ABSTRACT

Clustering in High Dimension and Choosing Cluster Representatives for SimPoint

Joshua Benjamin Johnston, M.S.

Mentor: Gregory J. Hamerly, Ph.D.

In computer architecture, researchers compare new processor designs by simulating them in software. Because simulation is slow, researchers simulate small parts of a workload to save time. The widely successful SimPoint approach identifies these key parts with k-means clustering. The extremely high-dimensional nature of these workloads causes difficulties for k-means, so SimPoint must reduce the dimension before clustering. We propose clustering workload data with the exponential Dirichlet compound multinomial (EDCM), a new relative of the multinomial probability distribution and the first model that has been used to cluster workload data without the need for dimension reduction. The EDCM mixture produces good models which have far fewer clusters than models generated by k-means, significantly reducing the amount of time spent in simulation. The EDCM mixture converges quickly and is a good model for "bursty" traits which appear in workloads. We discuss model selection and choosing cluster representatives for the EDCM mixture.

Clustering in High Dimension and Choosing Cluster Representatives for Simpoint

by

Joshua Benjamin Johnston, B.S.

A Thesis

Approved by the Department of Computer Science

_____

Donald L. Gaitros, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science

Approved by the Thesis Committee

_____

Gregory J. Hamerly, Ph.D., Chairperson

_____

David B. Sturgill, Ph.D.

_____

Dennis A. Johnston, Ph.D.

Accepted by the Graduate School
August 2007

_____

J. Larry Lyon, Ph.D., Dean

Page bearing signatures is kept on file in the Graduate School.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE

Introduction

Consider the complexity of a modern microprocessor. Billions of transistors are arranged, in precise nanometer scale, to form arithmetic units, caches, execution pipelines, and all the other microscopic structures needed to power today's computer systems. Now consider the planning needed to design the next generation of super-scalar processors. Planning out the right combination of design parameters is a daunting task. How big should one make the level one (L1) cache? How many stages should the execution pipeline have? What memory latency should be used? All of these questions must be answered at the same time, examining the trade-offs between various statistics and parameter configurations, in order to achieve the highest level of performance for a new processor design.

Testing a particular set of these parameters is painstaking. Unable to test each different set of parameters in hardware, one must use a software simulator to gather runtime statistics about a particular set of parameters. One may feed the simulator a standard set of benchmarks, like the SPEC CPU2000 suite . On the SimpleScalar simulator (Burger and Austin 1997), which can process about 400 million instructions per hour (depending on the speed of the computer system on which it runs), running the full set of SPEC CPU2000 benchmarks, with 4.06 trillion instructions to execute, will take 1.16 years of CPU time. This is only for one particular set of parameters. These same simulations must be run for many combinations if one is to find the best set of parameters to use for a design.

The SimPoint application uses the machine learning method of data clustering to determine which portions of a benchmark's execution are similar. By only running detailed simulations on parts of these similar portions, researchers are able to

get estimated statistics of their hardware design with only a small amount of error compared to running the full benchmark through a detailed simulation. The CPU time required to simulate these small portions is cut by several orders of magnitude, from roughly 1.16 years to around 6 days.

We are researching a new model for clustering the high dimensional program traces clustered by SimPoint. The exponential Dirichlet compound multinomial (EDCM) possesses several qualities that make it attractive over the current clustering method used by SimPoint, the k-means algorithm. The EDCM is able to handle data in high dimension and is able to account for temporal locality, a phenomenon often encountered in a program's execution.

1.1 What is Clustering

In the most general sense, clustering is concerned with dividing a set of examples into groups of similar items and, at the same time, separating those that are dissimilar. The definitions of what it means for two items to be similar or dissimilar can vary widely, depending on the problem context and clustering algorithm used. However, there must be a mathematical foundation and metric for comparing two examples so the clustering can be carried out automatically by a computer program.

Clustering is considered to belong in the machine learning subfield of unsupervised learning. This means clustering is performed without a training signal to tell the clustering algorithm whether or not it is correct. Often times, we don't know how many clusters we should be looking for, what the clusters might possibly look like, or how the data should relate to one another. Contrast this to supervised learning, where we have a training signal, a set of correct answers for a given set of data. Hence the difficulty and unique challenge posed by clustering—our algorithms must perform accurate clustering (accuracy can be measured many ways) without any *a priori* knowledge (besides basic assumptions imposed by the choice of clustering

algorithm) about how the data will be organized or how the examples in our set of data are related (Jain *et al.* 1999).

In many applications, the purpose of clustering is to find a concise representation for a large set of examples by analyzing and determining a structure among the data and fitting a set of clusters to them. If we can gather all similar examples into one group and find some way to represent the group as a whole, we have summarized the information for all the data examples in that cluster. The more clusters we have, the more precise and fine-grained our representation of the overall dataset can be. The fewer clusters we have, the greater savings we enjoy in the compression of data representation, at the cost of losing finer detail about the dataset. Some different types of representations that may be used to summarize a clustering may be the cluster centroids (if using distance-based geometric similarity) or the parameters used to fit a probability distribution to the features of the data in the cluster (using something like a multinomial probability distribution).

Choosing a similarity metric can be difficult. Even within a particular clustering algorithm, one may have many methods by which data items can be compared. The choice is not always straightforward and different methods will have various strengths and weaknesses, even for the same clustering problem. In some cases, the most intuitive or appropriate metric to use may be some measure of distance—how far apart things are. Or, it may be some measure of probability—how likely it is that a particular cluster contains the data that it does. These metrics will be examined in more detail later.

Jain and Dubes (1988) propose a series of general steps taken to solve a clustering problem.

(1) Determine the number of clusters (k) to be found and the features of the data to be used by the clustering process (which might not be an option for the user).

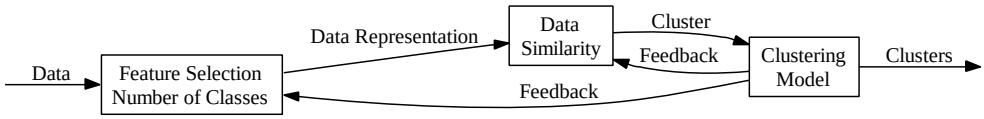


Figure 1.1: Feedback loop used in many clustering contexts. Information about the final clusterings is often used to refine initial assumptions about feature choice or similarity metrics, giving a new and better clustering at the end of the next iteration. Most of the time, the loop is terminated when there is not a significant change in the clusterings from one iteration to the next, or when we have determined to have found the best clustering (if possible). (Jain *et al.* 1999)

(2) Define or choose a metric to be used for data similarity.

(3) Perform the clustering using the appropriate clustering algorithm for the domain (see below for a discussion about different types of algorithms), yielding a set of clusters.

(4) Extract a summarization of the data from the clustering.

Often, the first three steps are repeated in a feedback loop (Figure 1.1) to optimize the clustering, using measurements on the clusters given in step three to reinforce certain important features, updating the number of clusters (k) needed to more accurately represent the data, or updating parameters for the similarity metric. The final clustering will be one that converges without a significant change in the clusters from one iteration to the next (with the definition of significance depending again on the problem context), or until we have determined to have found the best clustering (if possible). Again, the definition of what makes a clustering better than another depends on the problem context.

1.2 Categories of Clustering Algorithms

Clustering is a broad field and there are many different algorithms for clustering different sets of data. Depending on both the context of the clustering problem and the needs of the practitioner, there are many different approaches to clustering

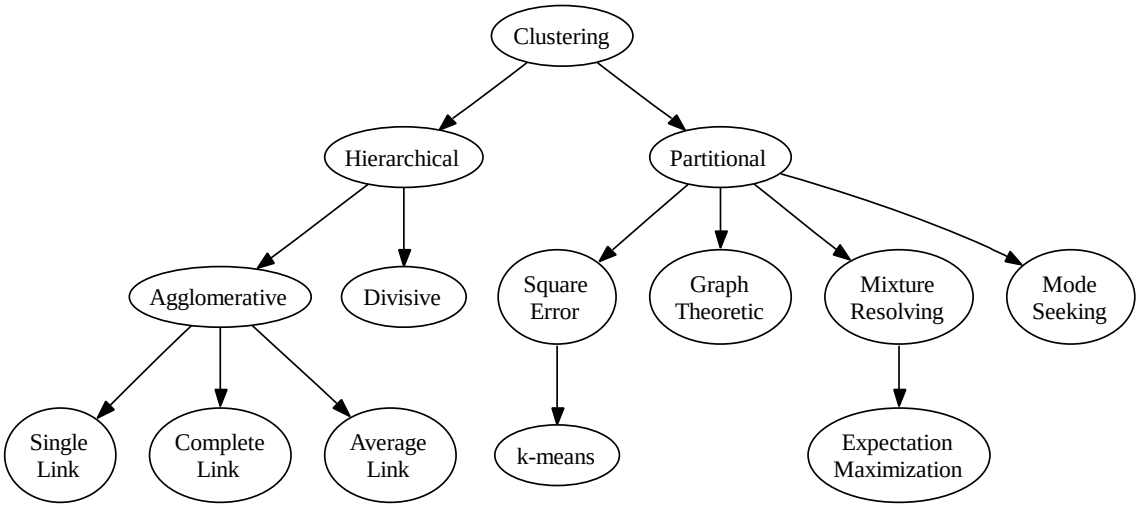


Figure 1.2: General hierarchy of clustering algorithms. Many algorithms will span multiple types, as these divisions are not perfectly disjoint. (Jain and Dubes 1988)

which fall into two broad categories, hierarchical and partitioning. Figure 1.2 (Jain and Dubes 1988) gives a visual synopsis of some of the most common clustering types. Obviously, every variation on these themes cannot be represented in such a concise figure. Nor are all the divisions of the algorithms perfectly disjoint. Across the division of clustering algorithms offered by Jain and Dubes lie many common considerations that can be applied to a variety of situations, depending on the needs of individual algorithms (1988).

Hierarchical clustering algorithms fall into two basic categories, agglomerative and divisive. Hierarchical clustering methods allow one to select the amount of granularity and detail used in data representation by choosing different levels of the hierarchy.

Agglomerative methods start by considering each data example in its own cluster. Through successive iterations of the algorithm, clusters are combined into larger groups until all the data examples are contained in one large cluster. However, all the data about the sub-clusters is preserved. Agglomerative methods consider which clusters to combine using single-link, complete-link, or average-link methods (there are

more methods which can be used). The three linkage methods consider the distance between two clusters to be some function of the distances between pairs of points, where one point lies in one cluster, and the other point lies in the other cluster. Single-link and complete-link methods consider the distance between two clusters to be the minimum or maximum distance, respectively, between all such pairs of points. Average-link is, as the name implies, a trade-off between the other two. The distance between two clusters is the average distance between all such pairs. In any case, the two closest clusters are combined each round until there is only one large cluster encompassing all the examples.

Divisive hierarchical methods work in the opposite manner. To begin, all data points are contained in one large cluster. Through successive iterations of the algorithm, the cluster(s) are divided at some point until each data input resides in a cluster by itself. Divisive methods are not as widely used or researched as agglomerative methods (Hastie *et al.* 2001).

Partitioning clustering algorithms are also generally called flat algorithms, because they do not create hierarchies of information. Instead, they partition the set of data examples into groups based on some metric.

Squared-error algorithms are generally the most easy to understand and implement, seeking to minimize the squared distance of each data point from the centroid of the cluster it is assigned to. k-means is a very popular algorithm using the squared error approach and is the clustering algorithm used in the current version of the SimPoint application (Sherwood *et al.* 2002), which will be discussed in detail later.

Mixture-resolving methods assume that data examples are generated by some number of probability distributions (like multinomials). The goal of these types of algorithms is to estimate the various parameters of the distributions mixtures. Usually, the log likelihood of the model is maximized through iterative methods, like the

Expectation-Maximization algorithm (Dempster *et al.* 1977). The EM algorithm is described in more detail later in the thesis.

Graph theoretic algorithms consider the data examples to be part of a graph data structure and perform different transformations on the graph. For example, Zahn (1971) considers a minimum weight spanning tree of all the examples. The examples are nodes connected by edges of weight equal to the distances between them. Clusters are generated by cutting the edges with the greatest weights.

Spectral clustering is a rather new area of clustering that also uses graph theory. Consider a set of examples to be vertices in a graph, with edges between them having weights corresponding to the distances between examples in some geometric space. One might think of separating the graph into k sections, cutting along the edges with the highest weights, to generate k groups of vertices that are all close together. Finding an optimal partitioning (minimizing the number of cuts and the weight of the edges between the nodes in each cluster) is NP-hard, but it can be approximated using spectral techniques, where we examine the eigenvectors of the graph's Laplacian (Ng *et al.* 2001).

There are many other types of clustering algorithms other than the ones shown here. The algorithms listed above serve as a general taxonomy rather than an exhaustive compendium. The k-means and EM algorithms are the main focus of this thesis.

1.3 Motivation of the Thesis

The primary focus of this thesis is applying the EM algorithm to cluster data in high dimension. Clustering data in high dimension must handle many interesting problems that do not occur in lower dimension. Many common clustering domains occur naturally in high dimension spaces, however, and it is important that clustering techniques are developed to handle them.

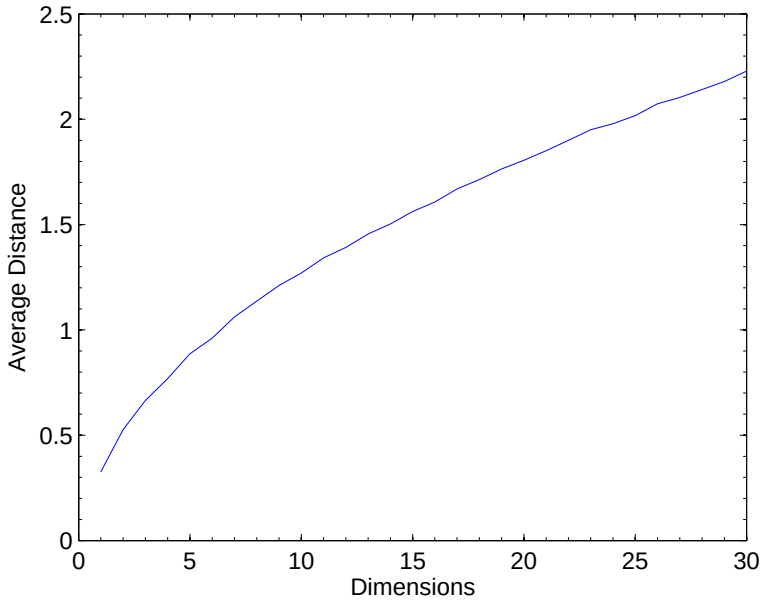


Figure 1.3: Relation of number of dimensions to average distance between points. The number of dimensions, D, is varied and the average distance between any two i.i.d. random points (distributed uniformly) selected from the D-dimensional unit hypercube is computed. The horizontal axis shows increasing number of dimensions. The vertical axis shows the average distance between any two random points. As the number of dimensions increases, so does the average distance between any two points. (Hastie *et al.* 2001)

As an example of data occurring in high dimension, consider text documents. Each word in the corpus' dictionary is counted as a dimension. The value of each dimension for a particular example is the number of times that word occurs in that document. Therefore, it is not uncommon to see examples with hundreds of thousands of dimensions. Or consider data like basic block vectors (BBVs), the program trace data clustered by SimPoint. Each dimension represents a count of the number of times a program basic block executes within an interval of machine instructions. There may be thousands or millions of basic blocks that can occur in each interval (with perhaps millions of instructions per interval), and thus each BBV has to maintain a correspondingly large number of counts, yielding data having incredibly high dimension to cluster.

Data in high dimension poses many interesting obstacles, not just for machine learning applications, but all of mathematics. With an increase in dimension, the distance between objects tends to increases as well. Consider Euclidean (straight line) distance. We see that as the number of dimensions increases, so does the average distance between any two pairs of random points. Figure 1.3 shows the relationship between the average Euclidean distance between any two uniformly i.i.d. random points chosen within the D-dimensional unit hypercube. D is the dimensionality of the data and increases from 1 to 30 dimensions.

Dimensionality also affects the number of examples needed to compute statistics over our data. When computing covariance, we need on the order of $O(D^2)$ examples (relating each dimension to every other) to get accurate estimations (where D is the number of dimensions in the data). When computing joint probabilities, however, we often need a number of examples exponential in the number of dimensions. Thus, if we have many dimensions, we need a lot of data to get a full picture of the relationships between the examples.

Another problem using data in high dimension is dealing with the curse of dimensionality. The curse of dimensionality is a mathematical principle by which the number of examples needed to densely populate a volume of space scales exponentially with the number of dimensions. Consider a D-dimensional hypercube with sides of length l. The volume of the region is

$$V = \prod_{d=1}^{D} l \tag{1.1}$$

$$= l^D. \tag{1.2}$$

Let l be bounded above by L, so we have a sub-cube of side length l within a larger cube with side length L ($0 \leq l \leq L$). Speaking in terms of the fraction of volume the smaller cube encompasses within the larger cube, as we increase the number of dimensions linearly, in order to maintain the same fraction of volume within the larger cube, l must increase multiplicatively. In other words, it takes more and more data

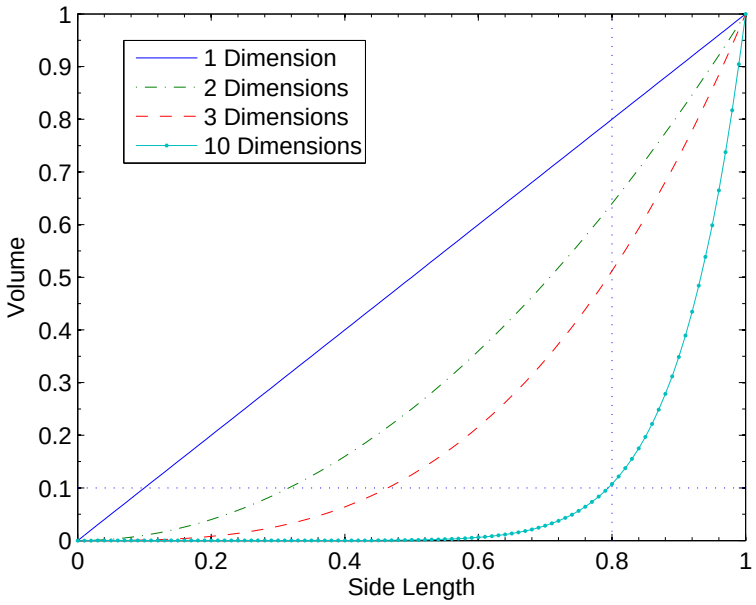


Figure 1.4: Curse of Dimensionality. The horizontal axis denotes the side length l of a hypercube, expressed as some fraction of a maximum length L (side length of an enclosing hypercube). The vertical axis denotes the volume of the smaller hypercube, some fraction of the volume of the larger hypercube. As the number of dimensions increases, the smaller hypercube must become larger and larger to contain the same fraction of the larger hypercube's volume. Note that in 10 dimensions, l must be 80% of the maximum length L for the hypercube to contain 10% of the maximum volume (L^{10}). (Hastie *et al.* 2001)

examples (analogous to l) to express the same amount of information (the volume of the hypercube) in higher dimension. Figure 1.4 expresses this idea (Hastie *et al.* 2001).

Our research with the exponential Dirichlet compound multinomial (EDCM) (Elkan 2006) mixture shows it is well suited to working with the data in its full dimension, avoiding most of the problems mentioned above. Clustering in the full dimension of the data is important for several reasons which will be fleshed out later in the thesis. With a strong algorithm for clustering in the full dimension of the data, we can expand this research into other areas that also deal with data in high dimension. Elkan has already used it to cluster text documents. Other areas of high

dimension clustering include program trace clustering (i.e. SimPoint), speech processing, and genome classification.

The EDCM also accounts for a natural phenomenon known as burstiness. A feature is said to be bursty if, when it occurs, it has a high probability of occurring again in the same example. Words in text documents are bursty—if an author uses a particular word once, especially if that word is content-bearing, it is more likely to appear again. The basic block vectors clustered by SimPoint exhibit bursty behavior. The temporal locality of program execution means that if a basic block is executed within an interval, there is a high chance that it will be executed again soon (for example, the contents of a loop). Burstiness and its importance are described in more detail later.

We consider several different methods of choosing cluster representatives. Representatives are the means by which a cluster can be summarized. They also serve as the simulation points for detailed simulations run on prototype hardware designs. With the EDCM, we can use the probabilities assigned to each data example in the clusters to select a representative (e.g. the example with the highest probability). We can also use basic geometric interpretations, such as Euclidean and Manhattan distance, as well as the vector dot product, to select cluster representatives.

We can select simulation points from EDCM clusters that give prediction accuracies comparable to the k-means clustering algorithm currently used by SimPoint. However, we find the EDCM algorithm is able to achieve comparable accuracies while using half the number of clusters as SimPoint, meaning time saved when the full simulations are run on the simulation points (there is one simulation point per cluster). Additionally, clustering with the EDCM mixture model is theoretically appealing for several reasons. First, it is able to deal with the program trace data in its full dimension, without the need for dimension reduction techniques. Second, the EDCM is able to handle the phenomenon of burstiness, a natural trait of the data which we

hope will help to identify similarities between examples. Third, k-means makes the rigid assumption that the clusters have spherical covariance. In most cases, for the program trace data we are examining, this is not true. The EDCM model avoids this limiting assumption.

1.4 Thesis Organization

The remainder of the thesis is organized as follows. In chapter two, we discuss the background work the thesis is based upon and related approaches at solving the problems. In chapter three, we discuss our methodology, analyzing and providing experimental results for it in chapter four. Chapter five ends with our conclusions and directions for future work.

CHAPTER TWO

Related Work

The application of the research in this thesis focuses on clustering the high dimensional program trace data clustered by the SimPoint application. In this section, we discuss SimPoint and the clustering problem it addresses. We discuss the k-means algorithm, the current clustering solution for SimPoint, and address motivations for seeking a different clustering algorithm. We then take a look at clustering algorithms, other than k-means, that are practical solutions to many real-world problems. We discuss why these algorithms do not work well for clustering high dimensional, bursty data, specifically the basic block vector data clustered by SimPoint

2.1 SimPoint

The SimPoint application was developed by researchers at the University of California, San Diego (Sherwood *et al.* 2002). It is the *de facto* standard for use in industry, including corporations such as Intel and IBM, for speeding up the simulation tests of new chip designs.

2.1.1 Hardware Simulation

When a microprocessor manufacturer is creating a new chip design, there are many factors to consider when choosing the various parameters of the chip specification. For example, one may modify the number of stages in the pipeline, the number of pipelines, the size of the L1 cache, the memory latency, etc. It is infeasible to fabricate the different combinations of these parameters in actual hardware prototypes and test the performance of the chip. Instead, a simulation is setup in software to emulate the hardware. The software simulation provides a flexible, cheap, and ac-

Table 2.1: Number of intervals in program trace data. We count the number of intervals for each of the program trace datasets from the SPEC CPU2000 suite of benchmarks. Each interval is 100 million instructions long. There are a total of 40,676 intervals (and therefore $40.676e3 \times 100e6 = 4.07e12$ instructions) within the suite.

Dataset	Intervals	Dataset	Intervals
ammp-ref	3266	gcc-00-scilab-ref	621
applu-ref	2239	gcc-00-train	52
art-110	418	gzip-graphic-ref	1038
art-470	451	gzip-log-ref	396
bzip2-graphic-ref	1436	gzip-program-ref	1689
bzip2-program-ref	1250	gzip-random-ref	822
bzip2-source-ref	1089	gzip-source-ref	844
crafty-ref	1919	lucas-ref	1424
eon-cook-ref	807	mesa-ref	2817
eon-kajiya-ref	1013	perlbmk-diffmail-ref	400
eon-rushmeier-ref	579	perlbmk-makerand-ref	21
equake-ref	1316	perlbmk-splitmail-ref	1109
facerec-ref	2111	swim-ref	2259
gap-ref	2696	vortex-one-ref	1190
gcc-00-166-ref	470	vortex-three-ref	1331
gcc-00-200-ref	1087	vortex-two-ref	1387
gcc-00-expr-ref	121	vpr-place-ref	35
gcc-00-integrate-ref	132	vpr-route-ref	841

curate method for adjusting and testing the parameters of a prototypical hardware design. Different benchmarks, like the SPEC CPU2000 suite, are then run through the simulator. This allows hardware designers to gauge the performance of a design with flexibility and relatively low cost.

The problem with software simulation, however, is that running one full benchmark can take months of CPU time. Therefore, trying to run a diverse array of benchmarks contained in a suite will take years. This is only for one combination of parameters, so trying to test every combination to find the best will take much longer. For example, within the SPEC CPU2000 suite, there are a total of 36 program trace datasets (benchmark-input pairs) comprised of a total of 4.06 trillion instructions. Table 2.1 gives the number of intervals per dataset. If these datasets are to be

put through a detailed simulation on the SimpleScalar simulator (Burger and Austin 1997), which runs at about 400 million instructions per hour (depending, obviously, on the speed of the computer system it is running on), running the entire suite will take roughly 1.16 years of CPU time for one combination of parameters.

SimPoint addresses this problem by splitting the execution of each benchmark up into intervals of program execution. An interval is a consecutive sequence of machine instructions. For example, some of the program trace data SimPoint examines are intervals of 100 million instructions each, though intervals of any length could work. Researchers have found, however, that an interval length of 10 to 100 million instructions seems to offer a good balance between the number of intervals within a program's execution and the simulation time required to run each interval through the simulator. The intervals are clustered by the k-means algorithm into groups with similar behavior. One interval from each cluster is chosen as a simulation point and run through the simulator to obtain detailed statistics on its performance. So for k simulation points, we obtain k performance estimates that are used in a weighted average to tell us the performance of the benchmark as a whole. The savings are tremendous in terms of CPU time, and the partial simulations on the simulation points are extremely accurate, $< 2\%$ error in predicted performance statistics compared with running a full simulation of the entire benchmark. Instead of over a year, the CPU time required to run these simulations on the simulation points is reduced to less than one week.

2.1.2 Basic Block Vectors

The execution of each benchmark is split into intervals. Within each interval, the number of times each basic block executes is tallied. The final count is multiplied by the number of instructions in that basic block, yielding a vector of weighted counts, a basic block vector (BBV). A basic block is an atomic grouping of instructions that

```
 1:  START  MOVE 11, AX
 2:         MOVE 0, BX
 3:         MOVE 0, CX
 4:   LOOP  DEC AX
 5:         JZ END
 6:         MOVE 1, DX
 7:         AND AX, DX
 8:         JZ EVENS
 9:   ODDS  INC BX
10:         JMP LOOP
11:  EVENS  INC CX
12:         JMP LOOP
13:    END  NOP
```

Figure 2.1: Example pseudo-assembly code snippet showing basic blocks. The initialization steps in lines 1–3 is a basic block. Line 4 starts a new basic block because the program's execution jumps back here each iteration of the loop. Lines 4–5 are the next basic block, decrementing the loop's counter and jumping if the loop should be terminated. Lines 6–8 form a basic block that tests if a variable is even (a conditional), with basic blocks from lines 9–10 and 11–12 making the bodies of the two branches of execution stemming from the conditional. Line 13 is the last and simplest basic block that concludes the program's execution.

execute as a unit during a program's execution. By atomic we mean a section of code with one entry point and one exit point, such as the body of a simple `for` loop or a conditional statement (Sherwood *et al.* 2001).

Figure 2.1 gives a snippet of toy assembly code for a very simple program. It loops over a series of integers (1 through 10) and counts up how many of them are odd and how many are even. We give the code as an example of how we split basic blocks up. Lines 1–3 comprise the first basic block, initializing the parameters. Lines 4–5 are a basic block, entering at line 4 and possibly exiting at line 5. Lines 6–8 are a basic block checking to see if the current integer is even. Lines 9–10 and 11–12 make up two basic blocks, the bodies of the two branches depending on whether or not the integer was even or odd. Finally, line 13 is a basic block that concludes the program.

If there are 100,000 unique basic blocks, then the dimensionality of each BBV is $D = 100,000$, and a BBV for one interval is $x = [x_1, \ldots, x_D]$. Suppose we are working

with intervals of 100 million instructions each. Each interval is not composed of one hundred million basic blocks. Rather, an interval is composed of a certain number of basic blocks which contain a total of one hundred million instructions all together. For a program that executes 52 billion instructions, split into intervals of one hundred million instructions each, we have $N = 52e9/1e8 = 520$ BBVs. If we choose to look at programs with an even finer granularity, say with intervals of 10 million instructions each, N grows by a factor of 10.

Each BBV gives a synopsis of that interval's behavior. Each basic block corresponds to a different action taken by the program at that step in execution. Each BBV, then, gives the overall behavior of the interval from which it is derived. One important note, however, is that the ordering of the execution of the basic blocks is not recorded. All we know is that basic block d executed a certain number of times corresponding to a total of x_d instructions within this interval (Sherwood *et al.* 2001).

With each interval concisely represented as a vector $x = [x_1, \ldots, x_D]$, we can begin to consider how two BBVs might look similar to one another. If we can find a group of BBVs that look similar and bring them together into one cluster, we can simulate just *one* of them and use the gathered statistics to represent the others as well. This problem falls naturally into the realm of unsupervised learning and clustering.

