\infty f(x)dx \Bigg]$$

We have reached this point without any assumptions about the form of $f(x)$, except for its non-negativity. We now assume that the data takes the form of Gaussian, so that:

$$x_i \sim \mathcal{N}(0, \sigma^2)$$

thus,

$$f(x; \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-x^2/(2\sigma^2)}$$

We have assumed without loss of generality that the distribution has zero mean. Now we want to find the $a$ for which the expected distortion is minimized for Gaussian data. We remove the arg min directive, and work with its quantity, which we will call $R(a)$.

$$\begin{aligned} R(a) &= -2c_1^* \int_{-\infty}^a x \frac{e^{-x^2/(2\sigma^2)}}{\sqrt{2\pi\sigma^2}} dx + c_1^{*2} \int_{-\infty}^a \frac{e^{-x^2/(2\sigma^2)}}{\sqrt{2\pi\sigma^2}} dx \\ &\quad -2c_2^* \int_a^\infty x \frac{e^{-x^2/(2\sigma^2)}}{\sqrt{2\pi\sigma^2}} dx + c_2^{*2} \int_a^\infty \frac{e^{-x^2/(2\sigma^2)}}{\sqrt{2\pi\sigma^2}} dx \\ &= \left[ \frac{2c_1^* \sigma e^{-x^2/(2\sigma^2)}}{\sqrt{2\pi}} + c_1^{*2} \Phi(x/\sigma) \right]_{-\infty}^a + \left[ \frac{2c_2^* \sigma e^{-x^2/(2\sigma^2)}}{\sqrt{2\pi}} + c_2^{*2} \Phi(x/\sigma) \right]_a^\infty \end{aligned}$$

Where here $\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-t^2} dt$ is the cumulative distribution function of the of the normal distribution. Evaluating integral bounds and combining terms, we get

$$R(a) = \frac{2\sigma}{\sqrt{2\pi}} e^{-a^2/2\sigma^2} (c_1^* - c_2^*) + \Phi(a/\sigma)(c_1^{*2} - c_1^{*2}) + c_2^{*2}$$

We now evaluate $c_1^*$ and $c_2^*$ and their squares according to our assumed distribution:

$$c_1^* = \frac{\int_{-\infty}^a x \frac{1}{\sqrt{2\pi\sigma^2}} e^{-x^2/(2\sigma^2)} dx}{\int_{-\infty}^a \frac{1}{\sqrt{2\pi\sigma^2}} e^{-x^2/(2\sigma^2)} dx}$$

$$
\begin{aligned}
&= \frac{\left[-\frac{\sigma}{\sqrt{2\pi}}e^{-x^2/(2\sigma^2)}\right]_{-\infty}^{a}}{[\Phi(x/\sigma)]_{-\infty}^{a}} \\
&= \frac{-\sigma e^{-a^2/(2\sigma^2)}}{\sqrt{2\pi}\Phi(a/\sigma)} \\
c_1^{*2} &= \frac{\sigma^2 e^{-a^2/\sigma^2}}{2\pi\Phi(a/\sigma)^2}
\end{aligned}
$$

similarly,

$$
\begin{aligned}
c_2^{*} &= \frac{\sigma e^{-a^2/(2\sigma^2)}}{\sqrt{2\pi}(1-\Phi(a/\sigma))} \\
c_2^{*2} &= \frac{\sigma^2 e^{-a^2/\sigma^2}}{2\pi(1-\Phi(a/\sigma))^2}
\end{aligned}
$$

Now that we have these defined in terms of the Gaussian distribution, we can substitute them into $R(a)$:

$$
\begin{aligned}
R(a) &= \frac{2\sigma}{\sqrt{2\pi}}e^{-a^2/(2\sigma^2)}\left(\frac{-\sigma e^{-a^2/(2\sigma^2)}}{\sqrt{2\pi}\Phi(a/\sigma)} - \frac{\sigma e^{-a^2/(2\sigma^2)}}{\sqrt{2\pi}(1-\Phi(a/\sigma))}\right) \\
&\quad +\Phi(a/\sigma)\left(\frac{\sigma^2 e^{-a^2/\sigma^2}}{2\pi\Phi(a/\sigma)^2} - \frac{\sigma^2 e^{-a^2/\sigma^2}}{2\pi(1-\Phi(a/\sigma))^2}\right) \\
&\quad +\frac{\sigma^2 e^{-a^2/\sigma^2}}{2\pi(1-\Phi(a/\sigma))^2}
\end{aligned}
$$

