The Dissertation Committee for Jason Victor Davis
certifies that this is the approved version of the following dissertation:

Mining Statistical Correlations with Applications to Software Analysis

Committee:

Inderjit Dhillon, Supervisor

Emmett Witchel, Supervisor

Joydeep Ghosh

Greg Hamerly

Raymond J. Mooney

Mining Statistical Correlations with Applications to Software Analysis

by

Jason Victor Davis, B.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2008

To my beautiful fiancé Kay and to my dog Monster.

Acknowledgments

I would like to acknowledge my co-advisors Professors Inderjit Dhillon and Emmett Witchel for their invaluable expertise, insight, and suggestions. I would also like to thank my primary collaborators, Brian Kulis, Prateek Jain, and Suvrit Sra for the countless discussions that took place in developing some of the core ideas and algorithms in my thesis. Finally, I would like to acknowledge several other colleagues who also contributed to some of the ideas in my thesis: Michael Bond, Jungwoo Ha, Justin Brickell, Matthew Taylor, Matyas Sustik, along with Emmett's Clarify research group: Don Porter, Hany Ramadan, Chris Rossbach, and Inderajit Roy.

Mining Statistical Correlations with Applications to Software Analysis

Publication No. _____

Jason Victor Davis, Ph.D.
The University of Texas at Austin, 2008

Supervisors: Inderjit Dhillon
Emmett Witchel

Machine learning, data mining, and statistical methods work by representing real-world objects in terms of feature sets that best describe them. This thesis addresses problems related to inferring and analyzing correlations among such features. The contributions of this thesis are two-fold: we develop formulations and algorithms for addressing correlation mining problems, and we also provide novel applications of our methods to statistical software analysis domains.

We consider problems related to analyzing correlations via unsupervised approaches, as well as algorithms that infer correlations using fully-supervised or semi-supervised information. In the context of correlation analysis, we propose the problem of correlation matrix clustering which employs a k-means

style algorithm to group sets of correlations in an unsupervised manner. Fundamental to this algorithm is a measure for comparing correlations called the log-determinant (LogDet) divergence, and a primary contribution of this thesis is that of interpreting and analyzing this measure in the context of information theory and statistics. Additionally based on the LogDet divergence, we present a metric learning problem called Information-Theoretic Metric Learning which uses semi-supervised or fully-supervised data to infer correlations for parametrization of a Mahalanobis distance metric. We also consider the problem of learning Mahalanobis correlation matrices in the presence of high dimensions when the number of pairwise correlations can grow very large.

In validating our correlation mining methods, we consider two in-depth and real-world statistical software analysis problems: software error reporting and unit test prioritization. In the context of Clarify, we investigate two types of correlation mining applications: metric learning for nearest neighbor software support, and decision trees for error classification. We show that our metric learning algorithms can learn program-specific similarity models for more accurate nearest neighbor comparisons. In the context of decision tree learning, we address the problem of learning correlations with associated feature costs, in particular, the overhead costs of software instrumentation. As our second application, we present a unit test ordering algorithm which uses clustering and nearest neighbor algorithms, along with a metric learning component, to efficiently search and execute large unit test suites.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many of today's state-of-the-art machine learning, data mining, and statistical methods represent real world objects via the building blocks or features that best describe them. For example, popular bag-of-words models used in text domains represent documents via individual words and their respective counts. In statistical software analysis domains, program executions can be represented by the methods called during runtime. In collaborative filtering applications, movies or songs are represented by users' ratings.

A fundamental analysis task that arises in such domains is that of inferring how features are related or correlated with one another. For example, understanding the behavior and correlations between method calls can be used in analyzing program behavior. Knowing that the calling behavior of method A is similar to that of the calling behavior of a seemingly unrelated method B can be helpful in debugging program errors. In the context of collaborative filtering, realizing that two users have similar or highly correlated movie tastes can be useful in determining users' preferences. Such inferences are the building blocks of recommender systems.

Inferring correlations is useful for not only analysis, but can also be

used as a means to a different end goal, particularly when data is fully or semi-supervised. In text analysis domains, understanding the relation between Google, Yahoo, and Microsoft (technology companies) as compared to the relation between Exxon, Texaco, and BP (oil companies) can be used as an important signal in text classification tasks. In fact, many popular machine learning and data mining algorithms use such relations in achieving their end tasks.

A principal goal of this thesis is that of investigating algorithms, models, and formulations for inferring and analyzing correlations. The primary machine learning contribution of this thesis is that of estimating correlations for distance function parametrization. We present algorithms and formulations for optimizing a Mahalanobis distance function, a generalization of the standard squared Euclidean distance. In doing so, we also consider various models for comparing correlations, proposing a new method for clustering correlations. Additionally, we tackle the problem of learning distance metrics in high dimensions when the number of pairwise correlations is very large (on the order of millions or more).

The primary application considered here is that of statistical software analysis. Statistical software analysis takes a discriminative approach in analyzing software by comparing program executions via standard machine learning feature representations. We consider various correlation inference problems in the context of statistical software analysis. In particular, we present novel applications of our distance metric learning algorithms to two different

statistical software analysis problems. We also consider other related statistical software analysis algorithms that work by inferring correlations, including classification tasks as well as cost-sensitive learning problems.

1.1 Learning with Correlations

The learning contributions in this thesis can be summarized in terms of four parts: (1) methods for comparing correlations, (2) clustering correlations, (3) learning constrained correlations for distance function parametrization, and (4) learning correlations in high-dimensions.

1.1.1 Comparing Correlations

The examples given above describe correlations in the context of comparing relations between several objects. However, in many problems, objects can be naturally represented themselves by a set of correlations. For example, consider the problem of modeling software method calls with respect to a set of errors or bugs. Vector representations that rely on the average method calling behavior for each error discount potentially rich information regarding how this behavior is distributed. Representing software methods via pairwise comparisons of each error's distribution can capture additional information that may be helpful in understanding program behavior. As another example, sensor networks are small wireless devices that work by continually monitoring their surrounding environmental conditions. While each sensor can be modeled by its mean temperature, humidity levels, etc., information regarding

covariance between these measures can also be important. For example, how do light levels correlate with temperature?

A fundamental contribution of this thesis is that of investigating measures for comparing correlations. A major focus of this work is that of exploring the properties of matrix divergences between two correlation matrices, in particular, the log-determinant (LogDet) matrix divergence. We analyze this measure under a variety of contexts, including connections with information theory and statistics, and also computational issues. Many of the algorithms presented in this thesis are based on this measure.

1.1.2 Clustering Correlations

Clustering is a common data analysis task [40], and when objects are represented in terms of their second order correlation information, we propose a clustering algorithm in Chapter 3. This method uses the LogDet divergence, and assumes that objects are represented by both their mean vectors along with covariance matrices. The problem assumes that these given vectors and corresponding covariance matrices parametrize a Gaussian distribution, and the clustering objective is to minimize the relative entropy from each distribution to its cluster centroid. Mathematically, we show that the relative entropy between two Gaussian distributions can be expressed as the convex combination of two Bregman divergences—a Mahalanobis distance between mean vectors and a LogDet matrix divergence between the covariance matrices. From this, we formulate efficient algorithms to optimize our problem

objective.

1.1.3 Correlations for Distance Metric Learning

In many applications, fully-supervised or semi-supervised constraint information relating pairs of points may be available. In Chapter 5, we introduce the Information-theoretic metric learning (ITML) problem which learns a Mahalanobis distance function from such constraint information. The Mahalanobis distance function is parametrized by a $(d \times d)$ matrix containing pairwise correlations between each of d features. Whereas standard maximum-likelihood covariance estimation methods compute this matrix in an unsupervised manner, the metric learning problem computes this correlation matrix in a semi-supervised or fully-supervised fashion.

ITML formulates the metric learning problem in an information-theoretic setting, exploiting a connection between the LogDet divergence and the relative entropy between two Gaussian distributions. ITML learns a distance metric subject to constraints over pairs of objects: objects that are similar should have smaller distances, and objects that are dissimilar should have larger distances. Mathematically, we show that this problem can be expressed as a convex optimization problem with a LogDet objective and linear constraints. The resulting algorithm uses relatively simple and closed-form projections, obviating the need for costly eigenvalue computations typically required to enforce positive definiteness of the learned matrix (which is necessary to ensure the distance function does not return negative values).

1.1.4 Learning Correlations in High Dimensions

Learning pairwise correlations in the presence of a large number of features presents several challenges. Statistical software analysis representations can have upwards of one hundred thousand features. Pairwise comparisons over this feature space require the estimation of billions of parameters. Like any learning algorithm, our metric learning algorithm ITML can be viewed as a statistical inference procedure. Estimating such a large number of parameters is challenging for even the most robust algorithms and also requires a very large amount of training data. Furthermore, evaluating the distance between two objects using a Mahalanobis distance learned by ITML requires computation that is quadratic in the dimensionality. Performing nearest neighbor searches with respect to such a distance function will not scale to large, high dimensional modern data sets.

In Chapter 6, we present structured Mahalanobis metrics for learning distance functions in high dimensional settings. Instead of learning arbitrary dense matrices that are learned by ITML, we propose low-parameter methods that use a number of parameters that is linear in the dimensionality. This enables the Mahalanobis distance function to be learned, stored, and evaluated efficiently in the context of high dimensionality. In particular, we present formulations and algorithms for learning two types of structured Mahalanobis correlation matrices: a low-rank representation suited for applications in which high recall is important, and a diagonal plus low-rank representation which optimizes both precision as well as recall. In Chapter 6, we also present

experiments over several modern high dimensional data sets, including text, collaborative filtering, and statistical software analysis applications.

1.2 Applications to Statistical Software Analysis

In this thesis, we explore two novel and exciting statistical software analysis applications: software error reporting and unit test prioritization. As we will see, estimating correlations is a fundamental problem for each of these applications.

1.2.1 The Clarify System

The Clarify system is presented in Chapter 8 and is motivated by the fact that reporting errors caused by black box components can be quite difficult. Clarify presents a solution where programs are monitored in a third party fashion, and error reports are determined using available program features collected after an error has occurred.

Clarify achieves better error messaging by exploiting a database of previous program executions. These executions are assumed to represent several runs of the program exhibiting various error types. Each execution is represented by features collected at runtime, for example, method counts. Clarify improves error messaging by leveraging machine learning and data mining algorithms that analyze this database.

When the database is fully supervised (i.e. error types are known for each program execution), errors are predicted using a decision tree classifier

that is trained directly from the database and its associated labels. Decision trees predict class labels by analyzing correlations between features values. Each leaf node of the tree determines classification predictions and is reached only if the given object satisfies each of its parent nodes.

When the execution database is unsupervised or semi-supervised, Clarify uses nearest neighbor searches to provide better error messaging. Baseline measures such as the Euclidean distance compare program executions by giving equal weight to each features. In reality, however, some program features (e.g. program methods) are much more critical than others. Here, distance metric learning can be used learn the importance and relations between features in order to improve the quality of the nearest neighbor search.

1.2.2 Cost-Sensitive Decision Trees

In many real-world problems, collecting features comes at a cost. Statistical software analysis is no exception, in particular, the Clarify system. The costs incurred here are that of instrumentation overhead incurred in collecting program features such as method calls, basic block calls, etc. In Chapter 9, we present a method which learns a decision tree classifier that maximizes both classification accuracy as well as minimizing the costs required to collect features tested in the tree.

1.2.3 Unit Test Ordering via Metric Learning

Unit tests are small functional software tests that verify software correctness. Large unit tests sets are necessary to ensure adequate testing coverage and reduce program errors. However, the time required to execute large sets of tests makes unit testing an inefficient part of code development. In Chapter 10, we present an algorithm which executes tests in an out-of-order manner so that failing tests can be quickly identified. This is achieved through the use of clustering and nearest neighbor searches, allowing efficient searching and exploration over the set of all unit tests. As with the Clarify system, using a domain-specific distance metric is critical to the quality of not only the nearest neighbor searches, but also the clustering algorithms. To this end, we present a novel application of our metric learning methods in which information about previously introduced software bugs and their associated failing tests are used as semi-supervised constraint information in learning a program-specific Mahalanobis distance function.

Chapter 2

Analyzing Correlations Between Features

The problem of mining statistical correlations consists of two parts: that of being able to quantify the degree to which any two variables (or features, attributes, etc.) are statistically correlated, and then applying data mining or machine learning algorithms for the purpose of discovering interesting trends across this set of correlations. If two variables a and b are completely dependent, they are said to have a high correlation (or possibly anti-correlation). Two variables that are independent have no correlation. Classical statistical methods focus mainly on just the problem of quantifying the correlation between variables. In this chapter, we present some basic methods used in statistics for inferring the correlation between two variables.

2.1 Inferring Statistical Correlations: Classical Methods

A standard means of inferring the correlation between a set of random variables is by assuming that the variables are multivariate Gaussian distributed. The *multivariate Gaussian* distribution is the multivariate generalization of the standard univariate case. The probability density function (pdf) of a d-dimensional multivariate Gaussian is parametrized by mean vector

$\boldsymbol{\mu}$ and positive definite covariance matrix $\boldsymbol{\Sigma}$:

$$p(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{\frac{d}{2}}|\boldsymbol{\Sigma}|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right),$$

where $|\boldsymbol{\Sigma}|$ is the determinant of $\boldsymbol{\Sigma}$.

The standard approach for inferring correlations under this framework is to find a covariance S, along with mean $\boldsymbol{m}$ that maximize the probability of the multivariate Gaussian. Given a set of independently drawn samples $\boldsymbol{x}_1, ..., \boldsymbol{x}_n$, the so-called maximum likelihood estimate maximizes the quantity:

$$\prod_{i=1}^{n} p(\boldsymbol{x}_i|\boldsymbol{m}, S), \tag{2.1.1}$$

where p has the form of a multivariate Gaussian. It can be shown [74], by taking the log of the above quantity and differentiating appropriately, that the optimal mean $\boldsymbol{m}$ and covariance S admit simple, closed form solutions. In particular, the means is the arithmetic mean over the input points, and the (i, j) entry of the sample covariance matrix is:

$$S_{ij} = \frac{1}{n}\sum_{k=1}^{n}(\boldsymbol{x}_k(i) - \boldsymbol{m}(i))(\boldsymbol{x}_k(j) - \boldsymbol{m}(j)).$$

2.2 Comparing Correlations with the LogDet Divergence

A fundamental task in machine learning, data mining, and statistics is that of comparing two representations. In particular, two important types of representation comparisons are (1) comparing two objects' representation

and (2) comparing two models' representations. The problem of comparing two objects' vector representations is relatively well studied. We review some of these methods below and then outline some important problems where representations in terms of correlations or covariance matrices arise naturally.

Many algorithms require that two objects be compared. For example, the k-means clustering algorithm groups objects into a set of k clusters such that the (normalized) pairwise distance between points in each cluster is minimized. K-means compares objects via their vector representations, and the distance computations used by the algorithm is that of the standard squared Euclidean distance. Nearest neighbor searches are another large class of algorithms that work by comparing pairs of objects. Here, nearest neighbors are determined by computing pairwise distances from a candidate object to every other object in a database. The nearest neighbors are defined as those objects which have the smallest distances to the candidate object.

Whereas many unsupervised methods such as k-means or nearest neighbor searches compare objects' representations directly, many supervised learning methods work by comparing models' representations. This is often manifested through a regularization component which works by incorporating a bias into the problem solution. For example, in Bayesian methods, this bias is parametrized via explicitly defined statistical distributions and is referred to as a prior. In fact, all learning algorithms have some sort of bias, and oftentimes this bias is modeled mathematically as the distance between the models' parameter vector and some baseline vector.

Consider a linear support vector machine (SVM), which seeks a separating hyperplane with minimum L_2 norm [60]. This norm can equivalently be viewed as the distance between the hyperplane and the origin. In the context of SVMs, this regularization can be viewed as a preference for solutions in which each feature is given equal weightings. AdaBoost is another popular machine learning algorithm in which a vector of weights are learned over a set of features (or in the context of boosting algorithms, base classifiers). The AdaBoost algorithm can be viewed as a greedy procedure for maximizing the entropy of these weights [25]. In terms of distance comparisons, maximizing the entropy is equivalent to minimizing the Kullback-Liebler divergence to the L_1-normalized unit vector (i.e. the uniform distribution).

The SVM and AdaBoost algorithms detailed above all deal with distances between vectors. However, in many important applications, representations naturally take on matrix forms. As we will see, this includes the problem of learning a Mahalanobis distance function parametrized by a $(d \times d)$ correlation matrix.

A straightforward generalization of vector divergences to matrix divergences is that of generalizing the standard squared Euclidean distance to the Frobenius distance, which compares two matrices by pairwise computation of the squares of their differences:

$$D_f(X, Y) = \sum_{ij} (X_{ij} - Y_{ij})^2 = \|X - Y\|_2^2.$$

The Frobenius measure computes distances between matrices in a highly localized manner. However, in practice, matrices that parametrize covariances or correlations often have structure that can only be described through more global measures. For example, principal components analysis (PCA) describes data through an eigen-analysis of the dominant components of the sample covariance matrix for the data (equivalently, through the SVD of the 'centered' data matrix). When comparing two covariance matrices, accounting for this structure is critical.

A central measure used throughout this dissertation is a matrix divergence called the log-determinant (LogDet) divergence. Here, we introduce this measure and compare it with the Frobenius distortion. In later chapters, we will introduce other properties of the LogDet divergence, including interpretations in the contexts of information theory and statistics.

The LogDet divergence is defined between two positive semi-definite matrices X and Y:

$$D_{\ell\mathrm{d}}(\boldsymbol{X}, \boldsymbol{Y}) = \mathrm{tr}(\boldsymbol{X}\boldsymbol{Y}^{-1}) - \log|\boldsymbol{X}\boldsymbol{Y}^{-1}| - d,$$

where tr is the trace of a matrix, and $|X|$ is the determinant of the matrix X.

We now present a real-world example comparing the Frobenius distance with the LogDet divergence. This example is taken from Clarify [49], a statistical software analysis system that uses classifiers and nearest neighbor searches to provide better error messaging. One way that Clarify represents

system is through the method counts of a program's executions. Here, the size of the vector model is equal to the number of methods in the program, and the value taken by these features is equal to the number of times the method is called during a single program execution. Clarify is trained over a set of known program errors. For each program error, several program executions and their method counts are used to train the system.

Figure 2.2 shows twelve histograms for a Clarify benchmark, an open source mp3 player, Mpg321. Each histogram represents the calling behavior of a single method, restricted to those instances which exhibit a single program error. For example, the plot in the upper left-hand side shows the calling behavior of method A restricted to the 'Unsupported Format' error. The histogram in this plot shows that the method count is always zero, which means that the method is never called when the 'Unsupported Format' error occurs. Similarly, the plot to the right of this shows a distribution in which the method is either called zero times, or is called five times. Overall, we can see that methods A and B exhibit similar calling behaviors. This is because for errors 'Corrupted Frame' and 'Normal', the calling behavior for the two methods is somewhat similar, as they both have very similar bimodal structures. The 'Unsupported Format' and the 'Corrupted Tag' errors also have similar calling behavior, as neither methods A nor B are executed when this error occurs. Overall, the calling behaviors of methods A and B is quite different from that of method C, in which all four errors have similar calling behavior.

We now consider how to formalize this observation through precise

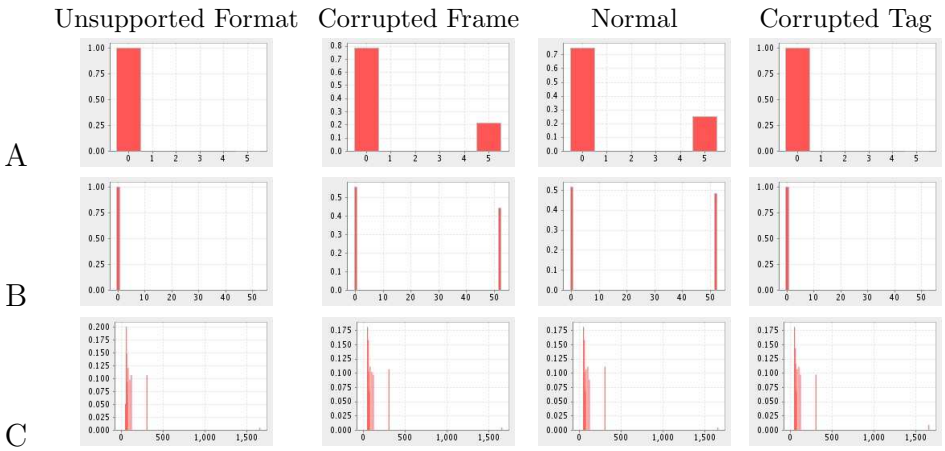
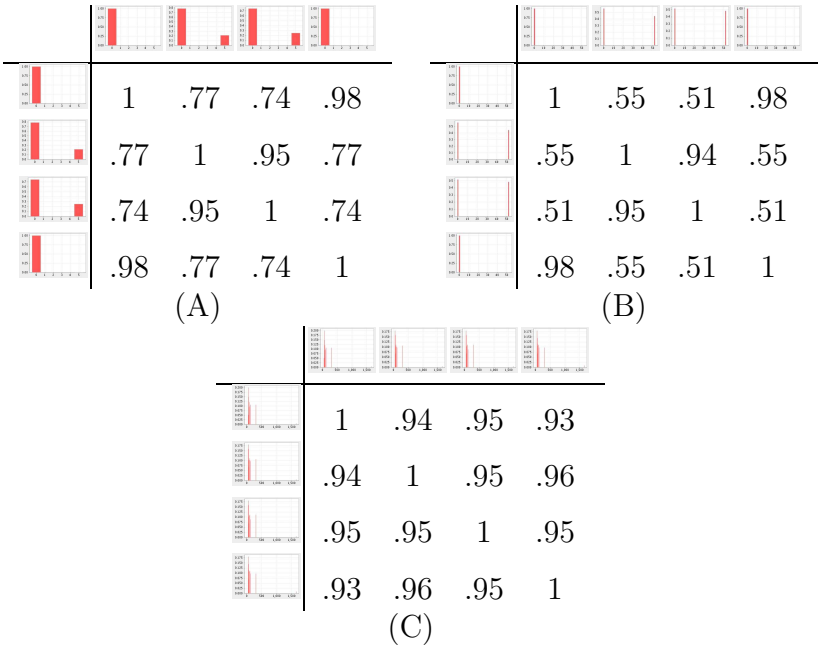


Figure 2.1: A comparison of the method counts for four different error types (across each of the four columns) for three different methods in the Mpg321 program. As we can see, methods A and B have similar calling behavior which is different from method C.

mathematical representations. In this example, we can see that while methods A and B have bimodal structure for the 'Corrupted Frame' and 'Normal' errors, the values that these distributions take are quite different. In fact, data normalization is quite challenging for statistical software analysis, as program features (i.e. method counts used here) tend to take on unpredictable and highly non-parametric forms.

To get around this, we factor out the distribution form by comparing each method across its calling behavior for each of the error types. Instead of representing each error/method pair by its distribution, we represent each method by its correlations between each pair of errors. Figure 2.2 shows such correlation matrices for each of the three methods. These correlation matrices are computed by pairwise comparisons among each of the four errors. The



	1	.77	.74	.98
	.77	1	.95	.77
	.74	.95	1	.74
	.98	.77	.74	1

(A)

	1	.55	.51	.98
	.55	1	.94	.55
	.51	.95	1	.51
	.98	.55	.51	1

(B)

	1	.94	.95	.93
	.94	1	.95	.96
	.95	.95	1	.95
	.93	.96	.95	1

(C)

Figure 2.2: Correlations matrices for each of the three different program methods.

value in the (i, j) position of the correlation matrix represents the overlap of the distribution of the i^{th} and j^{th} errors for the given method. If the distributions are identical, this overlap will be one, and this measure will approach zero as the distributions become dissimilar.

We now consider the distance between the calling behavior of these three different methods, as measured by the Frobenius divergence, as compare to that if measured by the LogDet divergence. Before doing so, we note that the correlation matrices between methods (A) and (B) can be described in terms of their two dominant eigenvalues. One eigenvalue corresponds to an eigenvector in which the 'Corrupted Frame' and 'Normal' errors take on large

Measure	D(A, B)	D(A, C)
LogDet	**.22**	6.2
Frobenius	.46	**.41**

Table 2.1: The divergence measures between the three methods given in figure 2.2. The LogDet divergence reflects the fact that methods A and B are similar, whereas A and C are not. The Frobenius distortion does not.

values, whereas the other eigenvector places more weighting on the other two errors. The final two eigenvalues of this matrix are quite small and represent noise in the problem. Similarly, matrix (C) can be described in terms of one component in which all errors are given equal weighting.

Table 2.2 shows the distance between methods A and B, and also the distance between methods A and C. The LogDet divergence reflects the notion that methods A and B are more similar the methods A and C, as the distance of 0.22 between (A,B) is much smaller than the distance 6.2 between (A,C). However, the Frobenius distortion provides exactly the opposite results, with (A,C) having a smaller distance than (A,B).