SimPoint's concern is how we go about clustering the BBVs. If we can form accurate clusters, where each BBV in a group is similar, we assume the behaviors (i.e. performance) of all the intervals represented in a cluster are similar. Because a BBV represents a portion of code execution, intervals with similar BBVs execute similar code and should ultimately have similar performance. Even though the ordering of the basic blocks within a vector is not taken into consideration, we assume similar BBVs will have similar statistics within the simulation because they exhibit similar behaviors.

Once our set of BBVs is clustered, the next task is to select a representative from each cluster. It is important that we select an interval that is the most similar to every other member in the group. Or, we may discuss selecting the interval that most similar to the average behavior of the cluster. This fits nicely with our requirement that a selected simulation point will be representative of all its peers in the same cluster.

2.2 k-Means

k-means is the clustering algorithm SimPoint uses. k-means is a partitional algorithm that moves cluster centroids around in geometric space. At the end of the iterative process, the examples are clustered based on which centroid they are closest to.

2.2.1 Algorithm

With k-means (Figure 2.2) there are k cluster centroids $\{\mu_1, \cdots, \mu_k\}$, one centroid per cluster. These k centroids may be initialized in many different ways. One way is simply to select k random points in the D-space spanned by all the different BBVs x_i in our interval set X. Another is to select k random $x_i \in X$. There are different heuristics that may be used in this approach to ensure the initial k are as spread out throughout the data space (Hastie *et al.* 2001).

Once each center μ_j $(1 \leq j \leq k)$ is initialized, each data point $x_i \in X$ is assigned to the cluster c_j whose center μ_j it is closest to, according to the L_2 (Euclidean) distance.

$$L_2(a, b) = ||a - b|| = \sqrt{\sum_{d=1}^{D} (a_d - b_d)^2}, \tag{2.1}$$

where $a = [a_1, \cdots, a_D]$ and $b = [b_1, \cdots, b_D]$. We define cluster c_j as

$$c_j = \left\{ x_i : j = \arg\min_l ||x_i - \ \mu_l|| \right\}, \tag{2.2}$$

Pre-Conditions: $k > 0$

1: Initialize k cluster centers $\{\mu_1, \cdots, \mu_k\}$.
2: **repeat**
3: **for all** $x_i \in X$ **do**
4: Find $c_{(i)}$ using Equation 2.3 (can also be thought of as determining the sets of each cluster c_j via Equation 2.2).
5: **end for**
6: **for** j from 1 to k **do**
7: Recompute μ_j using Equation 2.4.
8: **end for**
9: **until** The centers μ_j do not change

Figure 2.2. k-means Clustering Algorithm

the set of all examples x_i that are closer to centroid μ_j than any other centroid. The assigned cluster for point x_i, $c_{(i)}$, is given as

$$c_{(i)} = \arg\min_j ||x_i - \mu_j||, \tag{2.3}$$

the centroid to which x_i is closest. Once each point x_i is assigned to some cluster, the cluster centers μ_j are recomputed as the mean of all the $x_i \in c_j$.

$$\mu_j = \frac{1}{|c_j|} \sum_{x_i \in c_j} x_i \tag{2.4}$$

The new cluster centers are then used to reassign each x_i to the closest center, and the process repeats until the centers do not change.

2.2.2 Shortcomings

Unfortunately, k-means does not meet all of our needs and has several limiting characteristics. First, because k-means uses distance metrics to determine cluster assignments, it does not work well in high dimension. This is because all examples tend to be far apart in higher dimension, making it hard to determine which centroid really is the closest. This is especially evident when considering data with the high number of dimensions of the SimPoint BBVs (upward of 100,000 dimensions). In order to get around the problem of high dimension, the current SimPoint approach

is to use random projections. While this method is well motivated and sound, it introduces several undesirable properties into the clustering. The first is that random projections may collapse true clusters together into one new cluster. The second is that the original features of the data are lost in the projection. If we want to examine the data's original structures, we can do so, but it requires an extra step of mapping clusters back to the original space. Also, the features are lost to the clustering algorithm, which might have helped produce better clusterings. Third, the randomness of the projections adds another level of indirection to our algorithm and makes replicating the exact results of an experiment potentially that much more difficult, yielding different results each time the algorithm is run on the same dataset.

k-means, in the way that it handles cluster assignments by using simple distance from the centroids μ_j, makes an underlying assumption that the covariance of each cluster is spherical. This is very rarely the case, and certainly does *not* hold true for program trace data. Thus, there must be a better approach to representing such clusters.

SimPoint and the k-means algorithm also do not currently account for burstiness occurring within the data. As described earlier, burstiness is an important phenomenon that occurs extensively in program execution (due to temporal locality). It would be useful to have a method for exploiting this characteristic.

2.3 Other Clustering Algorithms

There are many different clustering algorithms, several of which have been applied either specifically to the program trace data SimPoint clusters, or to trying to address some of the shortcomings of the k-means algorithm. We discuss two algorithms briefly here, the PG-Means algorithm and multinomial mixtures.

2.3.1 Gaussian Mixtures

Other methods have been attempted to resolve some of the issues that k-means raises. Feng and Hamerly used mixtures of Gaussians but did not see a significant improvement in clustering results. Their PG-means (projected Gaussian) algorithm fits k Gaussian clusters to the data and is able to find clusters which overlap and exist in high dimension. Projections are used to simplify the clustering, so this method does not avoid the need for such simplification of the data beforehand. However, there is not an assumption that the clusters have spherical covariance, although the data still is assumed to be Gaussian. (Feng and Hamerly 2006)

2.3.2 Multinomial Mixtures

Sanghai *et al.* (2005) proposed a method for clustering program trace data using multinomial mixtures. They projected the data to fewer dimensions and did not use the full dimension of the data. However, this method was shown to not improve the basic k-means algorithm. The multinomial approach does not do well in high dimension and does not improve prediction error rates (Hamerly *et al.* 2006).

CHAPTER THREE

Design and Implementation

In this section, we describe the research employed to discover new clustering models that address some of the shortcomings of the current methods of clustering program trace data in SimPoint. As with any clustering algorithm, especially a new one, we examine the basic questions that get asked, such as how we can choose the number of clusters to be used in the model without *a priori* knowledge and how we can select representative examples from each cluster.

To clarify, in much of the discussion and explanation in this chapter, we assume a generative view of clustering for explanatory purposes (as opposed to a discriminative view). The generative idea is that given a learned model, such as a multinomial, we could generate a set of examples if we wished. This is not actually happening. We are not actually using a model to generate data. In truth what we are doing is learning a model that has a high likelihood of having generated the data. For instance, we could consider a large set of basic blocks (our model), reaching into the bag multiple times and generating a program trace in such a manner. This is the generative view. What we are actually doing is using existing program traces to figure out what the bag of basic blocks would look like, if one were to exist.

Note also that one popular domain of discussion for clustering algorithms, as well as many of the probability distributions that they operate on, is text document clustering. Because our research is not concerned with clustering text documents, we will modify any discussion of clustering algorithms and other related topics to deal with program traces and the basic block vectors clustered by SimPoint. The arguments of text document clustering apply to program trace data. Both domains contain high dimensional data, tend to be rather sparse in their count-vector repre-

sentations (only a rather small subset of the total set of all basic blocks occurs within an interval, like a small subset of the entire vocabulary appears in a document), and exhibit burstiness.

3.1 Clustering With EDCM Mixtures

We investigate a new clustering algorithm to cluster program trace data. We are searching for an alternative to the k-means algorithm currently used by SimPoint. One reason we would like another algorithm is we want a method that can operate in the full dimension of the data. Most contemporary clustering algorithms have difficulties with data in high dimension, such as the program trace data clustered by SimPoint. Dimension reduction techniques used to bring the dimensionality of the data down to an acceptable amount, such as random projections, can mask natural features of the data and collapse true clusters together. Another reason is that we want to avoid the limiting assumptions k-means makes about the data, namely that each of the clusters has a spherical covariance. Lastly, we want to account for the phenomenon of burstiness. We find that SimPoint data, like text documents, exhibits burstiness and would like a way to harness this characteristic of the data to our advantage.

3.1.1 Statistical Background

The exponential Dirichlet compound multinomial (EDCM) model provides a robust framework for clustering data in high dimension. It is an exponential-family approximation of the Dirichlet compound multinomial (DCM), which in turn is a variant of the multinomial model. This mathematical foundation yields several useful traits that we can exploit to our advantage.

3.1.1.1 *Multinomial distribution.* The multinomial is a probability distribution that provides a simple and effective framework for representing data in high

dimension. Consider a set of D random variables within a vector x. Each variable x_d $(1 \leq d \leq D)$ represents the number of times event d occurs over n observations $(\sum_d x_d = n)$. The different events are mutually independent, meaning the occurrence of one has no effect on the occurrence of another, even another occurrence of the same event. Each event has a probability θ_d of occurring $(\theta_d > 0, \sum_d \theta_d = 1)$. The probability of all the events occurring with the counts given in x, given the probabilities θ, is

$$\Pr(x|\theta) = \frac{n!}{\prod_{d=1}^{D} x_d!} \prod_{d=1}^{D} \theta_d^{x_d} \tag{3.1}$$

$$= \frac{\Gamma(n+1)}{\prod_{d=1}^{D} \Gamma(x_d+1)} \prod_{d=1}^{D} \theta_d^{x_d} \tag{3.2}$$

(Johnson *et al.* 1997).

Equations 3.1 and 3.2 are equivalent to each other, using the identity $\Gamma(\omega+1) = \omega!$ (for $\omega \in \mathbb{Z}^+$). It is more usual to see the factorial ($n!$) notation, rather than the Γ notation, since the multinomial is a discrete distribution for handling different counts of events, which are by definition integral and non-negative (an event cannot occur a fractional number a times).

The multinomial coefficient is similar to the binomial coefficient and is, in fact, just a generalization of it. The binomial coefficient gives the number of possible combinations of choosing k out of n possible items (ordering not considered). Read "n choose k," the binomial coefficient is

$$C(n,k) = \binom{n}{k} = \frac{n!}{k!(n-k)!} \qquad 0 \leq k \leq n \tag{3.3}$$

(Johnson *et al.* 2005). $n!$ by itself gives the number of possible permutations of n items, which are all the possible orderings of n items. We divide by $k!$ and $(n-k)!$, the sizes of the two groups the items are divided into. Each of these values represents the number of different ways to order the items in the respective groups. Dividing the total number of orderings by the number of orderings within the groups eliminates

the ordering consideration from the coefficient. The multinomial coefficient is

$$\binom{n}{k_1, \cdots, k_D} = \frac{n!}{k_1! \times \cdots \times k_D!} = \frac{n!}{\prod_{d=1}^{D} k_d!}, \tag{3.4}$$

with $\sum_{d=1}^{D} k_d = n; n, k_d \geq 0$. It is easy to see the multinomial is simply a generalization of the binomial, where instead of "n choose k," giving one group of k items and one group of $(n-k)$ items, we have n total items and D groups of $k_1, \cdots, k_D$ items each. The multinomial simplifies to a binomial when $D = 2$, with $k_1 = k$ and $k_2 = n - k$ (Johnson *et al.* 1997).

With a multinomial distribution, the ordering of the n events is not taken into consideration. Each count x_d only considers the number of times event d occurs, not when it occurs in relation to others. Because each dimension is considered independent of the others, we find the multinomial to be well suited to representing data in high dimension. We don't need an exponential number of examples to learn the relationships between all the dimensions because we are not concerned with such relationships, only the individual magnitudes, considered to be mutually independent of one another within a cluster.

One of the problems with the multinomial is that it is not well suited to expressing data that has traits that are "bursty" in nature. Burstiness is described in more detail later, but as a short example consider the domain of program trace clustering with SimPoint. Within this domain, we have a set of D basic blocks. We represent each interval of program execution x as a basic block vector, counting up the number of times each basic block d $(1 \leq d \leq D)$ occurs and storing this value (multiplied by the number of instructions in the basic block) in x_d. The probability of basic block d occurring in our interval is θ_d. Suppose we are analyzing some interval containing basic block d, which is perhaps the body of an `if` statement that takes the square root of a number. Because the programmer who wrote the code we are analyzing has already used the basic block, temporal locality tells us it is likely to appear again in the interval, even if $\theta_d \ll 1$ (in regards to the entire set of intervals,

taking the square root of a value may be rare, even if it occurs many times within this specific interval). However, as the number of times the basic block occurs within the interval increases, the term $\theta_d^{x_d}$ (from Equation 3.1) becomes very small, reducing the overall probability of the interval. We would like for a basic block, once it occurs, to be expected to appear more often so that we can use basic block occurrences as content (and therefore behavioral) identifiers. Instead, the multinomial penalizes the probabilities of the intervals containing bursty basic blocks.

The multinomial makes the naïve Bayes independence assumption about the distribution of the probabilities of each event in an example. This is the assumption that all the basic blocks in an interval are drawn independently of one another. While this simplifies the model and allows the multinomial to escape the curse of dimensionality, it is not well suited to representing program trace data (as described in the previous paragraph). The reality is that basic are indeed related to each other. It is especially the case that for content carrying basic blocks, certain groups of basic blocks tend to occur together, and that when a basic block is seen once it is very likely to be seen again. The most common basic blocks (perhaps something like a decrement followed by a jump if zero) can be accurately described under the naïve Bayes assumption, but do not help to identify content and interval similarity.

3.1.1.2 *Dirichlet distribution.* The Dirichlet distribution is a distribution over the parameters of another probability distribution (in our case, a distribution of the parameters θ of a multinomial), defined as follows. The joint probability of parameters $\theta_1, \cdots, \theta_D$, with $\theta_d \geq 0$ $(1 \leq d \leq D)$ and $\sum_{d=1}^{D} \theta_d \leq 1$, is

$$\Pr(\theta_1, \cdots, \theta_D | \alpha) = \frac{\Gamma\left(\sum_{d=0}^{D} \alpha_d\right)}{\prod_{d=0}^{D} \Gamma(\alpha_d)} \prod_{d=1}^{D} \theta_d^{\alpha_d - 1} \left(1 - \sum_{d=1}^{D} \theta_d\right)^{\alpha_0 - 1} \tag{3.5}$$

(Kotz *et al.* 2000). The parameters of the Dirichlet distribution are $\alpha_0, \cdots, \alpha_D$. Parameters $\alpha_1, \cdots, \alpha_D$ are the parameters for each $\theta_1, \cdots, \theta_D$. α_0 is a parameter

assigned to handle the portion $1 - \sum_{d=1}^{D} \theta_d$, which is treated like another example and is dependent on the sum of the observed data.

The Dirichlet distribution is a generalization of the beta distribution, much like the multinomial is a generalization of the binomial distribution. For parameters $\alpha, \beta > 0$ and $0 < \theta < 1$, the beta distribution has probability density function

$$\Pr(\theta|\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1} \tag{3.6}$$

(Balakrishnan and Nevzorov 2003). The connection between Equations 3.5 and 3.6 is apparent when we let α_0 and α_1 from Equation 3.5 be β and α, respectively, from Equation 3.6, and $D = 1$ with $\theta = \theta_1$ to convert the input of Equation 3.5 to that of Equation 3.6.

The last term in Equation 3.5 is not usually present in the definition of the Dirichlet distribution given in most texts. This is an artifact of the way the Dirichlet is derived and one can see it is not necessary. One of the assumptions about the Dirichlet is that $\sum_{d=1}^{D} \theta_d \leq 1$. If $\sum_{d=1}^{D} \theta_d < 1$, there will be some remaining portion to get the examples to sum to 1 $(1 - \sum_{d=1}^{D} \theta_d)$. The last term in Equation 3.5 is just assigning a parameter, α_0, to handle this portion, treating it like another example that is dependent on the sum of the observed data. Thus we could define the Dirichlet as

$$\Pr(\theta_1, \cdots, \theta_D|\alpha) = \frac{\Gamma\left(\sum_{d=0}^{D} \alpha_d\right)}{\prod_{d=0}^{D} \Gamma(\alpha_d)} \prod_{d=1}^{D} \theta_d^{\alpha_d - 1} \tag{3.7}$$

for $\theta_d > 0$ $(1 \leq d \leq D)$ and $\sum_{d=1}^{D} \theta_d = 1$, as it is defined by Minka (2003). The abuse of notation between Equations 3.5 and 3.7, namely reusing the variables θ_d when they have different definitions, is employed to illustrate the similarities and relationship between the two equations.

The Dirichlet distribution is very similar in form to the multinomial distribution (Equation 3.1). The multinomial coefficient's terms (Equation 3.4) are concerned strictly with the data, and the probabilities θ are raised to the values of the data x.

As we've seen, this means that increasing counts in x reduces the overall probability $\Pr(x|\theta)$. The Dirichlet, on the other hand, is often referred to as a distribution over parameters θ. The normalizing coefficient is the analogue of the multinomial coefficient, but using the parameters α_d rather than the data x_d. Remember, $n = \sum_{d=1}^{D} x_d$, like $\sum_{d=1}^{D} \alpha_d$, and $\Gamma(\omega + 1) = \omega!$ for $\omega \in \mathbb{Z}^+$. The major difference is in the term $\prod_{d=1}^{D} \theta_d^{\alpha_d - 1}$. Here, the example is raised to the parameter, the opposite of the multinomial (which is θ^x). With increasing values of θ_d, the probability of the example in the Dirichlet, $\Pr(\theta|\alpha)$, increases. Thus, we see a framework that allows us to more naturally express burstiness.

3.1.1.3 *Multivariate Pólya-Eggenberger distribution.* The multivariate Pólya-Eggenberger distribution is often used as the basis to describe other multivariate distributions. It is based on the Pólya urn scheme. Consider an urn filled with balls of two different colors, say w white balls and b black balls. Each round, we draw one ball from the urn and record its color. We then replace the ball and add c balls of the same color. The Pólya-Eggenberger distribution describes the probability that x white balls are drawn during n rounds (Johnson *et al.* 2005). The multivariate Pólya-Eggenberger distribution, then, considers an urn filled with balls of D different colors (Johnson *et al.* 1997).

By replacing the ball of color d back into our urn and adding another c instances of color d to the urn, we make it more likely that a ball of color d will be drawn the next round. For example, say we start with $D = 2$ and $c = 1$, with $d_1 =$ white and $d_2 =$ black. The first round, say we happen to draw a black ball from the urn. At the beginning of the second round, the urn will contain one white ball and two black balls. We now have bursty behavior because a black ball has a higher chance $\left(\frac{2}{3}\right)$ of being drawn than the white ball $\left(\frac{1}{3}\right)$. Once a ball of a certain color is seen, it is more likely for that color to appear again in later trials (Elkan 2006).

3.1.1.4 *Dirichlet compound multinomial distribution.* The Dirichlet compound multinomial (DCM) was used by Madsen, Kauchak, and Elkan to cluster text documents, specifically for its ability to handle word burstiness (2005). The DCM uses the Dirichlet distribution to assign probability mass to the parameters of a multinomial distribution, θ, which can then be used to assign a probability to the data, x (Johnson *et al.* 1997).

The DCM is also called the multivariate Pólya-Eggenberger (or just multivariate Pólya, for short) distribution, the same as described above, because the final counts of the colors drawn in n trials follow a DCM distribution. We let α_d, the parameters of the DCM, be the initial count of the number of balls of color d in the urn (not necessarily integral or summing to 1) (Elkan 2006).

The DCM is a distribution over probability distributions. Namely, the DCM uses a Dirichlet distribution to generate multinomials. Madsen *et al.* (2005) use the analogy that a multinomial is a "bag of words," or, in our case, a bag of basic blocks. The bag is filled with basic blocks such that each basic block d has probability θ_d of being drawn. Taking a generative view, we can think of drawing n basic blocks to generate some interval x. With the DCM, we have a bag of bags of basic blocks. From the Dirichlet distribution (bag of bags), we draw a multinomial (bag of basic blocks), and from that multinomial we draw the basic blocks for an interval. The DCM allows us to handle burstiness for different examples by creating different multinomial distributions. For the cases where a particular interval x has a burst of a certain basic block d, the θ_d for that multinomial for that particular interval is higher than in any other set of multinomial parameters (which would be used to generate different intervals). Rather than thinking of individual multinomials for every interval, however, we usually think of the DCM as generating multinomials for clusters of intervals. Thus, all the intervals in a particular cluster are assumed to share many common basic blocks between them that may not be present in any of the other clusters. Additionally,

we have seen how the Dirichlet distribution itself encourages, rather than penalizes, bursty traits in the data.

The probability that the DCM, with parameters α, will generate a multinomial, with parameters θ, is

$$\Pr(\theta|\alpha) = \frac{\Gamma\left(\sum_{d=1}^{D} \alpha_d\right)}{\prod_{d=1}^{D} \Gamma(\alpha_d)} \prod_{d=1}^{D} \theta_d^{\alpha_d - 1}. \tag{3.8}$$

Notice Equation 3.8 is just the Dirichlet distribution (Equation 3.7) However, we're not concerned with the multinomials that are drawn from the DCM, but with the basic block that are generated from the multinomials. Madsen *et al.* (2005) give the probability for an example x given a DCM with parameters α as

$$\Pr(x|\alpha) = \int_{\theta} \Pr(x|\theta) \Pr(\theta|\alpha) d\theta \tag{3.9}$$

$$= \frac{n!}{\prod_{d=1}^{D} x_d!} \frac{\Gamma\left(\sum_{d=1}^{D} \alpha_d\right)}{\Gamma\left[\sum_{d=1}^{D} (x_d + \alpha_d)\right]} \prod_{d=1}^{D} \frac{\Gamma(x_d + \alpha_d)}{\Gamma(\alpha_d)}. \tag{3.10}$$

Thus, in practice, we never actually talk about the multinomial parameters θ when using the DCM. Minka (2003) gives solutions for computing the likelihood of the DCM and for estimating the parameters α given a set of training examples.

One point to make is that because the parameters of the Dirichlet distribution, α, are not required to sum to 1, we have an extra free parameter when compared to the multinomial, whose parameters θ are required to sum to 1. For D dimensions/features, the multinomial has $D-1$ free parameters because the D^{th} parameter can be computed as $1 - \sum_{d=1}^{D} \theta_d$. Because we do not know what $\sum_{d=1}^{D} \alpha_d$ should equal, we cannot assume a value for the D^{th} parameter of the Dirichlet distribution. This gives the DCM (and by virtue the EDCM), because of its basis on the Dirichlet distribution, one extra free parameter over the multinomial, giving DCM models a bit more flexibility than multinomial models in their ability to fit the data.

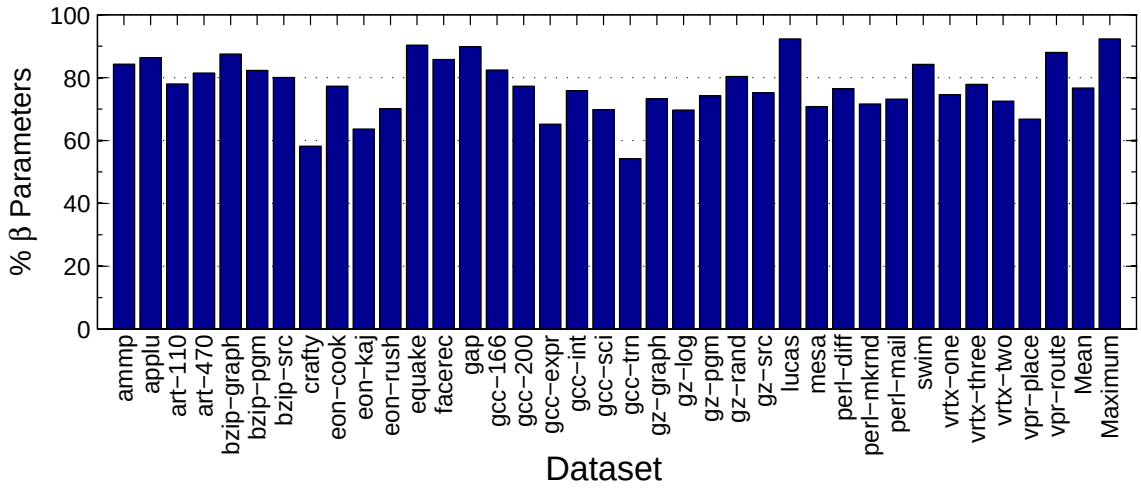


Figure 3.1: Percentage of small β for program trace data. To compute the percentages, we counted up, per dataset, the percentage of all β parameters (after the data had been clustered using the EDCM mixture model) less than 0.01. It is assumed, for the sake of argument, that $0.01 \ll 1$. On average, nearly 80% of all values β_d are < 0.01.

3.1.2 Exponential Dirichlet Compound Multinomial

The exponential Dirichlet compound multinomial (EDCM) is an exponential-family approximation of the DCM. It was derived and first used to cluster text documents by Elkan (2006).

We can approximate the DCM as follows. First, note that since we are dealing with sparse data (word count vectors or SimPoint data), it is not necessary to consider any x_d that are zero. In Equation 3.10, we can ignore any $x_d = 0$ because $x_d!$ and $\frac{\Gamma(x_d+\alpha_d)}{\Gamma(\alpha_d)}$ are both equal to 1, and do not affect the products they appear in, when this is the case. Second, Elkan points out that for domains such as text document clustering, the parameters α_d tend to be very small. A quick examination of the EDCM mixture model parameters used to cluster program trace data for SimPoint shows that its parameters, β_d, also tend to be very small. Figure 3.1 shows that, on average, nearly 80% of all parameters β_d are < 0.01 and $\ll 1$. The identities

$$\lim_{\alpha \to 0} \frac{\Gamma(x+\alpha)}{\Gamma(\alpha)} - \Gamma(x)\alpha = 0 \qquad (x \geq 1) \tag{3.11}$$

and

$$\Gamma(x) = (x-1)! \qquad (x \in \mathbb{Z}^+) \tag{3.12}$$

(the counts of basic blocks in SimPoint data are positive integers) allow us to approximate Equation 3.10, giving the EDCM as

$$\Pr(x|\beta) = \frac{n!}{\prod_{d:x_d \geq 1} x_d} \frac{\Gamma(s)}{\Gamma(s+n)} \prod_{d:x_d \geq 1} \beta_d. \tag{3.13}$$

We use parameters β for the EDCM to distinguish it from the DCM, which uses parameters α. Here, $n = \sum_{d=1}^{D} x_d$ and $s = \sum_{d=1}^{D} \beta_d$. Since the EDCM is an approximation of the DCM and not an exact distribution, it is much more efficient to compute than the DCM.

Because of its use of the multinomial model (via the DCM being a distribution over multinomials), the EDCM is well suited to handling data in high dimension. Each dimension is treated independently under the naïve Bayes assumption—no joint probabilities are computed to say which basic blocks occur together in an interval or to reflect a particular ordering of the basic blocks within an interval. The naïve Bayes assumption does not hinder our performance here, as it does with a more simple multinomial model, because the EDCM deals with probabilities of parameters θ rather than working directly with the data. The multinomial had to deal with data, where the naï ve Bayes assumption does not, in truth, hold. With the EDCM, the assumption is made over the parameters of the multinomial θ. Because we can cluster in the full dimension of the data we see several direct benefits, such as avoiding the dependence on random projections. Random linear projections may mask natural identifying features within the data, making it harder to discern which examples exhibit similarities. Worse, random projections may collapse true clusters that are well separated in high dimension into an amalgam that is not easily separable in low dimension. Also, the randomness of the projections introduces another level of uncertainty into our algorithm, meaning we must save even more data if we are to reproduce exact results.