After a lot of algebra which we will not replicate here, we obtain the much simpler equation:

$$
R(a) = -\frac{\sigma^2 e^{-a^2/\sigma^2}}{2\pi\Phi(a/\sigma)(1-\Phi(a/\sigma))}
$$

Though it is not easy to use the derivative of this function to find the minimum, it is clear that this function is minimized when $a = 0$. We prove this in two steps by looking at the numerator and denominator separately. The numerator has the term $e^{-a^2/\sigma^2}$, which has a maximum value of 1 when $a = 0$. However, the numerator has a leading negative sign, so this maximum of the exponential function is a minimum of the numerator. Therefore, the numerator is minimized

when $a = 0$. The denominator is of the form $\Phi(a/\sigma)(1 - \Phi(a/\sigma))$, which also has its maximum value at $a = 0$ of $0.5(1 - 0.5) = 0.25$. This maximum value in the denominator contributes to the minimum of the overall equation. Therefore, $a = 0$ is the minimum of $R(a)$, and hence of $r(X)$. We illustrate this in Figure A.1.

If $a = 0$, then we can restate the optimal center locations in simpler terms:

$$\begin{aligned} c_1^* &= -\frac{2\sigma}{\sqrt{2\pi}} \\ c_2^* &= \frac{2\sigma}{\sqrt{2\pi}} \\ R(0) &= -\frac{2\sigma^2}{\pi} \end{aligned}$$

This proves the result we used in Chapter III for optimal center initialization of the G-means algorithm.

We also wish to know the value of $r(X)$, which is the minimum distortion for two centers in the data, so we must use $R(0)$ and add back in the constant term we removed for minimization.

$$\begin{aligned} r(X) &= \int_{-\infty}^{\infty} x^2 \frac{1}{\sqrt{2\pi\sigma^2}} e^{-x^2/(2\sigma^2)} dx + R(0) \\ &= \left[ -\frac{\sigma x}{\sqrt{2\pi}} e^{-x^2/(2\sigma^2)} \right]_{-\infty}^{\infty} + \left[ \sigma^2 \Phi(x/\sigma) \right]_{-\infty}^{\infty} + R(0) \\ &= (0 - 0) + \sigma^2(1 - 0) + R(0) \\ &= \sigma^2 - \frac{2\sigma^2}{\pi} \\ &= \sigma^2 \left( 1 - \frac{2}{\pi} \right) \end{aligned}$$

It should be obvious that the value of $\int x^2 \frac{1}{\sqrt{2\pi\sigma^2}} e^{-x^2/(2\sigma^2)} dx$ for zero-mean data is $\sigma^2$. I have derived it above for completeness.

This result shows that the expected optimal distortion (per point) for two centers in data sampled from one univariate Gaussian is slightly less than

the distortion for one center – specifically, it is less by $\frac{2\sigma^2}{\pi}$.

We can apply this result to multivariable distortion (two centers in one true multivariate Gaussian of any shape) along the axis with maximal spread (the first prinicpal component). Without loss of generality, we can rotate any multivariate Gaussian data so that the principal component is aligned with the first dimension, and then apply this result to that dimension. The distortion in other orthogonal dimensions remains unchanged, and can be integrated out due to symmetry.

This result is used in Chapter III to give the optimal starting point for the $k$-means algorithm run with two centers on data drawn from one Gaussian. This best starting point will be at $\mu \pm 2s\sigma/\sqrt{2\pi}$, where in univariate data $s = 1$, and in multivariate data $s$ is the normalized vector indicating the principal component of the data.