In the next chapter, we introduce a clustering algorithm which uses the LogDet measure as its underlying distance. There, we will provide more mathematical insights regarding the benefits of the LogDet divergence.

Chapter 3

Clustering Correlation Matrices

In the previous chapter, we provided an overview of standard covariance estimation methods, and we introduced a divergence called the LogDet divergence which can be used to compare two covariance matrices. In this chapter, we present a clustering algorithm in which objects being clustered are represented by their mean as well as their covariance [28]. We provide an algorithm based on the LogDet divergence, which we use to model the problem in an information-theoretic setting.

3.1 Problem Overview

Gaussian data is pervasive in all walks of life and many learning algorithms—e.g. k-means, principal components analysis, linear discriminant analysis, etc—model each input object as a *single* sample drawn from a multivariate Gaussian. For example, the k-means algorithm assumes that each input is a single sample drawn from one of k (unknown) isotropic Gaussians. The goal of k-means can be viewed as the discovery of the mean of each Gaussian and recovery of the generating distribution of each input object.

However, in many real-life settings, each input object is naturally rep-

resented by *multiple* samples drawn from an underlying distribution. For example, a student's scores in reading, writing, and arithmetic can be measured at each of four quarters throughout the school year. Alternately, consider a website where movies are rated on the basis of originality, plot, and acting. Here, several different users may rate the same movie. Multiple samples are also ubiquitous in time-series data such as sensor networks, where each sensor device continually monitors its environmental conditions (e.g. humidity, temperature, or light). Clustering is an important data analysis task used in many of these applications. For example, clustering sensor network devices has been used for optimizing routing of the network and also for discovering trends between sensor nodes [34]. If the k-means algorithm is employed, then only the means of the distributions will be clustered, ignoring all second order covariance information. Clearly, a better solution is needed.

In this chapter, we consider the problem of clustering input objects, each of which can be represented by a multivariate Gaussian distribution. The "distance" between two Gaussians can be quantified in an information theoretic manner, in particular by their differential relative entropy. Interestingly, the differential relative entropy between two multivariate Gaussians can be expressed as the convex combination of two Bregman divergences—a Mahalanobis distance between mean vectors and a LogDet matrix divergence between the covariance matrices. We develop an EM style clustering algorithm and show that the optimal cluster parameters can be cheaply determined via a simple, closed-form solution. Our algorithm is a Bregman-like clustering

method that clusters both means and covariances of the distributions in a unified framework.

We evaluate our method across several domains. First, we present results from synthetic data experiments, and show that incorporating second order information can dramatically increase clustering accuracy. Next, we apply our algorithm to a real-world sensor network dataset comprised of 52 sensor devices that measure temperature, humidity, light, and voltage. Finally, we use our algorithm as a statistical debugging tool by clustering the behavior of functions in a program across a set of known software bugs.

3.2 Preliminaries

We first present some essential background material. The *Bregman divergence* [15] with respect to ϕ is defined as:

$$D_\phi(\boldsymbol{x}, \boldsymbol{y}) = \phi(\boldsymbol{x}) - \phi(\boldsymbol{y}) - (\boldsymbol{x} - \boldsymbol{y})^T \nabla\phi(\boldsymbol{y}),$$

where ϕ is a real-valued, strictly convex function defined over a convex set $Q = dom(\phi) \subset \mathbb{R}^d$ such that ϕ is differentiable on the relative interior of Q. For example, if $\phi(\boldsymbol{x}) = \boldsymbol{x}^T\boldsymbol{x}$, then the resulting Bregman divergence is the standard squared Euclidean distance. Similarly, if $\phi(\boldsymbol{x}) = \boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x}$, for some arbitrary non-singular matrix $\boldsymbol{A}$, then the resulting divergence is the Mahalanobis distance $M_{\boldsymbol{S}^{-1}}(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x}-\boldsymbol{y})^T\boldsymbol{S}^{-1}(\boldsymbol{x}-\boldsymbol{y})$, parameterized by the covariance matrix $\boldsymbol{S}$, $\boldsymbol{S}^{-1} = \boldsymbol{A}^T\boldsymbol{A}$. Alternately, if $\phi(\boldsymbol{x}) = \sum_i (x_i \log x_i - x_i)$, then the resulting divergence is the (unnormalized) relative entropy. Bregman

divergences generalize many properties of squared loss and relative entropy.

Bregman divergences can be naturally extended to matrices, as follows:

$$D_\phi(\boldsymbol{X}, \boldsymbol{Y}) = \phi(\boldsymbol{X}) - \phi(\boldsymbol{Y}) - \mathrm{tr}((\nabla\phi(\boldsymbol{Y}))^T(\boldsymbol{X} - \boldsymbol{Y})),$$

where $\boldsymbol{X}$ and $\boldsymbol{Y}$ are matrices, ϕ is a real-valued, strictly convex function defined over matrices, and $\mathrm{tr}(\boldsymbol{A})$ denotes the trace of $\boldsymbol{A}$. Consider the function $\phi(\boldsymbol{X}) = \|\boldsymbol{X}\|_F^2$. Then the corresponding Bregman matrix divergence is the squared Frobenius norm, $\|\boldsymbol{X} - \boldsymbol{Y}\|_F^2$. The LogDet matrix divergence is generated from a function of the *eigenvalues* $\lambda_1, ..., \lambda_d$ of the positive definite matrix $\boldsymbol{X}$: $\phi(\boldsymbol{X}) = -\sum_i \log \lambda_i = -\log|\boldsymbol{X}|$, the LogDet entropy of the eigenvalues. The resulting LogDet matrix divergence is:

$$D_{\ell\mathrm{d}}(\boldsymbol{X}, \boldsymbol{Y}) = \mathrm{tr}(\boldsymbol{X}\boldsymbol{Y}^{-1}) - \log|\boldsymbol{X}\boldsymbol{Y}^{-1}| - d. \tag{3.2.1}$$

As we shall see later, the LogDet matrix divergence will arise naturally in our application. Let $\lambda_1, ..., \lambda_d$ be the eigenvalues of $\boldsymbol{X}$ and $\boldsymbol{v_1}, ..., \boldsymbol{v_d}$ the corresponding eigenvectors and let $\gamma_1, ..., \gamma_d$ be the eigenvalues of $\boldsymbol{Y}$ with eigenvectors $\boldsymbol{w_1}, ..., \boldsymbol{w_d}$. The LogDet matrix divergence can also be written as

$$D_{\ell\mathrm{d}}(\boldsymbol{X}, \boldsymbol{Y}) = \sum_i \sum_j \frac{\lambda_i}{\gamma_j}(\boldsymbol{v_i}^T\boldsymbol{w_j})^2 - \sum_i \log\frac{\lambda_i}{\gamma_i} - d.$$

From the first term above, we see that the LogDet matrix divergence is a function of the eigenvalues as well as of the *eigenvectors* of $\boldsymbol{X}$ and $\boldsymbol{Y}$.

The *differential entropy* of a continuous random variable $\boldsymbol{x}$ with probability density function f is defined as

$$h(f) = -\int f(\boldsymbol{x}) \log f(\boldsymbol{x}) d\boldsymbol{x}.$$

It can be shown [26] that an n-bit quantization of a continuous random variable with pdf f has Shannon entropy approximately equal to $h(f) + n$. The continuous analog of the discrete relative entropy is the differential relative entropy. Given a random variable $\boldsymbol{x}$ with pdf's f and g, the differential relative entropy is defined as

$$D(f||g) = \int f(\boldsymbol{x}) \log \frac{f(\boldsymbol{x})}{g(\boldsymbol{x})} d\boldsymbol{x}.$$

3.3 Clustering Multivariate Gaussians via Differential Relative Entropy

Given a set of n multivariate Gaussians parameterized by mean vectors $\boldsymbol{m_1}, ..., \boldsymbol{m_n}$ and covariances $\boldsymbol{S_1}, ..., \boldsymbol{S_n}$, we seek a disjoint and exhaustive partitioning of these Gaussians into k different clusters, $\pi_1, ..., \pi_k$. Each cluster j can be represented by a multivariate Gaussian parameterized by mean $\boldsymbol{\mu_j}$ and covariance $\boldsymbol{\Sigma_j}$. Using differential relative entropy as the distance measure between Gaussians, the problem of clustering may be posed as the minimization (over all clusterings) of

$$\sum_{j=1}^{k} \sum_{\{i:\pi_i=j\}} D(p(\boldsymbol{x}|\boldsymbol{m_i}, \boldsymbol{S_i})||p(\boldsymbol{x}|\boldsymbol{\mu_j}, \boldsymbol{\Sigma_j})). \tag{3.3.1}$$

3.3.1 Differential Relative Entropy and Multivariate Gaussians

We first show that the differential entropy between two multivariate Gaussians can be expressed as a convex combination of a Mahalanobis distance between means and the LogDet matrix divergence between covariance

matrices.

Consider two multivariate Gaussians, parameterized by mean vectors $\boldsymbol{m}$ and $\boldsymbol{\mu}$, and covariances $\boldsymbol{S}$ and $\boldsymbol{\Sigma}$, respectively. We first note that the differential relative entropy can be expressed as $D(f||g) = \int f \log f - f \log g = -h(f) - \int f \log g$. The first term is just the negative differential entropy of $p(\boldsymbol{x}|\boldsymbol{m}, \boldsymbol{S})$, which can be shown [26] to be:

$$h(p(\boldsymbol{x}|\boldsymbol{m}, \boldsymbol{S})) = \frac{d}{2} + \frac{1}{2}\log(2\pi)^d|\boldsymbol{S}|. \tag{3.3.2}$$

We now consider the second term:

$$\begin{aligned}
& \int p(\boldsymbol{x}|\boldsymbol{m}, \boldsymbol{S}) \log p(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \\
= & \int p(\boldsymbol{x}|\boldsymbol{m}, \boldsymbol{S}) \left[-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}) - \log(2\pi)^{\frac{d}{2}}|\boldsymbol{\Sigma}|^{\frac{1}{2}} \right] \\
= & -\frac{1}{2}\int p(\boldsymbol{x}|\boldsymbol{m}, \boldsymbol{S}) tr(\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})(\boldsymbol{x}-\boldsymbol{\mu})^T) \\
& - \int p(\boldsymbol{x}|\boldsymbol{m}, \boldsymbol{S}) \log(2\pi)^{\frac{d}{2}}|\boldsymbol{\Sigma}|^{\frac{1}{2}} \\
= & -\frac{1}{2} tr\left(\boldsymbol{\Sigma}^{-1}\mathrm{E}\left[(\boldsymbol{x}-\boldsymbol{\mu})(\boldsymbol{x}-\boldsymbol{\mu})^T\right]\right) - \frac{1}{2}\log(2\pi)^d|\boldsymbol{\Sigma}| \\
= & -\frac{1}{2} tr\left(\boldsymbol{\Sigma}^{-1}\mathrm{E}\left[((\boldsymbol{x}-\boldsymbol{m}) + (\boldsymbol{m}-\boldsymbol{\mu}))((\boldsymbol{x}-\boldsymbol{m}) + (\boldsymbol{m}-\boldsymbol{\mu}))^T\right]\right) \\
& -\frac{1}{2}\log(2\pi)^d|\boldsymbol{\Sigma}| \\
= & -\frac{1}{2} tr\left(\boldsymbol{\Sigma}^{-1}\boldsymbol{S} + \boldsymbol{\Sigma}^{-1}(\boldsymbol{m}-\boldsymbol{\mu})(\boldsymbol{m}-\boldsymbol{\mu})^T\right) - \frac{1}{2}\log(2\pi)^d|\boldsymbol{\Sigma}| \\
= & -\frac{1}{2} tr\left(\boldsymbol{\Sigma}^{-1}\boldsymbol{S}\right) - \frac{1}{2}(\boldsymbol{m}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{m}-\boldsymbol{\mu}) - \frac{1}{2}\log(2\pi)^d|\boldsymbol{\Sigma}|.
\end{aligned}$$

The expectation above is taken over the distribution $p(\boldsymbol{x}|\boldsymbol{m}, \boldsymbol{S})$. The second to last line above follows from the definition of $\boldsymbol{\Sigma} = E[(\boldsymbol{x}-\boldsymbol{m})(\boldsymbol{x}-\boldsymbol{m})^T]$ and

also from the fact that $E[(\boldsymbol{x}-\boldsymbol{m})(\boldsymbol{m}-\boldsymbol{\mu})^T] = E[\boldsymbol{x}-\boldsymbol{m}](\boldsymbol{m}-\boldsymbol{\mu})^T = \mathbf{0}$. Thus, we have

$$\begin{aligned}
& D(p(\boldsymbol{x}|\boldsymbol{m},\boldsymbol{S})||p(\boldsymbol{x}|\boldsymbol{\mu},\boldsymbol{\Sigma})) \\
= \; & -\frac{d}{2} - \frac{1}{2}\log(2\pi)^d|\boldsymbol{S}| + \frac{1}{2}tr(\boldsymbol{\Sigma}^{-1}\boldsymbol{S}) + \frac{1}{2}\log(2\pi)^d|\boldsymbol{\Sigma}| \qquad (3.3.3) \\
& + \frac{1}{2}(\boldsymbol{m}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{m}-\boldsymbol{\mu}) \\
= \; & \frac{1}{2}\left(tr(\boldsymbol{S}\boldsymbol{\Sigma}^{-1}) - \log|\boldsymbol{S}\boldsymbol{\Sigma}^{-1}| - d\right) + \frac{1}{2}(\boldsymbol{m}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{m}-\boldsymbol{\mu}) \\
= \; & \frac{1}{2}D_{\ell\mathrm{d}}(\boldsymbol{S},\boldsymbol{\Sigma}) + \frac{1}{2}M_{\boldsymbol{\Sigma}^{-1}}(\boldsymbol{m},\boldsymbol{\mu}), \qquad (3.3.4)
\end{aligned}$$

where $D_{\ell\mathrm{d}}(\boldsymbol{S},\boldsymbol{\Sigma})$ is the LogDet matrix divergence and $M_{\boldsymbol{\Sigma}^{-1}}(\boldsymbol{m},\boldsymbol{\mu})$ is the Mahalanobis distance, parameterized by the covariance matrix $\boldsymbol{\Sigma}$.

We now consider the problem of finding the optimal representative Gaussian for a set of c Gaussians with means $\boldsymbol{m_1},...,\boldsymbol{m_c}$ and covariances $\boldsymbol{S_1},...,\boldsymbol{S_c}$. For non-negative weights $\alpha_1,...\alpha_c$ such that $\sum_i \alpha_i = 1$, the optimal representative minimizes the cumulative differential relative entropy:

$$\begin{aligned}
p(\boldsymbol{x}|\boldsymbol{\mu}^*,\boldsymbol{\Sigma}^*) &= \arg\min_{p(\boldsymbol{x}|\boldsymbol{\mu},\boldsymbol{\Sigma})} \sum_i \alpha_i D(p(\boldsymbol{x}|\boldsymbol{m_i},\boldsymbol{S_i})||p(\boldsymbol{x}|\boldsymbol{\mu},\boldsymbol{\Sigma})) \qquad (3.3.5) \\
&= \arg\min_{p(\boldsymbol{x}|\boldsymbol{\mu},\boldsymbol{\Sigma})} \sum_i \alpha_i \left(\frac{1}{2}D_{\ell\mathrm{d}}(\boldsymbol{S_i},\boldsymbol{\Sigma}) + \frac{1}{2}M_{\boldsymbol{\Sigma}^{-1}}(\boldsymbol{m_i},\boldsymbol{\mu})\right) \qquad (3.3.6)
\end{aligned}$$

The second term can be viewed as minimizing the Bregman information with respect to some fixed (albeit unknown) Bregman divergence (i.e. the Mahalanobis distance parameterized by some covariance matrix $\boldsymbol{\Sigma}$). Consequently, it has a unique minimizer [7] of the form

$$\boldsymbol{\mu}^* = \sum_i \alpha_i \boldsymbol{m_i}. \qquad (3.3.7)$$

Next, we note that equation (3.3.6) is strictly convex in $\mathbf{\Sigma}^{-1}$. Thus, we can derive the optimal $\mathbf{\Sigma}^*$ by setting the gradient of (3.3.6) with respect to $\mathbf{\Sigma}^{-1}$ to 0:

$$\frac{\partial}{\partial \mathbf{\Sigma}^{-1}} \sum_{i=1}^{n} \alpha_i D(p(\boldsymbol{x}|\boldsymbol{m_i}, \boldsymbol{S_i})||p(\boldsymbol{x}|\boldsymbol{\mu}, \mathbf{\Sigma})) \;=\; \sum_{i=1}^{n} \alpha_i \left(\boldsymbol{S_i} - \mathbf{\Sigma} + (\boldsymbol{m_i} - \boldsymbol{\mu}^*)(\boldsymbol{m_i} - \boldsymbol{\mu}^*)^T\right).$$

Setting this to zero yields

$$\mathbf{\Sigma}^* = \sum_i \alpha_i \left(\boldsymbol{S_i} + (\boldsymbol{m_i} - \boldsymbol{\mu}^*)(\boldsymbol{m_i} - \boldsymbol{\mu}^*)^T\right). \tag{3.3.8}$$

Figure 3.1 illustrates optimal representatives of two 2-dimensional Gaussians with means marked by points A and B, and covariances outlined with solid lines. The optimal Gaussian representatives are denoted with dotted covariances; the representative on the left uses weights, $(\alpha_A = \frac{2}{3}, \alpha_B = \frac{1}{3})$, while the representative on the right uses weights $(\alpha_A = \frac{1}{3}, \alpha_B = \frac{2}{3})$. As we can see from equation (3.3.7), the optimal representative mean is the weighted average of the means of the constituent Gaussians. Interestingly, the optimal covariance turns out to be the average of the constituent covariances plus rank one updates. These rank-one changes account for the deviations from the individual means to the representative mean.

3.3.2 Algorithm

Algorithm 1 presents our clustering algorithm for the case where each Gaussian has equal weight $\alpha_i = \frac{1}{n}$. The method works in an EM-style framework. Initially, cluster assignments are chosen (these can be assigned randomly). The algorithm then proceeds iteratively, until convergence. First, the

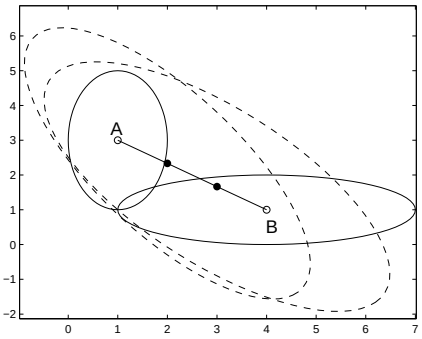


Figure 3.1: Optimal Gaussian representatives (shown with dotted lines) of two Gaussians centered at A and B (for two different sets of weights). While the optimal mean of each representative is the average of the individual means, the optimal covariance is the average of the individual covariances plus rank-one corrections.

mean and covariance parameters for the cluster representative distributions are optimally computed given the cluster assignments. These parameters are updated as shown in (3.3.7) and (3.3.8). Next, the cluster assignments π are updated for each input Gaussian. This is done by assigning the i^{th} Gaussian to the cluster j with representative Gaussian that is closest in differential relative entropy. Since both of these steps are locally optimal, convergence of the algorithm to a local optima can be shown. Note that the problem is NP-hard, so convergence to a global optima cannot be guaranteed [45].

We next consider the running time of Algorithm 1 when the input Gaussians are d-dimensional. Lines 6 and 9 compute the optimal means and covariances for each cluster, which requires $O(nd)$ and $O(nd^2)$ total work, respectively. Line 12 computes the differential relative entropy between each input Gaussian and each cluster representative Gaussian. As only the arg min

over all $\boldsymbol{\Sigma_j}$ is needed, we can reduce the LogDet matrix divergence computation (equation (3.2.1)) to $\text{tr}(\boldsymbol{S_i}\boldsymbol{\Sigma_j}^{-1}) - \log|\boldsymbol{\Sigma_j}^{-1}|$. Once the inverse of each cluster covariance is computed (for a cost of $O(kd^3)$), the first term can be computed in $O(d^2)$ time. The second term can similarly be computed once for each cluster for a total cost of $O(kd^3)$. Computing the Mahalanobis distance is an $O(d^2)$ operation. Thus, total cost of line 12 is $O(kd^3 + nkd^2)$ and the total running time of the algorithm, given τ iterations, is $O(\tau kd^2(n+d))$.

Algorithm 1 Differential Entropic Clustering of Multivariate Gaussians

1: $\{\boldsymbol{m_1}, ..., \boldsymbol{m_n}\} \leftarrow$ means of input Gaussians
2: $\{\boldsymbol{S_1}, ..., \boldsymbol{S_n}\} \leftarrow$ covariance matrices of input Gaussians
3: $\pi \leftarrow$ initial cluster assignments
4: **while** not converged **do**
5: **for** $j = 1$ to k **do** {update cluster means}
6: $\boldsymbol{\mu_j} \leftarrow \frac{1}{|\{i:\pi_i=j\}|} \sum_{i:\pi_i=j} \boldsymbol{m_i}$
7: **end for**
8: **for** $j = 1$ to k **do** {update cluster covariances}
9: $\boldsymbol{\Sigma_j} \leftarrow \frac{1}{|\{i:\pi_i=j\}|} \sum_{i:\pi_i=j} \left(\boldsymbol{S_i} + (\boldsymbol{m_i} - \boldsymbol{\mu_j})(\boldsymbol{m_i} - \boldsymbol{\mu_j})^T\right)$
10: **end for**
11: **for** $i = 1$ to n **do** {assign each Gaussian to the closest cluster representative Gaussian}
12: $\pi_i \leftarrow \text{argmin}_{1 \le j \le k} D_{\ell\text{d}}(\boldsymbol{S_i}, \boldsymbol{\Sigma_j}) + M_{\boldsymbol{\Sigma_j}^{-1}}(\boldsymbol{m_i}, \boldsymbol{\mu_j})$ {B is the LogDet matrix divergence and $M_{\boldsymbol{\Sigma_j}^{-1}}$ is the Mahalanobis distance parameterized by $\boldsymbol{\Sigma_j}$}
13: **end for**
14: **end while**

3.4 Experiments

We now present experimental results for our algorithm across three different domains: a synthetic dataset, sensor network data, and a statistical

debugging application.

3.4.1 Synthetic Data

Our synthetic datasets consist of a set of 200 objects, each of which consists of 30 samples drawn from one of k randomly generated d-dimensional multivariate Gaussians. The k Gaussians are generated by choosing a mean vector uniformly at random from the unit simplex and randomly selecting a covariance matrix from the set of matrices with eigenvalues $1, 2, ..., d$.

In Figure 3.2, we compare our algorithm to the k-means algorithm, which clusters each object solely on the mean of the samples. Accuracy is quantified in terms of normalized mutual information (NMI) between discovered clusters and the true clusters, a standard technique for determining the quality of clusters [39]. Figure 3.2 (left) shows the clustering quality as a function of the number of clusters when the dimensionality of the input Gaussians is fixed ($d = 4$). Figure 3.2 (right) gives clustering quality for five clusters across a varying number of dimensions. All results represent averaged NMI values across 50 experiments. As can be seen in Figure 3.2, our multivariate Gaussian clustering algorithm yields significantly higher NMI values than k-means for all experiments.

3.4.2 Sensor Networks

Sensor networks are wireless networks composed of small, low-cost sensors that monitor their surrounding environment. An open question in sensor

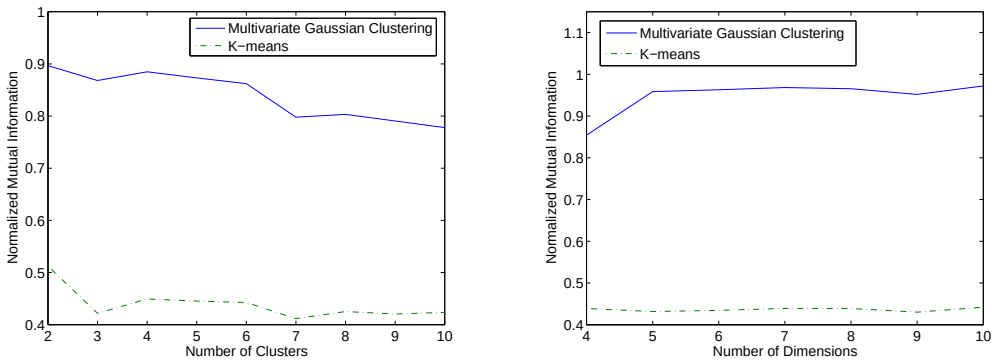


Figure 3.2: Clustering quality of synthetic data. Traditional k-means clustering uses only first-order information (i.e. the mean), whereas our Gaussian clustering algorithm also incorporates second-order covariance information. Here, we see that our algorithm achieves higher clustering quality for datasets composed of four-dimensional Gaussians with varied number of clusters (left), as well as for varied dimensionality of the input Gaussians with $k = 5$ (right).

networks research is how to minimize communication costs between the sensors and the base station: wireless communication requires a relatively large amount of power, a limited resource on current sensor devices (which are usually battery powered).