The EDCM is in the exponential family of distributions. The binomial, multinomial, and Dirichlet distributions are all members of the exponential family—the DCM is not. Therefore, the EDCM is superior to the DCM on this basis. Given some data x and some parameters θ, a probability distribution is in the exponential family if it can be expressed as

$$\Pr(x|\theta) = f(x)g(\theta)\exp\left(t(x)\cdot h(\theta)\right) \tag{3.14}$$

(Elkan 2006). f and t are real-valued functions that depend only on the data, not on the parameters, with $f(x) \geq 0$. g and h are real-valued functions that depend only on the parameters, and not the data, with $g(\theta) \geq 0$. The notation $t(x)\cdot h(\theta)$ denotes the inner product, which could also have been written

$$t(x)\cdot h(\theta) = \sum_{d=1}^{D} t_d(x)h_d(\theta)$$

for D-vectors x and θ (Casella and Berger 2002). Elkan rearranges Equation 3.13 to show its conformance to Equation 3.14, and thus its inclusion in the family of exponential distributions, as

$$\Pr(x|\beta) = \left(\prod_{d:x_d\geq 1}\frac{1}{x_d}\right)\left(n!\frac{\Gamma(s)}{\Gamma(s+n)}\right)\exp\left[\sum_{d=1}^{D} I(x_d \geq 1)\log\beta_d\right], \tag{3.15}$$

with

$$\begin{aligned} f(x) &= \prod_{d:x_d\geq 1}\frac{1}{x_d} \\ g(\theta) &= n!\frac{\Gamma(s)}{\Gamma(s+n)} \\ h(\theta) &= \langle \log\beta_1, \cdots, \log\beta_D\rangle \\ t(x) &= \langle I(x_1 \geq 1), \cdots, I(x_D \geq 1)\rangle \end{aligned}$$

Thus, the sufficient statistics for the EDCM is a vector of 1 or 0 indicating which of the elements d in the vector are non-zero (I is the identity function and returns 0 if the parameter is false, 1 if it is true) (Elkan 2006). Distributions in the exponential

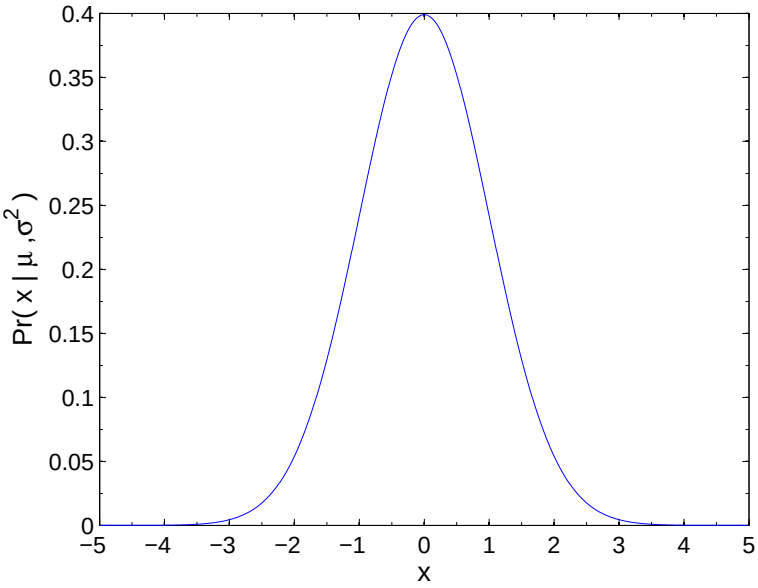


Figure 3.2: Gaussian distribution. We say that x is distributed normally, $x \sim N(\mu, \sigma^2)$, if its probability density function (pdf) is the Gaussian with parameters μ, the mean of the distribution, and σ^2, the variance of the distribution. The above figure shows the pdf of a Gaussian with $\mu = 0$ and $\sigma^2 = 1$.

family have several nice properties, of which the sufficiency principle is possibly the most significant (Casella and Berger 2002).

The sufficiency principle states that given a set of data x and a set of parameters θ, everything that we might want to infer about θ we can do so using a sufficient statistic $t(x)$ (the same $t(x)$ in Equation 3.14). A sufficient statistic gives all the information there is to know about a set of data. Given a sufficient statistic, we can determine properties of the data without having to depend on the parameters θ, even if the original sample x is not known to us. Any information about the data x, besides $t(x)$, does not yield any more information other than what is given by the sufficient statistic. For example, if we have two samples x_1 and x_2, and it is the case that $t(x_1) = t(x_2)$, then any inferences made about θ will yield the same value whether we actually observe x_1 or x_2 (Casella and Berger 2002).

As an example, consider the Gaussian distribution (standard normal distribution $N(\mu, \sigma^2)$) for the simplicity of its parameters μ, the mean of the distribution,

and σ^2, the variance of the distribution. Figure 3.2 gives a visual representation of a univariate Gaussian (a distribution over one variable). A random variable X is distributed normally, $X \sim N(\mu, \sigma^2)$, if, for observed values $x_i \in \mathbb{R}, 1 \leq i \leq n$,

$$\Pr(X|\theta) = \Pr(X|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(\sum_{i=1}^{n} x_i - \mu)^2}{2\sigma^2}\right) \tag{3.16}$$

(Casella and Berger 2002). The parameters θ (a vector of 2 parameters) are $\theta_1 = \mu$ and $\theta_2 = \sigma^2$. Elkan (2005) shows the Gaussian distribution to be in the exponential family via the rearrangement

$$\Pr(X|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{n\mu^2}{2\sigma^2}\right) \exp\left(-\frac{\sum_{i=1}^{n} x_i^2}{2\sigma^2} + \frac{\mu \sum_{i=1}^{n} x_i}{\sigma^2}\right). \tag{3.17}$$

$t(x)$ and $h(\theta)$ in Equation 3.14 are allowed to be vectors. Assigning the terms of Equation 3.17 to Equation 3.14, we have

$$\begin{aligned}
f(x) &= 1 \\
g(\theta) &= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{n\mu^2}{2\sigma^2}\right) \\
t_1(x) &= \sum_{i=1}^{n} x_i^2 \\
t_2(x) &= \sum_{i=1}^{n} x_i \\
h_1(\theta) &= -\frac{1}{2\sigma^2} \\
h_2(\theta) &= \frac{\mu}{\sigma^2}.
\end{aligned}$$

Thus, the sufficient statistics for the Gaussian distribution are $\sum_{i=1}^{n} x_i$ and $\sum_{i=1}^{n} x_i^2$. Note that the number of samples, n, is not dependent on the data itself. We can see that the parameters for the Gaussian, μ and σ^2, can be expressed in terms of the sufficient statistics. First,

$$\begin{aligned}
\mu &= \frac{1}{n} \sum_{i=1}^{n} x_d \\
&= \frac{1}{n} t_2(x).
\end{aligned}$$

We can express the variance of the Gaussian as

$$\begin{aligned}
\sigma^2 &= \sum_{i=1}^{n}(x_i - \mu)^2 \\
&= \sum_{i=1}^{n}(x_i^2 - 2x_i\mu + \mu^2) \\
&= \sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} 2x_i\mu + \sum_{i=1}^{n} \mu^2 \\
&= t_1(x) - 2\mu t_2(x) + n\mu^2.
\end{aligned}$$

Thus, the sufficient statistics t_1 and t_2 are all that we need to compute the parameters of the Gaussian. We don't need all of the data after we have computed the sufficient statistics on a sample, and no matter how much data we might sample, we can infer no more about θ than we can with the two sums $\sum_{i=1}^{n} x_i$ and $\sum_{i=1}^{n} x_i^2$.

3.1.3 Burstiness

As stated previously, burstiness is a trait of data wherein if a particular feature appears once, it is more likely to appear again. This trait occurs naturally in many domains, including the basic block vectors clustered by SimPoint. Church and Gale (1995) (along with Salva Katz) were among the first to quantify and study the burstiness phenomenon. They noticed that the typical bag of words representation for text documents failed to capture all the behaviors of the different words, noting a "bunching-up effect" (i.e. burstiness) of certain words not only within documents, but within genres (topics), paragraphs, and even sentences within a document. Katz (1996) makes several distinctions between different types of burstiness, each relating to the occurrences of content words that express the "ideas, feelings, facts, and events" of language. Hooked together by function words, the content words express the topic of a document and convey meaning and often occur in document-level bursts (where a word occurs many times over the course of a document) and within-document bursts (where a word will occur many times within a sentence, paragraph, or otherwise close

together). Church and Gale (1995) and Katz (1996) all discuss burstiness and handling it with Poisson distributions. We use a different approach with the DCM, but the underlying structure of burstiness is the same.

We have given the discussion of burstiness within the text setting. Now, consider burstiness within the execution of a benchmark that gets analyzed to form a set of BBVs for SimPoint. Programs undergo different phases of behavior. For example, the first phase of a program may be to read in some certain input. The next may be to perform various calculations or other tasks based on the input. The third may be to report any results and perform any cleanup. Within each of these phases, the phenomenon of temporal locality (when an instruction is executed, it is likely to be executed again soon—the reason caches work so well) suggests that there will probably be some repeated behaviors. For example, within the second phase of the above behaviors, we may have a `for` loop that iterates over all the data. Obviously, the instructions executed within this loop will get repeated many times in relatively quick succession. When we split the benchmark's execution up into fixed length intervals, we may find that the phase containing the loop gets split into several intervals. However, each interval, because it comes from the section of code containing the `for` loop, will have bursts of the occurrences of the basic blocks within the loop.

The multinomial penalizes burstiness, as we have already discussed, where one trait has a high number of occurrences. The EDCM, because of the definition of the Dirichlet distribution, is able to handle burstiness (like the Pólya urn scheme). Elkan (2006) expresses the DCM (and by virtue, the EDCM) in terms of the Pólya urn scheme. Let the parameters α_d be the initial count of the number of balls of color d within the urn. Unlike the counts of balls within an urn, however, we can have $\alpha_d \notin \mathbb{Z}$ and $\alpha_d < 1$ (whereas ball counts must necessarily be positive integers). Elkan makes the remark that the sum $s = \sum_{d=1}^{D} \alpha_d$ can be used to assess the total burstiness of the multivariate Pólya-Eggenberger distribution. Lower values of s mean higher levels of

burstiness, because drawing one ball increases its count relatively quickly to balls of another color. Conversely, higher values for s mean a lower amount of burstiness in the system because even if a particular color is drawn multiple times, its count may not increase significantly compared to other colors. As $s \to \infty$, the DCM (and by virtue, the EDCM) begins to behave like a multinomial, where every ball drawn is independent of all the others, and drawing a particular color does not make it more likely to be drawn again (because there are so many balls of each color).

The multinomial performs well when the event occurrences are uniform. It is the "rare" words, as quantified by Madsen *et al.* (2005), that the multinomial fails to model correctly. This set of words carries the content and tends to identify a document's topic, making them important markers for clustering . Therefore, it is essential that we model them correctly and not penalize bursty occurrences. Common words, while tending to occur with different frequencies in different corpora, do not exhibit burstiness (Church and Gale 1995).

3.1.4 Expectation-Maximization Algorithm

The Expectation-Maximization algorithm was formulated by (Dempster *et al.* 1977) in order to learn a model in the face of partial or incomplete data. We can use it to iteratively maximize the likelihood of a model, even when we have missing values or censored or truncated data. In the case of unsupervised learning and clustering, the missing data are which mixture component generated each example.

The EM algorithm (Figure 3.3) operates in two basic steps, giving the algorithm its name. The EM algorithm computes some parameters θ describing a probability distribution mixture (of which we know or assume a general structure, perhaps a mixture of EDCM distributions) using data x sampled from the distributions. The unknown parameters θ are both the parameters of the distributions within the mixture (such as the β parameters for the EDCM distributions) and the prior probabilities

Pre-Conditions: Data x
Post-Conditions: Parameters θ have a local maximum likelihood $\Pr(x|\theta)$

1: Initialize parameters θ
2: **repeat**
3: *Estimation (E) Step*: Using the current parameters θ and the observed data x, estimate the expected log likelihood of the complete data (z) given a revised hypothesis θ' to give us $Q(\theta'|\theta)$. In effect this computes the expected values of the latent, unobserved variables (e.g. the likelihood of each point being generated by each distribution in the mixture).

$$Q(\theta'|\theta) \leftarrow E[\ln \Pr(y|\theta')|\theta, x]$$

4: *Maximization (M) Step*: Update the set of parameters by maximizing the likelihood of the model assuming the latent values expected in the E step.

$$\theta \leftarrow \arg\max_{\theta'} Q(\theta'|\theta)$$

5: **until** Convergence on θ

Figure 3.3. Expectation-Maximization algorithm

of each distribution ($\Pr(c_j)$ for $1 \leq j \leq k$, where there are k distributions in the mixture). Our incomplete data is x, since we are dealing with multiple distributions within the mixture and don't know from which particular distribution each individual x_i is drawn. Let z be our missing data (latent variables telling us from which distribution each example was drawn) and let $y = x \cup z$ be the full set of data generated by the distributions we are computing θ for.

As an example, consider the k-means algorithm (given in 2.2), which is a specific instance of the EM algorithm. Let data we observe, x, be a set of two dimensional points in Euclidean space generated by k spherical Gaussian distributions. The clustering problem is to determining the means of each of the k Gaussians $\mu_1, \cdots, \mu_k$ (since we are assuming the variance is 1 by letting the Gaussians be spherical). The hidden variables z_{ij} tell us if example x_i was generated by Gaussian j (in which case $z_{ij} = 1$) or not ($z_{ij} = 0$). For the expectation step, given a set of means, we compute the expected value of all the variables z_{ij}, thus maximizing the likelihood of the latent

variables. Because we have assumed the underlying Gaussians to be spherical, each point x_i is simply assigned to the cluster to which it is the closest (via Euclidean distance). For the maximization step, we use the newly computed cluster assignments and adjust our estimates of the means by setting the mean of Gaussian j, μ_j, as the average of all the points assigned to that distribution in the expectation step.

3.1.4.1 *Likelihood.* The likelihood of a model M is the probability that the model generated the data x it represents (using the generative view). When we speak of data we use the term probability. When we speak of models (an EDCM mixture), we use the term likelihood. If we maximize the likelihood of a model, we adjust the model so it has the greatest probability of producing the data.

Assuming the examples x to be independently drawn, we express the likelihood, L, of x given a linear mixture model M (i.e. a clustering) as

$$\mathrm{L}(M|x) = \prod_{i=1}^{N} \Pr(x_i|M) \tag{3.18}$$

$$= \prod_{i=1}^{N} \sum_{j=1}^{k} \Pr(x_i|c_j) \Pr(c_j). \tag{3.19}$$

Each example x_i has a likelihood under the model M. The product of each of each $\Pr(x_i|M)$ gives the likelihood of the model given all the data. The model itself is composed of several constituent parts. In the case of clustering, each part is a specific cluster c_j. For the EDCM mixture model, each cluster c_j is represented by a set of parameters β_j. Thus, for each x_i, we sum the probability for that example for each cluster c_j ($1 \leq j \leq k$) multiplied by the prior probability of the cluster c_j. The prior, $\Pr(c_j)$, tells us how likely instances of the cluster c_j are to occur in general, or how often we predict cluster c_j in general. The probability $\Pr(x|c_j)$ tells us that given cluster c_j, how likely is it that example x_i was drawn/generated from the distribution describing c_j. Thus, the product $\Pr(x|c_j) \Pr(c_j)$ gives a value proportional to the

total probability for a particular data point belonging to a particular cluster. To compute the actual probability, we would have to compute

$$\Pr(c_j|x_i) = \frac{\Pr(x_i|c_j)\Pr(c_j)}{\Pr(x_i)}, \tag{3.20}$$

according to Bayes rule. However, the probability of x_i, $\Pr(x_i)$, is not dependent on a cluster and is often left out of the computation of the likelihood. Summing along the clusters gives the probability for a particular data point. Multiplying these probabilities for all data points gives the total likelihood of a model.

It is often the case that we deal with the log of the likelihood for mathematical reasons. We find the log likelihood ($\mathcal{L}$) is easier to maximize than the likelihood itself.

$$\begin{aligned} \mathcal{L}(M|x) &= \log(\mathrm{L}(M|x)) & (3.21)\\ &= \sum_{i=1}^{N} \log\left(\sum_{j=1}^{k} \Pr(x_i|c_j)\Pr(c_j)\right) & (3.22) \end{aligned}$$

To maximize the log likelihood, we take the first order derivative, set to zero, and solve for the variable we want to compute (like any other function we would want to maximize using calculus). The likelihood function for the EDCM is given in Equation 3.13. The log likelihood of the EDCM for one example is therefore

$$\mathcal{L} = \log n! + \log\Gamma(s) - \log\Gamma(s+n) + \sum_{d:x_d\geq 1} (\log\beta_d - \log x_d). \tag{3.23}$$

Consider the likelihood of a set of N examples, with example x_i having length $n_i = \sum_{d=1}^{D} x_{id}$. Solving for β_d, we have

$$\beta_d = \frac{\sum_{i=1}^{N} I(x_{id}\geq 1)}{\sum_{i=1}^{N}\Psi(s+n_i) - |N|\Psi(s)}. \tag{3.24}$$

The digamma function, Ψ, is defined as the derivative of the logarithm of the gamma function, Γ.

$$\Psi(x) = \frac{d}{dx}\log\Gamma(x) \tag{3.25}$$

We cannot solve directly for β_d because it depends on the unknown s. We can put Equation 3.24 in terms of s by remembering $s = \sum_{d=1}^{D} \beta_d$.

$$s = \frac{\sum_{d=1}^{D} \sum_{i=1}^{N} I(x_{id} \geq 1)}{\sum_{i=1}^{N} \Psi(s + n_i) - |N|\Psi(s)}. \tag{3.26}$$

We will examine how to solve for s when it is defined in terms of itself when discussing the EM algorithm for EDCM mixtures.

3.1.4.2 *Expected value.* The EM algorithm relies on the idea of the expected value of a variable (hence the name of the expectation step), which is simply defined. Given a random variable x with some probability density function $f(x)$, the expected value is simply the value that one expects to see on any given observation of x, the "average." It is the sum of the probabilities of each value of x multiplied by that value. In the case that x has a continuous probability distribution function $f(x)$, we can express the expected value as

$$E[x] = \int_{-\infty}^{\infty} x f(x)\ dx. \tag{3.27}$$

For discrete distributions, where each x has a probability $p(x)$ of occurring, we can express the expected value as

$$E[x] = \sum_{i=1}^{N} x_i p(x_i). \tag{3.28}$$

As an example, consider the uniform distribution over the interval $x \in [0, 1]$ (pdf $f(x) = 1, \forall x \in [0, 1], 0$ otherwise). Because we are dealing with the uniform distribution, each value of $x \in [0, 1]$ has just as likely a chance of occurring. The expected value is

$$\begin{aligned} E[x] &= \int_{-\infty}^{\infty} x \cdot 1 dx && (3.29)\\ &= \frac{1}{2}x^2\Big|_{-\infty}^{\infty} dx && (3.30)\\ &= \frac{1}{2}x^2\Big|_{0}^{1} && (3.31)\\ &= \frac{1}{2}. && (3.32) \end{aligned}$$

Consider a discrete example, a six sided, fair die. Each side of the die, 1–6, has an equal probability of being rolled ($p(x) = \frac{1}{6}$). The expected value of a roll of the die is

$$\begin{aligned} E[x] &= \sum_{i=1}^{6} \frac{1}{6} i && (3.33) \\ &= \frac{1}{6}1 + \frac{1}{6}2 + \frac{1}{6}3 + \frac{1}{6}4 + \frac{1}{6}5 + \frac{1}{6}6 && (3.34) \\ &= \frac{1+2+3+4+5+6}{6} && (3.35) \\ &= \frac{21}{6} && (3.36) \\ &= 3.5 && (3.37) \end{aligned}$$

Thus, we see expected value is not always in the domain of x (3.5 is not a valid value for a roll of the die), nor is it always the median of the values (which would be 3, in this case).

3.1.4.3 *EM with the EDCM.* We use the EM algorithm to maximize the log likelihood (Equation 3.22) of a mixture of EDCM distributions. We assume our data x is generated from a group of k EDCM distributions. The missing data is what EDCM distribution each data point is generated by. After the EM algorithm is run, we are left with a set of parameters β_j for each EDCM distribution in our mixture ($1 \leq j \leq k$). Elkan (2006) gives the outline for computing the EDCM mixture using the EM algorithm and derives all of the following formulas.

We initialize our parameters β for the EDCM mixture by first considering the data as one large cluster. We compute a set of parameters β_0 to fit an EDCM to the entire dataset. Because we are considering all examples to belong to one distribution, there are no latent variables and we do not have to use the EM to calculate β_0. From β_0, we derive k sets of individual parameters β_j ($1 \leq j \leq k$) using small, random perturbations (Elkan 2006). Each dimension d of each β_j is a slight, random modification of β_{0d} ($\pm 5\%$). In other words, multiply the dimensions of β_0 by a random

value in $[0.95, 1.05]$ to generate k β_j parameter vectors. These are the initial sets of parameters for each distribution in our mixture model. We also initialize a uniform set of k prior probabilities, with $\Pr(c_j) = 1/k$ being the prior for model j.

With an initial set of parameters, β_j, and priors, $\Pr(c_j)$, for each cluster, we can begin the EM algorithm. For the expectation step, we compute cluster assignments m_{ji}, the probability that example x_i is generated by cluster j (with parameters β_j).

$$m_{ji} = \Pr(j|x_i) = \frac{\Pr(c_j)\Pr(x_i|\beta_j)}{\sum_{l=1}^{k}\Pr(c_l)\Pr(x_i|\beta_l)} \tag{3.38}$$

The EDCM uses a soft assignment scheme. That is, each example has a probability of being generated by each EDCM in the mixture model, $\Pr(c_j)\Pr(x_i|\beta_j)$. Soft clustering gives the EDCM flexibility as it moves through the parameter space, allowing multiple EDCMs to be responsible for generating individual examples and sharing traits between them. The assignments for each example are normalized such that summing the probabilities for one example across all clusters equals 1. If we have to make hard clustering decisions, we can choose, for each example, the cluster that is most likely to have generated the example. The values $\Pr(x_i|\beta_j)$ can be computed directly from Equation 3.13, with one caveat. We must compute the probability in log space to prevent computational overflow within the computer. The magnitude of n can be very large (remember $n = \sum_d x_d$). For the program trace data clustered by SimPoint, this means the n for most examples can be something like $100e6$, depending on the interval size we are using. Trying to compute $100e6!$ will result in overflow. Therefore, we compute the probability $\Pr(x_i|\beta_j)$ in log space (we can compute $\log(n!)$ using $\log\Gamma(n+1)$). Now, because we have log probabilities, we must compute the cluster assignments in log space. We cannot simply exponentiate the probabilities and work in normal space because the log probabilities are so negative that they would underflow. To avoid underflow when computing the cluster assignments, we use

$$m_{ji} = \frac{\exp(\log\Pr(c_j) + \log\Pr(x_i|\beta_j) - c)}{\sum_{l=1}^{k}\exp(\log\Pr(c_l) + \log\Pr(x_i|\beta_l) - c)} \tag{3.39}$$

with

$$c = \max_j \left[\log \Pr(c_j) + \log \Pr(x_i|\beta_j)\right]$$

(Elkan 2006). Subtracting c in log space is equivalent to dividing by c in normal space, normalizing the product $\Pr(c_j)\Pr(x_i|\beta_j)$ (which is a sum in log space) to be closer to 1. Because the number is no longer so small, we can exponentiate it without underflowing.

An additional difficulty in using log probabilities is computing the log likelihood of our mixture model. Recall Equation 3.22, with cluster c_j being represented by its parameters β_j. To compute the log likelihood and avoid underflow, we must calculate $\mathcal{L}$ as

$$\mathcal{L}(M|x) = \sum_{i=1}^{N} \log \left[\sum_{j=1}^{k} \exp\left[\log \Pr(x_i|\beta_j)\right] \Pr(c_j)\right]. \tag{3.40}$$

We define

$$\rho_i = \max_j \log \Pr(x_i|\beta_j), \tag{3.41}$$

the maximum probability, over all clusters, of example x_i. We normalize the other probabilities of x_i by this value to avoid underflow. Modifying

$$\begin{aligned} \sum_{j=1}^{k} \exp\left[\log \Pr(x_i|\beta_j)\right] \Pr(c_j) &= \sum_{j=1}^{k} \exp\left[\log \Pr(x_i|\beta_j) - \rho_i + \rho_i\right] \Pr(c_j) \quad (3.42) \\ &= \exp(\rho_i) \sum_{j=1}^{k} \exp\left[\log \Pr(x_i|\beta_j) - \rho_i\right] \Pr(c_j) \quad (3.43) \end{aligned}$$

we are effectively dividing all the probabilities $\Pr(x_i|\beta_j)$ (subtraction in log space is division in normal space) to bring them closer to 1. Since they are no longer quite so small, we can exponentiate them for use in the computation of $\mathcal{L}$ without the risk of underflow. We multiply the sum by ρ_i to cancel out the effects of the division after the exponentiation has taken place. We can then calculate the log likelihood as

$$\mathcal{L} = \sum_{i=1}^{N} \left[\rho_i + \log \sum_{j=1}^{k} \exp\left[\log \Pr(x_i|\beta_j) - \rho_i\right] \Pr(c_j)\right] \tag{3.44}$$

without risking underflow associated with exponentiating such a small number.

For the maximization step, we use the cluster assignments m to recompute the parameters of our EDCM mixture and the priors of each model. Recall from Equations 3.24 and 3.26 that we must start by calculating s, and we must do so via numerical analysis since s is defined in terms of itself.

We initialize each s_j to some random value and compute a final value using fixed point iteration. Fixed point iteration updates a variable's value iteratively until there is no longer a significant change (where significance is defined as a percent change above some small threshold ϵ, not statistically). Modifying Equation 3.26 to take into account the cluster assignments, Elkan (2006) computes the s_j for some set of parameters β_j as

$$s'_j = \frac{\sum_{d=1}^{D} \sum_{i=1}^{N} m_{ji} I(x_{id} \geq 1)}{\sum_{i=1}^{N} m_{ji} \Psi(s_j + n_i) - M\Psi(s_j)}, \tag{3.45}$$

with $M = \sum_{i=1}^{N} m_{ji}$ being the sum, over all examples, of a particular cluster's assignments and s' being the updated value per iteration on s. We iterate until the percent change from s to s' is less than some small ϵ. Minka (2003) proves the iteration is convergent.

After computing s_j, Elkan (2006) computes β_j, using a modification of Equation 3.24 to account for our cluster assignments, in one, closed form step.

$$\beta_{jd} = \frac{\sum_{i=1}^{N} m_{ji} I(x_{id} \geq 1)}{\sum_{i=1}^{N} m_{ji} \Psi(s_j + n_i) - M\Psi(s_j)} \qquad 1 \leq d \leq D \tag{3.46}$$

Any dimensions that are always 0 are non-informative and are removed from the data.

We iterate the EM algorithm until there is not a significant change in the log likelihood of the model. We compare the likelihood in one iteration to the likelihood in the next, terminating iteration if the difference between the two likelihoods is less than some small ϵ.

3.1.4.4 *Random restarts.* There are several points in the EM algorithm where we use random values to initialize variables. For instance, we use random perturbations to initialize the k sets of parameters β_j. We also start the fixed point iteration

of s with a random value. It may be the case that these random values lead to poor local minima of the log likelihood function resulting in sub-optimal configurations of β for our EDCM mixture. To try and counteract the possible negative effects of random initialization, we restart the EM process for each value of k multiple times and choose the set of parameters β that has the highest log likelihood.

The number of times we should restart the EM algorithm is an easily configurable parameter. Setting the value higher will increase the chances that we find a model giving a higher log likelihood. However, this relates directly to an increase in running time as we must now perform linearly more clusterings (linear in the number of random restarts). We can decrease the number of random restarts and our clustering will run faster. However, we do this at the risk of lowering our chances of finding a clustering with a higher log likelihood.

3.1.4.5 *Deterministic annealing.* The EM algorithm is very sensitive to the initialization of the parameters. Though it is guaranteed to converge on a maximum log likelihood, this is only a local maximum and the solution may not be globally optimal. The difficulty with the initialization of the EM algorithm comes in the early iterations. At this point, especially when first starting, we are not sure which examples should belong to which clusters. However, our division of cluster assignments for each example is very important as early decisions will affect later iterations. If we happen to initialize to a poor set of parameters, it will be difficult, if not impossible, for the EM algorithm to converge on a model with a globally high log likelihood.

To try and reduce the sensitivity of EM to initialization and to try and avoid the phenomenon of local optima, Elkan (2006) proposes using deterministic annealing with the EM. Deterministic annealing draws from the idea of the principle of maximum entropy (Ueda and Nakano 1998). In the principle of maximum entropy, one seeks to maximize the entropy of a system—make everything more uniform. A tem-

perature parameter is varied over the course of the EM algorithm, starting at some maximum and decreasing to 1. At each temperature step, the EM algorithm is run to convergence, the resultant model fed into the initialization of the next temperature step. We repeat the EM process, moving from high to low temperatures, until we are left with a final model.

Annealing works by using the temperatures to smooth out the probabilities $\Pr(y|\theta')$ in the expectation step. It performs the smoothing by raising the probabilities to the inverse of the current temperature, call it τ. When computing the expectation over the latent variables we perform as calculations as normal but using $\Pr(y|\theta')^{1/\tau}$. Working in log spaces and with log probabilities, annealing is equivalent to dividing the log probability by the temperature. At higher temperatures, this has the effect of driving all probabilities toward 1 and making them seem more similar. Lowering the temperature decreases the effect of annealing, allowing the natural peaks in the probabilities to resurface and for the EM algorithm to climb slowly toward the highest local peak, rather than being stuck at a suboptimal local peak because of initial parameter choices. Ueda and Nakano (1998) give a good visual representation of deterministic annealing, Figure 3.4.

3.1.5 Running Time and Algorithmic Complexity

The running time of the EM algorithm is dependent on many variables. First, we dealing with data matrices sized $n \times D$—n examples of D dimensions each. Ultimately, we compute a β matrix sized $k \times D$—k clusters and D dimensions—and a matrix m of cluster assignments sized $n \times k$—n examples belonging to k clusters. The mathematics to compute the values, and all the values needed in between (such as the k-vector $s = \sum_{d=1}^{D} \beta_d$), result in matrix multiplications and additions bounded by $O(ndk)$.