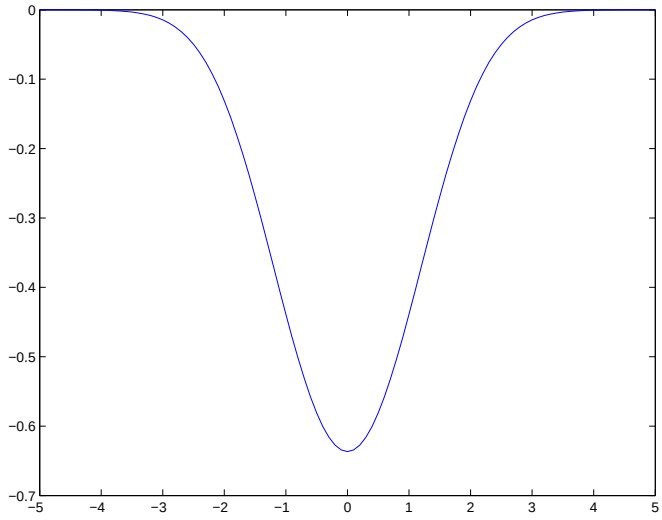


Figure A.1: The function showing the minimum-distortion splitting point for Gaussian data. This is the function $R(a)$ from the text.

## A.B A distribution with higher expected distortion

It is easy to think of a distribution of data which has a lower expected $r(X)$ than when the data is Gaussian, such as data sampled a mixture of two Gaussians where the two Gaussians are far apart. Here we show that there also exist distributions that have a higher expected $r(X)$ than the Gaussian distribution, and that it is appropriate to split into two centers when those are encountered.

**Lemma 3 (Distribution with higher distortion than Gaussian).** *Define a distribution which is a mixture of three delta functions:*

$$f(x;\alpha,\epsilon) \quad = \quad \epsilon\delta(x+\alpha) + (1-2\epsilon)\delta(x) + \epsilon\delta(x=\alpha)$$

*where* $\delta(x) = 1$ *if* $x = 0$, *0 otherwise, and* $2\epsilon\alpha^2 = \sigma^2$. *This distribution has a higher distortion for two centers than under Gaussian with variance* $\sigma^2$ *when*

$$\epsilon \quad < \quad 1 - \frac{\pi}{4}$$

*Proof.* First consider the one-center distortion:

$$\epsilon(-\alpha)^2 + (1-2\epsilon)(0)^2 + \epsilon(-\alpha)^2 \quad = \quad 2\epsilon\alpha^2$$

We set this equal to the distortion in the Gaussian case ($\sigma^2$), so that

$$2\epsilon\alpha^2 \quad = \quad \sigma^2$$

Now we consider where two centers $c_1$ and $c_2$ may be placed. We do not try to optimize this, but we consider a reasonable solution where $c_1$ claims the leftmost delta function, and $c_2$ claims the middle and rightmost delta functions. Using Lemma 2:

$$c_1 \quad = \quad -\alpha\epsilon/\epsilon$$

$$\begin{aligned} &= -\alpha \\ c_2 &= \frac{0(1-2\epsilon)+\alpha\epsilon}{1-2\epsilon+\epsilon} \\ &= \frac{\alpha\epsilon}{1-\epsilon} \end{aligned}$$

Now we find the two-center distortion using these two centers:

$$\begin{aligned} & (-\alpha+\alpha)\epsilon + \left(0-\frac{\alpha\epsilon}{1-\epsilon}\right)^2(1-2\epsilon) + \left(\alpha-\frac{\alpha\epsilon}{1-\epsilon}\right)^2\epsilon \\ = & \frac{\alpha^2\epsilon^2(1-2\epsilon)}{(1-\epsilon)^2} + \frac{(\alpha(1-\epsilon)-\alpha\epsilon)^2\epsilon}{(1-\epsilon)^2} \\ = & \frac{\alpha^2\epsilon^2 - 2\alpha^2\epsilon^3 + \alpha^2\epsilon - 4\alpha^2\epsilon^2 + 4\alpha^2\epsilon^3}{(1-\epsilon)^2} \\ = & \frac{2\alpha^2\epsilon^3 - 3\alpha^2\epsilon^2 + \alpha^2\epsilon}{(1-\epsilon)^2} \\ = & \frac{\alpha^2\epsilon(1-2\epsilon)(1-\epsilon)}{(1-\epsilon)^2} \\ = & \frac{\alpha^2\epsilon(1-2\epsilon)}{1-\epsilon} \end{aligned}$$