A recently proposed sensor network system, BBQ [34], reduces communication costs by modelling sensor network data at each sensor device using a time-varying multivariate Gaussian and transmitting only model parameters to the base station. We apply our multivariate Gaussian clustering algorithm to cluster sensor devices from the Intel Lab at Berkeley [72]. Clustering has been used in sensor network applications to determine efficient routing schemes, as well as for discovering trends between groups of sensor devices. The Intel sensor network consists of 52 working sensors, each of which monitors ambient

temperature, humidity, light levels, and voltage every thirty seconds. Conditioned on time, the sensor readings can be fit quite well by a multivariate Gaussian.

Figure 3.3 shows the results of our multivariate Gaussian clustering algorithm applied to this sensor network data. For each device, we compute the sample mean and covariance from sensor readings between noon and 2pm each day, for 36 total days. To account for varying scales of measurement, we normalize all variables to have unit variance. The second cluster (denoted by '2' in figure 3.3) has the largest variance among all clusters: many of the sensors for this cluster are located in high traffic areas, including the large conference room at the top of the lab, and the smaller tables in the bottom of the lab. Since the measurements were taken during lunchtime, we expect higher traffic in these areas. Interestingly, this cluster shows very high covariation between humidity and voltage. Cluster one is characterized by high temperatures, which is not surprising, as there are several windows on the left side of the lab. This window faces west and has an unobstructed view of the ocean. Finally, cluster three has a moderate level of total variation, with relatively low light levels. The cluster is primarily located in the center and the right of lab, away from outside windows.

3.4.3 Statistical Debugging

Leveraging program runtime statistics for the purpose of software debugging has received recent research attention [109]. Here we apply our al-

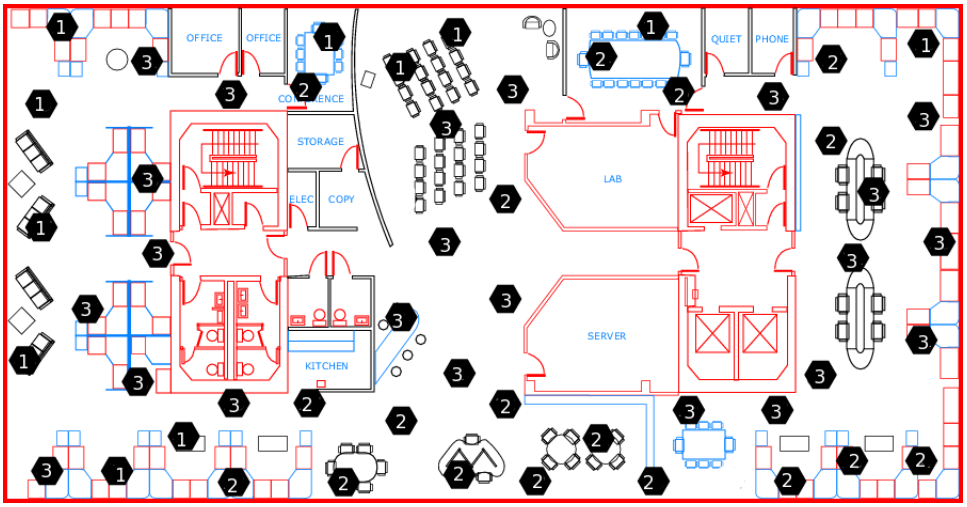


Figure 3.3: To reduce communication costs in sensor networks, each sensor device may be modelled by a multivariate Gaussian. The above plot shows the results of applying our algorithm to cluster sensors into three groups, denoted by labels '1', '2', and '3'.

gorithm to cluster functional behavior patterns over software bugs in the `latex` document preparation program. The data is taken from the Clarify system [49], a system that uses machine learning to provide better error messaging. The dataset contains four software bugs, each of which is caused by an unsuccessful `latex` compilation (e.g. specifying an incorrect number of columns in an array environment) with ambiguous or unclear error messages provided. `Latex` has notoriously cryptic error messages for document compilation failures—for example, the message "LaTeX Error: There's no line here to end" can be caused by numerous problems in the source document. More details of Clarify will be presented in Chapter 8.

Each function in the program's source is measured by the frequency

with which it is called across each of the four software bugs. We model this distribution as a 4-dimensional multivariate Gaussian, one dimension for each bug. The distributions are estimated from a set of samples; each sample corresponds to a single `latex` file drawn from a set of grant proposals and submitted computer science research papers. For each file and for each of the four bugs, the `latex` compiler is executed over a slightly modified version of the file that has been changed to exhibit the bug. During program execution, function counts are measured and recorded. More details can be found in [49].

Clustering these function counts can yield important debugging insight to assist a software engineer in understanding error dependent program behavior. Figure 3.4 shows three covariance matrices from a sample clustering of eight clusters. To capture the dependencies between bugs, we normalize each input Gaussian to have zero mean and unit variance. Cluster (a) represents functions that are highly error independent—i.e. the matrix shows high levels of covariation among all pairs of error classes. Conversely, clusters (b) and (c) show that some functions are highly error dependent. Cluster (b) shows a high dependency between bugs 1 and 4, while cluster (c) exhibits high covariation between bugs 1 and 3, and between bugs 2 and 4.

3.4.4 Clustering Program Features Using Non-Paramatric Kernels

The method presented is derived assuming that the matrices that specify the uncertainty of each object represent the covariance of object's Gaussian representation. Algorithmically, the only condition on these matrices is that

$$\begin{bmatrix} 1.00 & 0.94 & 0.94 & 0.94 \\ 0.94 & 1.00 & 0.94 & 0.94 \\ 0.94 & 0.94 & 1.00 & 0.94 \\ 0.94 & 0.94 & 0.94 & 1.00 \end{bmatrix} \quad \begin{bmatrix} 1.00 & 0.58 & 0.58 & 0.91 \\ 0.58 & 1.00 & 0.55 & 0.67 \\ 0.58 & 0.55 & 1.00 & 0.68 \\ 0.91 & 0.67 & 0.68 & 1.00 \end{bmatrix} \quad \begin{bmatrix} 1.00 & 0.58 & 0.95 & 0.58 \\ 0.58 & 1.00 & 0.58 & 0.95 \\ 0.95 & 0.58 & 1.00 & 0.58 \\ 0.58 & 0.95 & 0.58 & 1.00 \end{bmatrix}$$

(a) (b) (c)

Figure 3.4: Covariance matrices for three clusters discovered by clustering functional behavior of the `latex` document preparation program. Cluster (a) corresponds to functions which are error-independent, while clusters (b) and (c) represent two groups of functions that exhibit different types of error dependent behavior.

they are positive definite. This allows for other, more general means of representing object's uncertainty to be used.

Here, we present a correlation measure that is motivated by statistical software analysis problems. Recall that the covariance estimation presented in the previous section required that program executions be paired up with one another—i.e. multiple runs of `latex` over a single input file were used to comprised a single sample. Over each run, the document is slightly perturbed to exhibit each of the error types. We compute the correlation matrix for each feature by comparing the distributions that it takes on conditioned by each of k class values. Formally, given a feature a, the correlation of feature a given k class values $c = c_i$ and $c = c_j$ is a function of the similarity of the empirical distributions $P(a|c = c_i)$ and $P(a|c = c_j)$. It can be shown that if this similarity is measured in terms of the overlap of the empirical PDFs, then the resulting correlation matrix is guaranteed to be positive (semi) definite. It can also be shown that the rank of this matrix (modulo noise present in the smaller eigenvalues) is equal to the number of factors. Thus, good correlation

1	0.187	0.187	0.188	0.187	0.19	0.187	0.481	0.189
0.187	1	0.896	0.893	0.895	0.868	0.897	0.202	0.891
0.187	0.896	1	0.892	0.893	0.867	0.896	0.202	0.89
0.188	0.893	0.892	1	0.892	0.868	0.893	0.205	0.887
0.187	0.895	0.893	0.892	1	0.871	0.894	0.202	0.889
0.19	0.868	0.867	0.868	0.871	1	0.868	0.2	0.865
0.187	0.897	0.896	0.893	0.894	0.868	1	0.202	0.89
0.481	0.202	0.202	0.205	0.202	0.2	0.202	1	0.201
0.189	0.891	0.89	0.887	0.889	0.865	0.89	0.201	1

Figure 3.5: A cluster representative matrix for the `latex` benchmark. The correlation matrices represent the behavior across 9 error classes.

matrix clusters should retain this rank structure and hence the underlying error behavior. It was observed (although we do not present experiments here), that the LogDet matrix divergence and the Von Neumann matrix divergences resulted in much better clusters than those found using a Frobenius based version of the algorithm.

Here, we use this method to provide anecdotal evidence regarding how the clustering can be used to understand or debug software errors. The following cluster (along with its best rank-2 decomposition) shows a strong correlation between error types 'cline' and 'and'. The full set of errors (in order presented in the matrix) is: e_cline, e_ragged, e_asterisk, e_des, e_center, n, e_num, e_and, and e_foot.

The rank-decomposition is:

0.713 (1):	-0.1	-0.376	-0.375	-0.375	-0.375	-0.368	-0.375	-0.106	-0.374
0.153 (2):	0.703	-0.057	-0.056	-0.054	-0.057	-0.05	-0.056	0.696	-0.055

Two functions contained in this cluster are ‘initalign’, and ‘doendv’, which appear in two different cases of a switch statement with over 100 possibilities (`latex` source code is automatically generated and virtually human unreadable). A fragment of the source (pdftex3.c, lines 1242-1251) is below:

```
  case 235 :
    if ( privileged () )
    if ( curgroup == 15 )
*    initalign () ;
    else offsave () ;
    break ;
  case 10 :
  case 111 :
*    doendv () ;
    break ;
```

Another code fragment involving functions in this cluster is below:

```
#ifdef HAVE_PROTOTYPES
alignpeek ( void )
#else
alignpeek ( )
#endif
{
  /* 20 */ alignpeek_regmem
```

```
  lab20: alignstate = 1000000L ;
  do {
      getxtoken () ;
  } while ( ! ( curcmd != 10 ) ) ;
  if ( curcmd == 34 )
  {
    scanleftbrace () ;
    newsavelevel ( 7 ) ;
    if ( curlist .modefield == -1 )
    normalparagraph () ;
  }
  else if ( curcmd == 2 )
*  finalign () ;
  else if ( ( curcmd == 5 ) && ( curchr == 258 ) )
  goto lab20 ;
  else {

*    initrow () ;
*    initcol () ;
  }
}
```

Here, we see that the 'curcmd' variable is most likely correlated with the error behavior realized in this cluster. Note the complete lack of comments or clues

regarding the purpose of this variable.

3.5 Related Work

In this chapter, we showed that the differential relative entropy between two multivariate Gaussian distributions can be expressed as a convex combination of the Mahalanobis distance between their mean vectors and the LogDet matrix divergence between their covariances. This is in contrast to information theoretic clustering [36], where each input is taken to be a probability distribution over some finite set. In [36], no parametric form is assumed, and the Kullback-Liebler divergence (i.e. discrete relative entropy) can be computed directly from the distributions. The differential entropy between two multivariate Gaussians was considered in [95] in the context of solving Gaussian mixture models. Although an algebraic expression for this differential entropy was given in [95], no connection to the LogDet matrix divergence was made there.

Our algorithm is based on the standard expectation-maximization style clustering algorithm [40]. Although the closed-form updates used by our algorithm are similar to those employed by a Bregman clustering algorithm [7], we note that the computation of the optimal covariance matrix (equation (3.3.8)) involves the optimal mean vector.

In [79], the problem of clustering Gaussians with respect to the symmetric differential relative entropy, $D(f||g) + D(g||f)$ is considered in the context of learning HMM parameters for speech recognition. The resulting algorithm,

however, is much more computationally expensive than ours; in our method, the optimal means and covariance parameters can be computed via a simple closed form solution. In [79], no such solution is presented and an iterative method must instead be employed. The problem of finding the optimal Gaussian with respect to the first argument (note that equation (3.3.5) is minimized with respect to the second argument) is considered in [106] for the problem of speaker interpolation. Here, only one source is assumed, and thus clustering is not needed.

Chapter 4

Feature Correlations and Distance Functions

In this chapter, we discuss some straightforward yet fundamental connections between feature correlations and distance functions. Determining the similarity or distance between two objects should be a function of not only how much each individual feature differs, but should also consider the correlations and dependencies between the various features. Here, we given an example of this connection by illustrating the connections between an arbitrary $(d \times d)$ (positive definite) correlation matrix over a set of d features, and the Mahalanobis distance function, which uses this correlation matrix in determining the distance between instances.

In Section 2.1, we considered the problem of estimating the covariance S given a set of independently drawn samples from a Gaussian distribution. The inverse of this matrix represents the classical parameterization of the well-known Mahalanobis distance function, $d_{S^{-1}}(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x}-\boldsymbol{y})^T S^{-1}(\boldsymbol{x}-\boldsymbol{y}))$. The Mahalanobis distance function is a generalization of the standard squared Euclidean distance, as seen by the case where the sample covariance matrix S is the identity matrix. In cases where data is truly Gaussian, the Mahalanobis distance will be optimal (in a maximum likelihood sense) if parameterized by

S^{-1}. However, if data is not Gaussian distributed, or if "side-chain" information is available (i.e. information about class labels of instances or prior knowledge regarding the distance between points), then this distance function will be more effective if parameterized by some other positive definite matrix A. Note that the positive definiteness constraint ensures that the distance function will always evaluate to a positive number.

The next chapter provides an algorithm for learning such a distance function under limited supervision. The algorithm works by finding a positive definite matrix A that parametrizes a Mahalanobis distance satisfying a given set of constraints. In some sense, this matrix A describes feature correlations with respect to the prior information specified. For example, two variables X and Y may have relatively low correlation, yet when conditioned on some class value or other concept C may result in two highly correlated variables $X|C$ and $Y|C$. This is a very simple yet fundamental point is illustrated in Figure 4.1. Here, we see two variables that are (statistically) uncorrelated. Conditioning these points on the concept C results in strong correlations between the two variables.

4.1 Distance Metric Learning as Constrained Maximum Likelihood

The metric learning algorithm that we will describe in the next chapter is formulated in an information-theoretic context. Before presenting this, we first present an alternate formulation in which the problem can be viewed as

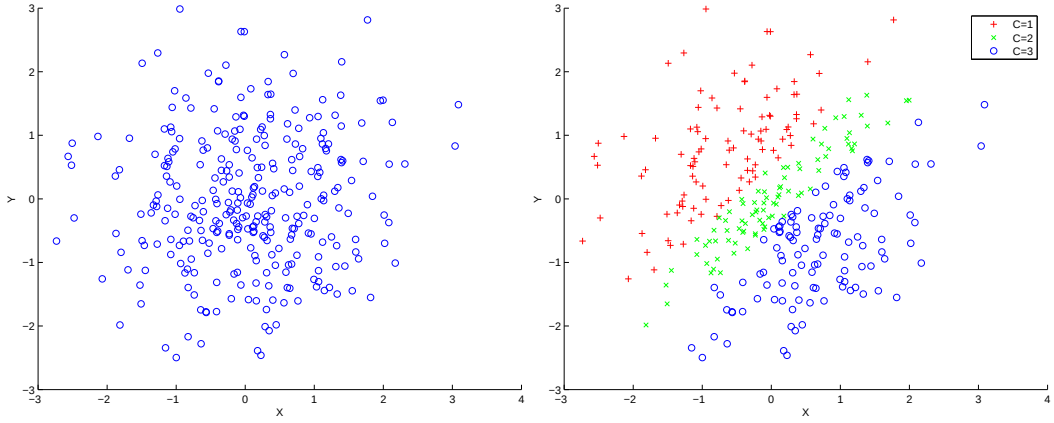


Figure 4.1: This figure illustrates a very simple yet important concept in this thesis: the problem of inferring correlations between features or variables is often dependent on the problem at hand. Here, we see that the variables X and Y are uncorrelated (left figure). However, conditioning these variables on the concept C results in much strong correlations between the two variables.

a constrained maximum likelihood problem.

As described above, the Mahalanobis distance function is classically parameterized by the sample precision matrix A_0 drawn from a multivariate Gaussian distribution, $p(\boldsymbol{x}; A_0) = \frac{1}{Z} \exp\left(-\frac{1}{2} d_{A_o}(\boldsymbol{x}, \boldsymbol{\mu})\right)$. The precision matrix [74] is defined as the inverse of the covariance matrix (i.e. $A_0 = S^{-1}$).

Unfortunately, standard unconstrained maximum likelihood suffers from several drawbacks in the context of estimating precision matrices for use by a Mahalanobis distance. First, the quality of A_0 depends heavily upon the number of data points available. Standard covariance maximum likelihood estimates are well known to yield relatively poor results when data is limited, particularly if the dimensionality is high. Second, such estimates will be biased in cases where the data is not Gaussian. Finally, classical approaches

are fundamentally limited in that they cannot incorporate supervised or semi-supervised information that is present in many problem settings. We saw an example of this in Figure 4.1.

Consider the problem of learning a Mahalanobis matrix as that of a constrained maximum likelihood estimation problem. Given a set of d-dimensional data points, the Wishart distribution [78] defines the probability that a $(d \times d)$ maximum-likelihood covariance matrix S estimated from these samples arises from a Gaussian with covariance Σ. The Wishart distribution is defined as

$$\mathcal{W}(S|\Sigma) = \frac{|S|^{\frac{n}{2}(n-d-1)} \exp(-\frac{n}{2}\operatorname{tr}(S\Sigma^{-1}))}{2^{\frac{nd}{2}}|\Sigma|^{\frac{n}{2}}\Gamma_d(\frac{n}{2})}, \tag{4.1.1}$$

where $\operatorname{tr}(\cdot)$ denotes the matrix trace, the function $\Gamma_d(\cdot)$ denotes the multivariate gamma function [74], and $|\cdot|$ the standard matrix determinant.

We formulate the constrained metric learning problem as that of finding the maximum likelihood precision matrix A with respect to $\mathcal{W}(A_0^{-1}|A^{-1})$ that also satisfies a given set of constraints over a set (or subset) $X = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\}$ of the data. Two points are constrained to be similar if their distance is at most some upper bound u. Similarly, two points are constrained to be dissimilar if their distance is at least some lower bound ℓ. Thus, our Wishart metric learning formulation is

$$\begin{aligned} \max_{A} \quad & \mathcal{W}(A_0^{-1}|A^{-1}) \\ \text{subject to} \quad & d_A(\boldsymbol{x}_i, \boldsymbol{x}_j) \leq u \qquad (i,j) \in S, \\ & d_A(\boldsymbol{x}_i, \boldsymbol{x}_j) \geq \ell \qquad (i,j) \in D. \end{aligned} \tag{4.1.2}$$

The sets S and D represent pairs of points constrained to be similar and dissimilar, respectively.

By taking the log-likelihood of the objective function of problem (4.1.2) with respect to A, we see that the problem can be viewed as a semi-definite programming problem in which the trace objective is regularized by a log-determinant term,

$$\begin{aligned} \log \mathcal{W}(A_0^{-1}|A^{-1}) &= \log |A_0^{-1}|^{\frac{n}{2}(n-d-1)} - \log |A^{-1}|^{\frac{n}{2}} - \frac{n}{2}\operatorname{tr}(A_0^{-1}A) + C \\ &\propto \log |A| - \operatorname{tr}(AA_0^{-1}) - d \qquad (4.1.3) \\ &= D_{\ell \mathrm{d}}(A, A_0) \qquad (4.1.4) \end{aligned}$$

Thus, we see here that our Wishart distribution-based maximum likelihood problem results in an objective in which the log-determinant (LogDet) divergence is minimized.

In the next section, we formally introduce the metric learning problem and provide more details regarding the problem formulation, constraints, and algorithms.

Chapter 5

Information-Theoretic Metric Learning

Metric learning can be viewed as a correlation inference procedure in which the correlation estimates are used to parametrize a distance function. This chapter presents a metric learning formulation based on the LogDet divergence [31].

5.1 Problem Overview

Selecting an appropriate distance measure (or metric) is fundamental to many learning algorithms such as k-means, nearest neighbor searches, and others. However, choosing such a measure is highly problem-specific and ultimately dictates the success—or failure—of the learning algorithm. To this end, there have been several recent approaches that attempt to learn distance functions. These methods work by exploiting distance information that is intrinsically available in many learning settings. For example, in the problem of semi-supervised clustering, points are constrained to be either similar (i.e. the distance between them should be relatively small) or dissimilar (the distance should be larger). In information retrieval settings, constraints between pairs of distances can be gathered from click-through feedback. In fully su-

pervised settings, constraints can be inferred so that points in the same class have smaller distances to each other than to points in different classes.

While existing algorithms for metric learning have been shown to perform well across various learning tasks, each fail to satisfy some basic requirements. First, a metric learning algorithm should be sufficiently flexible to support the variety of constraints realized across different learning paradigms. Second, the algorithm must be able to learn a distance function that generalizes well to new data, i.e. to test data. Finally, the algorithm should be fast and efficient.

In Chapter 4, we showed how pairwise feature correlations can be used to parametrize a Mahalanobis distance function. If these correlations are inferred using standard Gaussian-based maximum-likelihood methods, the resulting Mahalanobis matrix will be learnt in an unsupervised manner. In this chapter, we propose an approach to learning such a distance function using limited supervision. We model the problem in an information-theoretic setting by leveraging the relationship between the multivariate Gaussian distribution and the set of Mahalanobis distances. We translate the problem of learning an optimal distance metric to that of learning the optimal Gaussian with respect to an entropic objective. Our formulation can be viewed as a maximum entropy objective: maximize the differential entropy of a multivariate Gaussian subject to constraints on the associated Mahalanobis distance.

Our problem formulation is quite general: we can accommodate a range of constraints, including similarity constraints, dissimilarity constraints, and

relations between pairs of distances. Further, we can incorporate prior information regarding the distance function itself. For some problems, standard Euclidean distance may work well. In others, parameterizing a Mahalanobis distance using the inverse of the sample covariance may yield reasonable results. In such cases, our formulation finds a distance function that is 'close' to the specified distance function while also satisfying the given constraints.

We show an interesting connection to a recently proposed low-rank kernel learning problem [61], where a low-rank kernel K is learned that satisfies a set of given distance constraints by minimizing the LogDet divergence to a given kernel K_0. This equivalence allows the algorithm to be kernelized, resulting in an optimization over a larger class of non-linear distance functions. Algorithmically, the equivalence also implies that the problem can be solved very efficiently: it was shown that the kernel learning problem can be optimized using an iterative optimization procedure with cost $O(cd^2)$ per iteration, where c is the number of distance constraints, and d is the dimensionality of the data. In particular, this method does not require costly eigenvalue computations or semi-definite programming.

To demonstrate our algorithm's ability to learn a distance function that generalizes well to unseen points, we compare our method to existing metric learning algorithms. We apply the algorithms to Clarify, a recently developed system that classifies software errors using machine learning techniques. Here, we show that our algorithm effectively learns a metric for the problem of nearest neighbor software support. Furthermore, on standard UCI datasets,

we show that our algorithm consistently equals or outperforms the best existing methods when used to learn a distance function for k-NN classification.

5.2 Related Work

Most of the existing work in metric learning relies on learning a Mahalanobis distance, which has been found to be a sufficiently powerful class of metrics that work on many real-world problems. Earlier work by [105] uses a semidefinite programming formulation under similarity and dissimilarity constraints. More recently, [103] formulate the metric learning problem in a large margin setting, with a focus on k-NN classification. They also formulate the problem as a semidefinite programming problem and consequently solve it using a combination of sub-gradient descent and alternating projections. Globerson et. al. [46] proceed to learn a metric in the fully supervised setting. Their formulation seeks to 'collapse classes' by constraining within class distances to be zero while maximizing the between class distances. While each of these algorithms was shown to yield excellent classification performance, their constraints do not generalize outside of their particular problem domains; in contrast, our approach allows arbitrary linear constraints on the Mahalanobis matrix. Furthermore, these algorithms all require eignevalue decompositions, an operation that is cubic in the dimensionality of the data.

Other notable work wherein the authors present methods for learning Mahalanobis metrics includes [92] (online metric learning), Relevant Components Analysis (RCA) [93] (similar to discriminant analysis), locally-adaptive

discriminative methods [53], and learning from relative comparisons [89].

Non-Mahalanobis based metric learning methods have also been proposed, though these methods usually suffer from suboptimal performance, non-convexity, or computational complexity. Some example methods include neighborhood component analysis (NCA) [47] that learns a distance metric specifically for nearest-neighbor based classification; convolutional neural net based methods of [22]; and a general Riemannian metric learning method [64].