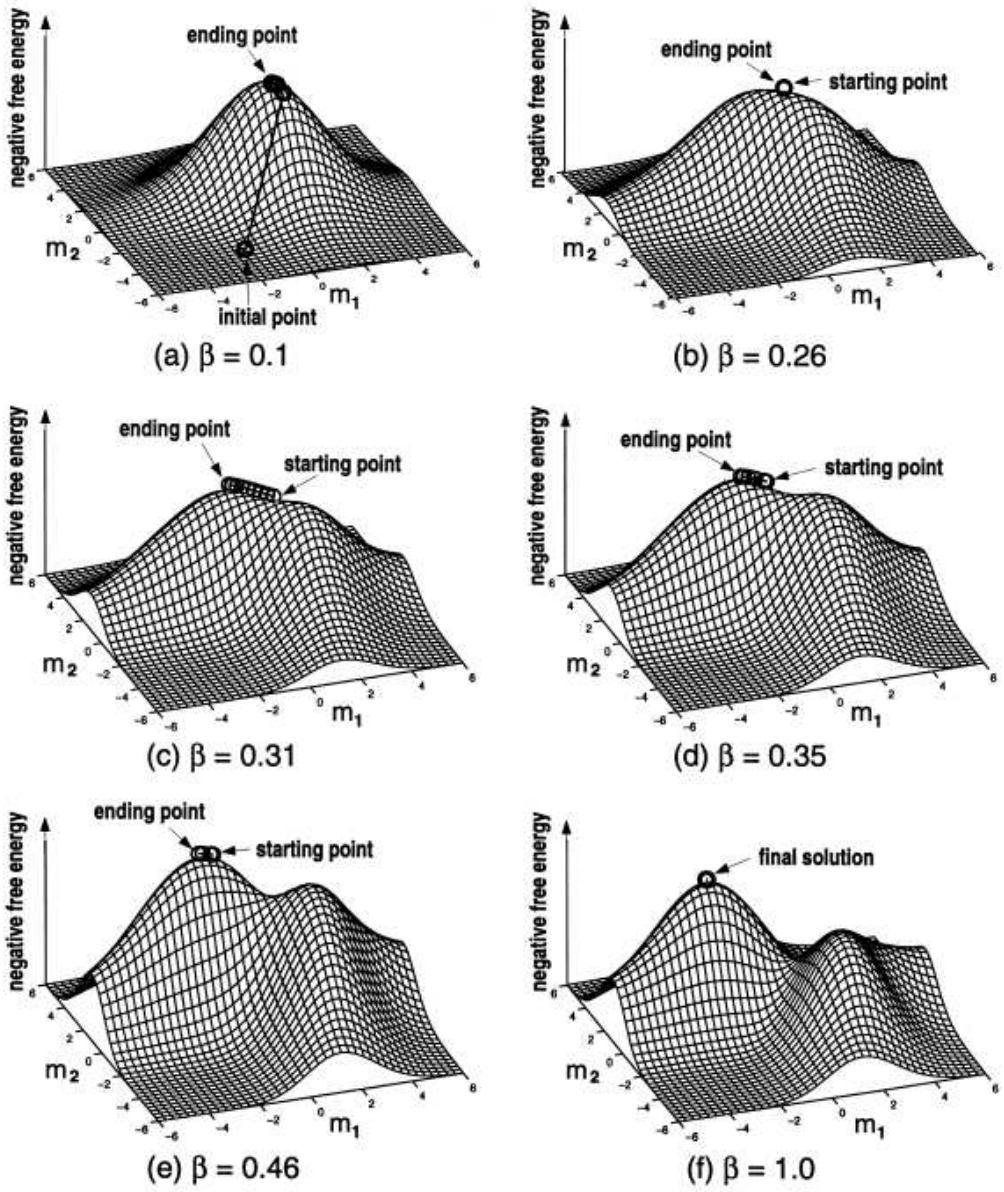


Figure 3.4: Visual illustration of the Deterministic Annealing EM (DAEM) algorithm. Ueda and Nakano (1998) give this visual synopsis of the running of the DAEM algorithm. $\beta = 1/\tau$, so as β increases to 1, the temperature τ is decreasing to 1. We see at high temperatures (low β) the probability density function's two peaks are driven together as all probabilities are smoothed out. As the temperature lowers, the probabilities assigned under the pdf are smoothed less and less, allowing the natural peaks in the probability space to emerge. The DAEM tracks the maximum as the peaks pull apart with lowering temperatures (increasing β). The term "negative free energy" used in the figure comes from the analogy and theory of maximizing the entropy of a system in thermodynamics.

However, one tricky point is that we must use fixed point iteration to compute our s values, since s is defined in terms of itself (Equation 3.45). As far as we know, there is not a theoretical upper bound on the number of iterations required to converge on a final value for s, only that convergence is guaranteed. Therefore, we let i denote the number of iterations needed to converge on a value for s. This brings the running time for one iteration of the EM algorithm (computing the cluster assignments, s, β, and any requisite values between), to $O(indk)$. Although we don't know the number of iterations it should take to converge on a value for s, we see in practice for training an EDCM mixture to cluster program trace data i is rarely greater than 9 (see the chapter on experimental results for more information).

Now we are left with trying to determine how many iterations it will take the EM algorithm to converge. There is a not, to our knowledge, a proven bound on the number of iterations needed for convergence, just that it is guaranteed to converge. The convergence rate of the EM algorithm depends largely on the values of the data, latent variables, and parameters and so is not easily proven. Therefore, we let e denote the number of iterations needed to converge the EM algorithm. This brings the asymptotic upper bound of the running time for the running of the EM to

$$O(eindk). \tag{3.47}$$

It takes e iterations of the EM to converge, and at each iteration we must compute s, taking i iterations to converge. Computing s costs $O(ndk)$ per iteration. We are making the assumption, of course, that the mathematical operations we need, such as the logarithm, square root, the Ψ function, and the Γ function, etc., all run in constant time for a scalar input. In the experimental results, we give empirical running times and comparisons with SimPoint.

If we are using deterministic annealing with T total temperatures, we have to iterate the EM for each temperature. This increases the running time of the EM algorithm to $O(Teindk)$. However, the temperatures are of user selected values and

so only increase the running time by a constant factor. For instance, if we choose five temperatures for our annealing process, we know $T = 5$ and can drop it from our discussion.

With regards to space, the EM algorithm performs rather well. We must store $n \times D$ values for the data examples, $n \times k$ values for the cluster assignments m and probabilities $\Pr(x_i|\beta_j)$, and $k \times D$ values for the β parameters. All the intermediate matrices needed are of the same size as these other matrices or smaller. Therefore, we express an upper bound on the space required to train an EDCM mixture with the EM algorithm as $O(nD + nk + kD)$ (some constant number of matrices, each one of these three sizes or smaller). We do not combine any of the terms because we do not control the values for n and D. Additionally, if we are choosing the number of clusters k without *a priori* knowledge using some automatic method, we also do not control k.

3.2 Choosing k

There are many parameters to consider when training an EDCM mixture to cluster a set of examples. We have seen how we can use the Expectation Maximization algorithm to train one EDCM mixture's parameters β to fit a set of examples with a maximum likelihood. However, we may not always know how many EDCM distributions we need to train within our mixture. In other words, how many clusters k best represent the data? This is a fundamental question in unsupervised learning, and is one of the most important pieces of information we do not know and must learn without *a priori* information.

3.2.1 Over- and Underfitting

It is often the case that adding more clusters (i.e. increasing k) improves a model's likelihood. Therefore, we could increase the likelihood of a model M by

increasing the number of clusters within M. In fact, we could maximize the likelihood of M by setting $k = n$, one cluster per example. This would mean every point was represented with perfect accuracy and our model would express the data with zero error (where the error of a model might be the sum of the distances from each point to the center of the clusters they are assigned to). However, expressing the data with fewer clusters gives us a greater amount of savings in the summarization of the data, leading to a balance that must be made.

Fitting too many clusters to the data is called overfitting. This is undesirable for many reasons. First, if there is any noise or error in the observations, the model will be fitted to the noise and will not be a good generalization of the underlying processes generating the data. Second, composing a model where we have more clusters than natural groupings of examples defeats the purpose of clustering in the first place. It is often the goal of clustering to find some way to group data and express the summarizations of group characteristics with cluster representatives, reducing the amount of information needed to describe the set of examples from on the order of n to on the order of k. If $n = k$ (in an extreme case of overfitting), we have not accomplished any savings. We can overfit a model to a set of examples without necessarily constructing one cluster for every individual data point. Any model that contains too many clusters is said to overfit the data and loses desired generality, beginning to cluster to subtle nuances in similarity that may not be significant. It is important not to overfit data because, unless we have an infinite number of perfectly drawn samples (the measurement or observation process introduces no error), we will always have a somewhat incomplete picture of the underlying structure of the data. Every sample size is always finite, resulting in a need to remain general in order to account for unseen or unobserved data.

The other extreme is underfitting, where we do not use enough clusters to express the data. Although we save in the amount of data needed to summarize the

examples, we do so at a loss of accuracy. If we don't have enough information to express our data well, we do not get a clear picture of what is going on and may lose many important details. For example, if a set of data contains five clusters in truth, but we only fit two clusters to it, we are in effect losing three clusters' worth of information.

3.2.2 Finding a Balance

The difficulty in choosing k is knowing where the point of diminishing returns lies. On one hand, we want a high enough value for k that we are adequately representing the data with enough clusters. On the other, we want a small enough k that our clustering is as concise as possible. The hard questions to answer are what it means for a model's representation to be adequate, and how concise we would like our summarization to be at the expense of details. There are many automatic methods for determining which model, given many with different values of k, should be used. The methods we examine here, the Akaike Information Criterion and the Bayesian Information Criterion, work by penalizing the log likelihood, how well a model fits a set of examples, by the complexity of the model. Increasing k will increase the log likelihood of the model, but at some point the penalty factor of increasing k, and, therefore, adding more complexity, will start to decrease a model's attractiveness.

The following criteria yield a score—an assigned value weighing the trade-off between complexity and likelihood. We perform a deterministic search through a bounded space of possible k and choose the model (with the corresponding value for k) that yields the optimal score. The optimal is not always the maximum score and depends on the formulation of the criterion, but it will always be a local optima, either maximal or minimal. To search for the optimal score, we begin with the lower bound on our value for k. Performing random restarts and deterministic annealing as appropriate for the experiment at hand, we choose the best model given the current k

(the model with the highest log likelihood) and compute its score within the criterion. Incrementing k, we continue to compute the scores for each model for each value of k, keeping track of the k yielding the optimal score (which may be a minimum or maximum depending on the score function). Experimentally, we find that each scoring method reaches a local optimum then falls away rapidly, allowing an easy search heuristic to be employed to optimize the score. When we find a new optimal score, we look a certain number of values of k greater than the current value to ensure there is not a more optimal score nearby (with a slightly larger k). As an easily modifiable parameter, we arbitrarily chose a value of five, observing empirically that optima are reached quickly and then fallen away from just as quickly. This is discussed in more detail in the experimental results.

3.2.3 Akaike Information Criterion

The Akaike Information Criterion (AIC) is of the form

$$AIC = -2\mathcal{L} + 2p, \tag{3.48}$$

with $\mathcal{L}$ being the log likelihood of our mixture model and p being the number of independent parameters estimated by our model (Akaike 1974). See below for a discussion on p.

Akaike noted that choosing a model based solely on the maximum likelihood estimate (MLE) would always choose the model with the highest number of independent parameters because the resultant model could easily overfit the examples. However, as discussed, a perfect fit to training data means we lose generality and possibly any goodness of fit to data that we have not seen yet.

3.2.4 Bayesian Information Criterion

The Bayesian Information Criterion (BIC, also called the Schwarz Criterion) was proposed as an alternative method of model selection. While the AIC was for-

mulated to deal with different number of parameters, the BIC was formulated specifically to select the number of dimensions that should be used to represent some set of data, noting that MLE methods always choose models with the maximum number of dimensions.

$$BIC = \mathcal{L} - \frac{1}{2}\log(N)p, \tag{3.49}$$

where $\mathcal{L}$ is the log likelihood of the model, N is the number of examples used to train the model, and p is the dimensionality of the model (Schwarz 1978). Although this method is specifically derived to handle changes in the dimensionality of a model, it is translatable into varying the number of clusters within a mixture model because adding a cluster effectively adds $D + 1$ more parameters to our representation (each cluster within the model is represented by some D-vector of parameters β_j, and we estimate another prior probability for the new cluster).

The primary difference between the AIC and the BIC is the penalty imposed on the number of independent parameters. There is a slight difference in the coefficients on each term in the two scoring methods, but this amounts to a constant factor. However, the BIC multiplies p by $\log N$, meaning the penalty includes a term dependent on the number of examples observed, and therefore will be higher than the AIC (for $\log(N) > 4$). This causes the BIC to prefer models with smaller p and allows the AIC to choose models with greater p. The difference in k chosen by each method is examined in the experimental results.

3.2.5 Model Complexity

Both the AIC and the BIC operate by penalizing the log likelihood of a model by some factor of the model's complexity. The complexity of a model is measured by the number of free parameters the model estimates. A free parameter is an unknown within our model that we vary freely to change the fit of our model. For example, the parameters of each EDCM in our mixture are free parameters, as are the prior

probabilities for each EDCM. We generally use p to represent model complexity and express it as

$$p = k + kD - 1. \tag{3.50}$$

We estimate p free parameters for a mixture of EDCM distributions, k parameter vectors β of D dimensions each and $k-1$ prior probabilities (one $\Pr(c_j)$ per EDCM). Because the k prior probabilities must some to 1, mathematically we only estimate $k-1$ of them, solving for the k^{th} prior as $1 - \sum_{j=1}^{k} \Pr(c_j)$.

The EM algorithm must also estimate a number of cluster assignments, the latent variables m_{ji}. The cluster assignments for each interval x_i must sum to 1 across the different clusters ($\sum_{j=1}^{k} m_{ji} = 1$). Therefore, the first $k-1$ cluster assignment variables are free parameters, with the k^{th} value being solved mathematically as $m_{ki} = 1 - \sum_{j=1}^{k} m_{ji}$. We compute a set of cluster assignments for all n intervals, meaning we estimate $n(k-1)$ latent variables. It might be the case that we need to modify our formulation of p to take into account these additional parameters,

$$p = k + kD - 1 + n(k-1). \tag{3.51}$$

However, for the remainder of the thesis, we use the formulation of p given in Equation 3.50, without considering the number of cluster assignments estimated.

3.3 Analysis of Clusterings

Once we've clustered a set of examples with an EDCM mixture, with k selected without the use of *a priori* knowledge (using either the AIC or BIC), random restarts to reduce the effects of random initialization of the EM algorithm, and possibly using the Deterministic Annealing EM algorithm to reduce the effects of local maxima, we want a way to determine how well a clustering fits our examples. Specifically, we wish to compare the clustering provided by the EDCM mixture to clusterings provided by the k-means algorithm used in SimPoint, since it is the *de facto* standard and motivation for the clustering problem at hand.

Much of this section will be covered in the experimental results, also. We will lay down motivations for the processes we took to analyze our clusterings here, separating the results and findings of the analysis into the other chapter.

3.3.1 Variation of Information

To compare two clusterings on the same dataset, we can use the Variation of Information (VI) metric. The VI is based on mutual information shared between two clusterings on the same dataset, regardless of the number of clusters in either model or the cluster assignments. The VI can be used regardless of how the clusters were obtained or whether the cluster assignments are hard or soft (Meilă 2002).

Meilă derives the VI in a straightforward manner. Consider a set of N examples assigned to k clusters $(c_1, \cdots, c_k)$. If we choose one of the examples at random (with each example having equal probability of being picked), the probability that it is assigned to cluster c_j is

$$\Pr(c_j) = \frac{|c_j|}{N}. \tag{3.52}$$

The entropy of these probabilities represents, for a particular clustering C, the amount of surprise or uncertainty we face when trying to decide which cluster some randomly drawn example will belong to.

$$H(C) = -\sum_{j=1}^{k} \Pr(c_j) \log \Pr(c_j) \tag{3.53}$$

If the examples are uniformly spread out over all the clusters, then the amount of uncertainty in our guess is high because the example could belong to any cluster just as well as any other. If all the examples are assigned to one cluster, the entropy will be low (minimized at 0) because we can be absolutely certain which cluster is correct.

Given two clustering models, C and C', with k and k' clusters respectively, Meilă (2002) computes the mutual information between them. Mutual information tells us how much about C' we can infer when we know C. First, consider another

set of probabilities,

$$\Pr(c_j, c'_{j'}) = \frac{|c_j \cap c'_{j'}|}{N}, \tag{3.54}$$

representing a $k \times k'$ matrix of probabilities that a randomly selected point is assigned to cluster c_j in model C and to cluster $c'_{j'}$ in model C'. The mutual information between the two clustering models C and C' is then

$$I(C, C') = \sum_{j=1}^{k} \sum_{j'=1}^{k'} \Pr(c_j, c'_{j'}) \log \frac{\Pr(c_j, c'_{j'})}{\Pr(c_j) \Pr(c'_{j'})}. \tag{3.55}$$

Meilă offers a concise and clear explanation of the meaning of $I(C, C')$.

> We are given a random point in [the domain of the data]. The uncertainty about its cluster in C' is measured by $H(C')$. Suppose now that we are told which cluster the point belongs to in C. How much does this change the uncertainty about C'? This reduction in uncertainty, averaged over all points, is equal to $I(C, C')$. (2002)

The Variation of Information between two clusterings C and C', the difference in information given by the two clusterings, is defined by Meilă as

$$\begin{aligned} VI(C, C') &= H(C) + H(C') - 2I(C, C') && (3.56) \\ &= [H(C) - I(C, C')] + [H(C') - I(C, C')]. && (3.57) \end{aligned}$$

Intuitively this is the amount of uncertainty present in both clusterings minus the amount of certainty we can infer about one clustering using the other. Therefore, it is the total amount of uncertainty in the system composed of the two clusterings C and C'. Meilă gives an elucidating illustration to help explain the logic and derivation of the VI, see Figure 3.5.

Upper and lower bounds are provided for the VI, allowing relative comparison from one set of clusterings to another. The variation of information for two identical clusterings is 0 (knowing one gives us total information about the other). Two upper bounds are given for the VI. The first is

$$VI(C, C') \leq \log(N) \qquad \forall N, \tag{3.58}$$

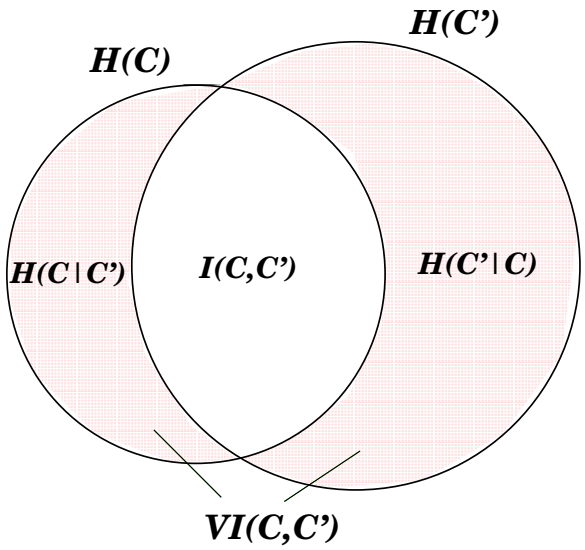


Figure 3.5: Variation of Information. Each circle represents the uncertainty, the entropy H, of each clustering C and C'. The overlap between the circles is the amount of information we can infer from one about the other. Adding together the entropies and subtracting out the mutual information (the middle section twice, once for each larger circle), leaves us with the total uncertainty given the two clusterings, the amount of information that given one clustering, we do not know about the other clustering. (Meilă 2002)

where N is the number of examples clustered by the models C and C'. The second upper bound is tighter but comes with more assumptions. If we let $K^* = \max(k, k')$ be the maximum number of clusters between C and C', and if $K^* \leq \sqrt{N}$ then it is the case that

$$VI(C, C') \leq 2 \log K^* \tag{3.59}$$

(Meilă 2002). We compare SimPoint and EDCM clusterings using the Variation of Information to determine how similar they are.

3.3.2 Cluster Characteristics

We don't just want to compare resultant clusterings from one process against those generated by another. Within the same process, the EDCM mixture model, we want to analyze the clusterings produced to gain an insight into how the EDCM

handles program trace data. We can look at how the examples are distributed among the clusters and how examples within the same cluster are related.

3.3.2.1 *Cluster weights.* The distribution of cluster weights tells us if our clustering algorithm is doing a good job of spreading the examples evenly across the different clusters. It is probably not the case that we always want a uniform distribution of data across the clusters (i.e. the same number of examples in each cluster). Being very data dependent, it is difficult, if not impossible, to describe what the proper distribution of cluster weights should be. Hence the reason clustering belongs in the field of unsupervised learning, because we do not know what the clusters should look like *a priori*.

However, one can examine the distribution of cluster weights among different runs of the EM algorithm for EDCM mixtures and make sure that it does not always favor a select few clusters and put most examples into one cluster. Additionally, if we -do- happen to have some idea of what the data should look like, even though we might not have specifics, we can verify part of our beliefs by examining the cluster weights to make sure the EDCM mixture model is behaving as expected.

3.3.2.2 *Coefficient of variation.* One can determine the amount of similarity of all examples within a cluster using a metric like the coefficient of variation (CoV). Simply defined, the CoV measures the amount of spread within a distribution normalized by the mean, making for easy comparisons between sets of examples with varying means. The CoV for a set of values is

$$CoV = \frac{\sigma}{\mu}, \tag{3.60}$$

the standard deviation divided by the mean of the set.

We use the CoV to determine the similarity of behavior between all intervals in a cluster. We choose the cycles per instruction (CPI) statistic as a marker for interval

behavior. CPI tends to give an overall picture of processor performance during the execution of a benchmark. Low CPI means our processor is doing a good job at getting the most performance out of each clock tick. Higher CPI means our processor is doing a poor job at tasks such as branch prediction and/or we have a lot of cache misses, etc. We choose to examine the CPI statistic because of its overall representation of processor performance. The CoV of the CPI statistics of the intervals within a cluster tells us how much behavior the intervals have in common. Lower CoV means the statistics are more similar to one another and have less variance. A higher CoV means the behaviors are more different.

We modify Equation 3.60 to handle multiple clusters and compute a weighted sum, giving the overall CoV for the entire clustering rather than just within one cluster. This allows us to gauge not just the per cluster variation, but also how closely grouped all the examples within all the clusters are with each other. For a mixture model with k clusters, each with variance σ_j and mean μ_j, the CoV of the CPI is

$$CoV = \sum_{j=1}^{k} \frac{\sigma_j}{\mu_j} w_j. \tag{3.61}$$

We use σ_j to denote the standard deviation in the set of CPI values for the intervals in cluster c_j and μ_j to represent the mean of the CPI values. The weight of each cluster is $w_j = |c_j|/N$, the percentage of all examples assigned to cluster c_j. Clusterings with relatively small CoV have done a better job at grouping similar items than clusterings with a higher CoV.

3.4 Choosing Cluster Representatives

The previous sections dealt with ways to cluster data, including high dimensional and bursty data, using the EM algorithm to train EDCM mixtures. We also examined ways in which we could verify and validate the resultant clusterings. Once we have a clustering that we feel confident is accurate of the clusters within the

underlying data, we may choose cluster representatives. Cluster representatives, appropriately enough, represent the clusters they are selected for. They are the means by which one may summarize the examples within the clusters.

In SimPoint, the cluster representatives are called simulation points. Detailed simulations are run on the simulation points and the gathered statistics are assumed to represent the behavior of all the intervals assigned to the same cluster. Therefore, it is important to not only have clusters that are indeed collections of examples (intervals) with strong similarities (behavior), but also that we choose cluster representatives that summarize all the examples in the cluster well.

3.4.1 Probabilistic Methods

Because the EDCM mixture uses underlying probability distributions to perform all the clustering, each example has a probability assigned to it for each cluster, $\Pr(x_i|\beta_j)$. A straightforward approach to choosing cluster representatives would be to select the example with the highest probability. We can select a simulation point, s_j, for cluster c_j using

$$s_j = \arg\max_{x_i \in c_j} \Pr(x_i|\beta_j). \tag{3.62}$$

The rationale behind using intervals with the highest probability as simulation points is that the parameters for a cluster, once the EM algorithm terminates, are fixed to maximize the likelihood of the cluster it describes and of the model as a whole. Each interval has a probability under the distribution described by the parameters for that particular EDCM. Therefore, it would seem that choosing the interval that maximized the probability with respect to the parameters, that maximized the log likelihood of the distribution, would give the most representative interval, because the selected interval would fit the cluster's parameters the best.

3.4.2 Geometric Methods

Alternatively, we may treat each example as a vector in some geometric space and use geometric methods to compare similarities. Examples with similar feature vectors will be relatively close together in the geometric space, however that sense of distance might be measured.

3.4.2.1 *Angular metrics.* One method of comparing vectors in geometric space is to normalize all the example vectors and the cluster representations (in the case of an EDCM mixture, the parameters β) to the unit hypersphere. Using the dot product, we select the normalized example with the smallest angle (using the dot product) from the normalized β vector. In D dimensions, the unit hypersphere is composed of the set of vectors x satisfying

$$\left\{ x : \sum_{d=1}^{D} x_d^2 = 1 \right\}. \tag{3.63}$$

To normalize all the examples to the unit hypersphere, we simply divide each example by the sum of the square of each dimension (letting x' denote the normalized example),

$$x' = \frac{x}{\sqrt{\sum_{d=1}^{D} x_d^2}}. \tag{3.64}$$

We normalize β in a similar fashion, giving β'.

When comparing x' to β', we want to choose as cluster representative the x' that is most similar to β'. On the surface of the unit hypersphere, or some other normalized surface, this equates to the x' with the smallest angle to β'. We can compute the angle between two vectors using the dot product. For two D-dimensional vectors a and b, the dot product, denoted $\langle a \cdot b \rangle$, is

$$\langle a \cdot b \rangle = \sum_{d=1}^{D} a_d b_d \tag{3.65}$$

$$= ||a||||b|| \cos\psi, \tag{3.66}$$

with $||a|| = \sqrt{\sum_{d=1}^{D} a_d^2}$ being the length of vector a. To minimize the angle ψ between a and b, we want to maximize the dot product. This is because when two vectors

are orthogonal, or at a ninety degree angle to one another, $\cos\psi = \cos 90^\circ = 0$ and $\langle a \cdot b \rangle$ is 0. If, however, the two vectors are collinear and point in the same direction (as opposed to pointing in opposite directions), $\psi = 0$ and $\cos\psi = 1$. For vectors pointing in opposite directions, $\cos\psi < 0$. Thus, we see that in order to minimize ψ, we need to maximize the dot product. Our simulation point, s_j, for cluster c_j is chosen as

$$s_j = \arg\max_{x'_i \in c_j} \langle x'_i \cdot \beta'_j \rangle. \tag{3.67}$$

3.4.2.2 *Distance metrics.* We can check for similarity between points in geometric spaces without using normalizations and the dot product by using distance metrics. Without normalizing our examples, we can treat them as points in some D-dimensional space. The x with the minimum distance to the cluster "centers" is chosen as the simulation point. A point that is closest to the center of a cluster is the point that is closest to the overall, or average behavior of the examples in that cluster.

There are several distance metrics we can choose from when deciding how to measure distance, but all are based off the L_p norm. For D-vectors a and b, the distance metric L_p is defined as

$$L_p(a, b) = \sqrt[p]{\sum_{d=1}^{D} |a_d - b_d|^p} \tag{3.68}$$

and is a general form for computing distance, with p being a user selected parameter. It has been shown that choosing smaller values for p yields more meaningful distance results in high dimension (Aggarwal *et al.* 2001). Recall that examples tend to look farther and farther apart as the number of dimensions is increased. Thus, for data in high, we wish to use a distance metric L_p with some small p. The two distance

metrics we examine are Euclidean (straight line) distance ($p = 2$),

$$L_2(a, b) = \sqrt{\sum_{d=1}^{D} (a_d - b_d)^2}, \tag{3.69}$$

and Manhattan distance ($p = 1$),

$$L_1(a, b) = \sum_{d=1}^{D} |a_d - b_d|, \tag{3.70}$$

which is simply the sum of the absolute values of the differences of each dimension. A common analogy to Manhattan distance is the distance from one point to another in a city where the streets are laid out along a square grid. One cannot simply take a straight line from point a to point b, one must travel along each dimension (east-west and north-south) independently.

We can also vary the way in which we choose cluster centers, using either the mean or median of the points in the clusters. Using the mean is the most straightforward method to thinking about choosing a center as it literally does, in fact, end up being a true geometrical center. The mean of a cluster with examples x in D-space is

$$\mu_j = \frac{1}{|c_j|} \sum_{x_i \in c_j} x_i. \tag{3.71}$$

μ_j represents the average value of some set of examples, but not necessarily the middle value. In a skewed distribution, the mean will not always be the most desirable way to express the majority behavior.

To counteract the problems that may be introduced by using the mean, we also examine using the median as the cluster center. The median is simply the middle value in a sorted list. If there are an even number of items within the list, there is not a single middle item, but a middle pair of items. One may choose the upper or lower element in the pair to act as the mean, or one may use the average of the two values as the median. So, given a list of values x, the median ν_j is the value that, given the list, satisfies $\Pr(X \leq \nu) = \Pr(X \geq \nu)$. So, for the above example, $\nu_j = 0$, a much more representative value for the cluster as a whole.