We now set this in an inequality against the expected distortion for two centers in a Gaussian, namely $\sigma^2(1-2/\pi)$. We also utilize the identity $2\epsilon\alpha^2 = \sigma^2$

$$\begin{aligned} \frac{\alpha^2\epsilon(1-2\epsilon)}{1-\epsilon} &> \sigma^2\left(1-\frac{2}{\pi}\right) \\ \frac{\sigma^2(1-2\epsilon)}{2(1-\epsilon)} &> \sigma^2\left(1-\frac{2}{\pi}\right) \\ \pi(1-2\epsilon) &> 2(\pi-2)(1-\epsilon) \\ \pi-2\pi\epsilon &> 2\pi-4-2\pi\epsilon+4\epsilon \\ 4-\pi &> 4\epsilon \\ \epsilon &< 1-\frac{\pi}{4} \end{aligned}$$

□

Thus, as long as $\epsilon < 1-\pi/4 \approx 0.2146$, this distribution will have a higher distortion for two centers than the Gaussian distribution. This proof shows that

there are distributions in which the distortion of two centers is higher, for the same overall distortion. This is important for the G-means statistic, as it was not previously known if any such distribution existed which had this property. As we can see, this distribution is a mixture of three clusters (delta functions), so in the case that two centers find a higher distribution than expected, it is appropriate to use more than one center to model the data.

# Bibliography


[1] Henry D. I. Abarbanel. *Analysis of observed chaotic data*. Springer-Verlag, New York, 1996.

[2] Charu C. Aggarwal, Alexander Hinneburg, and Daniel A. Keim. On the surprising behavior of distance metrics in high dimensional space. In *Proceedings of the 8th International Conference on Database Theory (ICDT)*, pages 420–434, 2001.

[3] U. Alon, N. Barkai, D.A. Notterman, K. Gish, S. Ybarra, D. Mack, and A.J. Levine. Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays. In *Proceedings of the National Academy Sciences USA*, volume 96, pages 6745–6750, 1999.

[4] Daniel Barbará and Ping Chen. Using the fractal dimension to cluster datasets. In *Proceedings of the sixth ACM SIGKDD International conference on Knowledge discovery and data mining*, pages 260–264. ACM Press, 2000.

[5] Daniel Barbará, Julia Couto, and Yi Li. COOLCAT: An entropy-based algorithm for categorical clustering. In *Eleventh International Conference on Information and Knowledge Management (CIKM 2002)*, pages 582–589, 2002.

[6] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An index structure for high-dimensional data. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, San Francisco, U.S.A., 1996. Morgan Kaufmann Publishers.

[7] James C. Bezdek. *Pattern Recognition with Fuzzy Objective Function Algorithms*. Plenum Press, New York, 1981.

[8] Horst Bischof, Aleš Leonardis, and Alexander Selb. MDL principle for robust vector quantisation. *Pattern analysis and applications*, 2:59–72, 1999.

[9] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.

[10] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998. http://www.ics.uci.edu/∼mlearn/MLRepository.html.

[11] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *Annals of Mathematical Statistics*, 29:610–611, 1958.

[12] Paul S. Bradley and Usama M. Fayyad. Refining initial points for K-Means clustering. In *Proc. 15th International Conf. on Machine Learning*, pages 91–99. Morgan Kaufmann, San Francisco, CA, 1998.

[13] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[14] George Casella and Roger L. Berger. *Statistical Inference*. Duxbury, Pacific Grove, CA, USA, 2002.