5.3 Problem Formulation

Given a set of n points $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\}$ in $\mathbb{R}^d$, we seek a positive definite matrix A which parameterizes the (squared) Mahalanobis distance.

$$d_A(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i - \boldsymbol{x}_j)^T A(\boldsymbol{x}_i - \boldsymbol{x}_j). \tag{5.3.1}$$

We assume that prior knowledge is known regarding interpoint distances. Consider relationships constraining the similarity or dissimilarity between pairs of points. Two points are similar if the Mahalanobis distance between them is smaller than a given upper bound, i.e., $d_A(\boldsymbol{x}_i, \boldsymbol{x}_j) \leq u$ for a relatively small value of u. Similarly, two points are dissimilar if $d_A(\boldsymbol{x}_i, \boldsymbol{x}_j) \geq \ell$ for sufficiently large ℓ. Such constraints are typically inputs for many semi-supervised learning problems, and can also be readily inferred in a classification setting where class labels are known for each instance: distances between points in the same class can be constrained as similar, and points in different classes can be constrained as dissimilar.

Given a set of interpoint distance constraints as described above, our problem is to learn a positive-definite matrix A that parameterizes the corresponding Mahalanobis distance (5.3.1). Typically, this learned distance function is used to improve the accuracy of a k-nearest neighbor classifier, or to incorporate semi-supervision into a distance-based clustering algorithm. In many settings, prior information about the Mahalanobis distance function itself is known. In settings where data is Gaussian, parameterizing the distance function by the inverse of the sample covariance may be appropriate. In other domains, squared Euclidean distance (i.e. the Mahalanobis distance corresponding to the identity matrix) may work well empirically. Thus, we regularize the Mahalanobis matrix A to be as close as possible to a given Mahalanobis distance function, parameterized by A_0.

We now quantify the measure of "closeness" between A and A_0 via a natural information-theoretic approach. There exists a simple bijection (up to a scaling function) between the set of Mahalanobis distances and the set of equal-mean multivariate Gaussian distributions (without loss of generality, we can assume the Gaussians have mean $\boldsymbol{\mu}$). Given a Mahalanobis distance parameterized by A, we express its corresponding multivariate Gaussian as $p(\boldsymbol{x}; A) = \frac{1}{Z} \exp\left(-\frac{1}{2} d_A(\boldsymbol{x}, \boldsymbol{\mu})\right)$, where Z is a normalizing constant and A^{-1} is the covariance of the distribution. Using this bijection, we measure the distance between two Mahalanobis distance functions parameterized by A_0 and A by the (differential) relative entropy between their corresponding multivariate

Gaussians:

$$\mathrm{KL}(p(\boldsymbol{x}; A_o) \| p(\boldsymbol{x}; A)) = \int p(\boldsymbol{x}; A_0) \log \frac{p(\boldsymbol{x}; A_0)}{p(\boldsymbol{x}; A)} \, d\boldsymbol{x}. \tag{5.3.2}$$

The distance (5.3.2) provides a well-founded measure of "closeness" between two Mahalanobis distance functions and forms the basis of our problem given below. This result was derived for the general case between two Gaussian distributions with arbitrary means in Chapter 3.

Given pairs of similar points S and pairs of dissimilar points D, our distance metric learning problem is

$$\begin{aligned} \min_{A} \quad & \mathrm{KL}(p(\boldsymbol{x}; A_0) \| p(\boldsymbol{x}; A)) \\ \text{subject to} \quad & d_A(\boldsymbol{x}_i, \boldsymbol{x}_j) \leq u \qquad (i,j) \in S, \\ & d_A(\boldsymbol{x}_i, \boldsymbol{x}_j) \geq \ell \qquad (i,j) \in D. \end{aligned} \tag{5.3.3}$$

In the above formulation, we consider simple distance constraints for similar and dissimilar points; however, it is straightforward to incorporate other constraints. For example, [89] consider a formulation where the distance metric is learned subject to relative nearness constraints (as in, the distance between i and j is closer than the distance between i and k). Our approach can be easily adapted to handle this setting. In fact, it is possible to incorporate arbitrary linear constraints into our framework in a straightforward manner. For simplicity, we present the algorithm under the simple distance constraints given above.

5.4 Algorithm

In this section, we first show that our information-theoretic objective (5.3.3) can be expressed as a particular type of Bregman divergence, which allows us to adapt Bregman's method [18] to solve the metric learning problem. We then show a surprising equivalence to a recently-proposed low-rank kernel learning problem [61], allowing kernelization of the algorithm.

5.5 Metric Learning as LogDet Optimization

It is known that the differential relative entropy between two multivariate Gaussians can be expressed as the convex combination of a Mahalanobis distance between mean vectors and the LogDet divergence between the covariance matrices. Assuming the means to be the same, we have

$$\mathrm{KL}(p(\boldsymbol{x}; A) \| p(\boldsymbol{x}; A_0)) = \frac{1}{2} D_{\ell \mathrm{d}}(A, A_0), \tag{5.5.1}$$

where $D_{\ell \mathrm{d}}(A, A_0)$ denotes the LogDet divergence between matrices A and A_0. The LogDet divergence is a Bregman matrix divergence generated by the convex function $\phi(X) = -\log \det X$ over the cone of positive-definite matrices, and it is defined as (for $n \times n$ matrices A, A_0)

$$D_{\ell \mathrm{d}}(A, A_0) = \mathrm{tr}(AA_0^{-1}) - \log |AA_0^{-1}| - n. \tag{5.5.2}$$

Historically the LogDet divergence originated as Stein's loss in the work of [58]. It can be shown that Stein's loss is the unique *scale invariant* loss-function for which the uniform minimum variance unbiased estimator is also a minimum

risk equivariant estimator [65]. In the context of metric learning, the scale invariance implies that the divergence (5.5.2) remains invariant under any scaling of the feature space. The result can be further generalized to invariance under any invertible linear transformation S, i.e.

$$D_{\ell\mathrm{d}}(S^T A S, S^T B S) = D_{\ell\mathrm{d}}(A, B). \tag{5.5.3}$$

We can exploit the equivalence in (5.5.1) to express the distance metric learning problem (5.3.3) as the following LogDet optimization problem:

$$\begin{aligned} \min_{A} \quad & D_{\ell\mathrm{d}}(A, A_0) \\ \text{subject to} \quad & \operatorname{tr}(A(\boldsymbol{x}_i - \boldsymbol{x}_j)(\boldsymbol{x}_i - \boldsymbol{x}_j)^T) \leq u \qquad (i,j) \in S, \\ & \operatorname{tr}(A(\boldsymbol{x}_i - \boldsymbol{x}_j)(\boldsymbol{x}_i - \boldsymbol{x}_j)^T) \geq \ell \qquad (i,j) \in D, \\ & A \succeq 0. \end{aligned} \tag{5.5.4}$$

Note that the distance constraints on $d_A(\boldsymbol{x}_i, \boldsymbol{x}_j)$ translate into the above linear constraints on A.

In some cases, there may not exist a feasible solution to (5.5.4). To prevent such a scenario from occurring, we incorporate *slack variables* into the formulation to guarantee the existence of a feasible A. Let $c(i,j)$ denote a function that indexes all constraints, and let $\boldsymbol{\xi}$ be a vector of slack variables, initialized to $\boldsymbol{\xi}_0$ (whose components equal u for similarity constraints and ℓ for dissimilarity constraints). Then we can pose the following optimization

problem:

$$\begin{aligned} \min_{A} \quad & D_{\ell \mathrm{d}}(A, A_0) + \gamma \cdot D_{\ell \mathrm{d}}(\mathrm{diag}(\boldsymbol{\xi}), \mathrm{diag}(\boldsymbol{\xi}_0)) \\ \text{s. t.} \quad & \mathrm{tr}(A(\boldsymbol{x}_i - \boldsymbol{x}_j)(\boldsymbol{x}_i - \boldsymbol{x}_j)^T) \leq \xi_{c(i,j)} \qquad (i,j) \in S, \\ & \mathrm{tr}(A(\boldsymbol{x}_i - \boldsymbol{x}_j)(\boldsymbol{x}_i - \boldsymbol{x}_j)^T) \geq \xi_{c(i,j)} \qquad (i,j) \in D, \\ & A \succeq 0. \end{aligned} \tag{5.5.5}$$

The parameter γ controls the tradeoff between satisfying the constraints and minimizing $D_{\ell \mathrm{d}}(A, A_0)$.

To solve the optimization problem (5.5.5), we extend the methods from [61]. The inputs to the algorithm are the starting Mahalanobis matrix A_0, the constraint data, and the slack parameter γ. The optimization method which forms the basis for the algorithm repeatedly computes Bregman projections—that is, projections of the current solution onto a single constraint. This projection is performed via the update

$$A_{t+1} = A_t + \beta A_t (\boldsymbol{x}_i - \boldsymbol{x}_j)(\boldsymbol{x}_i - \boldsymbol{x}_j)^T A_t, \tag{5.5.6}$$

where $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are the constrained data points, and β is a parameter of the projection computed by the algorithm. Each constraint projection costs $O(d^2)$, and so a single iteration of looping through all constraints costs $O(cd^2)$. We stress that no eigen-decomposition is required in the algorithm. Note that, if necessary, the projections can be computed efficiently over a factorization W of the Mahalanobis matrix, such that $A = W^T W$.

5.5.1 Equivalence to Low-Rank Kernel Learning

We now discuss a surprising equivalence between our metric learning optimization problem and a low-rank kernel learning problem. In addition to showing a connection between metric learning and kernel learning, this equivalence will be essential for kernelization of our metric learning algorithm. In this kernel learning formulation, a kernel K is optimized to satisfy a set of linear constraints while minimizing the LogDet divergence to a specified kernel K_0. Given $X = [\boldsymbol{x}_1\ \boldsymbol{x}_2\ ...\ \boldsymbol{x}_n]$, and the input $n \times n$ kernel matrix $K_0 = X^T A_0 X$, the optimization problem is:

$$\begin{aligned} \min_{K} \quad & D_{\ell \mathrm{d}}(K, K_0) \\ \text{subject to} \quad & K_{ii} + K_{jj} - 2K_{ij} \leq u \qquad (i,j) \in S, \\ & K_{ii} + K_{jj} - 2K_{ij} \geq \ell \qquad (i,j) \in D, \\ & K \succeq 0. \end{aligned} \tag{5.5.7}$$

In addition to being convex in the first argument, the LogDet divergence between two matrices is finite if and only if their range spaces are the same [61]. Thus, the learned matrix K can be written as a kernel $X^T A X$, for some $(d \times d)$ positive definite matrix A. The results below can be easily generalized to incorporate slack variables in (5.5.7).

First we establish that the feasible solutions to (5.3.3) coincide with the feasible solutions to (5.5.7).

Lemma 1. Given that $K = X^T A X$, A is feasible for (5.3.3) if and only if K is feasible for (5.5.7).

Proof. The expression $K_{ii}+K_{jj}-2K_{ij}$ can be written as $(\boldsymbol{e}_i-\boldsymbol{e}_j)^T K(\boldsymbol{e}_i-\boldsymbol{e}_j)$, or equivalently $(\boldsymbol{x}_i-\boldsymbol{x}_j)^T A(\boldsymbol{x}_i-\boldsymbol{x}_j) = d_A(\boldsymbol{x}_i,\boldsymbol{x}_j)$. Thus, if we have a kernel matrix K satisfying constraints of the form $K_{ii}+K_{jj}-2K_{ij} \leq u$ or $K_{ii}+K_{jj}-2K_{ij} \geq \ell$, we equivalently have a matrix A satisfying $d_A(\boldsymbol{x}_i,\boldsymbol{x}_j) \leq u$ or $d_A(\boldsymbol{x}_i,\boldsymbol{x}_j) \geq \ell$. □

We can now show an explicit relationship between the optimal solution to (5.3.3) and (5.5.7).

Theorem 1. Let A^* be the optimal solution to (5.3.3) and K^* be the optimal solution to (5.5.7). Then $K^* = X^T A^* X$.

Proof. We give a constructive proof for the theorem. The Bregman projection update for (5.5.4) is expressed as

$$A_{t+1} = A_t + \beta A_t(\boldsymbol{x}_i-\boldsymbol{x}_j)(\boldsymbol{x}_i-\boldsymbol{x}_j)^T A_t. \tag{5.5.8}$$

Similary, the Bregman update for (5.5.7) is expressed as

$$K_{t+1} = K_t + \beta K_t(\boldsymbol{e}_i-\boldsymbol{e}_j)(\boldsymbol{e}_i-\boldsymbol{e}_j)^T K_t. \tag{5.5.9}$$

It is straightforward to prove that the value of β is the same for (5.5.9) and (5.5.8). We can inductively prove that at each iteration t, updates K_t and A_t satisfy, $K_t = X^T A_t X$. At the first step, $K_0 = X^T A_0 X$, so the base case trivially holds. Now, assume that $K_t = X^T A_t X$; by the Bregman projection update, K_{t+1}

$$\begin{aligned}
&= X^T A_t X + \beta X^T A_t X(\boldsymbol{e}_i-\boldsymbol{e}_j)(\boldsymbol{e}_i-\boldsymbol{e}_j)^T X^T A_t X \\
&= X^T A_t X + \beta X^T A_t(\boldsymbol{x}_i-\boldsymbol{x}_j)(\boldsymbol{x}_i-\boldsymbol{x}_j)^T A_t X \\
&= X^T(A_t + \beta A_t(\boldsymbol{x}_i-\boldsymbol{x}_j)(\boldsymbol{x}_i-\boldsymbol{x}_j)^T A^T)X.
\end{aligned}$$

Comparing with (5.5.6), we see that $K_{t+1} = X^T A_{t+1} X$. Since the method of Bregman projections converges to the optimal A^* and K^* of (5.3.3) and (5.5.7), respectively [18], we have $K^* = X^T A^* X$. Hence, the metric learning (5.3.3) and the kernel learning (5.5.7) problems are equivalent. □

We have proven that the information-theoretic metric learning problem is related to a low-rank kernel learning problem. We can easily modify the algorithm to optimize for K—this is necessary in order to kernelize the algorithm. As input, the algorithm provides K_0 instead of A_0, the distance between two points is computed as $K_{ii} + K_{jj} - 2K_{ij}$, the projection is performed using (5.5.9), and the output is K.

5.6 Kernelizing the Algorithm

We now consider kernelizing our metric learning algorithm. In this section, we assume that $A_0 = I$; that is, the maximum entropy formulation that regularizes to the baseline Euclidean distance. Kernelizing for other choices of A_0 is possible, but not presented. If $A_0 = I$, the corresponding K_0 from the low-rank kernel learning problem is $K_0 = X^T X$, the Gram matrix of the inputs. If instead of an explicit representation X of our data points, we have as input a kernel function $\kappa(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x})^T \phi(\mathbf{y})$, along with the associated kernel matrix K_0 over the training points, a natural question to ask is whether we can evaluate the learned metric on new points in the kernel space. This

requires the computation of

$$d_A\big(\phi(\boldsymbol{x}), \phi(\boldsymbol{y})\big) = \big(\phi(\boldsymbol{x}) - \phi(\boldsymbol{y})\big)^T A\big(\phi(\boldsymbol{x}) - \phi(\boldsymbol{y})\big),$$
$$= \phi(\boldsymbol{x})^T A\phi(\boldsymbol{x}) - 2\phi(\boldsymbol{x})^T A\phi(\boldsymbol{y}) + \phi(\boldsymbol{y})^T A\phi(\boldsymbol{y}).$$

We now define a new kernel function $\tilde{\kappa}(\boldsymbol{x}, \boldsymbol{y}) = \phi(\boldsymbol{x})^T A\phi(\boldsymbol{y})$. The ability to generalize to unseen data points reduces to the ability to compute $\tilde{\kappa}(\boldsymbol{x}, \boldsymbol{y})$. Note that the size of the matrix A is equal to the dimensionality of $\phi(\boldsymbol{x})$, which can potentially be infinite (if the original kernel function is the Gaussian kernel, for example). This makes working with A impossible in general.

Even though we cannot explicitly compute A, it is still possible to compute $\tilde{\kappa}(\boldsymbol{x}, \boldsymbol{y})$. As $A_0 = I$, we can recursively "unroll" the learned A matrix so that it is of the form

$$A = I + \sum_{i,j} \sigma_{ij} \phi(\boldsymbol{x}_i)\phi(\boldsymbol{x}_j)^T.$$

This follows by expanding Equation (5.5.6) down to I. The new kernel function is therefore computed as

$$\tilde{\kappa}(\boldsymbol{x}, \boldsymbol{y}) = \kappa(\boldsymbol{x}, \boldsymbol{y}) + \sum_{i,j} \sigma_{ij} \kappa(\boldsymbol{x}, \boldsymbol{x}_i)\kappa(\boldsymbol{y}, \boldsymbol{x}_j),$$

and is a function of the original kernel function κ and the σ_{ij} coefficients. The σ_{ij} coefficients may be updated while minimizing $D_{\ell\mathrm{d}}(K, K_0)$ without affecting the asymptotic running time of the algorithm; in other words, by optimizing the low-rank kernel learning problem (5.5.7) for K, we can obtain the necessary coefficients σ_{ij} for evaluation of $\tilde{\kappa}(\boldsymbol{x}, \boldsymbol{y})$. This leads to a method for finding

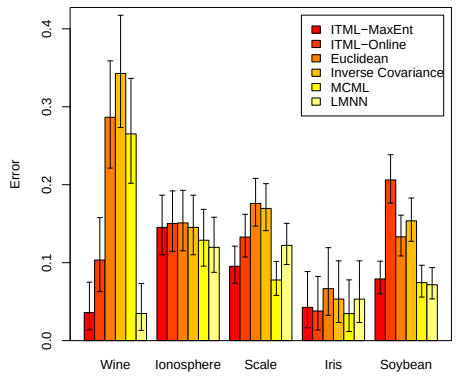
the nearest neighbor of a new data point in the kernel space under the learned metric which can be performed in $O(n^2)$ total time.

5.7 Experiments

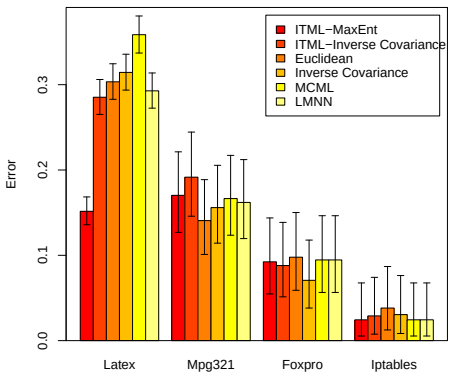
We compare our Information Theoretic Metric Learning algorithm (ITML) to existing methods across two applications: semi-supervised clustering and k-nearest neighbor (k-NN) classification.

We evaluate metric learning for k-NN classification via two-fold cross validation with $k = 4$. All results presented represent the average over 5 runs. Binomial confidence intervals are given at the 95% level.

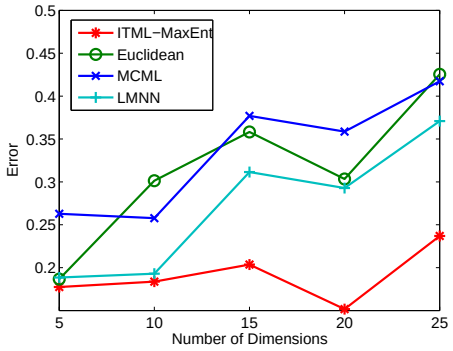
To establish the lower and upper bounds of the right hand side of our constraints (ℓ and u in problem (5.3.3)), we use (respectively) the 5^{th} and 95^{th} percentiles of the observed distribution of distances between pairs of points within the dataset. To determine the constrained points, we randomly choose $20c^2$ points, where c is the number of classes in the dataset. Points in the same class are constrained to be similar, and points with differing class labels are constrained to be dissimilar. Overall, we found the algorithm to be robust to these parameters. However, we did find that the variance between runs increased if the number of constraints used was too small (i.e., fewer than $10c^2$). The slack variable parameter, γ, is tuned using cross validation over the values $\{.01, .1, 1, 10\}$. Finally, the online algorithm is run for approximately 10^5 iterations. More details regarding the online variant of ITML and this learning rate can be found in [31].



(a) UCI Datasets



(b) Clarify Datasets



(c) Latex

Figure 5.1: Classification error rates for k-nearest neighbor classification via different learned metrics. We see in figures (a) and (b) that ITML-MaxEnt is the only algorithm to be optimal (within the 95% confidence intervals) across all datasets. ITML is also robust at learning metrics over higher dimensions. In (c), we see that the error rate for the Latex dataset stays relatively constant for ITML.

In Figure 5.1(a), we compare ITML-MaxEnt (regularized to the identity matrix) and the online ITML algorithm against existing metric learning methods for k-NN classification. We use the squared Euclidean distance, $d(\mathbf{x}, \mathbf{y}) = (\mathbf{x} - \mathbf{y})^{\mathbf{T}}(\mathbf{x} - \mathbf{y})$ as a baseline method. We also use a Mahalanobis distance parameterized by the inverse of the sample covariance matrix. This method is equivalent to first performing a standard PCA whitening transform over the feature space and then computing distances using the squared Euclidean distance. We compare our method to two recently proposed algorithms: Maximally Collapsing Metric Learning [46] (MCML), and metric learning via Large Margin Nearest Neighbor [103] (LMNN). Consistent with existing work [46], we found the method of [105] to be very slow and inaccurate. Overall, ITML is the only algorithm to obtain the optimal error rate (within the specified 95% confidence intervals) across all datasets. For several datasets, the online version is competitive with the best metric learning algorithms. We also observed that the learning rate η remained fairly constant, yielding relatively small regret bounds.

In addition to our evaluations on standard UCI datasets, we also apply our algorithm to the recently proposed problem of nearest neighbor software support for the Clarify system [49]. The basis of the Clarify system lies in the fact that modern software design promotes modularity and abstraction. When a program terminates abnormally, it is often unclear which component should be responsible for (or is capable of) providing an error report. The system works by monitoring a set of predefined program features (the datasets pre-

sented use function counts) during program runtime which are then used by a classifier in the event of abnormal program termination. Nearest neighbor searches are particularly relevant to this problem. Ideally, the neighbors returned should not only have the correct class label, but should also represent those with similar program configurations or program inputs. Such a matching can be a powerful tool to help users diagnose the root cause of their problem. More details of Clarify are provided in Chapter 8. The four datasets shown here are Latex (the document compiler, 9 classes), Mpg321 (an mp3 player, 4 classes), Foxpro (a database manager, 4 classes), and Iptables (a Linux kernel application, 5 classes).

The dimensionality of the Clarify dataset can be quite large. However, it was shown [49] that high classification accuracy can be obtained by using a relatively small subset of available features. Thus, for each dataset, we use a standard information gain feature selection test to obtain a reduced feature set of size 20. From this, we learn metrics for k-NN classification using the above described procedure. We also evaluate the method ITML-Inverse Covariance, which regularizes to the inverse covariance matrix. Results are given in Figure 5.1(b). The ITML-MaxEnt algorithm yields significant gains for the Latex benchmark. Note that for datasets where Euclidean distance performs better than using the inverse covariance metric, the ITML-MaxEnt algorithm that normalizes to the standard Euclidean distance yields higher accuracy than that regularized to the inverse covariance matrix (ITML-Inverse Covariance). In general, for the Mpg321, Foxpro, and Iptables datasets, learned metrics yield

Dataset	ITML-MaxEnt	MCML	LMNN
Latex	**0.0517**	19.8	0.538
Mpg321	**0.0808**	0.460	0.253
Foxpro	**0.0793**	0.152	0.189
Iptables	0.149	**0.0838**	4.19

Table 5.1: Training time (in seconds) for the results presented in Figure 5.1(b).

only marginal gains over the baseline Euclidean distance measure.

Figure 5.1(c) shows the error rate for the Latex datasets with a varying number of features (the feature sets are again chosen using the information gain criteria). We see here that ITML is surprisingly robust. Euclidean distance, MCML, and LMNN all achieve their best error rates for five dimensions. ITML, however, attains its lowest error rate of .15 at $d = 20$ dimensions.

In Table 5.1, we see that ITML generally learns metrics significantly faster than other metric learning algorithms. The implementations for MCML and LMNN were obtained from their respective authors. The timing tests were run on a dual processor 3.2 GHz Intel Xeon processor running Ubuntu Linux. Time given is in seconds and represents the average over 5 runs.

Finally, we present some semi-supervised clustering results. Note that both MCML and LMNN are not amenable to optimization subject to pairwise distance constraints. Instead, we compare our method to the semi-supervised clustering algorithm HMRF-KMeans [10]. We use a standard 2-fold cross validation approach for evaluating semi-supervised clustering results. Distances are constrained to be either similar or dissimilar, based on class values, and are drawn only from the training set. The entire dataset is then clustered into c

Dataset	Unsupervised	ITML	HMRF-KMeans
Ionosphere	0.314	**0.113**	0.256
Digits-389	0.226	**0.175**	0.286

Table 5.2: Unsupervised k-means clustering error, along with semi-supervised clustering error with 50 constraints.

clusters using k-means (where c is the number of classes) and error is computed using only the test set. Table 5.2 provides results for the baseline k-means error, as well as semi-supervised clustering results with 50 constraints. As with the classification experiments, error is defined as the number of correctly clustered examples divided by the total number of examples. Class predictions are determined from cluster labels by finding the mapping with minimum error.