3.4.3 Motivation for Choice of Method

It is an easy thing, given a clustering, to examine the different methods of choosing cluster representatives and simply decide to use the method that works best. However, we want to provide some greater insight into why one method might outperform another in choosing ideal simulation points. Additionally, outside of a research setting, a practitioner will not have statistics for all the intervals, as we have for our testing purposes. Clustering will be used in order to speed up the process of providing such statistics and statistics will not be available *a priori*. Therefore, simply examining which method works best without this knowledge is infeasible.

When judging the merit of a simulation point, we judge how well it represents the overall behavior of the cluster it is chosen from. The behavior of an interval is reflected in its statistics, which are not known *a priori*. We have them for our research because the time has been taken to gather all the statistics beforehand for the purposes of testing the SimPoint application's performance (Sherwood *et al.* 2001). We examine the cycles per instruction, CPI, of an interval to represent its behavior. Looking at the average CPI of a cluster c_j, we can compare it with the CPI of all the intervals $x_i \in c_j$. Moreover, if we examine the difference in CPI between an interval and the cluster average versus the distance (Euclidean or Manhattan) or angle between the example and the centroid (μ_j or ν_j), we hope to see that as distance or angle increases, so does the difference in CPI. This would mean that the centroid is indeed an accurate representative of overall cluster behavior.

CHAPTER FOUR

Experimental Results

The motivation and framework for the thesis has been laid out. Now we examine the ways in which we set up experiments to test our methodologies and expound upon the results.

4.1 Experimental Setup

For all the following experiments, we cluster program trace data in basic block format generated by Sherwood *et al.* (2001), split into fixed length intervals of 100 million instructions each. Full detailed statistics have been gathered on every interval, allowing us to judge the performance of our predictions.

For many experiments we talk about the percent error of our predictions as a basis for judging the merits of different methods used to generate the predictions. The prediction we are making per dataset is the average cycles per instruction (CPI). There are many other statistics we could examine for prediction purposes, such as the number of cache hits/misses, the number of branch prediction failures, etc. We choose to look at the CPI because it gives an overall picture of a hardware design's performance. After clustering the program trace data and choosing simulation points, we compute a weighted sum of the CPI for the dataset by multiplying the CPI of each simulation point by the weight of the cluster it represents. The weighted CPI statistics for all the clusters are summed, serving as the predicted CPI. The predicted CPI is compared to the true average CPI of all the intervals in the dataset and the percent error of the prediction is calculated.

4.2 Burstiness in Program Execution

Burstiness is an important phenomenon. As discussed previously, burstiness occurs not only in the domain of text documents, but is expected to occur in the program trace data clustered by SimPoint as well. We examine different ways burstiness can be defined and how bursty program execution tends to be. We also examine the effects of burstiness in our data.

4.2.1 Measuring Burstiness

In considering how we might quantify burstiness, we might consider the entropy of a feature (across all examples) to be a good measure, with low entropies being representative of bursty behavior and a feature with high entropy being non-bursty (because occurrences would be spread uniformly across all the intervals). Another measure of burstiness could be to examine the sum $s = \sum_{d=1}^{D} \beta_d$ for a particular cluster (Elkan 2006). Elkan notes the parameters of the DCM distribution can be thought of as representing the initial counts of differently colored balls in a Pólya urn scheme, where a smaller value for s results in more bursty behavior.

4.2.1.1 *Measuring burstiness with s.* We first examine expressing the burstiness of a distribution using the sum of the parameters of the distribution, s. Figures 4.1, 4.2, and 4.3 illustrate the bursty behavior of the balls in the Pólya urn scheme. The three graphs differ in the number of colors present in the urn. For each graph, we vary s, the number of balls initially in the urn. Note that for D different possible colors, we must have $s \geq D$. This is because every color must be initially represented within the urn and ball counts are integral. Otherwise, all initial color counts could be < 1 and s would not necessarily have to be $\geq D$. For each value of s, we initially fill the urn with s balls, the color of each ball determined by a uniform distribution over the D color choices (with at least one ball of each color present). Once the urn is initialized,

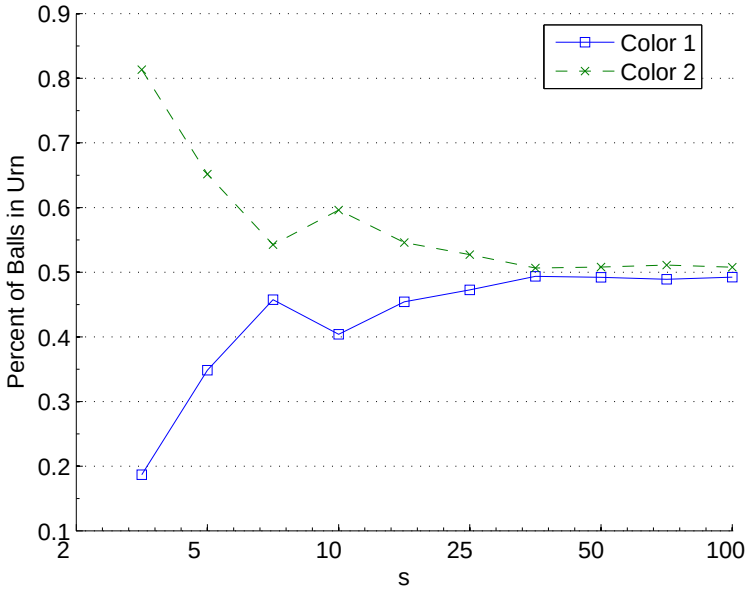


Figure 4.1: Burstiness of the Pólya urn scheme, 2 colors. Here we demonstrate the effect of varying s, the initial number of colored balls in Pólya's urn, on the amount of burstiness exhibited by the urn. For small s, we see that one color tends to dominate the other—the urn exhibits very bursty behavior. However, as we increase s, we see that the distribution of the colors of the balls in the urn becomes more uniform. For the experiment, the urn was initially filled with s balls, the color of each ball decided by a uniform distribution over all the colors (all colors equally probable of being represented initially). We then drew a ball out of the urn at random 5000 times (an easily modifiable parameter), replacing it and adding another ball of the same color. After taking all our samples, we counted up the number of balls of each color in the urn, denoting by 'Color 1' the color with the smaller proportion of balls, and by 'Color 2' the color with the larger proportion of balls.

we then take some large number of samples (this is an easily modified parameter and is set to 5000 for the figures given here), replacing the ball sampled each time and adding another ball to the urn of the same color. After we have drawn our samples, we count the number of balls of each color and calculate the percentage of the urn each color comprises. We denote the color comprising the smallest percentage of the urn Color 1, with the color comprising the highest percentage of the balls in the urn denoted Color D.

The process of initially filling the urn and sampling balls is completely random, but Figures 4.1, 4.2, and 4.3 show the empirically observed trend of burstiness. When

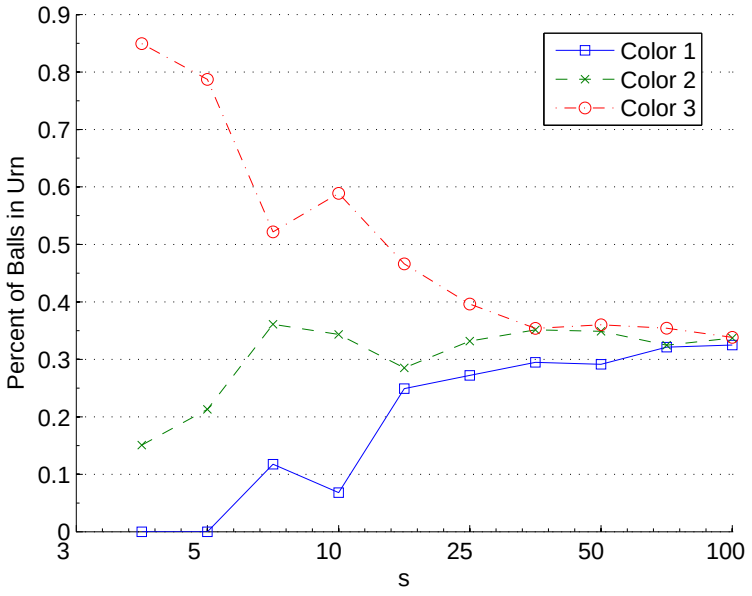


Figure 4.2: Burstiness of the Pólya urn scheme, 3 colors. This experiment was carried out just like the experiment with 2 colors (Figure 4.1), modifying only the number of colors. Note the consistent behavior, with widely differing proportions of colors converging to a uniform distribution of colors as s increases.

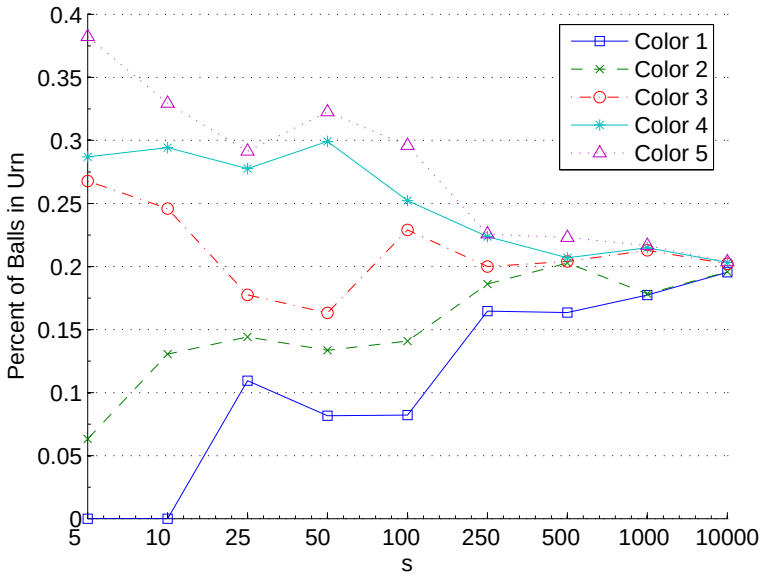


Figure 4.3: Burstiness of the Pólya urn scheme, 5 colors. The same experiment as Figures 4.1 and 4.2 but with 5 colors.

s is small, it is the case that there is some color that clearly makes up a higher percentage of the urn than all the others. Sometimes, the majority is not that pronounced, but it is always there. As s increases, we see that all colors begin to have more and more similar distributions throughout the urn. Finally, when s is very large, we find that the composition of the urn is fairly uniform with regards to color, with no single color having a true majority over any others.

We want to know, examining the values of s of the EDCM distributions, how bursty the program trace data clustered by SimPoint is. We examine the values of s for each dataset as described by one EDCM distribution and use it to describe how bursty that individual dataset is as a whole. Figure 4.4 shows the value of s for each dataset. We also compute, for comparison purposes, a theoretical maximum value for s. s would be maximized, according to the analogy drawn to the Pólya urn scheme by Elkan (2006), on a set of uniform data that does not exhibit burstiness. We create artificial datasets of the same number of dimensions as each of the real program trace datasets. The intervals in the artificial datasets are nearly uniform (with small random perturbations) and the artificial intervals all sum to 100 million, the number of instructions of each of the intervals in the real datasets. The theoretical maximum value of s is plotted above each of the bars, showing for each case, the actual s is much lower than the maximum. Figure 4.5 plots the values of s for the single EDCMs as a percentage of the maximum value of s. We see that, on average, the value of s is only about 12% of the maximum value, supporting the idea that program trace data is bursty because of the relatively small values of s.

4.2.1.2 *Measuring burstiness with entropy.* In addition to using the value of s to describe burstiness in program trace data, we also use the entropy of a basic block's occurrences in the intervals of a dataset. Entropy has been discussed several times already, but for clarity we present its formula again and explain how it is used. For

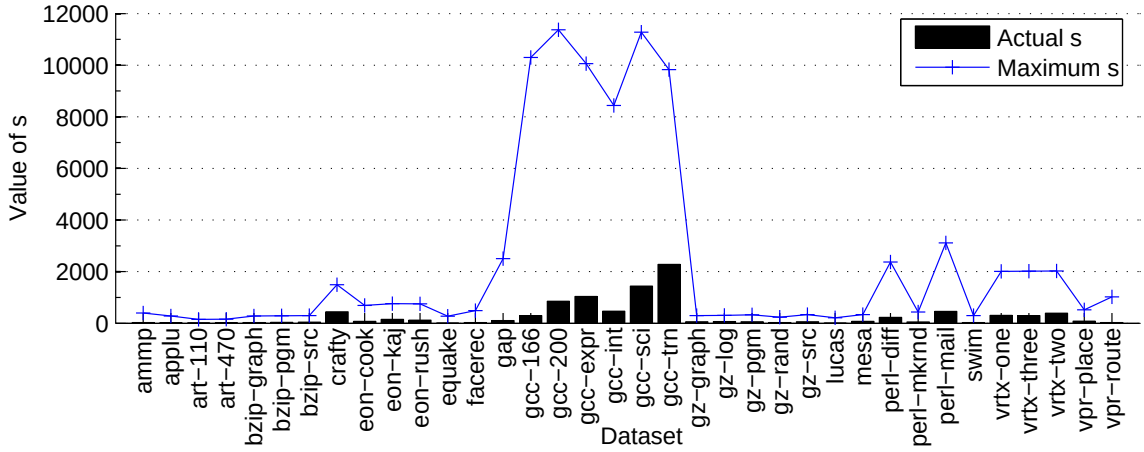


Figure 4.4: Value of s per dataset. Each dataset is fitted with a single EDCM. The value $s = \sum_{d=1}^{D} \beta_d$ for a dataset describes the amount of burstiness in the dataset. Lower values for s represent bursty datasets, as in the Pólya urn scheme. The bars for each dataset give the value of s. The plotted points above each bar give the theoretical maximum value of s, computed by fitting an EDCM to uniform data (with small random perturbations) with the same number of dimensions as the dataset. We see that although there are large differences in the actual values for s for each dataset, every one of them is far below the maximum values. This implies that the program trace datasets are indeed bursty.

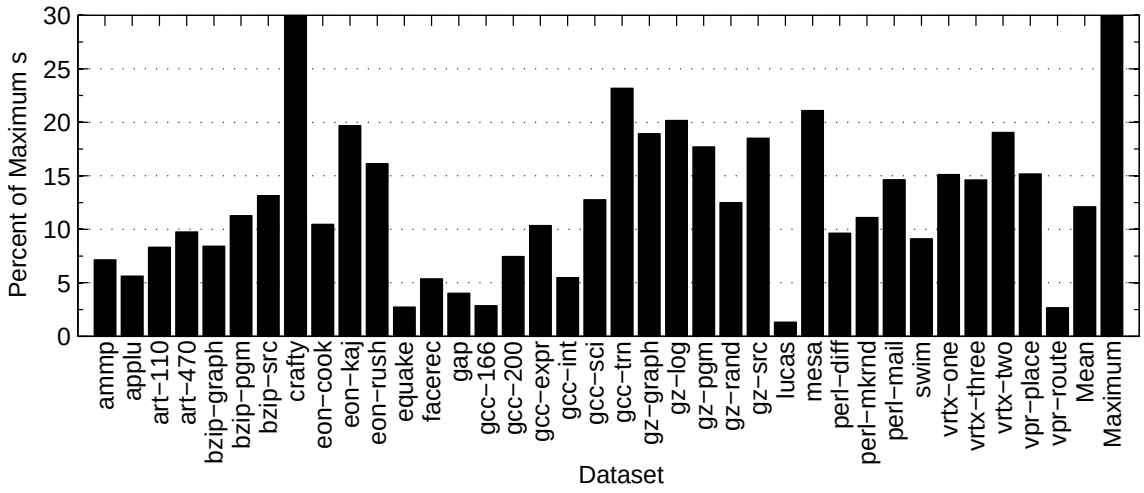


Figure 4.5: Percentages of maximum values of s. For each dataset, we compare the value of $s = \sum_{d=1}^{D} \beta_d$ for a single EDCM compared with the maximum value of s (as computed in Figure 4.4). We see that, on average, the value of s is only 12% of the maximum. We conclude that because of the small values of s, program trace data is bursty.

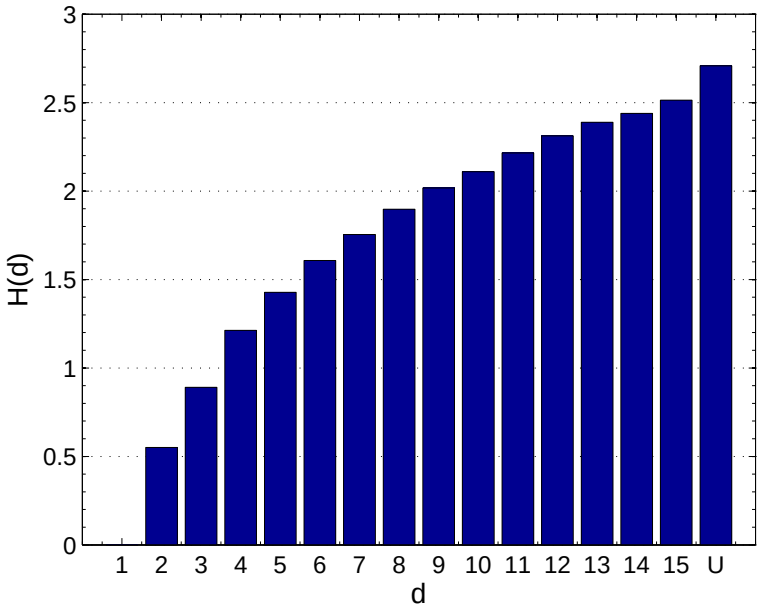


Figure 4.6: Entropy of burty features. Experiment showing that entropy is minimized when a feature is extremely bursty (i.e. all occurrences appear in one interval) and maximized when occurrences appear uniformly throughout the space of the intervals. The horizontal axis plots feature d, which occurs with random probability in d intervals ($1 \leq d \leq D$), meaning smaller d are more bursty. Feature U occurs uniformly over all D intervals (with probability $1/D$). The vertical axis plots the entropies of the features, averaged over multiple trials to smooth out the curve and reduce the effects of randomization.

basic block d, consider the counts of its occurrences within all intervals x_i. Consider the number of occurrences of the basic block in a particular interval, x_{id}. Define

$$\Pr(x_{id}) = \frac{x_{id}}{\sum_{l=1}^{N} x_{ld}}, \tag{4.1}$$

the probability that a particular basic block d executes within interval x_i. With this probability, we can state the entropy of a feature d as

$$H(d) = -\sum_{i=1}^{N} \Pr(x_{id}) \log \Pr(x_{id}). \tag{4.2}$$

The entropy of a basic block describes how uncertain we are about it's occurrence within a particular interval.

We expect bursty features to have a low entropy because occurrences of the basic block will be limited to a select few intervals, rather than having a probability

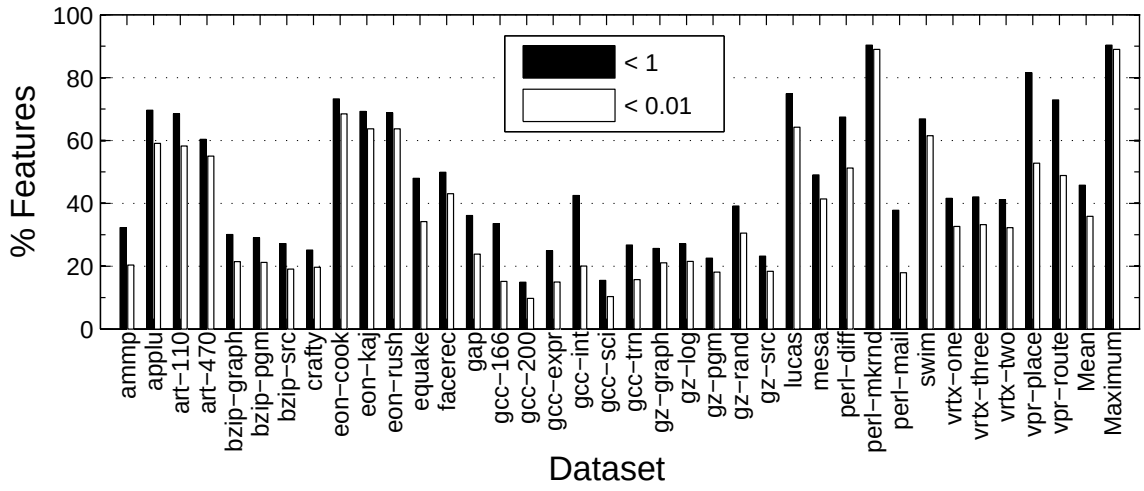


Figure 4.7: Percent of SimPoint features with low entropy. For the most part, features in program trace data tend to be bursty, with bursty features comprising the majority of many datasets. The horizontal axis gives data for each dataset, with the bars for each dataset giving the percentage of all feature entropies that are below the various thresholds. On average, we see that nearly 40% of all basic blocks in the program trace data are bursty.

of being spread over all the intervals. To test this hypothesis, we set up a simple experiment. Consider a set x of D features occurring in D intervals. Specific feature d occurs in d intervals with random distribution of probability, normalized such that $\sum_{i=1}^{D} x_{id} = 1$. Additionally, we add a feature $D+1$ (call it U for uniform) that is uniformly distributed, that is for all $1 \leq i, l \leq D, x_{id} = x_{ld} = 1/D$. We compute the entropy of each feature, expecting the entropy of feature 1 to be 0 because all its occurrences occur in 1 interval. The entropy of feature U should be the highest since its occurrences are uniform throughout the intervals. We repeat the experiment many times to remove measurement error introduced by the randomness of the data, averaging the entropy for each feature d over all the trials. Figure 4.6 plots the averaged entropy for each feature, showing that bursty features (where all occurrences occur in only a few intervals) have a lower entropy, with the entropy maximized when the occurrences are uniform through the intervals.

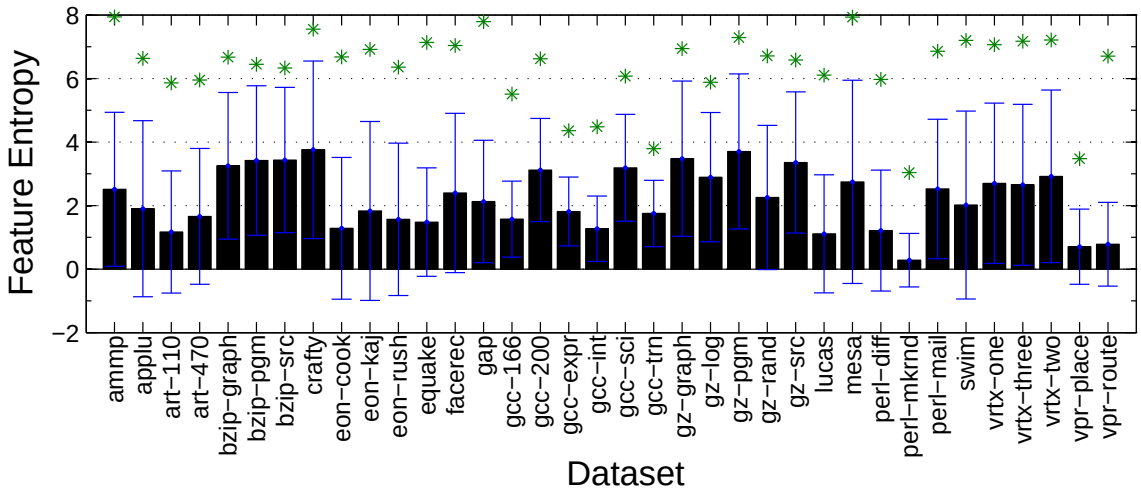


Figure 4.8: Average entropy of SimPoint features. An examination of the burstiness of features in program trace data. For each dataset, the entropy of every feature is calculated and the mean and standard deviation are calculated. The horizontal axis shows the different datasets, with the vertical axis giving entropy values. The bars show the average feature entropy, ± one standard deviation. The symbols along the top show the maximum observed entropy for all features in each dataset, giving a sort of upper bound on the observed entropies for comparison reasons (this is not the maximum theoretical entropy). However, we do see that the maximum observed entropies are very close to the maximum theoretical entropies.

We wish to examine program trace data and determine the amount of burstiness within it, to see how much of a role it may play in clustering. For each set of basic block vectors (each dataset), we compute the entropy of every feature. We calculate the percentage of all features with an entropy smaller than a certain threshold. This allows us to see just how many bursty features there are per dataset. Figure 4.7 shows the percentage of all features in each dataset that have entropies below the thresholds 1 and 0.01. We consider entropies < 0.01 to be very low, corresponding to features with a high amount of burstiness. From the figure, we see the majority of features for many datasets have low entropies and therefore high levels of burstiness.

We also examine the mean entropy of each interval, plotted as bars in Figure 4.8 with the maximum feature entropy of each dataset plotted with a star. Error bars show the mean ± one standard deviation. We see from the figure that it appears

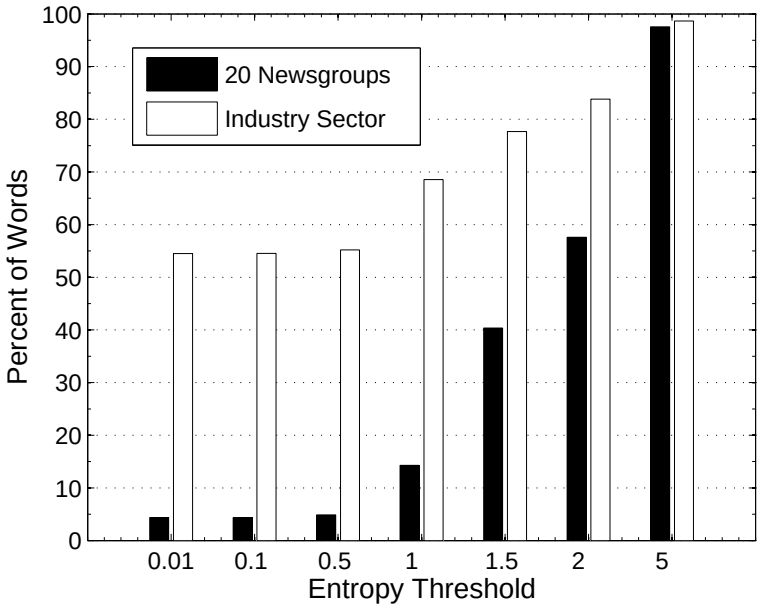


Figure 4.9: Percent small entropies of text corpora. Comparison of the percentage of all words in the two text corpora that have small entropies. We vary the threshold along the horizontal axis and count the number of words with an entropy below that threshold, plotting the percentage of all words along the vertical axis. We can see clearly from this graph that the 20 Newsgroups corpus is much less bursty than the Industry Sector corpus, with more than a fifty percent difference in the number of words that have an entropy < 1 between the two. On average, about 40% of basic blocks in program trace data have an entropy < 1, about halfway between the two text corpora.

features are, on average, more bursty than not (the mean, in all cases, is at least one standard deviation below the maximum). From empirical observation, we see that the maximum feature entropies observed are very close to the theoretically maximum entropy for each dataset (an entropy is maximized when the feature appears uniformly across the intervals).

4.2.1.3 *Burstiness of other domains.* Briefly, we compare the burstiness of program trace data, quantified using the feature entropy measure, to other domains. Specifically, we analyze the burstiness of two standard corpora of text documents, the 20 Newsgroups (Hettich and Bay 1999) and Industry Sector (McCallum) datasets.

Each word in the vocabulary of the sets of documents is a feature. We compute the entropy of every word in both corpora and examine them.

Figure 4.9 shows the percentage of all words in the text corpora that have entropies smaller than certain thresholds. We vary the thresholds widely, examining the change in percentages. Recall that for the program trace data clustered by SimPoint, the mean percentage of features that had an entropy < 0.01 was about 35%. For the text corpora, the percentage of words with entropy < 0.01 varies dramatically, from only 5% up to 55% depending on the corpus under consideration. We see that, by comparison with the number of bursty features, program trace data is about as bursty as text document data, more bursty than some corpora but less bursty than others.

4.2.2 Effects of Burstiness

We have established that basic block vector data, like text documents, is bursty. However, it is not clear what the consequences of having bursty features within our data are. We explore different facets of burstiness, including the probabilities assigned to bursty data by the EDCM and multinomial models, and possible correlations between the burstiness of data and the error in clustering it.

4.2.2.1 *Probabilities of bursty data.* We examine the probabilities assigned to examples of varying levels of burstiness by both multinomial and EDCM models. Given a number of parameters, D, in the model, we construct $D + 1$ examples of D dimensions each. Example d ($1 \leq d \leq D$) has d non-zero features, with each feature having some probability of occurrence and the sum of the probabilities within an example equal to 1. The features in example $D+1$ are distributed uniformly as $1/D$. We call this example U to denote its uniform composition.