[15] Dorin Comaniciu and Peter Meer. Mean shift: a robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2001.

[16] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, October 1996.

[17] Ralph B. D'Agostino and Michael A. Stephens. *Goodness-of-fit techniques*. Dekker, New York, 1986.

[18] Sanjoy Dasgupta. Experiments with random projection. In *Uncertainty in Artificial Intelligence: Proceedings of the Sixteenth Conference (UAI-2000)*, pages 143–151, San Francisco, CA, 2000. Morgan Kaufmann Publishers.

[19] N. E. Day. Estimating the components of a mixture of normal distributions. *Biometrika*, 56(3):463–474, December 1969.

[20] Gianna M. Del Corso. Estimating an eigenvector by the power method with a random start. *SIAM Journal on Matrix Analysis and Applications*, 18(4):913–937, 1997.

[21] A P Dempster, N M Laird, and D B Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society B*, 39:185–197, 1977.

[22] Chris Ding, Xiaofeng He, Hongyuan Zha, and Horst Simon. Adaptive dimension reduction for clustering high dimensional data. In *Proceedings of the 2nd IEEE International Conference on Data Mining*, 2002.

[23] Drineas, Frieze, Kannan, Vempala, and Vinay. Clustering in large graphs and matrices. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1999.

[24] Charles Elkan. A faster $k$-means algorithm. In *Proceedings of the 20th International Conference on Machine Learning (ICML 2003)*, 2003. To appear.

[25] Fredrik Farnstrom, James Lewis, and Charles Elkan. Scalability for clustering algorithms revisited. *SIGKDD Explorations*, 2(1):51–57, 2000.

[26] Charless Fowlkes, Serge Belongie, and Jitendra Malik. Efficient spatiotemporal grouping using the Nyström method. In *Proceedings of IEEE Computer Vision and Pattern Recognition Conference (CVPR)*, volume 1, pages 231–238, 2001.

[27] Yoav Freund and Robert Schapire. A short introduction to boosting. *Japanese Society for Artificial Intelligence*, 14:771–780, 1999.

[28] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, MA, 1984.

[29] Greg Hamerly and Charles Elkan. Bayesian approaches to failure prediction for disk drives. In *Proceedings of the eighteenth international conference on machine learning*, pages 202–209. Morgan Kaufmann, San Francisco, CA, 2001.

[30] Greg Hamerly and Charles Elkan. Alternatives to the k-means algorithm that find better clusterings. In *Eleventh International Conference on Information and Knowledge Management (CIKM 2002)*, pages 600–607, 2002.

[31] Greg Hamerly and Charles Elkan. Learning the $k$ in $k$-means. Submitted for publication, 2003.

[32] J. Haskins and K. Skadron. Minimal subset evaluation: Rapid warm-up for simulated hardware state. In *Proceedings of the 2001 International Conference on Computer Design*, September 2001.

[33] J. Haskins and K. Skadron. Memory reference reuse latency: Accelerating sampled microarchitecture simulations. Technical Report CS-2002-19, U of Virginia, July 2002.

[34] Peter J. Huber. Projection pursuit. *Annals of Statistics*, 13(2):435–475, June 1985.

[35] L. Hubert and P. Arabie. Comparing partitions. *Journal of Classification*, 2:193–218, 1985.

[36] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[37] Annaka Kalton, Pat Langley, Kiri Wagstaff, and Jungsoon Yoo. Generalized clustering, supervised learning, and data assignment. In *Proceedings of the Seventh International Conference on Knowledge Discovery and Data Mining*, pages 299–304, San Francisco, CA, 2001. ACM Press.

[38] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. The analysis of a simple $k$-means clustering algorithm. In *Proceedings of the sixteenth ACM Symposium on Computational Geometry*, pages 100–109, 2000.

[39] Robert E. Kass and Larry Wasserman. A reference Bayesian test for nested hypotheses and its relationship to the Schwarz criterion. *Journal of the American Statistical Association*, 90(431):928–934, 1995.