Chapter 6

Structured Metric Learning for High Dimensional Data

In Chapter 5, we presented a method for learning full-rank Mahalanobis distances. Our method was formulated using the LogDet divergence, and we provided an algorithm with running time quadratic in the dimensionality. Experimentally, we showed that ITML has excellent generalization performance for *low* dimensional problems. Here, we present methods for learning distance metrics when the number of dimensions is *high* [29].

6.1 Problem Overview

In *high* dimensional settings, the problem of learning and evaluating a Mahalanobis distance function and with its associated $(d \times d)$ matrix becomes quickly intractable due to the quadratic dependency on d. This quadratic dependency effects not only the running time for both training and testing, but also poses tremendous challenges in estimating a quadratic number of parameters. For example, a data set with 10,000 dimensions requires estimation of a symmetric positive definite matrix with roughly 50 million parameters! This represents a fundamental limitation of existing approaches, as many modern

data mining problems possess relatively high dimensionality. For example, in text-analysis domains, standard bag-of-words models can reach the size of thousands or even tens of thousands of features. Statistical software analysis applications that monitor program paths or method counts similarly have features sets with sizes of thousands or more. Finally, in collaborative filtering domains, objects are typically rated by thousands or even millions of reviewers. Methods in these domains typically compare content (e.g. movies, songs, etc.) using a representation in which each reviewer can be viewed as a single feature.

In this chapter, we present algorithms for learning structured Mahalanobis distance functions that scale linearly with the dimensionality [29]. Instead of searching for a full $(d \times d)$ matrix with $O(d^2)$ parameters, our methods search for compressed representations that typically have $O(d)$ parameters. This enables the Mahalanobis distance function to be learned, stored, and evaluated efficiently in the context of high dimensionality.

In particular, the technical contributions in this chapter are the problem formulations and algorithms that compute two types of structured low parameter matrices: a low-rank representation, and a diagonal plus low-rank representation. The low-rank representation, HDLR, results in a distance measure which is similar to that used by latent semantic analysis (LSA) [33]. This distance projects data into a low dimensional factor space, and the resulting distance between two examples is the distance between their projections. Our low-rank method can be viewed as a semi-supervised variant of LSA, and

is well suited for applications in which higher recall is desired. The second method, HD^{2}LR, learns a diagonal plus low-rank matrix, and is well suited for problems where both high recall as well as high precision are important. This is achieved by comparing examples at both the factor level in addition to a component that compares examples at a much finer, feature-level resolution.

Computationally, our algorithms are based on the information-theoretic metric learning (ITML) method presented in [31] and described in Chapter 5. Recall that the problem is formulated as that of learning a maximum entropy Mahalanobis distance that also satisfies a given set of constraints. Mathematically, this results in a convex programming problem with a matrix-valued objective function called the log-determinant (LogDet) divergence. We provide two new algorithms based on the LogDet divergence that enable learning Mahalanobis distances in high dimensions. Both of these algorithm scale linearly with dimensionality as $O(d)$.

Experimentally, we evaluate our methods in the context of several modern domains, including text, statistical software analysis, and collaborative filtering. We provide experimental evidence that existing metric learning algorithms do not scale to high dimensional data sets, while demonstrating that our methods can easily handle data with upwards of ten thousand dimensions. We compare our methods in the context of learning metrics for nearest neighbor searches on the basis of several criteria, including accuracy, precision and recall. As baseline measures, we use the standard Euclidean distance and LSA. Additionally, we compare our methods against a heuristic in which an existing

full-rank metric learning algorithm LMNN [103] is used to learn a low-rank distance function. In general, we show that our low-parameter metric learning algorithms learn high quality distance functions. For example, classification accuracy for the Classic3 data set is improved by 24% over standard Euclidean distance measures. Additionally, our methods achieve precision as high as all other methods while yielding recall values up to 20% higher than a baseline LSA approach.

The chapter is organized as follows. Section 6.2 introduces the problem and provides some background on related and existing metric learning methods. Section 6.3 outlines limitations of these methods and provides two low-parameter Mahalanobis distance forms. Section 6.4 formalizes our low parameter metric learning problems using the aforementioned proposed distance functions, and we provide efficient and scalable algorithms. Finally, we present experimental results in section 6.6.

6.2 Background

As we saw in the previous chapter where we introduced our Information-theoretic metric learning (ITML) algorithm, the Mahalanobis distance is one class of distance measures that has shown excellent generalization properties. Here, we provide additional background on existing metric learning algorithms. One commonality among existing methods is the regularization term found in the problem objective. In Xing et. al., a method is presented in which the learned matrix A is optimized with respect to a sum-of-squares Frobenius ob-

jective [105]. The LMNN method is formulated as a semi-definite programming problem with a linear objective over the trace of the matrix A. The ITML method seeks a matrix A that minimizes the differential relative entropy to some baseline matrix A_0. Mathematically, this entropic objective results in a problem objective that minimizes the *log-determinant* (LogDet) divergence with respect to some baseline matrix A_0.

The methods we present in this paper attempt to learn a structured positive semi-definite matrix using the LogDet problem framework, and are similar to ITML, which we now describe in detail. The problem assumes a given set of similarity constraints S and dissimilarity constraints D between pairs of examples. Constraints may be inferred from true labels (where examples in the same class are constrained to be similar and examples from different classes are constrained to be dissimilar), or in semi-supervised settings where constraints are explicitly provided. Other constraints that are linear in the entries of A can also be easily incorporated. Additionally, ITML assumes a baseline Mahalanobis distance function parametrized by a positive definite matrix A_0. The formal goal of the problem is to learn a Mahalanobis distance parametrized by A that has minimum LogDet divergence to a given baseline matrix A_0 while satisfying the given constraints:

$$\begin{aligned} \min_{A} \quad & D_{\ell \mathrm{d}}(A|A_0) \\ \text{subject to} \quad & d_A(\boldsymbol{x}_i, \boldsymbol{x}_j) \leq u \qquad (i,j) \in S, \\ & d_A(\boldsymbol{x}_i, \boldsymbol{x}_j) \geq \ell \qquad (i,j) \in D. \end{aligned} \tag{6.2.1}$$

The LogDet objective function $D_{\ell \mathrm{d}}(A|A_0)$ is a non-negative, matrix-valued convex function that in the absence of constraints is minimized when $A = A_0$. It is defined over $(d \times d)$ positive semi-definite matrices A and A_0:

$$D_{\ell \mathrm{d}}(A|A_0) = \mathrm{tr}(AA_0^{-1}) - \log |AA_0^{-1}| - d,$$

where $|X|$ denotes the determinant of the matrix X. In practice, slack variables for the constraints can be incorporated into the above formulation but are omitted for the sake of clarity.

6.3 Structured Mahalanobis Distances

A major shortcoming of ITML and other existing approaches is their quadratic (or cubic) dependency on the dimensionality. Learning full-rank Mahalanobis matrices for problems where dimensionality is large is prohibitive for several reasons. First, optimizing via algorithms with quadratic dependencies on the dimensionality can be quite expensive. Second, learning a full-rank matrix can be viewed as a statistical inference procedure over d^2 parameters. Text data sets with thousands of dimensions result in full-rank Mahalanobis matrices parametrized by millions of values. For even the most robust methods, learning under such conditions is prone to overfitting and requires a very large amount of supervision. Finally, computing the distance between two points with respect to a Mahalanobis distance function parametrized by a dense, full-rank matrix is an $O(d^2)$ operation. In this section, we present Mahalanobis distance functions that are parametrized by $O(d)$ values.

6.3.1 Low-Rank Mahalanobis Distances

A common representation used in text analysis applications are Tf-Idf models [3]. These models typically compute the distance between two examples $\boldsymbol{x}$ and $\boldsymbol{y}$ using the cosine similarity, $cos(\boldsymbol{x}, \boldsymbol{y}) = \frac{\boldsymbol{x}^T \boldsymbol{y}}{\|\boldsymbol{x}\|\|\boldsymbol{y}\|}$. Note that when $\boldsymbol{x}$ and $\boldsymbol{y}$ are normalized to have unit L_2 norm, the cosine similarity is equivalent to the standard Euclidean distance: $d_I(\boldsymbol{x}, \boldsymbol{y}) = 2 - 2 * cos(\boldsymbol{x}, \boldsymbol{y})$. In many high dimensional domains, feature representations tend to be very sparse, and Tf-Idf models are no exception. This poses several problems for standard Euclidean measures. In Tf-Idf models, two documents can have very similar contextual meaning, yet may not necessarily share many of the same words. Hence, the inner product between two documents can be quite small or even zero, resulting in large Euclidean distances. An example of this is shown in Figure 6.1, where we have three sentences about metric learning. Sentence pairs (A,B) and (B,C) share several common words. Sentences A and C, however, share no common words and the Euclidean distance between them will therefore be quite large. Thus, even though A and C are contextually similar, the model does not reflect this.

Latent factor models work by representing objects in terms of their context or underlying topics [40]. Instead of representing an object $\boldsymbol{x}$ in its original high dimensional space, latent factor models provide a mapping f that transforms $\boldsymbol{x}$ into some lower k-dimensional space. In Figure 6.1, we saw that if examples A and C are compared using the Euclidean distance via their original full-dimension representations, the resulting distances will be large.

A) **Metric learning** is an **important problem** in **data mining**.
B) **High dimensional problems** are common in **data mining**.
C) **Optimizing distance functions** in **high dimensions** is **challenging**.

	A	B	C
A	6	3	0
B	3	6	2
C	0	2	6

Word	A	B	C
metric	1	0	0
learn	1	0	0
important	1	0	0
problem	1	1	0
data	1	1	0
mining	1	1	0
high	0	1	0
dimension	0	1	1
optimiz	0	0	1
distance	0	0	1
function	0	0	1
dimension	0	1	1
challeng	0	0	1

Figure 6.1: The upper left corner lists three sentences. The table on the right shows word counts for each word from the sentences. We can see from the inner product matrix on the lower left that even though all three sentences are about metric learning, the distance between documents A and C is large. This toy example illustrates that Tf-Idf models are quite accurate when inner products are larger, yet can be inaccurate when inner products are small or zero.

This is in spite of the fact that the two examples are in fact quite similar. The goal of a latent factor model is to learn a mapping f such that $f(A)$ and $f(C)$ are similar.

A popular class of latent factor models such as latent semantic analysis (LSA) [33] are those that are parametrized by a $(d \times k)$ projection matrix R. Here, the factor model's mapping function is $f(\boldsymbol{x}) = R^T\boldsymbol{x}$. Consider the Euclidean distance between the latent factors of two points $\boldsymbol{x}$ and $\boldsymbol{y}$:

$$\begin{aligned} d_I\left(f(\boldsymbol{x}), f(\boldsymbol{y})\right) &= d_I(R^T\boldsymbol{x}, R^T\boldsymbol{y}) \\ &= (R^T\boldsymbol{x} - R^T\boldsymbol{y})^T(R^T\boldsymbol{x} - R^T\boldsymbol{y}) \\ &= d_{A_\ell}(\boldsymbol{x}, \boldsymbol{y}), \end{aligned} \tag{6.3.1}$$

where d_{A_ℓ} is a low-rank Mahalanobis distance defined by the low-rank matrix $A_\ell = RR^T$:

$$d_{A_\ell}(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x} - \boldsymbol{y})^T RR^T(\boldsymbol{x} - \boldsymbol{y}). \tag{6.3.2}$$

Whereas a full-rank Mahalanobis matrix is parametrized by $O(d^2)$ values, this low-rank matrix is parametrized by $O(dk)$ parameters comprising the $(d \times k)$ matrix R.

Computationally, we can see from equation (6.3.1) that low-rank Mahalanobis distances can also be computed efficiently in $O(dk)$ time, as the distance between two d-dimensional instances $\boldsymbol{x}$ and $\boldsymbol{y}$ can be computed by first projecting these vectors onto R, and then computing the standard squared Euclidean distance between the projected points.

6.3.2 Diagonal Plus Low-Rank Mahalanobis Distances

In Figure 6.1, we saw an example of two sentences that were of similar context but had large Euclidean distances due the zero overlap (i.e. zero inner product) of their respective feature sets. Because of this, it would be incorrect to conclude that in the context of Tf-Idf models, if two objects have zero or small inner products, then they are contextually different. However, the converse of this statement is much more likely to hold true. That is, if two documents share many commons words, then they are likely to be contextually similar. This is in comparison to low-rank models where this overlap is largely ignored when data is projected into a low dimensional space.

We will now consider this observation in the context of two standard measures used in information retrieval, precision and recall:

$$recall = \frac{\text{Number of Relevant Documents Returned}}{\text{Total Number of Relevant Documents}}. \tag{6.3.3}$$

Precision is measured as the number of relevant documents returned, divided by the total number of documents returned:

$$precision = \frac{\text{Number of Relevant Documents Returned}}{\text{Total Number of Documents Returned}}. \tag{6.3.4}$$

In fact, Tf-Idf models tend to result in higher precision, whereas low-rank factor models provide better recall.

Figure 6.3.2 illustrates this behavior for the Classic3 text data set. Here, recall and precision are measured with respect to the nearest neighbors from a single text document using a leave-on-out-cross-validation approach.

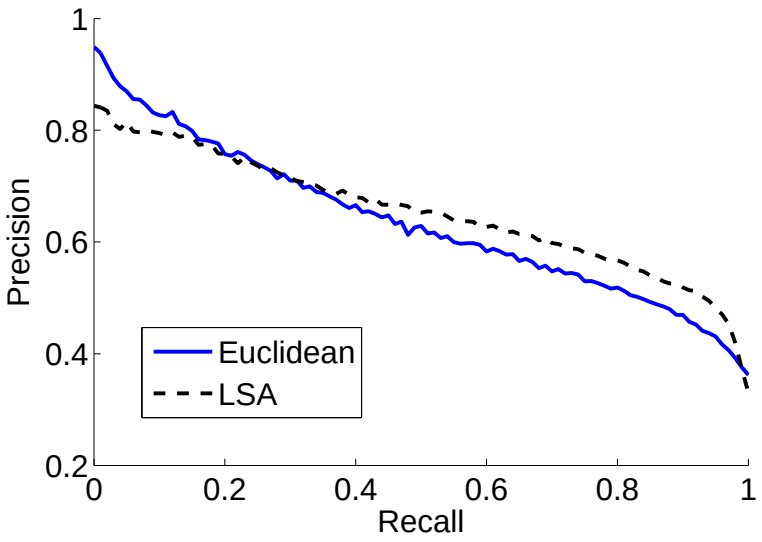


Figure 6.2: Precision-recall curve for the Classic3 text data set. Tf-Idf models using Euclidean distance yield relatively high precision, while the low-rank LSA method has higher recall.

More details regarding this data set will be presented in section 9.3. Precision is plotted for various recall values, comparing nearest neighbor searches using a Tf-Idf Euclidean distance model with that of LSA. We see that for relatively low recall values (i.e. when a relatively small number of documents are returned), the Tf-Idf model significantly outperforms a ten-dimensional LSA factor model. However, as recall increases, the precision for the Euclidean distance model starts to sharply decrease, after which LSA eventually achieves noticeably higher precision.

In domains where both high recall and high precision are needed, we propose a second Mahalanobis distance that incorporates benefits of both the Euclidean distance as well as a low-rank component. We propose a Maha-

lanobis distance parametrized by a matrix $I + A_\ell$, where A_ℓ is low-rank:

$$\begin{aligned} d_{I+A_\ell}(\boldsymbol{x}, \boldsymbol{y}) &= (\boldsymbol{x} - \boldsymbol{y})^T (I + A_\ell)(\boldsymbol{x} - \boldsymbol{y}) \\ &= (\boldsymbol{x} - \boldsymbol{y})^T (\boldsymbol{x} - \boldsymbol{y}) + (\boldsymbol{x} - \boldsymbol{y})^T A_\ell (\boldsymbol{x} - \boldsymbol{y}) \\ &= d_I(\boldsymbol{x}, \boldsymbol{y}) + d_{A_\ell}(\boldsymbol{x}, \boldsymbol{y}). \end{aligned} \tag{6.3.5}$$

Since this proposed measure compares vectors in both the original feature space as well in a projected low-rank factor feature space, one would expect it to achieve both high recall as well as high precision. Revisiting Figure 6.1, we can see that the Euclidean component is a good predictor for two of the three distances, resulting in relatively small distances when comparing (A,B) and (B,C). However, the Euclidean distance between A and C is large, and the second low-rank component is needed here to effectively compare sentences A and C.

6.3.3 Discussion

In this section, we compare our two low-parameter Mahalanobis distances and provide an overview of the strengths and weaknesses of each. Figure 6.3(a) shows a toy data set with three groups of points in three dimensions, X1, X2, and X3. Let the red squares along the X1 axis belong to class S, let the blue circles along the X3 axis belong to class C. Both classes S and C are equidistant to the group of black x's near the origin. The black ellipse represents a 2-dimensional basis spanning the X1-X2 axis.

We will now consider the two metrics for the case when the black x's

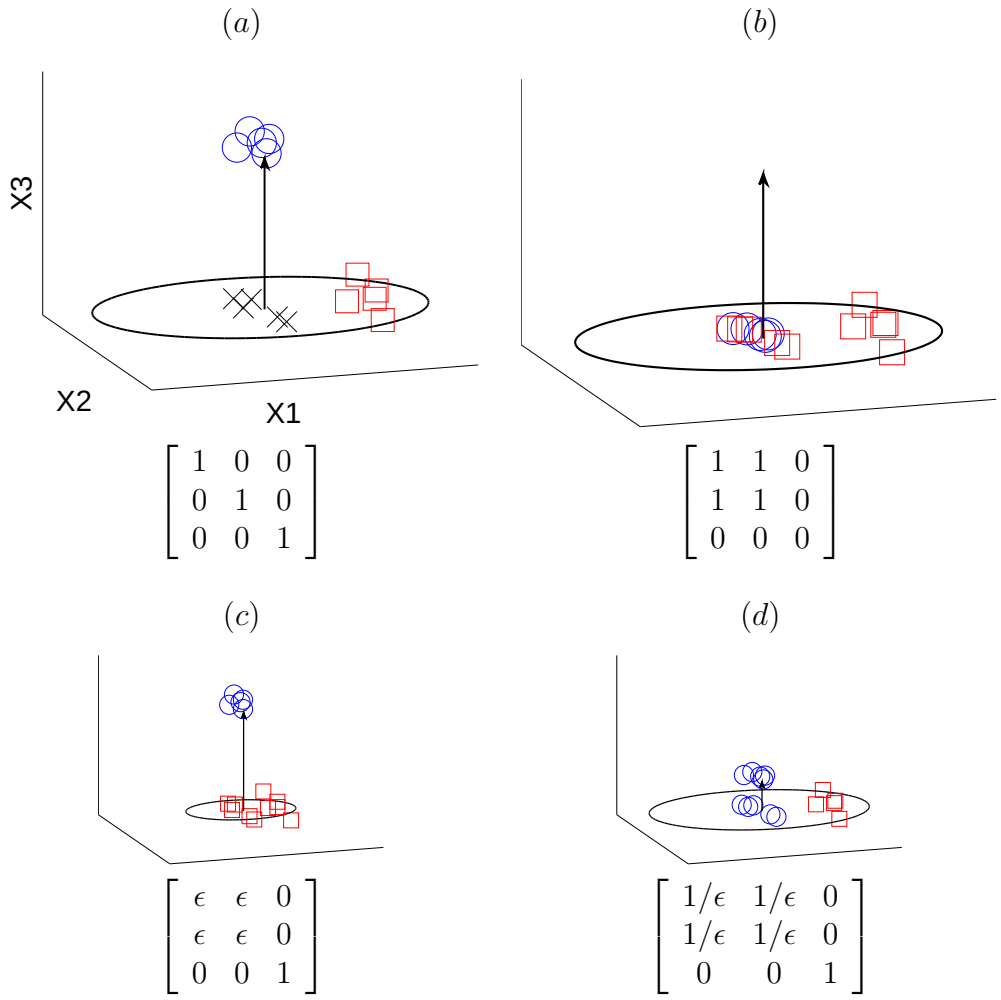


Figure 6.3: A two-class problem with three groups of points. In (b), we see here that a low-rank Mahalanobis distance may not be able to separate the two classes. Examples (c) and (d) show how classes can be separated if our diagonal plus low-rank metric is instead used.

take on the square class and also for the case when they take on the circle class. Figure 6.3(b) shows a shortcoming of the low-rank Mahalanobis distance (6.3.2 for the case when the black x's take on the square class. Here, the blue squares are projected directly on top of red circles, and low-rank metric will not be able to separate the two classes.

Figure 6.3(c) shows how our second metric (10.4.3) can effectively learn a metric here. By shrinking the low-rank basis, the blue circles are moved away from the red-circles lying in the X1-X2 plane. The arrow shown here represents the identity component X3 of the Mahalanobis metric that is orthogonal to the low-rank basis X1-X2. As seen in (c), preserving the distances along the identity component is critical to separating the two classes.

In (d) the class labels for the black x's take on the circle class instead. Here, we see that the diagonal plus low-rank Mahalanobis distance can learn a discriminative metric by increasing the scale of the low-rank basis. This effectively moves the square class outward along the X1-X2 basis, thus moving the group of points along the X3 axis relatively closer towards the points at the origin.

Due to limits of visualization, the points in figure 6.3 have been restricted to only three dimensions. Learning full-rank Mahalanobis distances over three dimensions can be easily achieved using existing methods. However, if we think of the X3 dimension as not only one dimension, but instead as a representation for a much larger set of sparse dimensions, then learning a full-rank matrix is no longer possible. Here, the identity plus low-rank method

can represent the set of dimensions represented by the X3 axis using a linear number of parameters.

So far, we have proposed and motivated two forms of low-parameter Mahalanobis distances. In the next section, we formalize these two problems and provide efficient algorithms to optimize them.

6.4 Learning Structured Metrics

6.4.1 Low-Rank Mahalanobis Distances

We now extend the full-rank ITML algorithm to learn low-rank matrices. Let R be the $(d \times k)$ factor matrix for the rank-k regularization matrix A_0, i.e. $A_0 = RR^T$. We formulate our high dimensional low-rank (HDLR) metric learning problem as:

$$\begin{aligned} \min_{A} \quad & D_{\ell \mathrm{d}}(A|RR^T) \\ \text{subject to} \quad & d_A(\boldsymbol{x}_i, \boldsymbol{x}_j) \leq u \qquad (i,j) \in S, \\ & d_A(\boldsymbol{x}_i, \boldsymbol{x}_j) \geq \ell \qquad (i,j) \in D \\ & rank(A) \leq k \end{aligned} \tag{6.4.1}$$

Comparing this to the full-rank ITML formulation (6.2.1), we see that A_0 here is low-rank, and an additional constraint has been added enforcing the rank of the optimal Mahalanobis matrix A. For the ITML problem, it was shown that if A_0 is positive definite, then the optimal solution A^* will also be positive definite. Here, we present an extension of this result for when A_0 is a low-rank, positive semi-definite matrix. Recent work by Kulis et. al. [61] considers

a related problem of learning kernel matrices subject to linear constraints on the matrix. It was shown that the LogDet divergence can be extended to the positive *semi*-definite cone. Two matrices have a finite LogDet divergence if and only if they share the same range space. Generalizing this result to our metric learning setting, we state the following result without proof:

Lemma 2. Let R be an arbitrary basis, and let A_0 be a positive semi-definite matrix with range space spanned by R. The objective of problem (6.4.1) is finite if and only if A is positive semi-definite with range space equal to R.

So, if the baseline Mahalanobis distance function is parametrized by a rank-k matrix, the optimal solution to the HDLR metric learning problem (6.4.1) will also have rank of k. Therefore, the rank constraint $rank(A) \leq k$ need not be explicitly enforced. In section 6.5, we present methods that can be used to choose an appropriate baseline matrix.

6.4.2 HDLR Algorithm

We now present an algorithm for solving our HDLR formulation (6.4.1), Algorithm 2. The algorithm optimizes a slightly modified version of problem (6.4.1) that incorporates slack variables to allow constraint violation in the case of incorrect or noisy constraints. The slack penalty parameter γ determines the relative weighting given to the LogDet component of the objective as opposed to the slack penalty component of the objective. When γ is large, more weighting is given to the slack terms, and the final solution will more closely satisfies the constraints. When γ is small, more emphasis is given

to the LogDet objective, yielding smoother solutions which are closer to the regularization matrix A_0.

The algorithm uses the method of cyclic projections [15] and works by iteratively projecting the current solution onto a single constraint. Instead of directly optimizing A, the algorithm instead optimizes its $(d \times k)$ factor matrix B. The main loop starting in line 2 iterates over each constraint until convergence. In practice, convergence can be checked by monitoring the change in the dual variables, λ. Steps 5-10 compute the projection parameter r. In step 11, this parameter is then used to update B via a rank-one update. Each projection can be computed in closed form and requires $O(dk)$ computation, where k is the rank of A_0. Finally, the optimal solution is $A = BB^T$. As shown in (6.3.1), the low-rank Mahalanobis distance between two points can be computed in $O(dk)$ time without the need to explicitly compute A.