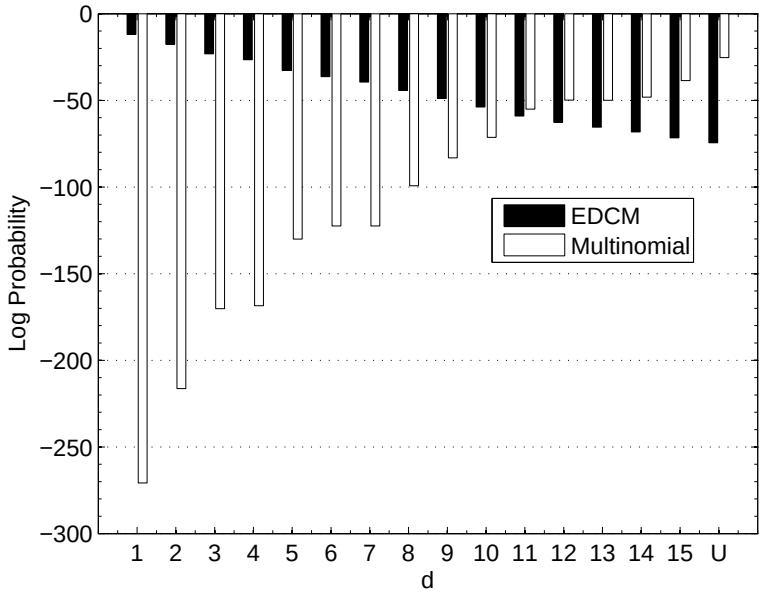


Figure 4.10: Probabilities of bursty data, EDCM vs. Multinomial. Comparison of probabilities assigned by the EDCM and multinomial distributions to randomly generated test data. The test data has D features, d of which are non-zero ($1 \leq d \leq D$)—varying along the horizontal axis. Example U is composed of D uniform features. The two sets of bars are the log probabilities assigned to each example via the different probability distributions. We see the EDCM gives higher probabilities to bursty data, while the multinomial gives higher probabilities to uniform data.

For each probability distribution, either the multinomial or EDCM, we compute the probabilities for all $D + 1$ examples using uniform parameters. That is, for each dimension d, $\theta_d = \beta_d = 1/D$ for the multinomial and EDCM, respectively. We compute the probabilities $\Pr(x|\theta)$ and $\Pr(x|\beta)$ (in log space to prevent overflow) and compare the results in Figure 4.10. Remember, burstiness *decreases* as d increases, with $d = 1$ being the most bursty example and U being a uniformly distributed example. Also, we are dealing with probabilities in log space, which will be negative. The magnitude of a number is directly related to the magnitude of its logarithm, however, so in this graph, values closer to zero correlate to higher probabilities. We can see that the EDCM gives the highest probability to the most bursty example, with the probabilities dropping slowly as the level of burstiness decreases toward U, the uniformly distributed example. The multinomial behaves in the opposite manner,

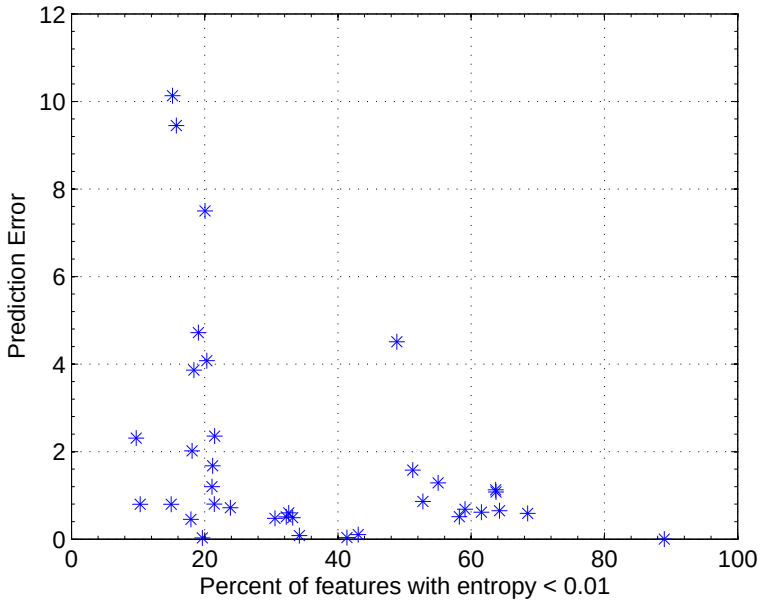


Figure 4.11: Relation between burstiness and prediction error. For each dataset (a point on the graph), we plot the percentage of all features with an entropy < 0.01 versus the prediction error, looking for a correlation. There may be a loose correlation between the two, with the datasets that have high prediction errors having lower percentages of bursty features. It may be easier to accurately cluster datasets that are more bursty.

and much more dramatically, with a very low probability assigned to the most bursty example and a sharp rise to the highest probability given to the uniform example. The EDCM has the higher probability until about 2/3 of all the features in the test examples are nonzero ($d = 11$).

4.2.2.2 *Burstiness and error.* We examine the relationship between burstiness and the prediction error for each dataset, looking for a correlation between the two. Figure 4.11 plots the two values against each other, with each point representing one dataset. As the number of bursty features in a dataset increases, the prediction error seems to decrease. However, it is not the case that all datasets with low burstiness have high prediction error, just that the datasets that do have higher prediction error

also have lower amounts of burstiness. It may be easier to cluster datasets that are more bursty.

4.3 Clustering with EDCM Mixtures

The EDCM is a new probability distribution approximation and has not been used to cluster program trace data for SimPoint before. We examine how effective the EM algorithm is in training an EDCM mixture model and how different additions to the algorithm, such as random restarts and deterministic annealing, affect its performance.

4.3.1 Rates of Convergence

The Expectation Maximization algorithm is fairly straightforward. There are, however, a few points of uncertainty concerning the rate at which it converges. The first is how many iterations are required to converge on a value for s using fixed point iteration (i). The second is how many iterations the EM algorithm itself takes to converge on a solution (e). Both these values factor directly into the algorithm's running time. We would like to see empirically what the values for e and i are, since it is not clear what a theoretical upper bound on either would be.

Recall that the maximization step of the EM algorithm involves computing values for β, the parameters of out mixture model. However, in order to do so, we must first compute s using some iterative method (since the solution to s depends on itself). One interesting question is how many iterations it takes to converge on a value of s. We also wish to know how the number of iterations needed to compute s changes with varying initial values of s, the data involved, and the number of clusters k. For these experiments, we clustered each of the datasets using the EDCM. For each dataset, we varied $k = \{2, 4, 6, 8, 10\}$ clusters without random restarts or deter-

Table 4.1. Table of statistics for iterations needed to compute s

	Mean	Std.	Number of Iterations					
Initial value	Iters	Dev.	5	6	7	8	9	10
$0 \leq s \leq 1$	8.22	0.42	0	0	0	77.60%	22.3%	$< 0.1\%$
$0 \leq s \leq 100$	7.51	0.68	0.15%	3.71%	47.46%	42.49%	6.40%	0

ministic annealing, simply fitting a model to the data and counting the number of iterations to compute s each time it was calculated.

For the first set of experiments, we initialized s to a random number $0 \leq s \leq 1$. We are only picking a starting point for a fixed point iteration and not the final value for s, so it is not necessary to choose an $s \geq D$ as we did for the Pólya urn experiments above. Additionally, because our β values (the parameters of the EDCM, analogous to initial counts of each color in a Pólya urn) tend to be $\ll 1$, we find that s is often $\ll D$ by several orders of magnitude.

Clustering every dataset for the values of k listed above, we counted the number of iterations needed for convergence in the fixed point iteration every time s was calculated. s was calculated a total of 2,460 times with a minimum of 8 iterations and a maximum of 10 iterations until convergence. The mean number of iterations needed for convergence was 8.22, with a standard deviation of 0.42. Of all 2,460 computations of s, 77.6% took 8 iterations to converge, 22.3% took 9 iterations to converge, and $< 0.1\%$ took 10 iterations to converge.

We experiment with modifying the initial value of s. We multiply our initial random value for s by 100, increasing it by two orders of magnitude. We run the same experiment, looking for differences in the number of iterations needed to converge on a final value for different initial values of s. There is not a significant difference between the number of iterations needed for the two initial values for s. In fact, there is almost no difference. The mean number of iterations for this method was 7.51, with a standard deviation of 0.68. Out of 2,048 calculations of s, 0.15% took 5 iterations,

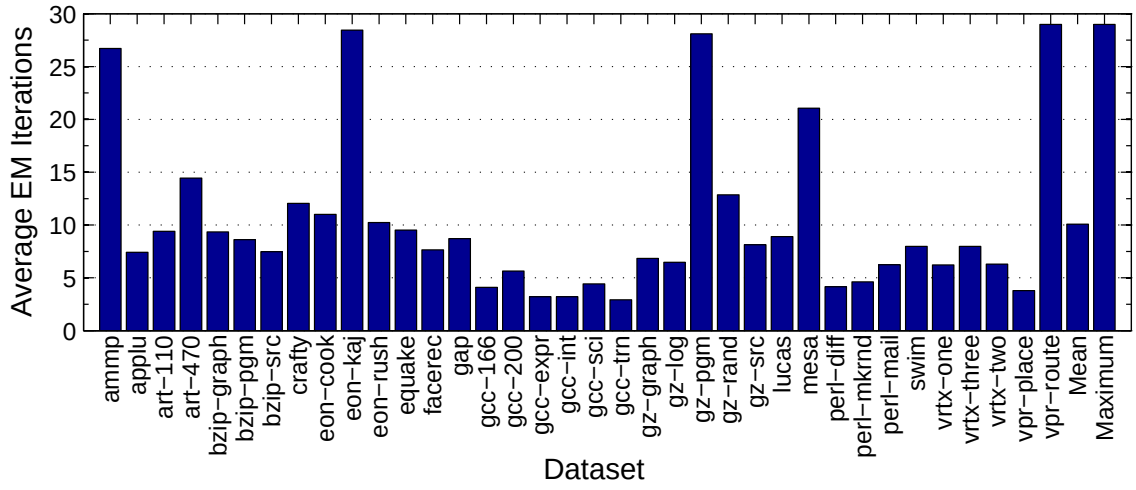


Figure 4.12: EM iterations per benchmark. We examine the average number of iterations needed to converge the EM algorithm for every run of every value of k per dataset. The average number of iterations is very low, around 10, with only a handful of datasets requiring more iterations for convergence.

3.71% took 6 iterations, 47.46% took 7 iterations, 42.49% took 8 iterations, and 6.40% took 9 iterations for convergence. Table 4.1 compares the number of iterations needed to converge on a value for s for both initial values.

The performance of the EM algorithm also depends directly on the number of iterations the EM algorithm must step through in order to converge on a set of cluster assignments and parameters. We used data from our actual clustering runs, where for every dataset, we used random restarts and a linear search through the values of k using the AIC to find the best clustering. Each time we trained a model (for a certain set of examples, value of k, and random restart) we counted the number of iterations the EM algorithm took to converge.

Figure 4.12 shows the average number of EM iterations needed for convergence per dataset. We see that on average, only around ten iterations are needed to converge on a set of parameters and cluster assignments for an EDCM mixture model. These averages are across all k, so for models of varying sizes, some with as many as 15–20

clusters, only an average of ten iterations are needed. One interesting point to note is that the datasets with the highest number of dimensions, such as `gcc`, took the fewest number of iterations, on average, to converge. To explore a possible relationship between the number of dimensions in the data and the number of iterations needed for the EM algorithm to converge, we plotted the two values against each other for each dataset, shown in Figure 4.13. We see that datasets with a high number of dimensions only undergo a few iterations of the EM algorithm. The EM algorithm must fit EDCM parameters to each dimension of the data. With such a large number of dimensions, the parameter space to explore is very large and it is easy for the EM algorithm to fall into a poor locally maximum log likelihood. This makes the EM algorithm terminate quickly even though the choice of parameters may be globally poor in terms of model log likelihood.

Figure 4.14 shows a slightly different perspective of the same data. Rather than average the number of iterations needed to converge the EM algorithm per dataset, we examine the average number of iterations per value of k. Each bar on the graph plots the average number of iterations needed to converge the EM algorithm on any model with k clusters. The number of iterations needed to fit data into a certain number of clusters starts to rise with k. This is because our models are more complex (more EDCM distributions in the mixture) and need more time to converge on a good local maximum. However, the more complex datasets, which tend to be the datasets with a larger number of dimensions, require more k to fit accurately than simpler datasets. As we have seen, more dimensions results in fewer iterations of the EM algorithm because we tend to fall into poor local maxima.

4.3.2 Deterministic Annealing

We examine the effectiveness of using deterministic annealing with different temperatures to help address the problems of local optima using the EM algorithm.

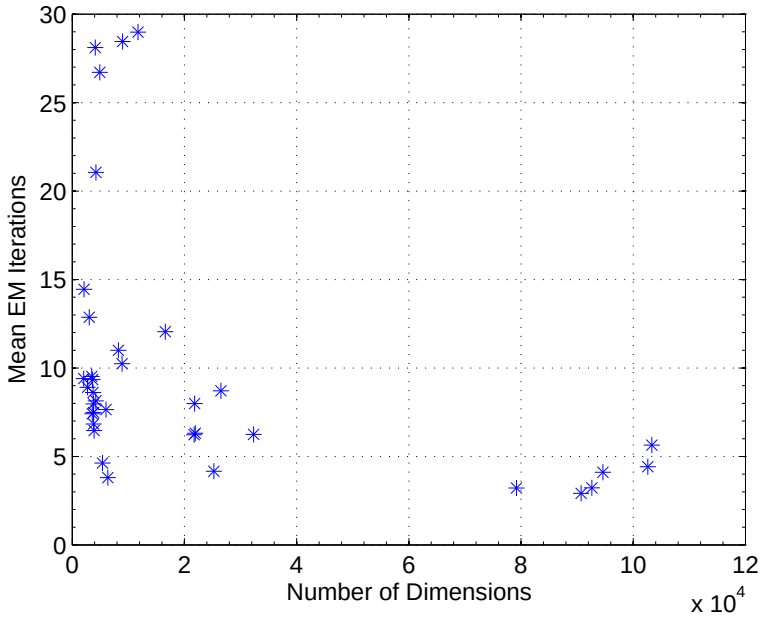


Figure 4.13: Number of EM iterations versus number of dimensions. For each dataset, we plot the average number of EM iterations versus the number of dimensions. We see a correlation between high numbers of dimensions and low iterations needed to converge for the EM algorithm. This is most likely because with such a large parameter space to explore in high dimension (we fit a parameter for each dimension), there are a multitude of local maxima which the EM algorithm can find quickly.

For the following set of experiments, we run three sets of trials. In each trial, we cluster a subset of the datasets with the deterministic annealing EM (DAEM) algorithm, using 5 random restarts to choose a model for each k, and searching through the clusterings using the BIC score to choose a model (as described below). For each trial, we use a different set of temperatures—a set of high temperatures $\{25, 5, 1\}$ (Elkan 2006), a set of low temperatures $\{2, 1.5, 1.1, 1\}$, and no annealing (temperature of 1). For annealing, we start with the higher temperatures and slowly cool distribution of log likelihoods down to a temperature of 1. At a temperature of 1, there is no change made to the distribution and annealing has no effect on clustering.

Figures 4.15 and 4.16 give the results for clustering with the DAEM algorithm. We see that, on average, annealing gives us no significant decrease in percent error compared to clustering without annealing, and that using the set of greater tem-

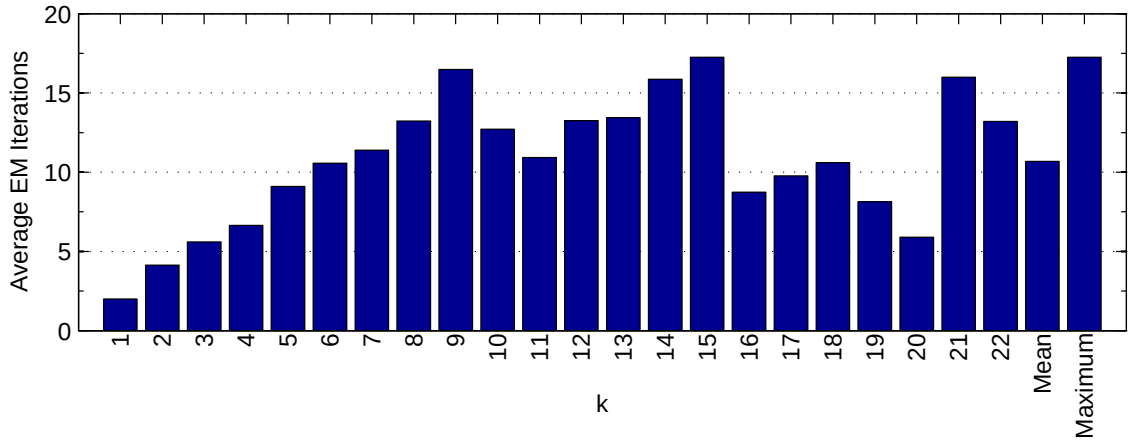


Figure 4.14: Average EM iterations per k. The number of iterations needed for convergence of the EM algorithm are averaged per value of k, regardless of dataset. This gives us more insight into how much work must be done by the EM algorithm as we increase the complexity of the models. There is some increase for smaller values of k due to the increasing complexity of the models. However, the higher values of k are fit to the more complex datasets, which tend to have the higher numbers of dimensions (see Figure 4.13) and converge quickly.

peratures give a better prediction error than using lower temperatures. Additionally, using annealing also results in choosing more clusters (higher values of k), on average. Thus, because we see a favoring of more complex models without significant benefits in percent error, we choose not to use annealing in our implementation of the EM algorithm.

When examining why the DAEM did not perform as well as we had hoped, we discovered that at high temperatures, when the distribution of log likelihoods was made more similar, the DAEM algorithm converged many of the clusters in the mixture to very similar sets of parameters. As the distribution cooled, the clusters were not able to separate themselves. Therefore, although the DAEM tended to select a higher value for k than the EM without annealing, the extra clusters might not have actually helped because several clusters were fit to the same data. As part of our

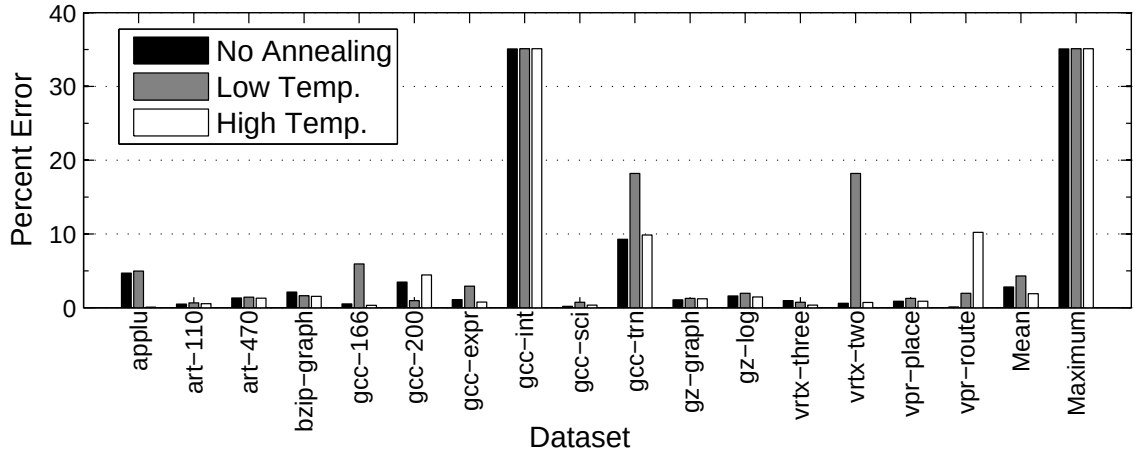


Figure 4.15: Deterministic Annealing EM (DAEM) algorithm—Percent error. We use the DAEM algorithm to reduce the effects of poor initializations leading to local optima. Only a select subset of datasets are clustered to save on computation time. Using the BIC to choose the number of clusters, we see that annealing, both with high $\{25, 5, 1\}$ and low $\{2.0, 1.5, 1.1, 1.0\}$ temperatures, gives no significant advantage over not using annealing with regards to percent error.

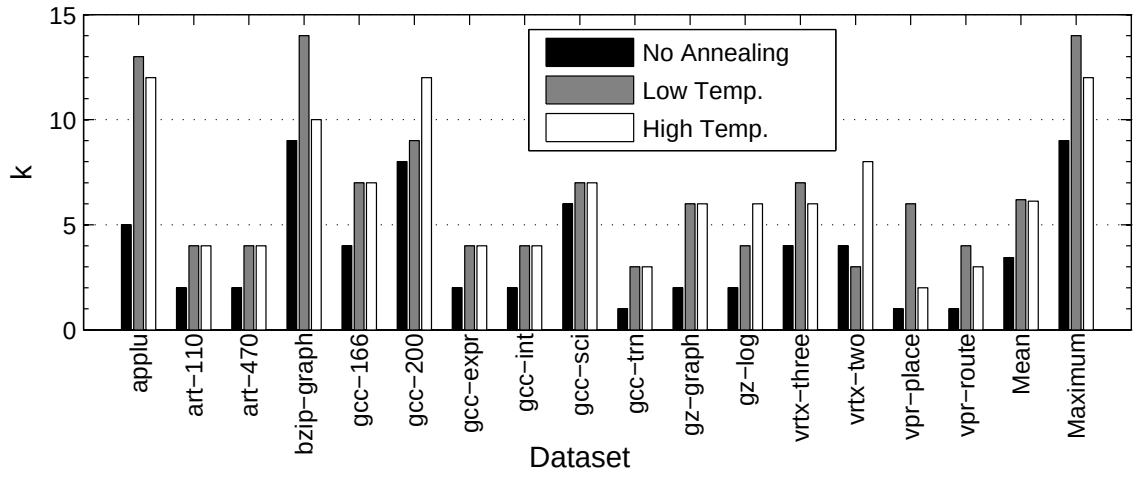


Figure 4.16: Deterministic Annealing EM (DAEM) algorithm—Choice of k. In the same experiment as Figure 4.15, we observe that using the DAEM results in choosing a larger number of clusters. Deterministic annealing may actually hurt the performance of the EM algorithm for the SimPoint domain, leading us to choose models with higher k without significant improvement in prediction error.

future work, we will investigate methods to separate the clusters as the temperatures of the DAEM algorithm are lowered.

4.3.3 Running Time

We have described the algorithmic complexity of the EM algorithm and explored the empirical values of the number of iterations needed to converge on a value for s and the EM algorithm itself. Here, we explore empirical running times of the EM and k-means algorithm for clustering the same data.

We use the EDCM to cluster all thirty six datasets, using five random restarts per value k and the AIC to automatically select the best k. On a dual-processor, hyper-threaded 3 GHz Intel Pentium 4 computer with 2 GiB of RAM, this takes approximately 25 hours of CPU time. Clustering the same data with the k-means algorithm, using five random restarts and the BIC to select k, takes 757 seconds on the same machine, about 128 times faster than the EDCM.

The discrepancy in running times occurs for several reasons. First, SimPoint uses random projections to reduce the data to fifteen dimensions before handing it to k-means, whereas with the EDCM we consider the data in its full dimension. The amount of data clustered by k-means is approximately 15 $\times$ average number of intervals per dataset $\times$ number of datasets. The EDCM, even though it uses the data in its full dimension, deals with data that is very sparse. We count up the number of non-zero elements in all the program trace datasets clustered by the EDCM and find it clusters about 132 times more raw data than k-means. Interestingly, this gives about a one to one correspondence between running times and the amount of data clustered. Second, SimPoint is a highly optimized application written in C++, while the our implementation of the EM algorithm to train the EDCM is unoptimized and written in MATLAB. In light of these facts, we see the EDCM's performance is actually quite good.

It is important to note that the amount of time spent clustering is, in the end, not of dire importance to researchers using SimPoint. More time spent clustering up front is worth the effort if it means more accurate and quicker partial simulations later, especially because the time needed to perform a design space exploration over processor configurations is much greater than the time needed to cluster the datasets. Clustering is performed once on all the program trace intervals to select a set of simulation points. These same simulation points are used to predict performance on all of the parameter combinations tested in the design space exploration. The EDCM, as we will explore shortly, also chooses fewer clusters to represent the program traces than k-means does. Fewer clusters means fewer simulation points and results in a direct savings in the amount of simulation time needed.

4.4 Choosing a Model

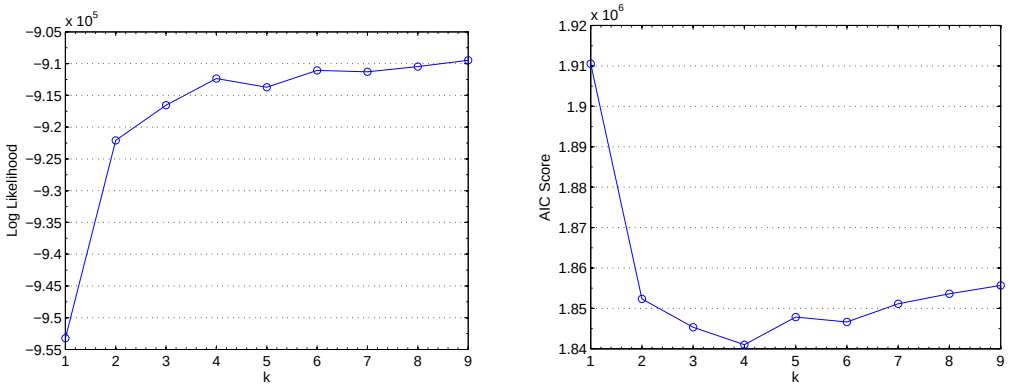
An important point of discussion with any clustering algorithm is not just how it can be applied to a new domain of data, as we have discussed, but how we can use it without *a priori* knowledge about the data. A difficult problem to address when using a clustering algorithm is how one chooses the number of clusters k. We have examined two methods for selecting a model, the Akaike and Bayesian information criteria. Here, we show how they perform choosing the value k for the EDCM mixture model.

4.4.1 AIC and BIC

Recall the formula for the AIC,

$$AIC = -2\mathcal{L} + 2p. \tag{4.3}$$

Likelihoods for a model are necessarily positive and ≤ 1 (it makes no sense for a probability to be negative), usually $\ll 1$, meaning the log likelihoods are negative numbers. Greater likelihoods mean greater (less negative and closer to 0) log likeli-



(a) Log likelihoods for `art-110`

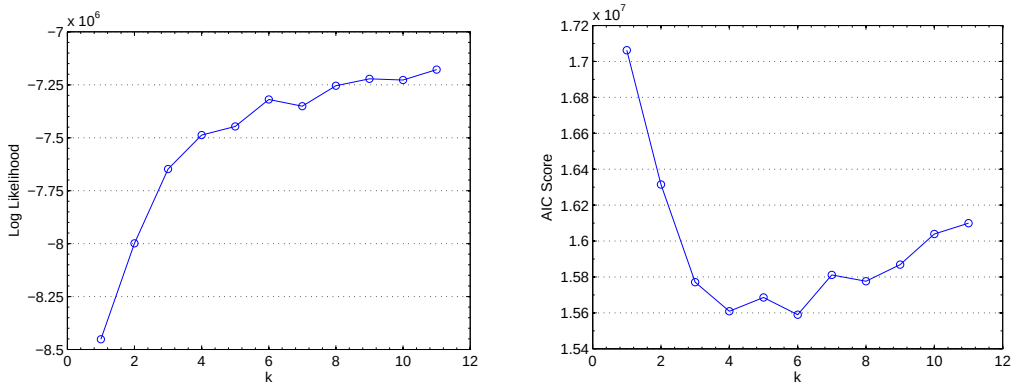
(b) AIC scores for `art-110`

Figure 4.17: Log likelihood and AIC for `art-110`. The horizontal axes plot increasing values of k, with the vertical plotting the log likelihood and AIC score of the model. (a) In general, increasing k means strictly increasing log likelihoods, with occasional decreases for bad local optima. (b) Increasing log likelihoods drive the AIC score down until the point where the penalty outweighs any increase in $\mathcal{L}$.

hoods and therefore smaller terms of $-2\mathcal{L}$, so we want to minimize the AIC score. To simplify our discussion, we examine two of the datasets (rather than all 36), choosing one of the easier (`art-110`) and one of the harder (`gcc-int`) to cluster. Figure 4.17 shows a comparison of the log likelihoods of `art-110` with the AIC scores. We can see that the log likelihood is usually increasing with increasing k. The AIC score drops quickly (we want low AIC scores) as the log likelihood increases. Soon, however, the penalty on the model complexity ($2p$) drives the AIC score up. Figure 4.18 shows the same data for the `gcc-int` dataset. Figure 4.19 gives two graphs, one per dataset, that compare the scaled AIC score with the percent prediction error for each value of k on both datasets, showing how our choice of k affects our prediction error. The prediction error is not available to the clustering algorithm while the AIC score space is being explored.