[40] Michael Kearns, Yishay Mansour, and Andrew Y. Ng. An information-theoretic analysis of hard and soft assignment methods for clustering. In *Proceedings of Uncertainty in Artificial Intelligence*, pages 282–293. AAAI, 1997.

[41] Michael J. Kearns, Yishay Mansour, Andrew Y. Ng, and Dana Ron. An experimental and theoretical comparison of model selection methods. In *Computational Learing Theory (COLT)*, pages 21–30, 1995.

[42] Jon Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 599–608, 1997.

[43] A. KleinOsowski, J. Flynn, N. Meares, and D. Lilja. Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research. In *Proceedings of the International Conference on Computer Design*, September 2000.

[44] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *Workload Characterization of Emerging Applications, Kluwer Academic Publishers*, September 2000.

[45] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[46] A. Likas, N. Vlassis, and J.J. Verbeek. The global k-means clustering algorithm. Technical report, Computer Science Institute, University of Amsterdam, The Netherlands, February 2001. IAS-UVA-01-02.

[47] J. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. LeCam and J. Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley, CA, 1967. University of California Press.

[48] Marina Meĭla and David Heckerman. An experimental comparison of model-based clustering methods. *Machine learning*, 42:9–29, 2001.

[49] Andrew W. Moore. The anchors hierarchy: Using the triangle inequality to survive high dimensional data. In *Proceedings of The Sixteenth Conference on Uncertainty in Artificial Intelligence*, pages 397–405, 2000.

[50] D. Mount and S. Arya. ANN: a library for approximate nearest neighbor searching, 1997. 2nd annual CGC workshop on computational geometry.

[51] Andrew Ng, Michael Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. *Neural Information Processing Systems*, 14, 2002.

[52] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[53] M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor designs. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[54] H. Peitgen, H. Jurgens, and D. Saupe. *Chaos and Fractals: New Frontiers of Science*. Springer-Verlag, 1992.

[55] Dan Pelleg and Andrew Moore. Accelerating exact $k$-means algorithms with geometric reasoning. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 277–281. AAAI Press, 1999.

[56] Dan Pelleg and Andrew Moore. $X$-means: Extending $K$-means with efficient estimation of the number of clusters. In *Proceedings of the 17th International Conf. on Machine Learning*, pages 727–734. Morgan Kaufmann, San Francisco, CA, 2000.

[57] J.M. Peña, J.A. Lozano, and P. Larrañaga. An empirical comparison of four initialization methods for the k-means algorithm. *Pattern recognition letters*, 20:1027–1040, 1999.

[58] Peter Sand and Andrew Moore. Repairing faulty mixture models using density estimation. In *Proceedings of the 18th International Conf. on Machine Learning*. Morgan Kaufmann, San Francisco, CA, 2001.

[59] Peter Sand and Andrew Moore. Fastmix clustering software, 2002. `http://www.cs.cmu.edu/~psand/`.

[60] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[61] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[62] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming*, October 2002.

[63] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. Technical Report CS2002-0710, UC San Diego, June 2002.

[64] Tim Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, 2002.

[65] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[66] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[67] M. A. Stephens. EDF statistics for goodness of fit and some comparisons. *American Statistical Association*, 69(347):730–737, September 1974.

[68] Michael A. Stephens. Asymptotic results for goodness-of-fit statistics with unknown parameters. *Annals of Statistics*, 4(2):357–369, 1976.

[69] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[70] Lofti Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

[71] Bin Zhang. Generalized k-harmonic means – boosting in unsupervised learning. Technical Report HPL-2000-137, Hewlett-Packard Labs, 2000.

[72] Bin Zhang, Meichun Hsu, and Umeshwar Dayal. K-harmonic means – a data clustering algorithm. Technical Report HPL-1999-124, Hewlett-Packard Labs, 1999.

[73] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: an efficient data clustering method for very large databases. In *ACM SIGMOD International Conference on Management of Data*, pages 103–114, Montreal, Canada, June 1996.