6.4.3 Diagonal Plus Low-Rank Distances

Here, we formulate the high-dimensional diagonal plus low-rank (HD^{2}LR) metric learning problem. Let R be some low-rank basis, and let U be an orthogonal representation of this basis (i.e. $U = R(R^TR)^{-\frac{1}{2}}$). We will constrain the low-rank component A_ℓ of our diagonal plus low-rank distance matrix A given in (10.4.3) to take on the form $UU^T(A - A_0)UU^T$. This term can be viewed as the difference of two matrices, UU^TAUU^T and $UU^TA_0UU^T$. The first term is a function of the learned matrix A, while the second term can be viewed as a low-rank offset provided by the regularization matrix A_0. If

Algorithm 2 High Dimension Low-Rank (HDLR) Metric Learning

Require: $A_0 = RR^T$: Baseline Mahalanobis matrix, γ: Slack penalty, $X = [\boldsymbol{x}_1, ..., \boldsymbol{x}_n]$: Set of constrained points, (S, D): Similarity/Dissimilarity constraints.

$B = R,\ \lambda_{ij} = 0\ \forall\ i, j,\ b_{ij} = 0\ \forall\ i, j$
while not converged **do**
  $(i, j) \leftarrow$ similarity or dissimilarity constraint
  $\delta \leftarrow 1$ if Similarity constraint, -1 if Dissimilarity constraint
  $d \leftarrow (\boldsymbol{x}_i - \boldsymbol{x}_j)^T B^T B (\boldsymbol{x}_i - \boldsymbol{x}_j)$
  $\eta \leftarrow \min\left(\lambda_{ij}, \frac{\delta\gamma}{\gamma+1}\left(\frac{1}{d} - \frac{1}{b_{ij}}\right)\right)$
  $\alpha \leftarrow \delta\eta/(1 + \delta\eta d)$
  $\lambda_{ij} \leftarrow \lambda_{ij} - \eta$
  $b_{ij} = \gamma b_{ij}/(\gamma + \delta\eta b_{ij})$
  $r = \sqrt{1 + \alpha} - 1$
  $B \leftarrow B + r(\boldsymbol{x}_i - \boldsymbol{x}_j)(\boldsymbol{x}_i - \boldsymbol{x}_j)^T B$
end while
$A \leftarrow BB^T$

$A = A_0$, the low-rank term will always be zero, and the distance function will reduce to that of the standard Euclidean distance. As A diverges from A_0, the low-rank term starts to dominate, giving more emphasis to the underlying factor model.

We will now present the HD^{2}LR metric learning problem that learns a full-rank Mahalanobis matrix with the constraint $A = I + UU^T(A - I)UU^T$. The analysis and algorithms presented below can be generalized to arbitrary

regularizers A_0 (i.e. $A = I + UU^T(A - A_0)UU^T$).

$$\begin{aligned} \min_{A} \quad & D_{\ell \mathrm{d}}(A|I) \\ \text{subject to} \quad & d_A(\boldsymbol{x}, \boldsymbol{y}) \leq u \qquad (i,j) \in S, \\ & d_A(\boldsymbol{x}, \boldsymbol{y}) \geq \ell \qquad (i,j) \in D, \\ & A = I + UU^T(A - I)UU^T. \end{aligned} \tag{6.4.2}$$

Consider the distance constraints used in this formulation:

$$\begin{aligned} d_{I+A_\ell}(\boldsymbol{x}, \boldsymbol{y}) &= d_I(\boldsymbol{x}, \boldsymbol{y}) + d_{A_\ell}(\boldsymbol{x}, \boldsymbol{y}) \\ &= d_I(\boldsymbol{x}, \boldsymbol{y}) - d_{UU^T}(\boldsymbol{x}, \boldsymbol{y}) + d_{UU^TAUU^T}(\boldsymbol{x}, \boldsymbol{y}) \end{aligned}$$

The first two terms are independent of the learned matrix A and are therefore constants in the context of the optimization problem. Moving these to the right hand side, we can rewrite the problem:

$$\begin{aligned} \min_{A} \quad & D_{\ell \mathrm{d}}(A|I) \\ \text{subject to} \quad & d_{UU^TAUU^T}(\boldsymbol{x}, \boldsymbol{y}) \leq u - c_{xy} \qquad (i,j) \in S, \\ & d_{UU^TAUU^T}(\boldsymbol{x}, \boldsymbol{y}) \geq \ell - c_{xy} \qquad (i,j) \in D, \\ & A = I + UU^T(A - I)UU^T, \end{aligned} \tag{6.4.3}$$

where $c_{xy} = d_I(\boldsymbol{x}, \boldsymbol{y}) - d_{UU^T}(\boldsymbol{x}, \boldsymbol{y})$. Recall that the original problem constrains the full-rank distance between points $\boldsymbol{x}$ and $\boldsymbol{y}$. Here, a low-rank approximation of the learned Mahalanobis distance is constrained. Recall that for the low-rank HDLR formulation presented in the previous section, if A_0 is low-rank, then the optimal solution is also low-rank. The next theorem characterizes the solution for problem (6.4.3), showing that the optimal solution satisfies

$A^* = I + UU^T(A^* - I)UU^T$, thereby obviating the need to explicitly enforce this identity plus low-rank constraints.

Theorem 2. Let A^* be the optimal solution to an instance of the information-theoretic metric learning problem (6.4.3) with similarity constraints S, dissimilarity constraints D, orthogonal projection matrix U, and regularization matrix $A_0 = I$. Then A^* satisfies $I + UU^T(A^* - I)UU^T$.

Proof. For appropriately defined constants c_{ij}, the Lagrangian of problem (6.4.3) can be written as

$$L(A, \lambda) = \operatorname{tr}(A) - \log|A| + \sum_{i,j} \delta_{ij}\lambda_{ij}\left(\operatorname{tr}(UU^T AUU^T(\boldsymbol{x}_i - \boldsymbol{x}_j)(\boldsymbol{x}_i - \boldsymbol{x}_j)^T) - c_{ij}\right),$$

where λ_{ij} are dual variables with $\lambda_{ij} \geq 0$. Here, δ_{ij} is $+1$ for similarity constraints and -1 for dissimilarity constraints. Using the fact that $\bigtriangledown_A \log|A| = A^{-1}$ [13], we differentiate the Lagrangian,

$$\bigtriangledown_A L(A, \lambda) = I - A^{-1} + \sum_{i,j} \delta_{ij}\lambda_{ij} UU^T(\boldsymbol{x}_i - \boldsymbol{x}_j)(\boldsymbol{x}_i - \boldsymbol{x}_j)^T UU^T.$$

Setting to zero and solving for A^{-1},

$$\begin{aligned}
(A^*)^{-1} &= I + \sum_{i,j} \delta_{ij}\lambda_{ij} UU^T(\boldsymbol{x}_i - \boldsymbol{x}_j)(\boldsymbol{x}_i - \boldsymbol{x}_j)^T UU^T \\
&= I + UU^T\left(\sum_{i,j} \delta_{ij}\lambda_{ij}(\boldsymbol{x}_i - \boldsymbol{x}_j)(\boldsymbol{x}_i - \boldsymbol{x}_j)^T\right)UU^T \\
&= I + UU^T PUU^T.
\end{aligned}$$

Thus, the inverse of the optimal solution has the form identity plus low-rank. To see that the solution A^* also has this form, we can use the Sherman-Morrison-Woodbury formula [48], which states that for any invertible A and $(d \times k)$ complex matrix R:

$$(A + RR^H)^{-1} = A^{-1} - A^{-1}R(I + R^H A^{-1} R)^{-1} R^H A^{-1}.$$

Note that R may be complex, and we denote R^H as its conjugate transpose. Applying the above equation for $A = I$, and for $R = UU^T C$, where C is defined such that $CC^H = P$, we have:

$$\begin{aligned} A^* &= (I + UU^T P UU^T)^{-1} \\ &= I - UU^T C(I + C^H UU^T UU^T C)^{-1} C^H UU^T \\ &= I + UU^T(\hat{A} - I)UU^T, \end{aligned}$$

for $\hat{A} = -C^H UU^T UU^T C$. Finally, using the fact that $U^T U = I^k$, we have

$$\begin{aligned} I &+ UU^T(A^* - I)UU^T \\ &= I + UU^T \left(\left(I + UU^T(\hat{A} - I)UU^T \right) - I \right) UU^T \\ &= I + UU^T(\hat{A} - I)UU^T \\ &= A^*. \end{aligned}$$

□

6.4.4 HD^{2}LR Algorithm

The metric learning problem HDLR formulated in the previous section learns a full-rank, $(d \times d)$ matrix. Using algorithms presented in [31], both run-

ning time and storage requirements for this algorithm would still be quadratic in the dimensionality. In this section, we show that problem (10.4.4) can be transformed to an equivalent problem in k dimensions. Via this transformation, the problem can then be solved in time quadratic in k. The solution can then by mapped back to the optimal solution of the original problem via a simple matrix operation.

Consider the following k-dimensional metric learning problem:

$$\begin{aligned} \min_{M} \quad & D_{\ell \mathrm{d}}(M|I^k) \\ \text{subject to} \quad & d_M(U^T\boldsymbol{x}, U^T\boldsymbol{y}) \leq u - c_{xy} \qquad (i,j) \in S, \\ & d_M(U^T\boldsymbol{x}, U^T\boldsymbol{y}) \geq \ell - c_{xy} \qquad (i,j) \in D, \end{aligned} \tag{6.4.4}$$

where the superscript notation I^k is used to emphasize the dimensionality of the matrix. We now prove a lemma that shows how to construct the optimal solution to problem (6.4.3) given M^*, the optimal solution to the above problem (6.4.4).

Lemma 3. Let M^* be an optimal solution to problem (6.4.4). Then the optimal solution to problem (6.4.3) can be constructed as $A^* = I + U(M^* - I)U^T$.

Proof. From theorem 2, we know that A^* satisfies $A^* = I + UU^T(A^* - I)UU^T$. Combined with the lemma statement, this implies $U^T A^* U = M^*$. We now show that problems (6.4.4) and (6.4.3) are equivalent. Specifically, we show that for any two matrices M and A satisfying $M = U^T AU$, (1) problem (6.4.4)

is feasible if and only if problem (6.4.3) is feasible, and (2) the problem objectives differ by at most a constant. To see equivalence with respect to feasibility,

$$\begin{aligned} d_M(U^T\boldsymbol{x}, U^T\boldsymbol{y}) &= (\boldsymbol{x}-\boldsymbol{y})^T UMU^T(\boldsymbol{x}-\boldsymbol{y}) \\ &= (\boldsymbol{x}-\boldsymbol{y})^T UU^T AUU^T(\boldsymbol{x}-\boldsymbol{y}) \\ &= d_{UU^T AUU^T}(\boldsymbol{x},\boldsymbol{y}) \end{aligned}$$

Next, we show that the objective functions of the two problems differ by at most a constant. We will first consider the trace term:

$$\begin{aligned} \mathrm{tr}(A) &= \mathrm{tr}(I^d + UU^T(A-I^d)UU^T) \\ &= \mathrm{tr}(U^T AUU^T U) + \mathrm{tr}(I^d - UU^T) \\ &= \mathrm{tr}(M) + \mathrm{tr}(I^d - UU^T). \end{aligned}$$

Since U is defined to be orthogonal, $U^T U = I^k$. Also, the second term is constant. Next consider the log det term. Let W be the orthogonal complement to U, i.e. a $(d \times d-k)$ matrix such that $I^d - UU^T = WW^T$ and $U^T W = 0$.

$$\begin{aligned} A &= I^d + UU^T(A-I^d)UU^T \\ &= UU^T AUU^T + I^d - UU^T \\ &= UU^T AUU^T + WW^T \\ &= [UW]\begin{bmatrix} U^T AU & 0 \\ 0 & I^{d-k} \end{bmatrix}\begin{bmatrix} U^T \\ W^T \end{bmatrix}. \end{aligned}$$

The first line is a result of the fact that in theorem 2, we showed that any solution to problem (6.4.3) satisfies $A = I + UU^T(A-I)UU^T$. The determinant of a matrix is invariant under orthogonal transformation $[UW]$, so

$$\log|A| = \log|U^T AU| + \log|I^{d-k}| = \log|M| + d - k.$$

Finally, we have

$$\begin{aligned} D(A|I^d) &= \operatorname{tr}(A) - \log|A| + d \\ &= \operatorname{tr}(M) + \operatorname{tr}(I^d - UU^T) - \log|M| + k \\ &= D(M|I^k) + C, \end{aligned}$$

for constant C that depends only on U and A_0. □

Algorithm 3 shows how this lemma can be used to efficiently solve the HD^{2}LR metric learning problem (10.4.4). Step 1 projects the original d-dimensional data onto a k-dimensional subspace using the low-rank basis U. Next, step 2 solves the (full-rank) ITML problem in this much lower, k-dimensional space, returning a $(k \times k)$ matrix M^*. The optimal solution can be constructed using Lemma 3 as $A^* = I + UM^*U^T$. Note that this matrix never needs to be explicitly constructed, since a Mahalanobis distance parametrized by A^* can be expressed as the sum of two Mahalanobis distances as shown in equation (10.4.3). Finally, it is possible that the right hand side of problem (6.4.3) is negative. This presents problems for the slack variables used in Algorithm 3. However, in practice, the goal is to learn a metric in which constraints are satisfied *relatively*, and this issue can be solved by adding a scalar $c > 0$ to the right hand side of (6.4.3) in order to ensure positivity.

6.5 Choosing an Appropriate Basis

The metric learning algorithms presented here are parametrized by a low-rank matrix. Algorithms 2 and 3 work by optimizing with respect to

Algorithm 3 Identity Plus Low-Rank Metric (HD^{2}LR) Algorithm

Require: U: Low-rank basis, γ: Slack penalty, $X = [\boldsymbol{x}_1, ..., \boldsymbol{x}_n]$: Set of constrained points, (S, D): Similarity/Dissimilarity constraints.

1: Form the projected data set $\hat{X} = [U^T x_1, ..., U^T x_n]$

2: Compute optimal solution M^* to full-rank ITML problem (6.4.4) with constraints D and S over the projected data set $\hat{X}$ using Algorithm 2

3: Return optimal solution $A^* = I + UM^*U^T$

a given basis. In order to maximize the quality of the learned metric, an appropriate basis should be chosen.

A standard basis that is used in unsupervised settings such as latent semantic analysis is the left singular vectors of the singular value decomposition (SVD) [48]. LSA works by taking the SVD of the so-called 'centered' data matrix. Let X be a $(d \times n)$ centered data matrix, where each column represents a d-dimensional instance and each row is normalized to have zero mean, and let the SVD of this matrix be UDV^T, where U and V are orthogonal matrices, and D is diagonal. Let U^k denote the first k columns of U. This matrix U^k is typically referred to as the top k principal components, and the matrix UU^T represents a projection from the original space onto a rank-k subspace. LSA uses this projection in computing distances (or cosine similarities) between points, $d_{LSA}(\boldsymbol{x}, \boldsymbol{y}) = D_{UU^T}(\boldsymbol{x}, \boldsymbol{y})$. Thus, the regularization matrix that results from this LSA basis is $A_0 = UU^T$.

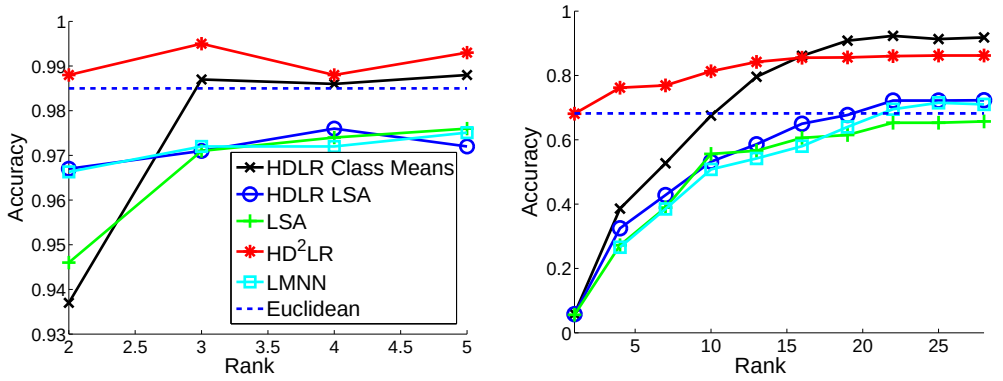
While the use of the SVD in methods such as LSA has been shown to achieve good results in many information retrieval settings, it is fundamentally an unsupervised method. When used with our metric learning algorithms

along with similarity and dissimilarity constraint information, the result is a semi-supervised form of LSA.

In cases where data is fully supervised, we propose a method which chooses a low-rank basis according to the class labels. Whereas LSA chooses vectors based on the SVD, the class-mean method forms vectors directly using the class labels. Let c be the number of distinct classes and let k be the size of the desired basis. If $k = c$, then each class mean is computed forming the basis $R = [\boldsymbol{r}_1 \dots \boldsymbol{r}_c]$. If $k < c$ a similar process is used but restricted to a randomly selected subset of k classes. If $k > c$, instances within each class are clustered into approximately $\frac{k}{c}$ clusters. Each cluster's mean vector is then computed to form the set of low-rank basis vectors R.

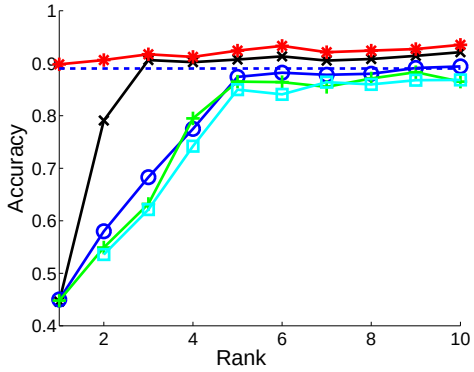
LSA's use of the SVD results in an orthogonal low-rank basis. The supervised class means method presented here will not generally result in an orthogonal basis. As a final step in forming a class means basis, we orthogonalize R, resulting in a regularization matrix $A_0 = R(R^TR)^{-1}R^T$. Orthogonalization reduces distortion of the low-rank distance. For example, distances between class means are preserved. Let $\boldsymbol{r}_i$ and $\boldsymbol{r}_j$ be two class mean vectors (i.e. columns i and j of R). Then

$$\begin{aligned} d_{A_0}(\boldsymbol{r}_i, \boldsymbol{r}_j) &= (\boldsymbol{r}_i - \boldsymbol{r}_j)^T R(R^TR)^{-1}R^T(\boldsymbol{r}_i - \boldsymbol{r}_j) \\ &= (\boldsymbol{e}_i - \boldsymbol{e}_j)R^TR(R^TR)^{-1}R^TR(\boldsymbol{e}_i - \boldsymbol{e}_j) \\ &= (\boldsymbol{e}_i - \boldsymbol{e}_j)R^TR(\boldsymbol{e}_i - \boldsymbol{e}_j) \\ &= (\boldsymbol{r}_i - \boldsymbol{r}_j)^T(\boldsymbol{r}_i - \boldsymbol{r}_j) = d_I(\boldsymbol{r}_i, \boldsymbol{r}_j), \end{aligned}$$



(a) Classic3

(b) 20-Newsgroups



(c) Newsgroups Politics Subset

Figure 6.4: Classification accuracy for Mahalanobis metrics of various ranks. Overall, our methods outperform existing methods.

where $\boldsymbol{e}_i$ is a vector of all zeros with a one in the i^{th} position.

6.6 Experimental Results

We now present some sample results for our methods from a variety of high-dimensional domains: text analysis, statistical software analysis, and collaborative filtering. These datasets can all be characterized by relatively high dimensionality (from 5,000 to more than 100,000 features) and represent

a broad sample of modern, high dimensional problems.

We evaluate performance of our learned distance metrics in the context of both classification accuracy for the k-nearest neighbor algorithm, as well as in the context of precision/recall performance for general nearest neighbor searches. The k-nearest neighbor classifier uses $k = 10$ nearest neighbors, breaking ties arbitrarily. Accuracy is defined as the number of correctly classified examples divided by the total number of classified examples. Recall and precision are computed as defined in equations (6.3.3) and (6.3.4).

Our experiments compare our two formulations, HDLR and HD^{2}LR, as learned by Algorithms 2 and 3. We also compare these algorithms to a heuristic based on the the Large-Margin Nearest Neighbor (LMNN) metric learning algorithm. In [103], LMNN is presented as a method for learning full-rank Mahalanobis distance matrices. Here, we use a heuristic in which data is first projected onto some low-rank, r-dimensional basis U. LMNN is then run over this r-dimensional problem, yielding a $(r \times r)$ matrix A. Finally, this matrix is transformed back to the original, high-dimensional space as UAU^T. We emphasize, that this procedure does not optimize a well-formed global objective, whereas our approaches optimize a log-determinant objective function. The LMNN implementation used is a Matlab implementation provided by Weinberger.

For our proposed algorithms, pairwise constraints are inferred from true labels. For each class 100 pairs of points are randomly chosen from within the class and are constrained to be similar, and 100 pairs of points are drawn

from different classes to form dissimilarity constraints. Given c classes, this results in $100c$ similarity constraints, and $100c$ dissimilarity constraints, for a total of $200c$ constraints. This number was determined empirically to provide a reasonable tradeoff between computational efficiency (more constraints result in longer training times) and final accuracy of the learned metric. The upper and lower bounds for the similarity and dissimilarity constraints are determined empirically as the 1^{st} and 99^{th} percentiles of the distribution of distances computed using a baseline Mahalanobis distance parametrized by A_0. Finally, the slack penalty parameter γ used by Algorithms 2 and 3 is cross-validated using values $\{.01, .1, 1, 10, 100, 1000\}$.

All metrics are trained using data only in the training set. Test instances are drawn from the test set and are compared to examples in the training set using the learned distance function. The test and training sets are established using a standard two-fold cross validation approach. For experiments in which a baseline distance metric is evaluated (for example, the squared Euclidean distance), nearest neighbor searches are again computed from test instances to only those instances in the training set.

6.6.1 Speed Comparison

We first compare the computational speed for our low-parameter algorithms as compared with existing full-rank methods, LMNN and ITML. Figure 6.5 shows time taken to learn metrics of dimensionality 50 to 2000 over a synthetic data set with 900 instances and 3 classes. All implementations are

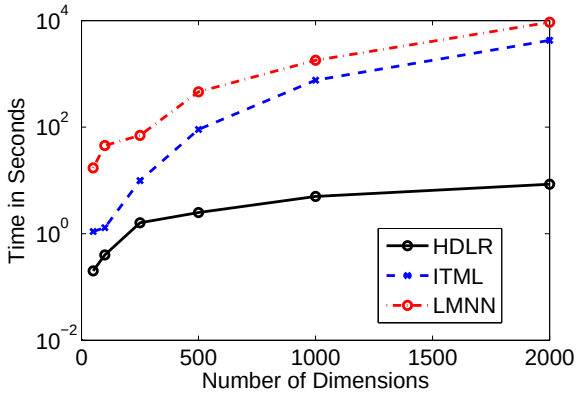


Figure 6.5: Running times for our high-dimensional algorithms compared to existing full-rank methods. Full-rank Mahalanobis distance learning algorithms do not scale well to high dimensionality, whereas our method HDLR does.

in Matlab and are run on an Intel Pentium processor with 4 GB of RAM. Noting that the time axis is displayed on a log-scale, we can see that our HDLR algorithm scales roughly linearly with dimensionality, whereas existing full-rank methods scale quadratically as dimensionality increases. Further, LMNN ran out of memory when learning a 3000-dimensional metric. Running-time of the HD^{2}LR method is comparable to those shown for the low-rank HDLR algorithm shown in the figure.

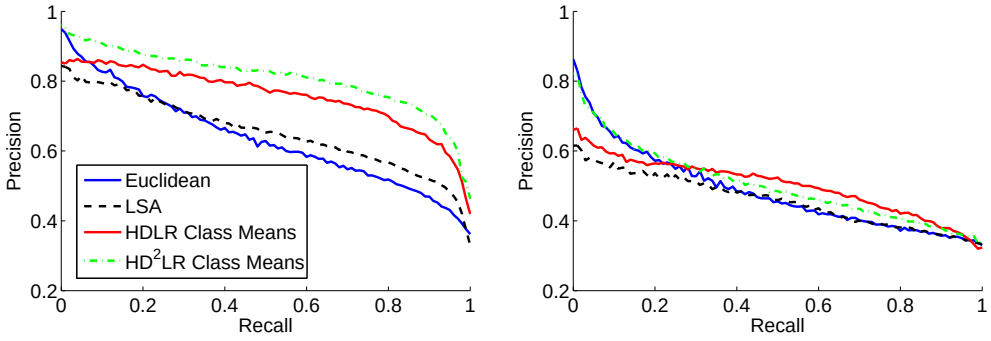
6.6.2 Text Analysis

Our text datasets are created by standard bag-of-words Tf-Idf representations. Words are stemmed using a standard Porter stemmer and common stop words are removed. The text models are limited to the 5,000 words with the largest document frequency counts. We provide experiments over two data

sets: CMU's 20-newsgroup data set [91], and the Classic3 data set [90]. Classic3 is a relatively small 3 class problem with 3,891 instances. The newsgroup data set is much larger, having 20 different classes from various newsgroup categories and 20,000 instances.