We also experiment using the BIC,

$$BIC = \mathcal{L} - \frac{1}{2} p \log N. \tag{4.4}$$

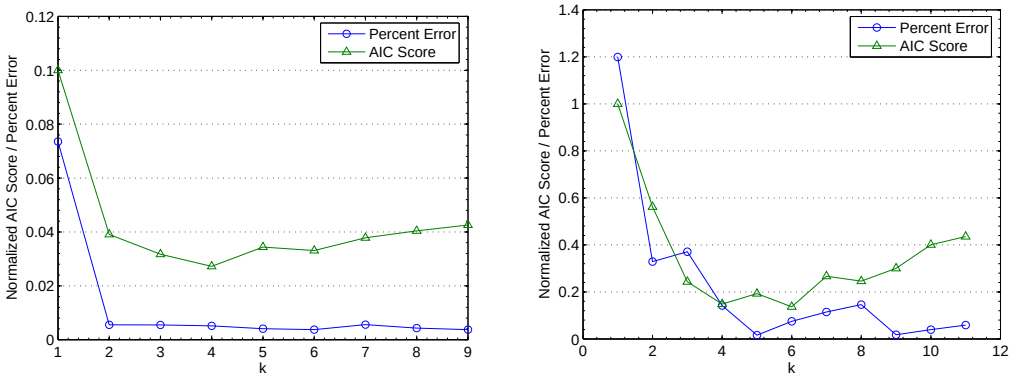


(a) Log likelihoods for `gcc-int` (b) AIC scores for `gcc-int`

Figure 4.18: Log likelihood and AIC for `gcc-int`. The horizontal axes plot increasing values of k, with the vertical plotting the log likelihood and AIC score of the model. (a) In general, increasing k means strictly increasing log likelihoods, with occasional decreases for bad local optima. (b) Increasing log likelihoods drive the AIC score down until the point where the penalty outweighs any increase in $\mathcal{L}$.

Similarly to the AIC, the likelihood of a model is a positive value < 1, meaning the log likelihood $\mathcal{L}$ is a negative value. Larger likelihoods mean less negative (closer to zero) log likelihoods, and subtracting off the penalty on model complexity ($\frac{1}{2}p \log N$) drives the value lower. Therefore, we want to maximize the BIC. We examine the same two datasets as we do with the AIC. Figure 4.20 gives the BIC scores for both datasets. We don't show the log likelihoods for either dataset as we did with the AIC, because the log likelihoods will be the same for both the AIC and BIC. Figure 4.21 shows the scaled BIC scores versus the prediction error for both datasets. For `gcc-int` (Figure 4.21(b)) we see how the BIC penalizes our models too harshly, not allowing us to select a higher value for k that would give significantly less prediction error.

We have mentioned the difficulty in clustering `gcc-int` while stating `art-110` is a relatively simple dataset to cluster. As a visual representation, we randomly project the data in the two datasets to two dimensions. We can see a small number of fairly distinct clusters of examples in `art-110`, Figure 4.22(a). Thus, we can easily express



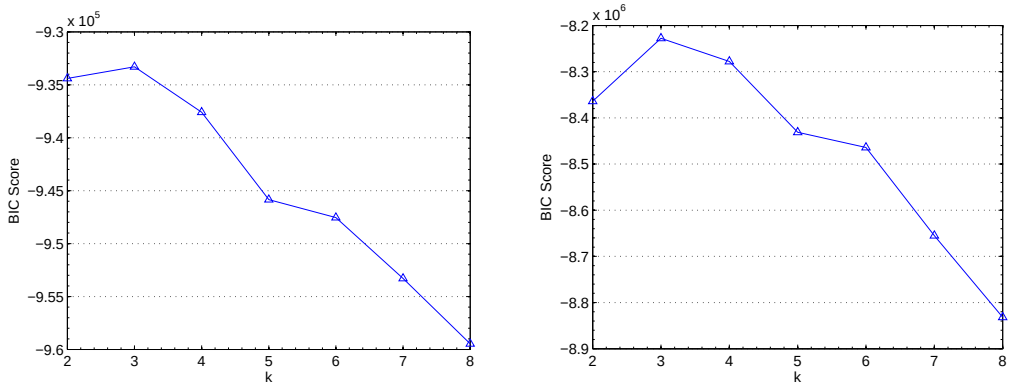
(a) AIC vs Percent error for `art-110`

(b) AIC vs Percent error for `gcc-int`

Figure 4.19: AIC versus percent error. We plot scaled AIC scores versus percent error for the different values of k. We do not know the percent error while clustering, so we cannot use it as a basis for selecting a model. However, it is useful to examine the percent error of a model compared to its AIC score so we can determine the quality of our choice. (a) `art-110` is such a simple dataset to cluster that increasing k above 2 does not improve the percent error. However, the AIC chooses $k = 4$. (b) `gcc-int` is a much more complex dataset. We see that with our choice for k, based on the minimum AIC (at $k = 6$), the percent error is not quite at its minimum. Here, the lowest prediction error was given by a lower k that had a higher AIC score (and therefore did not get selected, most likely a case of bad local optima).

`art-110` with small values of k, and adding more does not significantly improve the prediction error. However, with `gcc-int`, Figure 4.22(b), we see much more complex data and need far more clusters to accurately capture similarities. There is not a clear separation of the data. Thus, the BIC's high penalty on model complexity hurts the prediction error because it does not allow us to use enough clusters to capture all the data. The AIC, on the other hand, allows us to choose more clusters and gives better prediction error.

The way in which the spaces of AIC and BIC scores are searched for the best model (value of k) is an important point of discussion. Empirically, we see that we achieve local optima quickly (for relatively low values of k) in the curves of the scores. Because program trace data exists in such a high number of dimensions, and because the EDCM mixture model uses the data without dimension reduction, the complexity

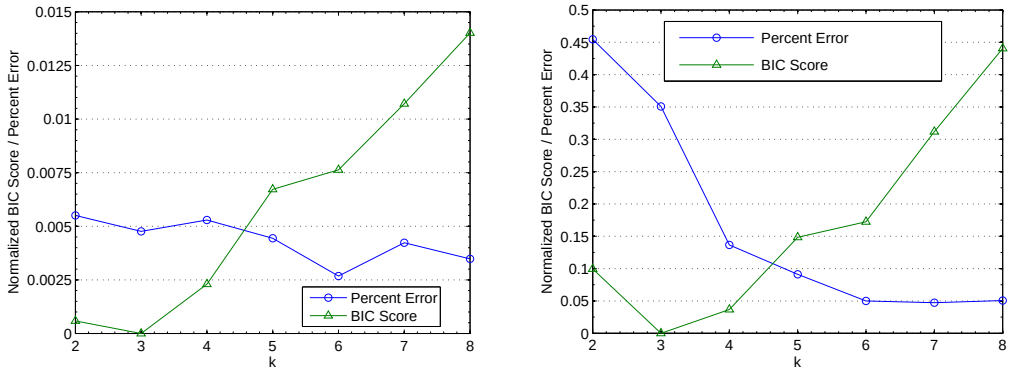


(a) BIC scores for `art-110`

(b) BIC scores for `gcc-int`

Figure 4.20: BIC scores for `art-110` and `gcc-int`. The BIC scores for the two datasets are surprisingly similar. Notice how quickly the scores drop off after peaking. This is due to the larger penalty on p imposed by the BIC (compared to the AIC, whose behavior is more gradual).

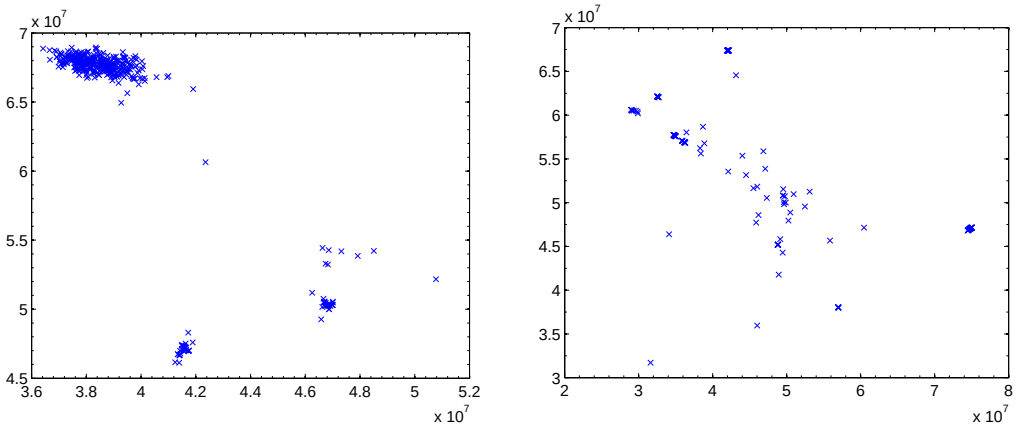
of our models is so great that it quickly overcomes strictly increasing log likelihoods when k is increased. Starting at some initial value ($k = 1$ for most of our clustering runs), we step through increasing k, computing the AIC/BIC score for each model. When we see a new minimum/maximum in the scores, we increase k a certain number of times (in the case of our experiments, 5, chosen based on empirical observation) to make sure there is not a lower minimum or higher maximum in the immediate vicinity. If there is not, we can be confident, based on empirical observation, that there will not be one any farther out. This may not be the case for all domains. For example, in SimPoint, with the projected data clustered by the k-means algorithm, the BIC scores are strictly increasing for increasing k until k gets very large. Searching the space of scores described above for the projected data yields a very large choice of k, larger than most practitioners are willing to accept. So instead of a linear search for a local maximum, a method of binary search is used as a heuristic to select k (Sherwood *et al.* 2002).



(a) BIC vs percent Error for `art-110`

(b) BIC vs percent Error for `gcc-int`

Figure 4.21: BIC versus percent error. Similarly to Figure 4.19, we scale the BIC score curve and plot it with the percent error curve for the values of k in order to examine the relationship between our choice of k and the error. (a) Again, `art-110` is such an easy dataset to cluster that any choice of k results in $\ll 1\%$ error. (b) For `gcc-int`, we see the BIC penalizes increasing k too harshly. If we would have chosen just a slightly higher k, say around $k = 5$ instead of $k = 3$, we could have achieved $< 10\%$ prediction error. Hence the difficulty with unsupervised learning, we aren't sure what the best answer is while we are deciding on it.



(a) Projection of `art-110`

(b) Projection of `gcc-int`

Figure 4.22: Projection of `art-110` and `gcc-int` to two dimensions. We see how simple the `art-110` dataset is compared to `gcc-int`. Thus, we can easily express the data in `art-110` with few clusters. However, the complexity of `gcc-int` drives a need for higher k. The BIC penalizes higher k such that we are not able to select a good model for `gcc-int` (Figure 4.21(b)).

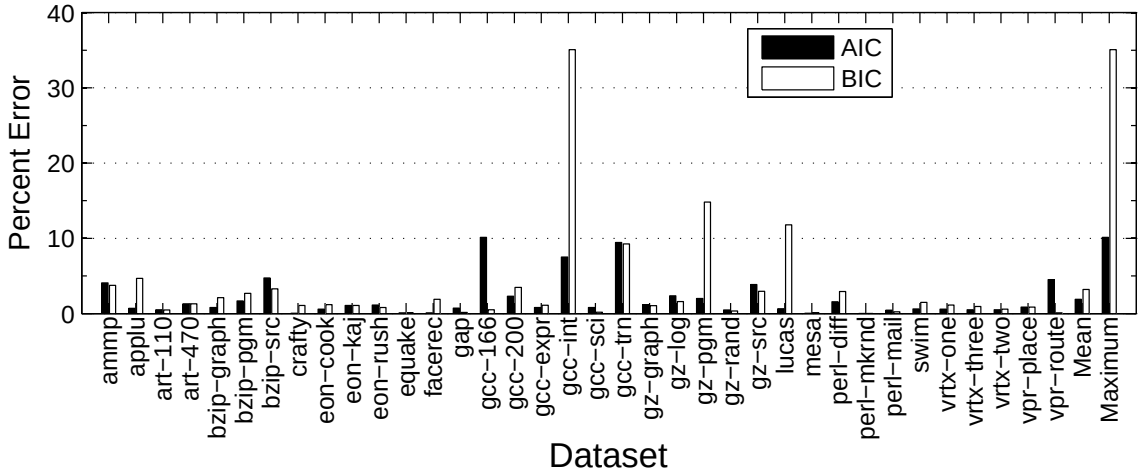


Figure 4.23: Model selection with the EDCM—Prediction error. Graph comparing the prediction errors for the EDCM models selected by the AIC and BIC. On average, both methods perform rather well, with $\approx$ 2% and 3% error. However, the BIC performs poorly on several datasets, especially so on `gcc-int`. This is because it penalizes the log likelihood too harshly (Figure 4.21(b)).

We compare the error and value of k for the models selected by the AIC and BIC scores for the EDCM mixture model. We see that the AIC chooses models that result in lower error, on average, than the BIC (Figure 4.23). However, the BIC chooses models with smaller values of k (Figure 4.24), at the cost of higher error. There is a trade-off that must be made between the error of a model and its complexity. The BIC chooses models with lower k because it penalizes the complexity of a model more than the AIC does (the BIC's penalty on p includes a $\log N$ factor which is not present in the AIC's penalty on p). However, because the AIC allows the model to have a larger value k, it is more flexible in the way it can fit the data and results in lower prediction error.

4.4.2 Comparison with SimPoint

As noted, SimPoint uses the BIC to choose values for k for the k-means algorithm. We compare SimPoint (which uses the BIC) with the EDCM (which uses the AIC). Figures 4.25 and 4.26 show the comparisons of the values of k chosen by the two

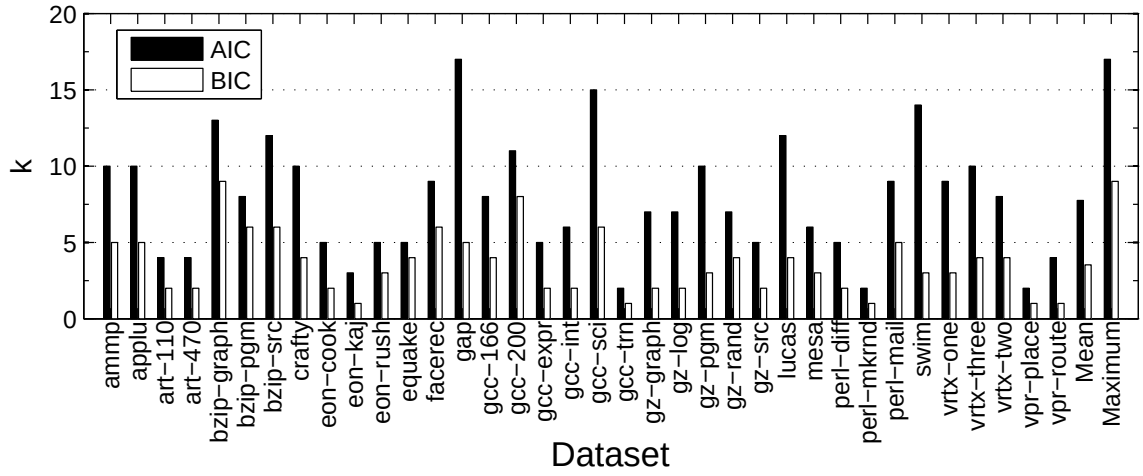


Figure 4.24: Model selection with the EDCM—Choice of k. Graph comparing the value k selected for each dataset by the AIC and BIC. We see that, on average, the BIC chooses about half as many clusters as the AIC (because the BIC penalizes the complexity of the model more than the AIC). However, as we have seen, this comes at the cost of increased prediction error.

methods and the percent errors of the resultant models. The EDCM performs within 1% of the percent error from SimPoint's clusterings. However, the EDCM chooses half the number of clusters as SimPoint. This is quite a significant amount of savings achieved with still a very low error rate. Even the maximum error of the EDCM, about ten percent, gives sufficiently low prediction error for comparing different processor designs. Remember, the ultimate purpose of SimPoint is not how accurate we can make the clusterings, but how much we can speed up the process of getting reliable results for comparing new processor designs. Fewer clusters (lower k) means fewer simulation points and less time performing full simulations on them. Running full simulations on the entire suite of SPEC CPU2000 benchmarks takes over a year of CPU time. Running full simulations on the simulation points selected by k-means takes about 6 days, whereas full simulations on the simulation points selected by the EDCM take 3 days of CPU time to run.

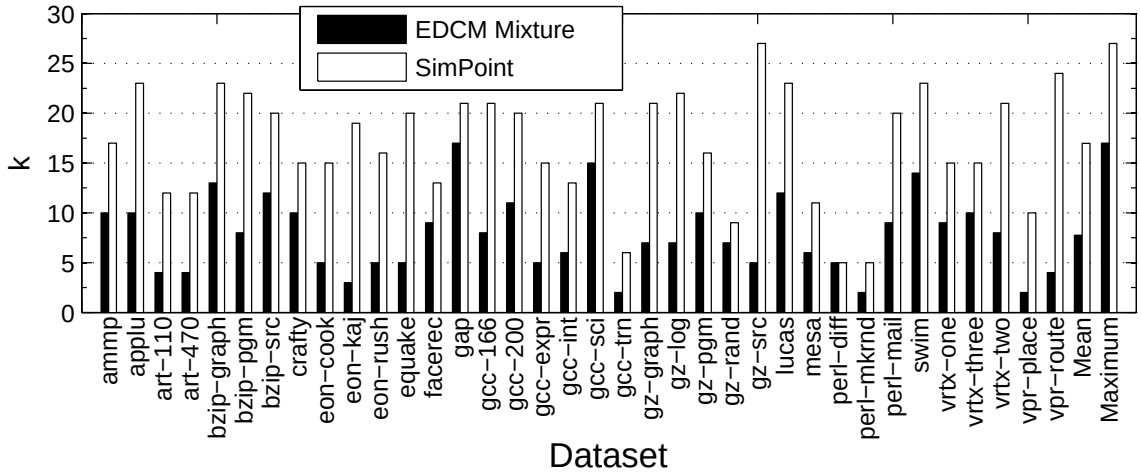


Figure 4.25: SimPoint (BIC) versus EDCM (AIC)—Choice of k. We compare the values selected for k via the BIC and AIC for SimPoint and the EDCM, respectively. We see that on average, the EDCM chooses half the number of clusters as SimPoint.

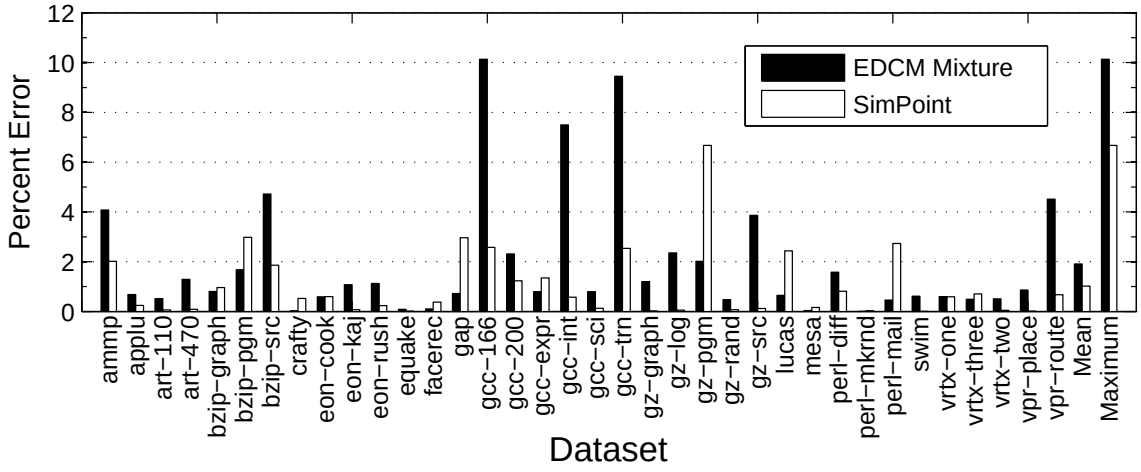


Figure 4.26: SimPoint (BIC) versus EDCM (AIC)—Percent error. We compare the prediction errors of the selected models for EDCM and SimPoint. We see that, on average, SimPoint predicts with about half the error as EDCM (1% compared to 2%). However, this small change in prediction error comes at the savings in number of clusters needed by EDCM. 2% error is still low enough for comparing processor designs with each other.

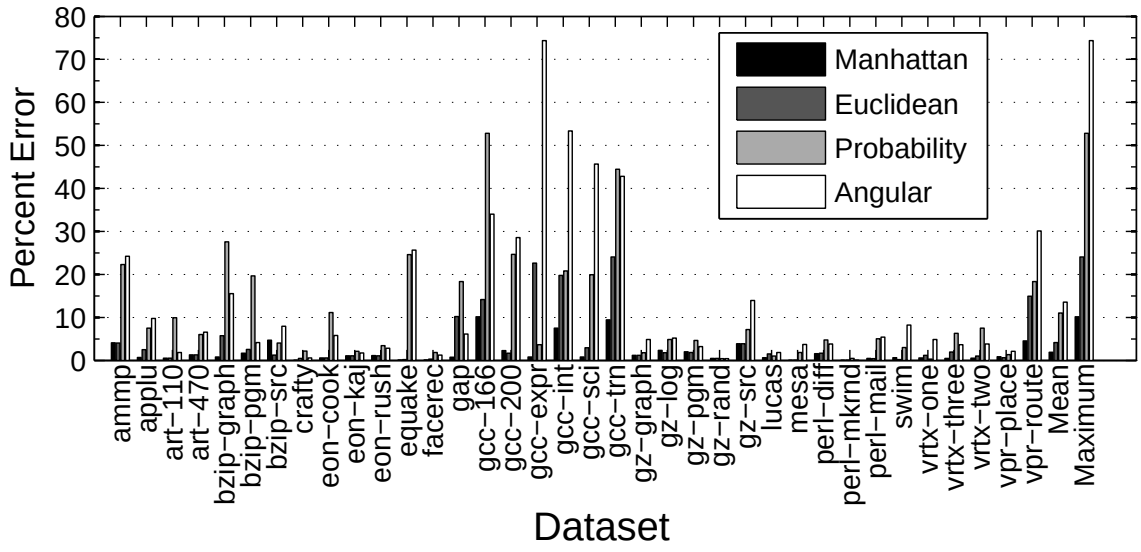


Figure 4.27: Comparison of all four methods for selecting cluster representatives. Overall, using the angular measure performs the worst in terms of prediction error, followed closely by using probabilities. The best methods were the two distance metrics, with about half the error, on average, as the other methods.

4.5 Choosing Cluster Representatives

After we've clustered the data and chosen a suitable value of k using an automatic method, the next difficulty is choosing cluster representatives. The choice of simulation points is very important to the SimPoint application. If we are not able to pick simulation points that are representative of the overall behaviors of the dataset we are performing clustering on, our prediction of performance will be off and we will not have gained anything by clustering. If however, we are able to accurately cluster the data and choose good simulation points, out predictions will be very accurate.

4.5.1 Methods for Choosing Simulation Points

We discussed several methods for choosing cluster representatives. For choosing a simulation point s_j for cluster c_j, we could consider the point with the highest probability.

$$s_j = \arg\max_{x_i \in c_j} \Pr(x_i|\beta_j) \tag{4.5}$$

We could also think of the examples as vectors in a geometric space. Normalizing the basic block vectors to the unit hypersphere,

$$x'_i = \frac{x_i}{\sqrt{\sum_{d=1}^{D} x_{id}^2}}, \tag{4.6}$$

we normalized β'_j in a similar fashion and choose as a simulation point the example satisfying

$$s_j = \arg\max_{x'_i \in c_j} \langle x'_i \cdot \beta'_j \rangle, \tag{4.7}$$

the x'_i with the smallest angle from β_j. Or, we may compute the mean (μ_j) for each cluster and choose the interval that is the closest using some distance metric L_p.

$$s_j = \arg\min_{x_i \in c_j} L_p(x_i, \mu_j) \tag{4.8}$$

Figure 4.27 compares all the methods for selecting simulation points. We used the EM algorithm with no annealing, 5 random restarts, and the AIC to choose the value for k. From the selected models for each dataset, we chose cluster representatives using each method and compared the prediction errors. We see that the angular measure performs with the highest percent error on average, with the probability method performing nearly as poorly. Figure 4.28 compares the probability and angular method, both methods with greater than ten percent error overall. In the worst case, both methods perform with over fifty percent error. The two distance metrics, Euclidean (L_2) and Manhattan (L_1), perform the best of all the methods. Figure 4.29 compares the distance metrics, showing that both methods achieve less than five percent error on average. However, Euclidean distance has a maximum error of almost twenty five percent, with six datasets having over ten percent error. The average error for Manhattan distance is about two percent, with only one dataset having just barely more than ten percent error.

One of the most interesting results when comparing the methods for selecting simulation points was the fact that the probability measure performed the worst. We

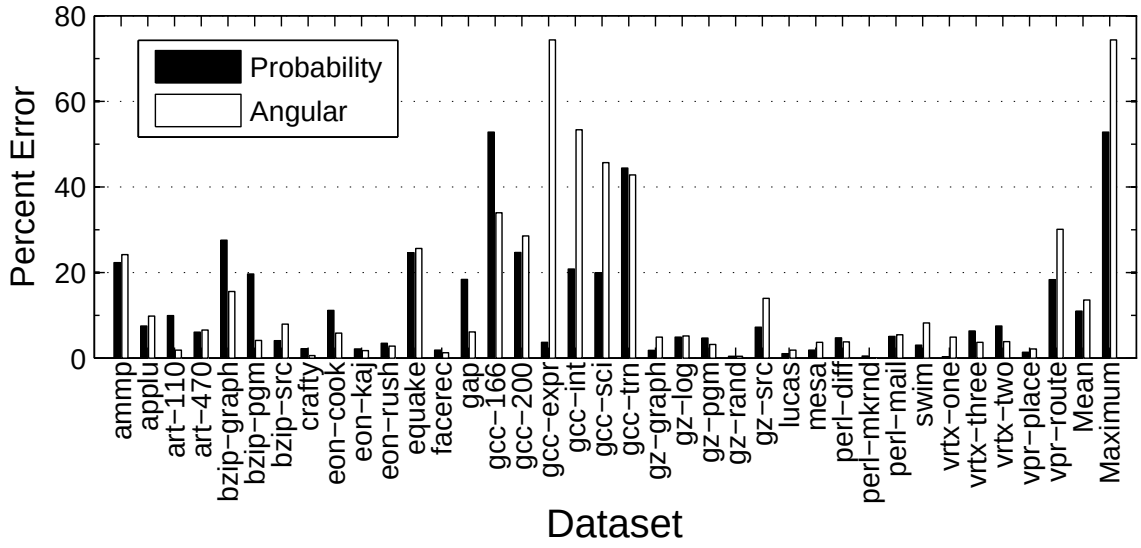


Figure 4.28: Comparison of the probability and angular based methods for selecting simulation points. Both methods have a mean error of over ten percent, with many datasets having over twenty percent error. Using probabilities did not work as well as we would have liked, due in part to burstiness.

had hoped, since the EDCM assigns probabilities to each example for each cluster, that we would be able to select the interval with the highest probability as a cluster representative. We concluded that probabilities did not work because of burstiness, exactly the problem we use the EDCM to help address. The problem is that certain intervals within the clusters are very bursty and get assigned higher probabilities than the other, more uniform examples. However, the bursty behavior of the single interval with the highest probability may not be representative of the cluster as a whole. In fact, more than likely, the overall behavior is going to be more uniform since, in the scope of one interval of program execution, a program will be doing more than one thing, even if it is just a small set of related tasks. It is our belief that if a method is devised to split the intervals based on boundaries of common behavior, rather than the current methods of using fixed length intervals, using assigned probabilities may end up being a viable method for choosing representatives.

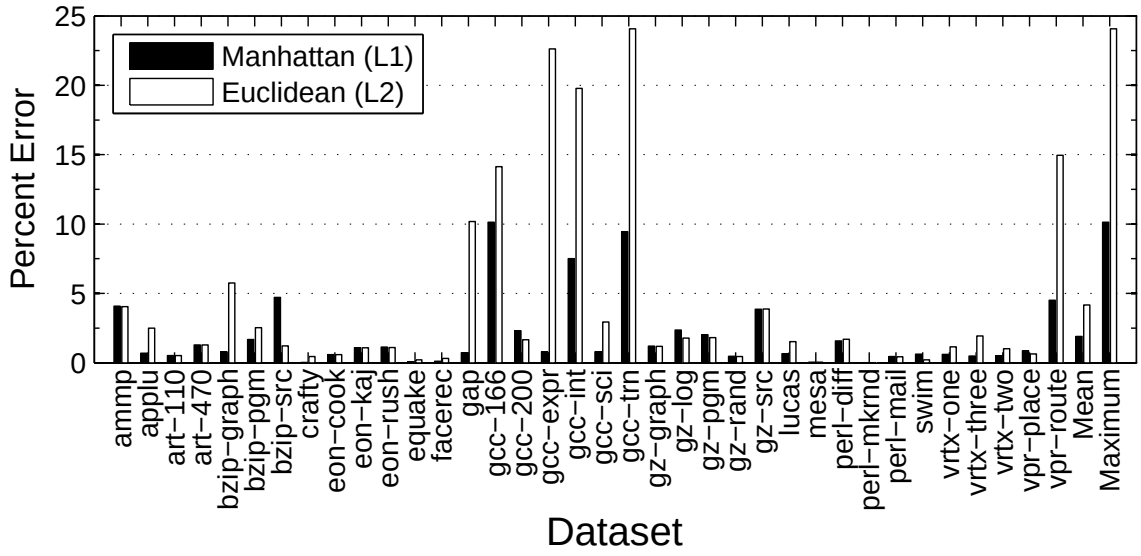


Figure 4.29: Comparison of using distance metrics to select cluster representatives. Both methods achieve less than five percent error on average. However, Euclidean distance has a maximum error of almost twenty five percent, with six datasets having over ten percent error. The average error for Manhattan distance is about two percent, with only one dataset having more than ten percent error.