Figure 6.4 shows classification accuracy across various ranks for the Classic3 dataset, along with the full newsgroup data set and a subset of the data restricted to the three politics related classes. Here, the diagonal plus low-rank method HD^{2}LR uses the class means basis as described in section 6.5. Comparing this to the baseline Euclidean measure, we can see that for low dimensionality, the accuracy of the two algorithms is similar, with HD^{2}LR having a slightly higher value. For larger ranks, the accuracy of the HD^{2}LR method slowly increases, while the accuracy of the low-rank HDLR class means method increases much more quickly. In fact, for the largest 20-class Newsgroup data set (b), we can see that for larger ranks, the HDLR method outperforms the HD^{2}LR method. Here, the HDLR method achieves accuracy 27% higher than the baseline Euclidean distance.

The HD^{2}LR and HDLR class means methods require full supervision in order to form the low-rank basis. In semi-supervised settings, forming this basis from the similarity and dissimilarity constraints used in our low-rank metric learning algorithms is not possible and a low-rank LSA basis may be used instead. Recall that the LSA basis described in section 6.5 requires no supervision. In Figure 6.4, we see that our HDLR method outperforms the baseline unsupervised LSA method across all data sets for most dimensions.



(a) Classic3 (a) Newsgroups Politics Subset

Figure 6.6: Precision-recall curves comparing our low-rank Mahalanobis distance functions to standard LSA and Tf-Idf measures for the Classic3 text data set, and the politics newsgroups subset.

This is compared to LMNN using the same LSA basis U, which generally performs only comparably to LSA.

Figure 6.6 shows recall-precision curves for four methods: standard Euclidean distance Tf-Idf measure, LSA, and our two methods using a class means basis. The rank used for LSA and our two methods is ten. We can see that for low recall values, both the Euclidean distance and the HD^{2}LR method achieve significantly higher precision values than the other two low-rank methods. As desired recall levels increase, the precision of the Euclidean distance decreases rather quickly, while the HD^{2}LR continues to achieve higher precision values that are comparable to or better than the low-rank methods. For the Classic3 data set, HD^{2}LR outperforms all other methods for all recall values, marking an improvement over LSA of up to 20%. For the newsgroups Politics subset, the HD^{2}LR method outperforms HDLR for low recall values, yet achieves slightly worse precision for higher recall values.

We additionally compare the accuracy of these methods against a multi-class support vector machine (SVM) using a linear kernel [98]. Here, the accuracy of the of the Classic3 data set is 99.7%, the politics subset is 96.3%, and the full newsgroup data set is 94.8%. This is somewhat higher than the accuracy achieved by our HD^{2}LR method (99.3%, 93.5%, and 91.8%, respectively).

6.6.3 Software Analysis

We now present results from the Clarify system [49] as described in the previous chapter. Complete details of the Clarify system are provided in Chapter 8.

Figure 6.7 provides accuracy results for a k-NN classifier used with our learned rank-10 distance metrics HDLR and HD^{2}LR . We compare this against four baseline methods. The class means method is a supervised method in which the class mean basis is used to parametrize a low-rank Mahalanobis distance without performing any additional learning. Confidence intervals intervals shown are computed for the 5^{th} and 95^{th} percentiles. Overall, we see that our HDLR and HD^{2}LR methods outperform the other four methods. In particular, we notice that for two of the four data sets (Mpg321 and Foxpro), the SVM performed significantly worse than these two methods.

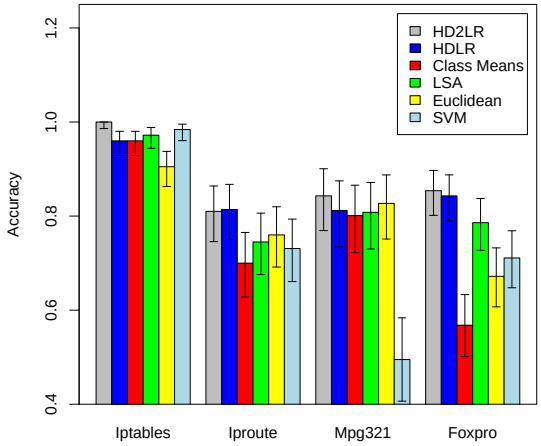


Figure 6.7: Classification accuracy for four statistical software analysis datasets across six different algorithms: HDLR, HD^{2}LR, a baseline class means method, latent semantic analysis (LSA), Euclidean distance, and a multi-class SVM

6.6.4 Collaborative Filtering Data

Finally, we present experiments over a set of Yahoo song reviews. Here, 14,596 songs are reviewed by a total of 120,397 reviewers. Each review is on the basis of a '1' (the reviewer does not like the song) to a '5' (the reviewer liked the song). Further, each song is categorized into one of five genres: Rock, R&B, Pop, Rap, and Country.

Many of today's recommender systems work by performing nearest neighbor searches over such collaborative data in order to help users find similar songs, movies, or products to the ones that he or she already enjoys. Here, we consider the problem of learning a distance function over the Yahoo song data that respects genres. Such a distance function is important as often times people prefer songs from a limited number of genres, yet genre information is not known for all songs. In fact, the 14,596 labelled songs used in this data set represent a very small subset ($< 1\%$) of the entire set of all songs in the Yahoo music data set, most of which have the genre type 'unknown'.

Figure 6.8 shows classification accuracy for this data set for four different methods across a varying number of dimensions. Here, we see that the low-rank method HDLR (64.5% accuracy) performed significantly better than HD^{2}LR (53.9% accuracy). This data set is extremely sparse, with an average of 33 reviews per song (99.97% sparse). This is compared to the newsgroup data set, that had an average of 212 words per document or 95.76% sparse. The primary motivation behind the diagonal component of the HD^{2}LR method was the fact that the diagonal element accounts for feature overlaps. Due to

the high degree of sparsity in the Yahoo data (as well as other collaborative filtering domains), the overlap here is almost always very small or zero. Here, the diagonal component actually degrades performance. Finally, the SVM performance for this data set (not shown in Figure 6.7) is quite high, at 70.06%.

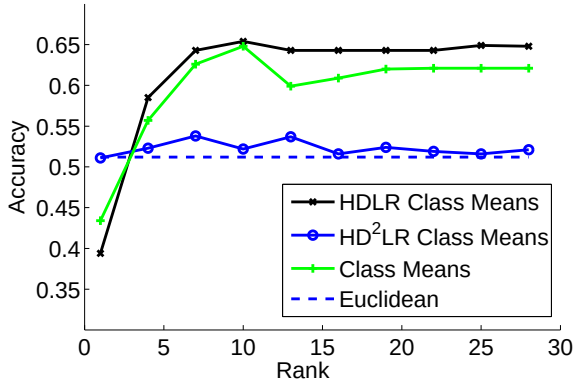


Figure 6.8: Accuracy for distance functions of various ranks for the Yahoo music data set. Here, the HDLR method significantly outperforms the HD^{2}LR method. This suggests that HDLR method may yield better results in problems that are extremely sparse.

Chapter 7

Statistical Software Analysis

In this chapter, we introduce the primary application of our methods, statistical software analysis. Modern software systems are becoming increasingly complex, pushing the limits of static analysis methods that directly analyze software code using program language models. Static analysis method are largely limited to the analysis of a single program method or other relatively small sections of code, and is further limited by the following factors:

- Modern programs are becoming increasingly modular and rely on black-box components. By definition, source code of black-box components is not available for use in static analysis methods.
- The use of dynamically generated code is becoming quite popular in today's popular programming languages, for example, the Ruby language [62], or the Java Hibernate framework [55]. Dynamically generated code is created during runtime and is therefore cannot be analyzed statically.

Statistical software analysis takes a data-driven approach in directly analyzing a program's runtime behavior. Statistical software analysis meth-

ods represent software using programming language independent features and can overcome issues regarding black-box components and dynamic code generation.

7.1 Problems

The problem of statistical software analysis can be framed in the following manner. Given a target software system, we represent a set of n executions of the system by a set of data objects $X = \{x_1, ..., x_n\}$. Optionally associated with each object x_i is a label $t_i \in T$, where T represents possible outcomes of the system. Depending on the application, T may represent errors encountered by a system, program crashes, levels of disk utilization, etc. Here, we detail several central problems, such as statistical debugging, program error detection, and automatic test prioritization.

The problem of statistical debugging can be viewed as that of comparing features found in non-buggy executions to those found in buggy executions. Here, the set T denotes either normal program execution types, or (one or more) buggy execution types. The problem of statistical debugging reduces to finding a particular component, line of code, basic block, or any program feature $x_i(j)$ of each execution that maximizes the probability $P(buggy|x_i(j))$ - i.e. to find the component of the system that is most predictive of buggy executions. In some sense, statistical debugging is a much weaker tool than automated software proving in that it can only minimize the probability of bug occurrences. Clearly, if a bug type t is never realized in a given data set, then

statistical software analysis will be of no use. However, statistical debugging has been successfully applied to many large, real-world system [66], whereas the success of automatic software proving has been largely limited to verifying the correctness of only relatively small system components. We also note that the basic problem of minimizing the occurrence of software bugs requires a distribution over which to be minimized, which can naturally be estimated via statistical methods.

A more recently proposed statistical software analysis task is that of predicting program errors [49] which we will present in Chapter 8. The Clarify system seeks to predict program errors in order to provide more informative messaging for a user. Clarify is motivated by the problem that black box components complicate the problem of error reporting. Components such as dynamically linked libraries or device drivers can be highly system dependent, prohibiting software engineers from writing a complete set of error messages for possible component mismatches. Clarify takes a user-centric, statistical approach which derives error reporting from real-world usage. For the problem of error prediction, the set T is comprised of the set of possible errors, and the problem is that of learning a function or classifier $f : x \rightarrow t \in T$ that maximizes predictive accuracy. Clarify uses the popular C4.5 decision tree classifier to approximate this function.

In many statistical software analysis settings, labeled data can be prohibitively expensive to acquire. This presents several challenges to the Clarify system, as its decision tree classifiers require labeled data in order to be

trained. In the absence of labeled data, Clarify proposes the nearest neighbor support model. Whereas the decision tree classifier directly provides a better error message, the nearest neighbor support model instead matches user's execution with similar executions experienced by others. Algorithmically, the model employs nearest neighbor algorithms to match users' executions. Given an execution x_i, the goal of the nearest neighbor software support algorithm is to find a small set of k highly similar executions $\{\boldsymbol{x}_{s_1}, ..., \boldsymbol{x}_{s_k}\}$.

Finally, the problem of automated test analysis seeks to predict software test failures. Modern software test suites can be very large, and large suites can take a long time to run, slowing the development cycle. The set of executions X in the test prioritization problem correspond to each individual software test, and their associated labels T denote if the test succeeded or failed. The goal of test prioritization is to predict failure probabilities for each test in order to quickly identify failing tests by running only a subset of all available tests. In Chapter 10, we present our statistical software analysis approach to test prioritization.

7.2 Challenges

One of the biggest challenges in any statistical, data mining, or machine learning method is that of representation. Choosing an appropriate feature set is fundamental to the success of any statistical software analysis algorithm, and for the most part can be characterized as a trade-off between minimizing computational costs of the data collection and maximizing the quality of the

resulting statistical analysis. For example: the basic block trace of an execution of a program consists of a sequential record of every basic block that was entered throughout the entire execution's history. Execution traces are the finest level of control-flow represent, and, for even moderately sized programs, their sizes can grow upwards of one gigabyte. Recording full traces of a program can incur program overheads of upwards of 1000x, which is impractical for any application where execution statistics are collected from users in a deployed setting. In the context of the Clarify system [49], it was shown that relatively 'simpler' methods that use a bag-of-words model to maintain counts of coarser control-flow features (i.e. function counts or call-site counts) yield excellent classification accuracy. Instrumentation overheads can additionally be reduced by collecting only a subset of all available features. Understanding feature correlations is critical here, as two features A and B may be highly correlated, yet one may be cheaper to collect than the other. Minimizing system overhead while maintaining an acceptable level of statistical analytic capabilities is a central issue in statistical software analysis.

Statistical software analysis data sets can be characterized by a large number of features as well as a large number of instances. Computationally, algorithms that optimize over such data sets must be efficient and scalable. High dimensional data suffers from the so-called 'curse of dimensionality', and algorithms must be carefully designed to avoid overfitting. Classifiers must generalize well to unseen data, while unsupervised methods must discover trends that are not merely anomalies present in the data but rather capture

the underlying concepts present. In terms of correlation mining, methods that compare features in a pairwise fashion will not scale to data sets with thousands or more features. A challenge here is developing robust inference methods for estimating correlations when dimensionality is high.

Complicating matters, feature values arising from program execution data often take on vastly different scales and have varying distributions. For example, a core library function may be called hundreds of thousands times during a program's execution, whereas the main function will be called only once. This presents problems for data normalization, which is further complicated by the fact that feature values often take on multi-modal distributions. Standard normalization methods such as normalizing features to have zero mean and unit covariance are not sufficient, and off-the-shelf distance measures that give equal weighting to each feature are not robust here. Previewed in Chapters 5 and 6, we will introduce in-depth applications of metric learning for improved statistical software analysis algorithms.

Chapter 8

Clarify: Improved Error Reporting for Software that Uses Black-Box Components

This chapter describes the Clarify system [49], a system that uses classifiers and nearest neighbor algorithms to provide better error messaging.

8.1 Problem Overview

Bad error reporting is more than an inconvenience for most users. A large part of modern software support cost comes from time wasted with bad error messages, which we define as any message that does not provide sufficient information for a user to fix the problem in a timely fashion. One recent study concluded that up to 25 percent of a system administrator's time may be spent following blind alleys suggested by poorly constructed and unclear messages [9]. The time and expertise required to administer modern computing systems is causing the cost of administrating, configuring and updating a machine to surpass the cost of the hardware [56]. Improving error reporting will keep down computer ownership costs and improve end-user satisfaction.

An *error* or *error behavior* is any program behavior that is not a successful completion of a task specified by a user. Errors include bugs, which

are program behaviors that do not match a program's specification. It is also an error when a program fails a consistency check on its inputs—possibly because the user entered bad input, or mis-configured the system. Errors cause programs to produce error reports, which are usually text messages or dialog boxes that inform the user that the requested action will not complete. Crashes and hangs cause the program to output the null error message. The user must interpret an error report to figure out how to get the program to complete her request, often resorting to search engines and support websites (like `support.microsoft.com`) for more information.

Consider the following model of error reporting. A given application has a set E of errors, and a set R of error reports. Unfortunately, one element $r \in R$ can correspond to multiple elements $e \in E$ because an error report is often ambiguous across multiple causes. For example, the Linux operating system uses the return code `EEXIST` to signal diverse error conditions, such as an attempt to create a file whose name already exists in a directory, or an attempt to put a rule in a routing table that conflicts with the routing table's current state. Define S as a set of vectors of runtime statistics about an application. Then the tuple $(r, s)|r \in R, s \in S$ could uniquely determine the proper $e \in E$, even though r alone fails. In fact, r might not be needed at all, s alone might suffice.

We introduce Clarify, a system to improve error reporting. Clarify consists of two parts: a runtime to monitor a black-box software component, and a classifier to interpret the output of the runtime. Clarify monitors the

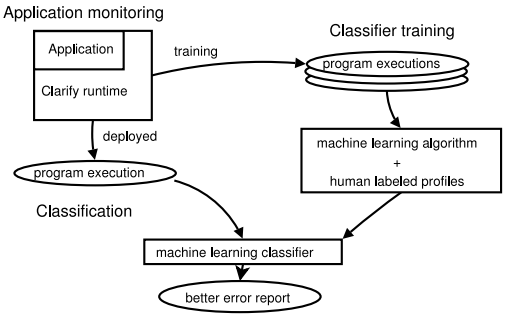


Figure 8.1: Clarify consists of a runtime monitor and a machine-learning classifier. The rectangles represent processes consuming and producing data. The Clarify runtime monitors a black-box component to generate a behavior profile that summarizes the execution history of the component. Section (A) shows a machine learning classifier, trained offline from behavior profiles. Section (B) shows the trained classifier classifying behavior profiles to produce improved error reports.

program using minimally invasive techniques like reading the program's memory or counting function calls. The Clarify runtime outputs a *behavior profile* (the $s \in S$). Clarify's users collect the behavior profiles generated when the program experiences a particular error, and train a machine learning classifier that recognizes the application's error behaviors. These users also write an improved error report that describes the error behavior and how to fix or work around it (the $e \in E$). Classifier training is done by a small minority of technically-savvy Clarify users such as support engineers who reproduce user problems in-house. End-users get the improved error reports by classifying their behavior profile. Clarify reduces the problem of improving error reports to the problem of classifying error behaviors.

Figure 8.1 shows the major components of Clarify: the runtime and

the machine-learning classifier. Each time the black-box component executes, the Clarify runtime generates a behavior profile. The behavior profile includes information about the control flow or data values of the program execution. Simple examples of a behavior profile would include counts of each function execution, or counts of how often each function returned zero.

Training the machine learning classifier happens in section (A) of Figure 8.1. A machine learning algorithm takes labeled behavior profiles as input and produces a classifier. The classifier takes a behavior profile as input and outputs a label. A label could be a non-ambiguous error code, or a lengthy description of the problem and how to resolve it. The classifier improves error reports because users can train the classifier to recognize very specific errors that have a generic error report. In settings where labeled data is not available, Clarify employs a nearest-neighbor software support method. Here, users are paired with others who have experienced similar errors.

Non-technical end-users get improved error reports from Clarify in section (B) of Figure 8.1. Clarify classifies an end-user's behavior profile, giving them more precise information about their error and how to resolve it. The machine learning classifier uses *features* from the behavior profile to determine the error classification. A feature is the value of a particular statistic, like the number of times the function `decode_audio_frame` was called in an execution of an mp3 player application. A value of zero can indicate an error where no audio frames were ever played.

The contributions of the Clarify system are:

- A system that combines runtime monitoring and machine learning in a novel way to improve error reports of black-box software components.

- A new profiling technique called *call-tree profiling*, that represents software behaviors more accurately, on average, than existing profiling techniques such as function profiling, or path profiling.

- Evaluation of a Clarify prototype on large, mature programs that currently produce unclear error messages and confusing error behavior, such as the `gcc` compiler, and the Linux operating system. Our evaluation includes an in-lab deployment of Clarify.

- Introduction of nearest-neighbor software support, where users are paired with other users who have experienced the same problem.

8.2 Behavior profiles

The problem of representation is fundamental to statistical software analysis. Here, we describe various instrumentation methods used by Clarify to monitor program runtime behavior. The Clarify runtime should collect the most expressive runtime features at the lowest cost. Expressive features are those that a machine learning algorithm can use to discriminate different error behaviors robustly. Intuitively, expressive features capture details of control flow or important data values that are caused by a particular error behavior. For instance, an incorrectly formatted URL passed to a web browser

can be correlated with the execution of functions that attempt every possible interpretation of the input URL before declaring the error.

Programs often have error-reporting routines, so one might think that the execution of such routines is a surefire indication of an error behavior. However, highly mature and factored programs, like `gcc`, reuse error-reporting code for other purposes, such as producing warnings during correct compilation. In every non-trivial program we have examined, simple correlations between an error condition and the execution of a given function or the presence of a given return code do not hold.

Clarify collects feature counts from black-box components using code instrumentation that does not require source code. Clarify must limit the number of features it collects. Error behavior is usually correlated with a small number of features, so collecting large numbers of features requires the machine learning algorithm winnow the large set of features down to the relevant few. Having more than about 70,000 features pushes the limits of many machine learning algorithms often causing address space exhaustion and unreasonable runtimes. This subsection discusses Clarify's strategy for collecting information about control flow and data values.

8.2.1 Control flow

Clarify counts features that are related to control flow because control flow is a good indicator of program behavior. In general, the more information Clarify collects about control flow, the more accurate its model of program

Behavior Profile	**Key**	**Value**
FP	<*function addr*>	# of times called
CSP	<*call-site addr*>	# of times invoked
PP	<*path in a func*>	# of times occurred
CTP	<*bitvector of caller, bitvector of callee*>	# of times executed
CSRV	<*call-site addr, predicated return value*>	# of times executed
SS	<*predicate*>	# of counts

Table 8.1: Summary of the types of features that are collected by the Clarify runtime.

behavior, but this accuracy comes at the price of greater CPU and memory overhead.

One form of behavior profiling counts the execution of function call sites. Another counts intra-procedural paths using path profiling [5]. Paths encode more information about control flow, but they are more expensive to collect than function counts. Clarify also introduces a new profiling method called *call-tree profiling* that summarizes the calling behavior of a function and its caller. The calling behavior contains some of the intraprocedural control flow that program paths represent, but it is less computationally intensive to gather.

Clarify uses counts because counts preserve rare events. Often a program will make a unique sequence of function calls before outputting a cryptic error report or crashing. Clarify uses those unique calls as the signature of the behavior. Some systems use event probabilities [12], which penalize the importance of rare code paths, especially for programs that run for long periods

of time.

We evaluate a number of approaches to behavior profiles that have different tradeoffs for performance overhead and level of execution detail.

Function and call-site profiling. The first method uses *function profiling* (FP) (sometimes called function call profiling [83]). Each function has a counter that is incremented when the function is executed. The order in which the functions are called is not retained. Function profiling is efficient and tends to be accurate when each behavior has a set of unique functions associated with it.

The second method is *call-site profiling* (CSP). This is similar to FP but the counter is associated with each call site, rather than with the call target. For direct calls, CSP differentiates among call sites, while FP does not.

Path profiling. The third control-flow based behavior profiling method is *path profiling* (PP) as described by Ball and Larus [5]. Each program path within a procedure (unique sequence of basic blocks) has a counter that is incremented when the path is executed. Path profiling distinguishes amongst program behaviors that result in different control flow within a function (intra-procedural control flow), something that function profiling cannot do.

Call-tree profiling. The fourth control-flow based profiling technique is *call-tree profiling* (CTP). Since each function in a program is one processing step, the dynamic call tree is a good representation of the program behavior.

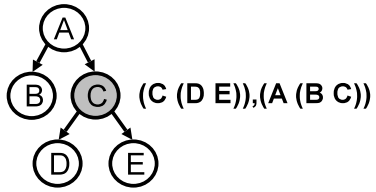


Figure 8.2: An example of call-tree profiling. The left side of the diagram is the rightmost subtree of the dynamic call tree with arrows pointing in the direction of function calls. The right side is the CTP feature that is collected when function C returns. The CTP feature is combination of call patterns of function C and its caller A.

However, the size of the whole dynamic call tree is enormous, it is impractical to use it for the classification directly. CTP counts the number of times a particular calling sequence occurs in the current function and its caller. It counts the sequence at every function return or loop backedge. CTP is an approximation to the subtree of depth 2 in the call tree.

Figure 8.2 shows an example of a dynamic call tree and the CTP pattern that is counted. Each arrow indicates the call direction and the left sibling is called before the right sibling. The tree shown is where A calls B (B may call other functions but that behavior is ignored by CTP) and then calls C, and C calls D and then E. When function C returns the call pattern for C is (C (D E)), and the call pattern for C's caller A is (A (B C)). Therefore, CTP increments by one a counter for the entire pattern of C and its caller, "(C (D E)),(A (B C))."

To implement CTP efficiently, each function gets a CTP bitvector, where each corresponds to a call site in the function. To reduce the number of

bits used, a bit is assigned only once per basic block because calls in a basic block happens in the same order. Some bits are shared for basic blocks that cannot be called together in a single path. When a function returns, CTP increments a counter for the concatenation of the function's and its caller's bitvector. CTP also increments the counter on loop backedges, clearing the current function's bitvector. In this way, CTP bitvectors remain compact.

Path profiling is able to preserve more fine-grained information about paths within the function than CTP's bitvector, but CTP preserves more information about calling context by concatenating the caller's bitvector. Because the order of the function calls can be decoded offline with the control-flow graph and the bitvector, CTP is distinct from calling context trees [2, 110] which are lossy with respect to calling sequence. Experimental results in subsection 8.6 show that CTP supports high classification accuracy.

8.2.2 Data

Data values can provide robust characterization of error behavior, though a naïve implementation can greatly increase the number of features thereby canceling any benefit. For instance, to associate return values with their call sites, Clarify can count call site, return value pairs. A function that has 100 distinct return values will increase the number of features by 100. Such encodings increase the complexity of the classification task considerably as machine learning algorithms have performance and accuracy problems when confronted with large numbers of features.