Very rarely do SimPoint and EDCM choose the same intervals as cluster representatives. SimPoint chooses simulation points using Euclidean distance from cluster means. Compared to the EDCM, using both Euclidean and Manhattan distance from the center mean, the two methods, on average, have less than one simulation point in common per dataset. This is with the EDCM having an average of around eight clusters per model and SimPoint having an average of about seventeen clusters per model.

4.5.2 Motivations for Different Methods

We want to compare different methods of choosing simulation points using more than a "what worked best" approach. We would like to have some basis for choosing Manhattan distance from cluster means as our method for choosing cluster representatives. To try and get some understanding why the Manhattan distance from the mean works best, we set up an experiment.

For each dataset, we find the mean and median of all the intervals and the average of the CPI statistic for the entire dataset. We compute the L_2 (Euclidean) and L_1 (Manhattan) distances from every interval to both the mean and median. We examine the difference of an interval's CPI from the mean CPI versus the L_2 distance from the mean, and the same difference in CPI against the L_2 distance from the median. Similar examinations are made for the L_1 distance. We compute the coefficient of correlation for all four sets of data, giving the amount that one variable (difference in CPI) varies with the other (distance from the mean/median). We observe that most often, L_1 distance from the mean gives the most accurate information in terms of difference in CPI (intervals that were closest to the mean via L_1 distance had the smallest differences from the average CPI). Additionally, L_1 distance from the mean tends to have higher a coefficient of correlation than the other methods, meaning the farther we get away from the mean, the larger the difference in the CPI. Figure 4.30 shows the four combinations of distance metric and mean/median for one dataset.

In high dimension, the reliability of distance metrics tends to break down (Aggarwal *et al.* 2001). For distance metrics L_p, smaller values of p work better at defining the distance between two points in high dimension. This is why using L_1 distance is better suited to selecting cluster representatives than L_2 distance.

4.6 Analysis of Clusterings

We want to analyze the clusters produced by the EM algorithm training an EDCM mixture model. We want to know how the EDCM mixture models groups things together, how similar the items within a cluster are, etc. These are questions are concerned with actual clustering characteristics.

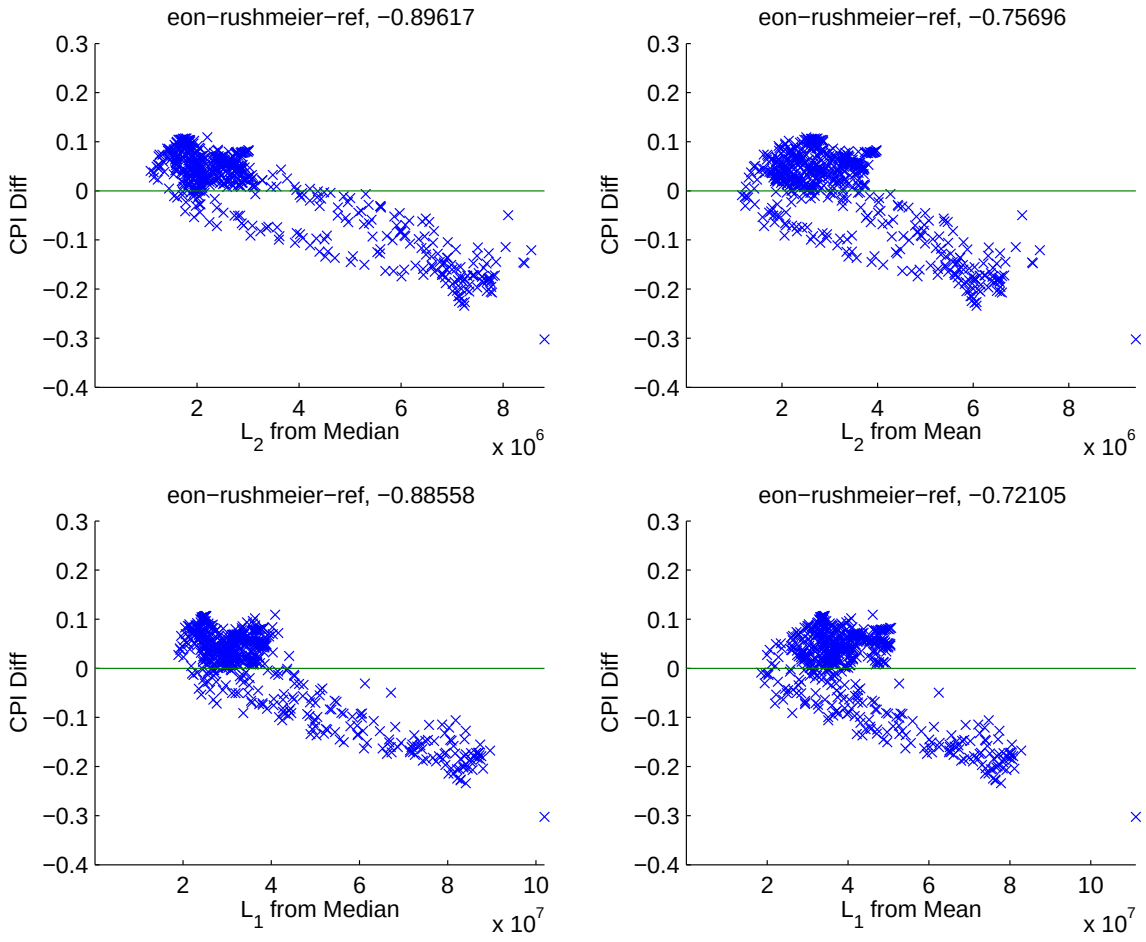


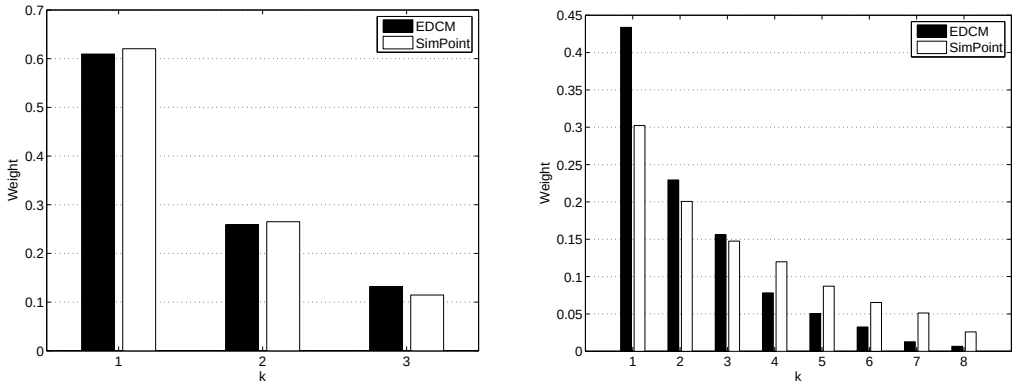
Figure 4.30: Choice of distance metrics and mean or median. We examine using the two different distance metrics, L_1 and L_2, and using the mean and the median. We compute the mean and median of every dataset and the distance (using both L_1 and L_2) of every interval to it. We also compute the average CPI for the dataset and the difference in CPI of every interval from it. For each interval, we plot the distance from the mean and median against the difference from the mean CPI. In terms of choosing the interval with the smallest difference in CPI, choosing the interval with the smallest L_1 distance from the mean is the best choice. Additionally, it gives the highest coefficient of correlation, meaning that the closer to the mean/median get, the better our prediction becomes. Here we show the different combinations of distance metrics and mean/median for one dataset, `eon-rushmeier`, giving the coefficient of correlation for each combination as well.

4.6.1 Distribution of Cluster Weights

One question worth asking is how well the EDCM mixture model spreads out cluster membership among the examples. There is no right answer for this question as it is almost completely data dependent. For one clustering method, two sets of data may have wildly different distributions of cluster weights just because the makeup of the data is different. Figure 4.31 shows the mean size of all the clusters, sorted by cluster size, across all models generated for all datasets while running clustering using the AIC for model selection.

Figure 4.31 gives the average distribution of cluster weights for two clusterings of the datasets, one with SimPoint and one with the EDCM. We see that both methods produce clusterings that have largely the same weights. However, SimPoint does tend to produce clusters that are more uniform in size, with the largest clusters being smaller than the largest EDCM clusters. This is because SimPoint uses random linear projections to reduce the number of dimensions in the data. The projected intervals may tend to look more similar to one another, resulting in clusters of more similar sizes, because there are fewer dimensions in which to express all the differences between the characteristics. Additionally, the differences that do exist in the features are muted because of the random linear projections.

We don't know what a correct distribution of cluster weights should be, especially not on average as each dataset will vary. However, we can see that the EDCM and SimPoint produce clusters with very similar weights. Thus, the EDCM does not tend to skew toward putting too much data in one cluster, or toward spreading out the data too much among the clusters. Though we do not know what the make up of each individual cluster is, we can assume that, for the most part, the clusters of the EDCM and SimPoint methods are composed of largely the same intervals (coupling the similarities between the cluster weights with the similarity in prediction error).



(a) Three Clusters (b) Eight Clusters

Figure 4.31: Comparison of cluster weights. We cluster all 36 datasets into 3 and 8 clusters using both SimPoint and the EDCM. We look at the cluster weights and compute the mean weight of the largest cluster for each dataset, the mean of the next largest cluster, etc. We see from the two graphs that both clustering methods produce clusters that are largely the same size, with SimPoint clusters tending to be just a bit smaller and more of the same size (evident in (b)). This is not surprising as the random linear projections would tend to make things look more similar (there are fewer dimensions in which to express dissimilarities between intervals).

4.6.2 Variation of Information

We use Meilă's Variation of Information (VI) metric to compare the clusterings generated by SimPoint and EDCM and see how similar they are to one another in the cluster assignments that have been made. Figure 4.32 shows the VI of every dataset along with the tighter theoretical upper bound on the VI (Meilă 2002). The upper bound on the VI between two clusterings C and C' is defined as

$$VI(C, C') \leq 2 \log K^*, \tag{4.9}$$

with $K^* = max(k, k')$ being the larger of the number of clusters in C and C', respectively. The upper bound holds for $K^* \leq \sqrt{N}$, which is the case for program trace data and the number of clusters chosen by SimPoint and the EDCM.

We see that, on average, the VI tends to be about 1/3 the maximum, with a VI of 0 meaning two clusterings share all information about one another. Thus, we conclude that while there is some difference in the clusterings of the EDCM and

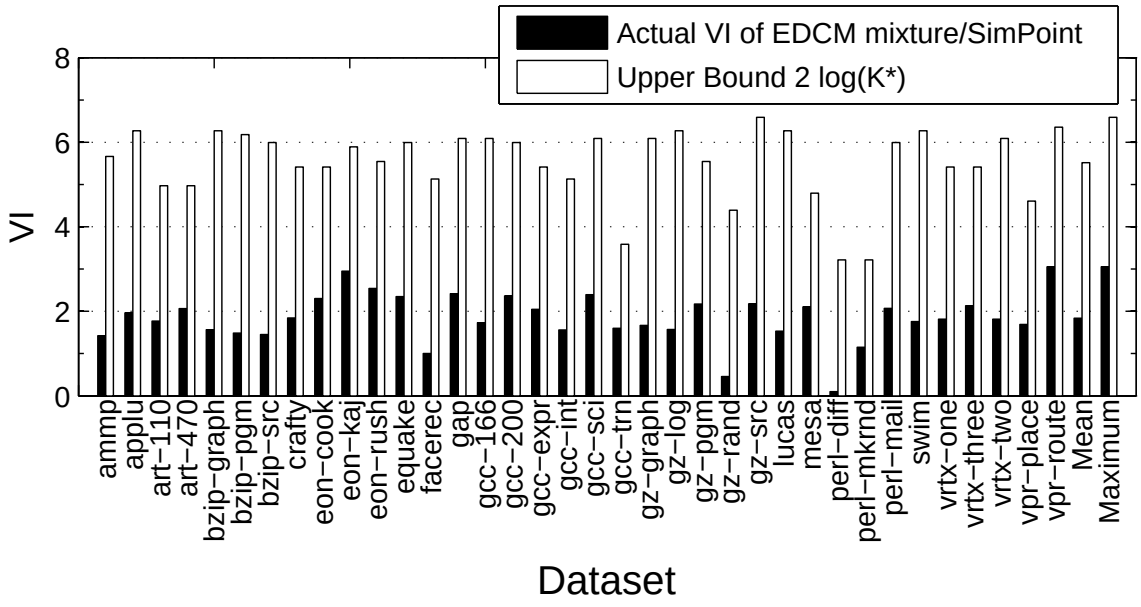


Figure 4.32: Variation of Information between EDCM and SimPoint. For each dataset, we compute the VI between the clustering given by SimPoint and the clustering given by the EM algorithm on the EDCM. We see that, on average, the VI between the clusterings is about 1/3 the theoretical maximum. Thus, we conclude that although the clusterings are obviously not identical, they are more similar than they are different. The term K^* in the upper bound is defined as $max(k, k')$, the maximum number of clusters in either of the clusterings compared using the VI, and holds for $K^* \leq \sqrt{N}$.

SimPoint, the two clustering methods are more similar than they are different. They will not be perfectly correlated, of course, because the random linear projections used by SimPoint are changing the data before it is clustered.

4.6.3 Coefficient of Variation

For each dataset, we compute the weighted sum coefficient of variation of the CPI metric. The CoV is the standard deviation of a variable divided by the mean, allowing us to compare the variance in populations with different means. Computing the CoV of the CPI for the clusters tells us how much, within a cluster, the CPI metric varies. A lower CoV means the CPI of intervals within a cluster are more similar. The more similar the CPI values are within a cluster, the better the job we

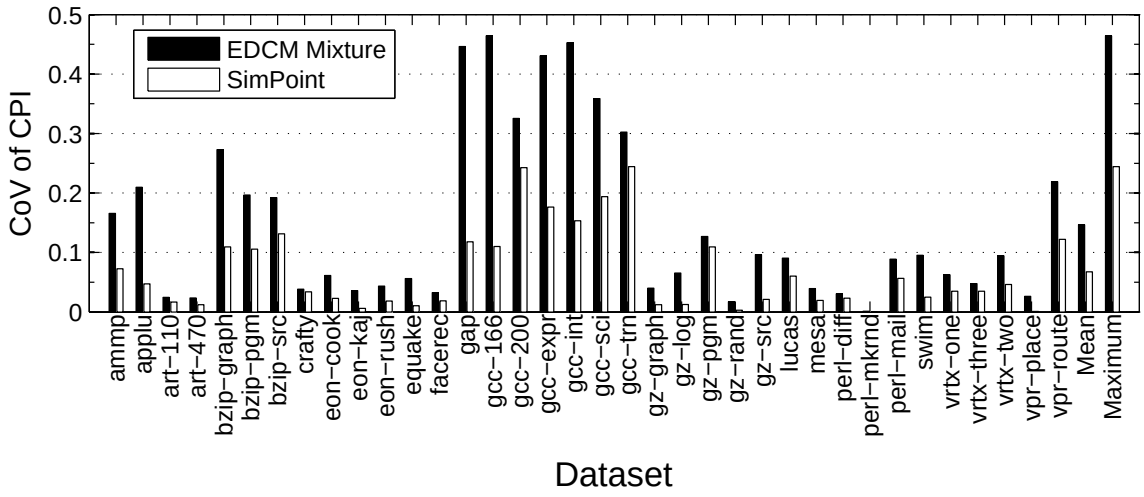


Figure 4.33: Coefficient of variation (CoV) of CPI. We compute the CoV of the CPI statistic for both clusterings, SimPoint and EDCM, for every dataset. Lower values for CoV mean more similar items within the clusters, meaning the clustering has done a good job at grouping similar items. We see that SimPoint gives clusterings with lower CoV values than the EDCM when each clustering method is allowed to choose its own value for k. However, SimPoint picks larger values of k than the EDCM, allowing it to more closely fit the differences in behavior in program trace data. Because it uses more clusters, SimPoint has a lower CoV of CPI.

have done at clustering examples with similar behavior. Figure 4.33 compares the CoV for the clusterings generated by SimPoint and the EDCM for every dataset. We see that, on average, SimPoint has a lower CoV, and therefore clusters with intervals exhibiting more similar behavior, than the EDCM.

SimPoint chooses a larger number of clusters than the EDCM. This gives it more flexibility to fit the data's behaviors. Because we can fit smaller differences in behavior better with a higher number of clusters, SimPoint clusterings have a smaller CoV within them. To reduce the effect of the larger number of clusters selected by SimPoint and the k-means algorithm, we force both clustering methods to use only $k = 3$ clusters to fit the data. Figure 4.34 compares the coefficient of variation of the CPI metric when we force SimPoint and the EDCM to use a small number of clusters ($k = 3$). This removes the advantage of being able to use more clusters that

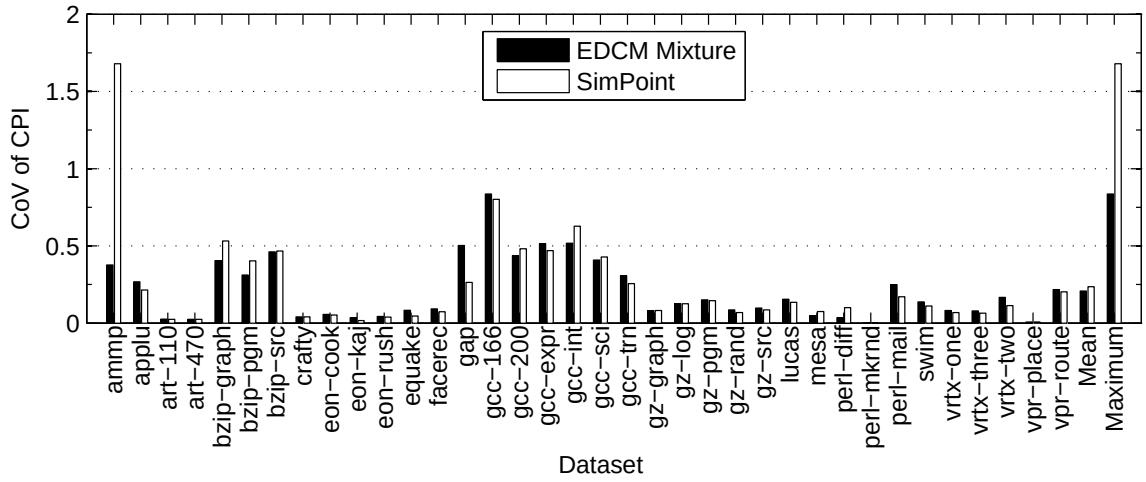


Figure 4.34: Coefficient of variation of CPI, small k. We force both SimPoint and the EDCM mixture to use a very small number of clusters, $k = 3$. We see that without its ability to use more clusters to fit data behavior more carefully, the CoV of the CPI statistic for k-means is nearly exactly equal to the EDCM (except for one large difference in the `ammp` dataset, attributable to either a large standard deviation or a small mean, < 1, in the CPI). We conclude that both the EDCM and k-means do just as well in clustering items with similar CPI statistics, and the difference is due to the difference in k.

SimPoint previously had over the EDCM when comparing the CoV. We see that, on average, the two clustering methods give nearly identical CoV measures. The large coefficient of variation given by SimPoint for the `ammp-ref` dataset can be attributed to either a very large standard deviation in the CPI, or a small mean (< 1). Overall, both methods produce clusters that have the same amount of variation of behavior per cluster.

4.7 Overview

We have shown many results in this chapter. At a high level, we have first shown the importance of the discussion of burstiness with regards to program trace data. Program trace data is indeed burstiness and it is important that we have a clustering method that can take this natural phenomenon into consideration. We showed that the EDCM is well suited to expressing burstiness.

The EM algorithm is well suited to training mixtures of EDCM distributions. The algorithm converges quickly and we see that the EM runs very quickly when training an EDCM mixture, especially considering the amount of data that is being clustered compared to SimPoint and the k-means algorithm. Out implementation of deterministic annealing did not help the EM algorithm, forcing the model selection methods to choose models with higher values for k without offering a significant improvement in percent prediction error.

The Akaike and Bayesian information criteria offer trade-offs that must be taken into consideration when deciding which model selection method to use. Because of its larger penalty on the model complexity, the BIC tends to choose models with a smaller number of clusters. While this would mean time saved when simulating a fewer number of cluster representatives, it also means less flexibility in the model in its ability to fit the behaviors present in the data. The AIC, on the other hand, is more flexible and can choose models with a higher value for k, meaning we can fit the data more accurately and see a reduction in prediction error.

Of the four methods of choosing cluster representatives that we examined, using the L_1 distance from the cluster means gave us the smallest amount of prediction error. The probability method did not work as well as we had hoped it might because the EDCM places the highest probability on the most bursty intervals, which may not be truly representative of overall cluster behavior (even though burstiness helps us to cluster similar intervals).

We also showed how similar the clusterings given by SimPoint and the EDCM are. Both methods produce clusters with largely the same coefficient and variation of the CPI metric (equal amounts of similarity between the intervals in the clusters), produce clusterings that are mostly the same size, and have values for the variation of information metric. SimPoint is a time-proven method for clustering program

behavior, and seeing the similarities the EDCM has with it increases our confidence that the EDCM mixtures are good models for expressing program trace behaviors.

CHAPTER FIVE

Summary

The SimPoint project is an important tool for researchers of new hardware designs. Running full benchmark simulations to test a collection of designs can be extremely time consuming, taking years of CPU time. SimPoint allows researchers to cut the time needed to get accurate estimations of performance to days, rather than years. To do so, SimPoint clusters benchmark data using the k-means algorithm, grouping intervals of program execution with similar behavior. From these clusters, simulation points are drawn that represent overall behaviors within the program's execution. Statistics are gathered on this handful of intervals rather than the entire benchmark, yielding accurate predictions at a fraction of the simulation cost.

The k-means algorithm does have some difficulties, however. First, program trace data exists in a very high number of dimensions. Benchmarks are split into basic block vectors, with each basic block of a program's execution, of which there may be thousands or millions, being one dimension. Because the k-means algorithm uses distance metrics to perform its clustering, and because points tend to look farther and farther apart with increasing dimension, SimPoint must use dimension reduction techniques before clustering the data down to a manageable number of dimensions (the default for SimPoint is 15). Dimension reduction may mask natural features that occur in the data. Or, worse, it may collapse two or more true clusters of behavior in the original space into one cluster in the projected space. We would like to use a clustering algorithm that does not require dimension reduction—one that can work in the full dimension of the data.

Additionally, we would like a method that allows us to account for the phenomenon of burstiness, which is present is program execution (temporal locality) and

therefore the program data clustered by SimPoint. The exponential Dirichlet compound multinomial (EDCM), developed by Elkan (2006), allows us to cluster in the full dimension of the data and to account for burstiness. To our knowledge, this research is the first application of the EDCM to a domain outside of text document clustering and is the first application to SimPoint. It is also one of the few methods able to cluster the program trace data used by SimPoint in its full dimension.

In applying the EDCM mixture model to the task of clustering SimPoint data, we also examine different methods for choosing the value k. Picking the number of clusters without *a priori* knowledge is a major problem in clustering. We find that the Bayesian Information Criterion penalizes too harshly on the large number of free parameters present in a model with high dimension. The Akaike Information Criterion is more lenient on the number of parameters and gives the EDCM mixture model more flexibility than the BIC. Thus, the AIC, while using more clusters, gives better prediction performance. The EDCM, using the AIC to select the number of clusters, chooses half the number of clusters as SimPoint, a significant savings.

Once a clustering has been fit to the data using the EM algorithm and the AIC, we need to choose a representative from each cluster to serve as simulation points. We examine several methods of selection, finding that the probability based method does not perform as well as we has hoped. However, using the L_1 distance from the cluster means yields simulation points that give very accurate prediction results, less than 2% error on average.

The EDCM performs very well in the task of clustering SimPoint data and choosing simulation points. The EDCM gives only slightly more error than SimPoint, while doing so with half the number of clusters needed. This is a significant result because while accuracy is important, it is not the purpose of SimPoint. Accuracy can be off slightly because we are using the simulation points to compare statistics of several different processor designs. As long as we use the same simulation points,

our results should be comparable. However, the savings in number of clusters is significant, because the main purpose of SimPoint is to save time performing estimation simulations. The fewer clusters we have, the fewer simulation points we have to simulate in full to get an accurate idea of performance, yielding a direct savings in the amount of simulation time needed. Theoretically, we also find the EDCM more appealing to work with that the k-means algorithm because we can cluster in the full dimension of the data and account for the phenomenon of burstiness.

Future work involving the EDCM includes implementing an optimized version of the EM algorithm used to train it in C++ and incorporating the method into SimPoint so others may use it. We would also like to find a way to make proper use of the probabilities assigned to the data by the EDCM mixture.

BIBLIOGRAPHY

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

Akaike, H. (1974, December). A new look at the statistical model identification. *IEEE Transactions on Automatic Control 19*(6), 716–723.

Balakrishnan, N. and V. B. Nevzorov (2003). *A Primer on Statistical Distributions*. Hoboken: Wiley-Interscience.

Burger, D. C. and T. M. Austin (1997, June). The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, U. of Wisconsin, Madison.

Casella, G. and R. L. Berger (2002). *Statistical Inference* (2 ed.). Australia: Duxbury.

Church, K. W. and W. A. Gale (1995). Poisson mixtures. *Natural Language Engineering 1*(2), 163–190.

Dempster, A. P., N. M. Laird, and D. B. Rubin (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistics Society 39*(1), 1–38.

Elkan, C. (2005, January). Lecture #8. Univ. California San Diego, CSE291: Statistical Learning, http://www-cse.ucsd.edu/users/elkan/291/lect08.pdf.

Elkan, C. (2006). Clustering documents with an exponential-family approximation of the Dirichlet compound multinomial distribution. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, New York, NY, USA, pp. 289–296. ACM Press.

Feng, Y. and G. Hamerly (2006). PG-means: learning the number of clusters in data. In B. Schölkopf, J. Platt, and T. Hoffman (Eds.), *Advances in Neural Information Processing Systems*, Volume 19, Cambridge, MA. MIT Press.

Hamerly, G., E. Perelman, and B. Calder (2006, March). Comparing multinomial and k-means clustering for SimPoint. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

Hastie, T., R. Tibshirani, and J. Friedman (2001). *The elements of statistical learning: data mining, inference, and prediction*. Springer Series in Statistics. New York: Springer.

Hettich, S. and S. D. Bay (1999). The UCI KDD archive: 20 newsgroups. University of California, Department of Information and Computer Science. http://kdd.ics.uci.edu/.

Jain, A. K. and R. C. Dubes (1988). *Algorithms for clustering data*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Jain, A. K., M. N. Murty, and P. J. Flynn (1999). Data clustering: a review. *ACM Comput. Surv. 31*(3), 264–323.

Johnson, N. L., A. W. Kemp, and S. Kotz (2005). *Univariate Discrete Distributions* (3 ed.). Wiley series in probability and statistics. Hoboken: Wiley-Interscience.

Johnson, N. L., S. Kotz, and N. Balakrishnan (1997). *Discrete Multivariate Distributions*. Wiley Series in Probability and Statistics. New York: Wiley-Interscience.

Katz, S. M. (1996). Distribution of content words and phrases in text and language modelling. *Nat. Lang. Eng. 2*(1), 15–59.

Kotz, S., N. Balakrishnan, and N. L. Johnson (2000). *Continuous Multivariate Distributions* (2 ed.), Volume 1: Models and Applications of *Wiley series in probability and statistics*. New York: Wiley-Interscience.

Madsen, R. E., D. Kauchak, and C. Elkan (2005). Modeling word burstiness using the Dirichlet distribution. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, New York, NY, USA, pp. 545–552. ACM Press.

McCallum, A. Industry sector. http://www.cs.umass.edu/∼mccallum/code-data.html.

Meilă, M. (2002, October). Comparing clusterings. Department of Statistics, University of Washington.

Minka, T. (2003). Estimating a Dirichlet distribution. http://research.microsoft.com/∼minka/papers/dirichlet/.

Ng, A. Y., M. I. Jordan, and Y. Weiss (2001, mar). On spectral clustering: Analysis and an algorithm. In *Advances in Neural Information Processing Systems (NIPS)*, Number 14.

Sanghai, K., T. Su, J. G. Dy, and D. R. Kaeli (2005). A multinomial clustering model for fast simulation of computer architecture designs. In *KDD*, pp. 808–813.

Schwarz, G. (1978, mar). Estimating the dimension of a model. *The Annals of Statistics 6*(2), 461–464.

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

Sherwood, T., E. Perelman, G. Hamerly, and B. Calder (2002). Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, pp. 45–57. ACM Press.

SPEC. SPEC CPU2000. http://www.spec.org/cpu2000/.

Ueda, N. and R. Nakano (1998). Deterministic annealing EM algorithm. *Neural Netw. 11*(2), 271–282.

Zahn, C. T. (1971, January). Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on Computers C-20*(20), 68–86.