Predication is a standard technique to reduce the feature space of data values [67]. We define nine predicates which are applied to Clarify data and return values; the predicates map raw values to feature values. The predicates indicate whether the raw value is equal to zero, equal to 1 or -1, is a small or large positive or negative integer, or is a pointer to the stack or heap. The thresholds for small and large positive and negative integers are arbitrary: any value with absolute value less than 100 is small, any value with absolute value greater than 100 that is neither a stack or heap pointer is "large".

Call-site profiling with predicated return values. Call-site profiling with predicated return values (CSRV) counts pairs of call sites and predicated return values. If call-site A returns 255 one hundred times and returns -1 once, then the feature <A, *large int*> has a count of 100 and the feature <A, *equals -1*> has a count of 1.

Stack scraping. Stack scraping (SS) is a behavior profile that relies only on the dynamic data values from an execution instance, rather than on control flow. The insight behind stack scraping is that the stack contains control flow history in the form of return addresses (some of them residual in memory below the current stack pointer) and status information like function return codes.

At the moment the program returns an error code, its execution is paused, the range of memory allocated to the program stack is traversed, and a feature vector representing that instance of execution is created by applying predication to each word in the stack range. The representation trades some

fidelity for convenience and compactness, compared to instrumentation-based control flow histories. The scraper obtains the stack and heap bounds dynamically (from `/proc/pid/maps` on Linux) so it can differentiate pointers to the stack and pointers to the heap.

Stack scraping is unique in Clarify feature sources in that it does not require instrumentation of the source program. It imposes very little runtime overhead, but it is also the least accurate feature source.

8.3 Deployment issues

This subsection discusses the different deployment scenarios for Clarify.

8.3.1 Forensic vs. live deployments

Clarify can be deployed in two ways: to improve any error report a program can give (*live deployment*), or to improve a fixed set of error reports (*forensic deployment*). A live deployment will instrument an entire executable, sacrificing some performance to collect data about the entire application's behavior. A forensic deployment only collects data that is known to help disambiguate a fixed set of error reports.

8.4 Minimizing human effort

Clarify requires humans to label or generate examples of faulty error reports in order to train a machine learning classifier. Even without a classifier, Clarify should help users. We describe nearest-neighbor software support, an

execution mode Clarify uses when it has no classifier. The subsection next describes distributing the work of labeling profiles among a software support community.

8.4.1 Nearest neighbor software support

Clarify needs a certain number of labeled examples to build an accurate machine learning classifier (exact numbers are problem-dependent and quantified in Subsection 8.6.4). Before it has trained a classifier, Clarify uses nearest-neighbor search to match similar behavior profiles. For instance, users of `mpg321` can give their email addresses to a support website. If a user has a problem that she does not understand, she sends her behavior profile to the site which runs Clarify. The site returns the emails of 5 other users who opted in to the system and who likely experienced the same application behavior. (The system might give out a particular email address only 3 times and take other steps to make sure participants are not overwhelmed with email or put on spam lists.) As the results in Subsection 8.6.6 show, nearest-neighbor search is sometimes highly effective, but it is not as accurate in general as building a classifier.

8.4.2 Labeling behavior profiles

The Clarify classifier must be built from labeled behavior profiles. There are three ways this labeling can be done.

- Members of a support organization can do all labeling. This approach

is human resource intensive, but provides high-quality labeling.

- End-users can label their profiles, distributing the work across many more people, but enabling malicious or inept users to add noise in the form of incorrect labels. End-user contributions can be graded by support staff or by peer reputation (like what is done on current support websites).

- Support engineers can write scripts to generate many variant inputs for each problem. All inputs exercise the same problem, so they all share the same label. We use this method to evaluate Clarify. It requires the most expertise, and the inputs are not guaranteed to accurately model real-life inputs.

8.5 Benchmarks

Clarify is intended to improve the error reporting of complex, black-box software components. To evaluate Clarify, we choose benchmarks that are common, heavily-used programs for which non-exotic error conditions lead to misleading or non-existent error messages. That common utilities provide shoddy error reporting makes clear the motivation for Clarify.

This subsection summarizes the benchmarks and the kinds of problematic errors they report. We explain the behavior underlying the error reports—it is this underlying behavior that Clarify is intended to discover. Three kernel benchmarks are also tested by the Clarify system but are not described here

(iptables, iproute2, and mount). Experimental results for these benchmarks are provided in the result section.

8.5.1 User-level programs

gcc. The GNU C compiler is a popular compiler, containing both hand-written and automatically generated source code. Our experiments use version 3.1, executing only the compiler (the `cc1` phase), using the ".i" file output of the pre-processor, drawn from a pool of 4,070 files pre-processed from the Linux kernel 2.6.13 distribution. A corruptor script randomly modifies correct source code to exhibit mistakes from four error classes: adding a semicolon after an `if()` that has an `else` clause, causing the compiler to fail on the `else`; omitting the closing curly bracket of a switch block causing an "end of file" error; deletion of a semicolon, yielding a generic syntax error, often on a very different line from the removed semicolon; misspelling a keyword which also generates a generic syntax error. All error classes result in confusing and imprecise error messages.

mpg321. `mpg321` is an mp3 player for Linux. This benchmark has three failure modes: file format error (e.g. trying to play a wav file as if it were an mp3), corrupted tag (mp3 metadata is stored in ID3 format tags, e.g., artist name), corrupted frames (mp3 frame data is corrupt). The Clarify classifier distinguishes between these three failure modes and normal execution. The application itself does not give any consistent error message for any of these error cases.

latex. Latex is a typesetting program widely used by the research community. Its error reporting is known to be obscure. Rubber [88] is a tool that filters `latex`'s output to make it more comprehensible to the user. However, many of `latex`'s error messages are generic and many have varied root causes, making it difficult for users to understand what went wrong and fix it.

Our `latex` benchmark has 26 ambiguous error cases, too many to summarize here, so we describe one illustrative example. A website [23] contains an explanation of all the classes.

If a `table`, `array` or `eqnarray` has more separator characters (ampersands) than columns, `latex` prints the obscure error message, "*!Extra alignment tab has been changed to* `\cr`". Most `latex` books and most `latex` support websites recommend checking the number of ampersands if a user receives this error. Some websites and books are helpful enough to suggest a missing end of row symbol \\ on the previous line. While forgetting the double backslash will cause the error report, the error report is not unique: misuse of the `\cline` command (a directive that draws a horizontal line in the table) will result in the same message if one of the arguments to `\cline` refers to a non-existent column in the table. Users who make the `\cline` mistake get an error message that almost all support options say are due to one of two possible causes, even though there is a third possible cause. Error reports are biased to their most likely cause, leaving a user who executes a less likely scenario scratching her head, potentially for a long time.

App.	inst.	Er	FP	CSP	CSRV	PP	CTP
latex81	34,677	81	395	6,802	61,202	1,504	23,296
latex27	11,528	27	395	2,191	21,425	1,504	20,761
mpg321	263	4	128	1,162	11,495	21,954	1,318
gcc	1,582	5	2,920	57,221	514,973	40,513	93,246
iptables	131	5	56	70	N/A	N/A	N/A
iproute2	146	4	146	475	N/A	N/A	N/A
mount	1,920	5	292	292	N/A	N/A	N/A

Table 8.2: Sizes of the Clarify behavior profiles for each benchmark. The second and third columns show the number of instances (program executions) and the number of error classes for each benchmark. The remaining columns show the number of features for each behavior representation. SS is not shown in the table since it always has 9 features. Kernel utilities only generate the first two behavior profiles due to limitations in how the kernel can be instrumented.

8.5.2 Complexity of Clarify benchmark datasets

Table 8.2 summarizes the complexity of the Clarify benchmark datasets. Each program has at least three ambiguous or misleading error classes and one normal class. Latex27 has 26 ambiguous error classes and 1 normal class. In general, more accurate profiles have more features. For instance, there are 533 functions in latex, but 6,802 call sites, and call-site profiling is more accurate than function profiling.

Our benchmarks all have approximately equal number of instances per error type. This distribution is not intended to model the frequency of bugs occurring in the field, but rather trains the classifier to distinguish among the given cases.

8.6 Evaluation

We evaluate Clarify according to four criteria: accuracy, performance, training cost, and scalability. First, Clarify must correctly classify program behaviors that share ambiguous error messages. Accuracy is summarized by the ratio of behavior profiles correctly classified to the total number of profiles (Subsection 8.6.1). A perfect classifier would correctly identify each error scenario from the behavior profile for each benchmark. As further validation of our classification models, we examine the decision trees generated by Clarify in Subsection 8.6.3. We show that the tree tests program features that intuitively correlate with the observed behavior.

The accuracy of Clarify must come at an acceptable performance cost, which is measured in Subsection 8.6.2. A successful deployment of the Clarify system should incur minimal overhead costs.

Labeled examples can be expensive to collect, as determining the error type of a given instance can require considerable human effort. Subsection 8.6.4 shows how many labeled behavior profiles are required to generate a Clarify classifier. In the absence of any labeled data, Clarify employs a nearest-neighbor algorithm, where users are paired with other users who have experienced the same problem (Subsection 8.6.6).

Finally, subsection 8.6.5 examines how the accuracy of Clarify's classifiers scale with the number of error classes. The robustness of the Clarify classifiers is demonstrated by the relatively high accuracy obtained for the

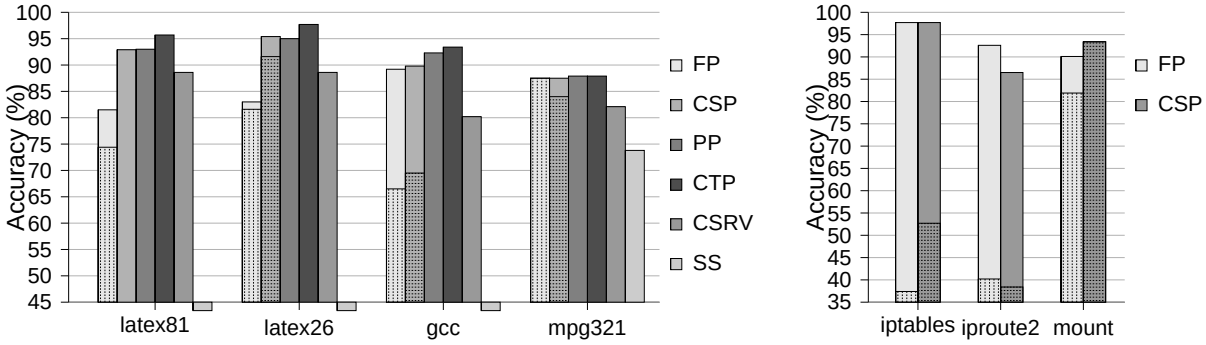


Figure 8.3: The figure shows the accuracy of the classifier used to distinguish the error cases, based on behavior profiles, for each benchmark. For each benchmark a classifier is built using different behavior profiles: function profiling (FP), call-site profiling (CSP), path profiling (PP), call-tree profiling (CTP), call-site profiling with predicated return values (CSRV), and stack scraping (SS). The figure also presents sampled versions of function profiling and call site profiling with a sampling rate of 10% (in the stippled, lower bar in the stacked FP or CSP entry).

latex benchmark with 81 classes.

8.6.1 Classification accuracy

Clarify uses decision trees to classify. Decision trees are nested if-then-else statements where each leaf corresponds to a single class prediction. An advantage of decision trees (over more continuous methods like support vector machines) is their ease of interpretation. It is possible for a software engineer to validate the classifier based on knowledge of program structure. Further, in the context of our experiments with Clarify, decision trees are as accurate as other machine-learning methods. Although Clarify's instrumentation computes thousands of features that describe each program execution, the task of

classifying error messages can be accomplished by analyzing only a few features. This can be seen through the relatively small size and high accuracy of Clarify's decision trees. In contrast, methods that optimize over the entire feature set—e.g. logistic regression or support vector machines—tend to yield over-fitted models with lower accuracy. Other algorithms that optimize over only a subset of features, such as rule learning and boosted decision stumps, yield classifiers we found to be competitive with decision trees.

Figure 8.3 shows the accuracy of user and kernel benchmarks, for several different behavior representations. These tables report accuracy using 5-fold cross validation, a standard technique for evaluating classifiers. The dataset is partitioned into five subsections, the classifier is trained and tested five times; it is trained on four subsections of the data and its accuracy tested on the remaining fifth. The average of these five tests is the reported accuracy of the classifier.

The decision trees are built using an implementation of the C4.5 algorithm [84] found in the WEKA machine learning package [104]. Call-tree profiling (CTP) demonstrates the best overall accuracy. Call-site profiling (CSP), path profiling (PP) and CTP have an accuracy of over 85% on every user-level benchmark, and call-site profiling has over 85% accuracy for kernel benchmarks. 85% accuracy is a significant help for improving error reports.

To evaluate sampling, we present results for sampling FP and CSP, with a sampling rate of 10% (which is generous for systems that use sampling [67]). For example, the sampled function counts record one of every ten function

App.	CSP		CTP	
	Forensic	Live	Forensic	Live
latex	4.2%	5.3%	1.15%	321%
mpg321	0.3%	1.2%	1.29%	105%
gcc	5.6%	11.3%	10.1%	371%
iptables	1.1%	3.2%	N/A	N/A
iproute2	4.7%	7.6%	N/A	N/A
mount	1.1%	3.1%	N/A	N/A

Table 8.3: Slowdown of programs running under the Clarify runtime using CSP and CTP for a forensic deployment (which can only classify errors known during training), and a live deployment (which can classify new errors found after deployment).

calls, uniformly at random. The sampled results are the stippled part of each bar, achieving lower classification accuracy than non-sampled data for almost every benchmarks. The poor accuracy of sampling confirms our intuition that sampling is the wrong approach for classifying program behavior, because Clarify must be sensitive to rare events.

8.6.2 Performance

Table 8.3 shows the performance of live and forensic deployments of call-site profiling. All timing runs are on a dual-processor Intel Xeon 3.0GHz with 2GB of RAM. Because there is no freely available static binary translator for the x86 architecture, the experiment modifies the assembly code of the programs to count call sites in exactly the way a binary modification tool would do it. On the x86 a count with a known address can be incremented with a single instruction. The counters reside in a memory mapped file, so the results can be collected after program termination.

Each benchmark runs several inputs to obtain a running time that is long enough to measure accurately: `gcc` compiles the 23 largest .i files from the Linux 2.6.16 distribution, `mpg321` decodes 256 frames of 200 mp3 files, and `latex` processes 5 files with a total of 27,587 lines. We average the user time of three executions. The remaining rows in Table 8.3 show benchmarks run on the 2.6.17 version of the Linux kernel. The kernel behavior profile is built using the kprobes [59], a dynamic instrumentation package that is standard in Linux. Kprobes uses breakpoints so it is a more expensive form of instrumentation. We use it to collect only function profiling and call-site profiling.

Overhead is low (less than 12%) for call-site profiling, both for forensic and live deployments. Forensic deployments are faster than live deployments (less than 6%), sometimes dramatically so. The machine learning model for the forensic deployment chooses features that training runs indicate are cheaper to collect, e.g., they reside in functions that are called infrequently. We use a cost-sensitive decision tree algorithm [32] that uses training data to find the minimum cost tree whose accuracy is within 1% with our cost-oblivious tree. Details of this algorithm are provided in the Chapter 9. Forensic deployment is an effective means of deploying richer behavior profiles like CTP at reasonable levels of overhead.

8.6.3 Verifying the machine learning model

Machine learning algorithms train classifiers without any domain knowledge regarding the underlying semantics of the program's behavior. It is pos-

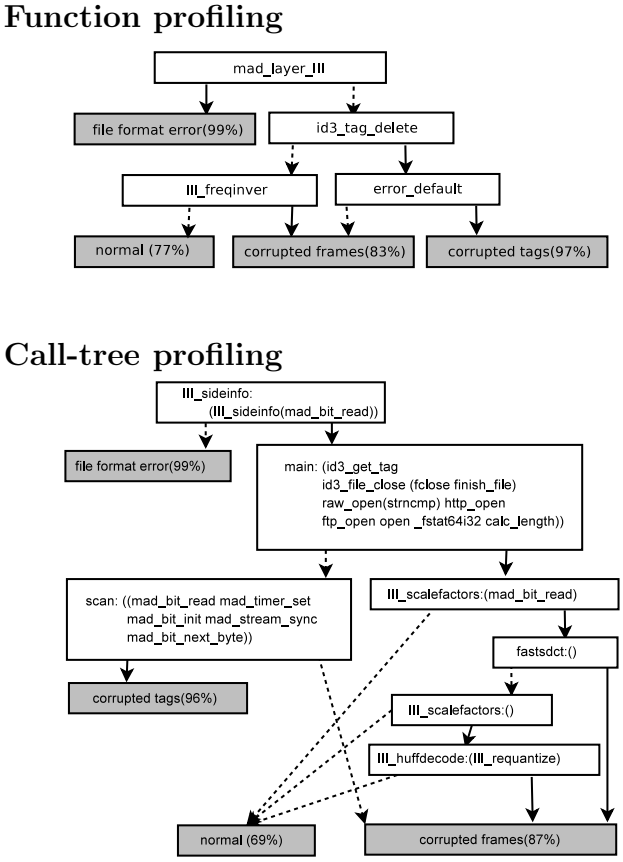


Figure 8.4: Decision trees produced for the `mpg321` benchmark. Dotted lines are taken when the normalized count of the feature value is less than or equal to a threshold, while the solid line is taken when it is greater than the threshold. The threshold is determined automatically for each benchmark by the decision tree algorithm, and can be different for each node in the tree. Clear boxes are features. FP features are normalized function counts, and call-tree profiling features are normalized counts of CTP subtrees (represented by the symbolic tree names in brackets, with function names for nodes in each call tree). Shaded boxes are error classes.

sible for a classifier to fail miserably on unseen data because the classifier examines features that are semantically unrelated to the behavior it classifies.

To make sure that Clarify classifiers use program features that intuitively relate to the behaviors they classify, we examined several classifiers by hand. Classifiers trained using function profiling and call-tree profiling for the mp3 player `mpg321` are shown in Figure 8.4. The trees show how each behavior profile provides different clues to the classifier about the same underlying behavior.

The function profiling tree is composed of a simpler set of rules that depict differences in control flow across the four error classes. At the root of the tree, the function `mad_layer_III` provides near perfect discriminative information for the 'wav' error class: the `mad_layer_III` routine is part of the `libmad` library and is called when the audio frame decoder runs. Since the wav format is among the formats not supported by mpg321, it will not successfully decode any audio frames, and the `libmad` library will never call `mad_layer_III`. The `id3_tag_delete` routine differentiates between the corrupted tag and other classes. The ID3 tag parser in the `libid3tag` library dynamically allocates memory to represent tags and frees them with `id3_tag_delete`. If tag parsing fails, the memory for a tag is not allocated. Since no tag parsing succeeds in the corrupted frames case, `id3_tag_delete` is never called to free the tag memory, making its absence discriminative for that class. The `libmad` audio library's default error handler `error_default` is used if the application does not specify one. `mpg321` does not specify its own error handler, so the presence of the function indicates corrupted audio frames, and its absence indicates the corrupted id3 tags case. Finally, `III_freqinver`, which performs subband frequency inversion for odd sample lines, is called very frequently as part of

the normal process of decoding audio frame data. When there are corrupted frames, this function is called less frequently, and the decision tree algorithm finds an appropriate threshold value to separate the normal from the corrupted case.

The decision tree built on call-tree profiling data has a richer combination of data sources than function profiling. Call-tree profiling uses the presence of the `libmad` library function `III_sideinfo` (which decodes frame side information from a bitstream) calling the utility function `mad_bit_read` as an indicator of successful audio frame decoding. The lack of that calling pattern reliably indicates a file format error. The corrupted frames class is once again differentiated from the normal class by a threshold value on a subtree of `libmad` functions that will only be called during successful decoding of audio frame data, such as `III_scalefactors`, the discrete cosine transform function `fastsdct`, `III_huffdecode`, and so on. The `libmad` function `scan` encapsulates the process of reading mp3 files. A CTP rule (decoded bitvector) wherein `scan` calls a function that calls a number of low-level stream manipulation routines such as `mad_bit_read`, and `mad_timer_set`, and so on, provides discriminative power in combination with a similarly complex control flow pattern in `main` for the corrupted tags error class. The decision tree node whose CTP rule involves `main, id3_get_tag`, and so on differentiates between normal and error conditions for the handling of ID3 tags, while the decision tree node whose CTP rule involves `scan` discriminates between successful and unsuccessful audio decoding. The high level pattern exposed by these rules

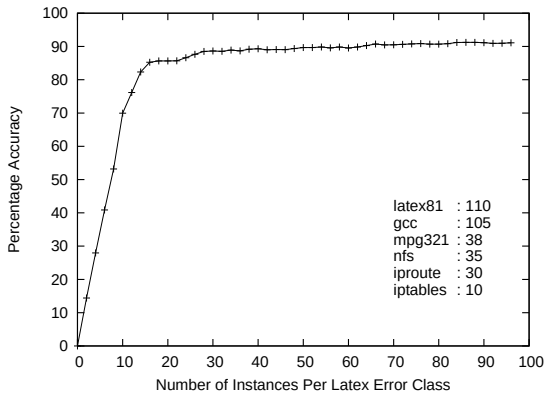


Figure 8.5: The curve shows how the accuracy increases as the number of training instances per error class is increased. The dataset is the latex benchmark with 75 classes. The text on the graph gives the minimum number of training instances needed for a benchmark to achieve accuracy within 1% of accuracy obtained using all the training data.

is the combination of failed ID3 tag parsing with successful audio decoding, which precisely describes the corrupted tag error class.

8.6.4 How many labeled behavior profiles are needed?

The classifiers used by Clarify are trained with labeled behavior profiles. Labeling profiles generally requires human effort, so it should be minimized. In general, classifiers trained with fewer labeled training instances will result in less accurate models. In this subsection, we investigate the tradeoff between classification accuracy and the amount of training data used in building the classifier. Figure 8.5 plots the classification accuracy of the latex benchmark as a function of the number of instances used in training (the benchmark

includes 75 of the 81 distinct error classes) . The C4.5 algorithm used to build the decision tree is surprisingly robust: with as few as 15 examples per class, the algorithm achieves an accuracy of 86%. Looking at the legend in Figure 8.5, we can see that to achieve accuracy within 1% of the maximum, only a small subset of the training data is required. For example, gcc needs only 105 instances to attain the accuracy level of 88.9% which is within 1% of the accuracy reached when we use all of the 300 available examples per class.

A human does not need to label each behavior profile individually. For our training sets we use a script to induce errors in the program input, producing large numbers of training examples with little human involvement. However, inducing errors by a script is not necessarily an accurate model for the errors that Clarify would see in deployment.

8.6.5 Scalability

In this subsection we analyze how Clarify scales as the number of error classes increases. `latex` has 247 unique error messages, and we evaluate the scalability on 81 behavior classes—about one third of all possible `latex` errors.

Table 8.4 shows how model creation time and classification accuracy scale as the number of error classes increases. We consider subsets of error classes with varying sizes. For each size, we picked 10 random subsets of error classes and ran our experiments. The curves shown in the graph are the average of the results for each size.

We can see that as the number of error classes increases the accuracy

# Classes	Accuracy	Creation Time
10	97.8 %	25min
20	97.5 %	1hr 37min
35	94.9 %	6hr 2min
50	94.3 %	10hr 26min
65	93.9 %	11hr 50min
81	93.6 %	18hr 28min

Table 8.4: The accuracy and time to create the classifier as the number of behaviors is increased in the `latex` benchmark.

drops from 97.8% to 93.6%. This decrease is acceptable considering that the number of behaviors has increased by a factor of 8. The training time of the model increases from under 30 minutes to more than 18 hours as the number of error classes increases from 10 to 81. This increase does not hinder scalability since the model is trained offline. Of greater practical concern is the execution time needed to evaluate the decision tree, as this largely determines the amount of processing done at the client end. Our experiments show that the time to execute the models averaged 10ns, with a maximum of 21ns, which is imperceptible for almost any application. Clarify scales to nearly one hundred error behaviors without much loss in accuracy or substantial increase in processing time.

8.6.6 Nearest neighbor software support

In contrast to the decision trees used by Clarify's classifiers which rely on only a small subset of all features, nearest-neighbor algorithms rely on averages over all features. For example, the Euclidean distance between two instances, a popular metric used for nearest-neighbor searches, is a function

of the average of the squares of the difference between each pair of feature values. Such distance functions are particularly susceptible to differences of scale among the various features.

In Clarify, features take on vastly different scales: some features may have a count under ten, while others may have upwards of one million occurrences. Furthermore, for some features, the count is a function of the length of the program execution, and for others it is independent of program execution. For example, a parsing-related feature for gcc will be called many more times for a longer file that contains many repetitions of particular construct, than a shorter file. Some subsections of code—e.g. initialization functions—will be called a (roughly) constant number of times and thus will take on values independent of the program execution length.

To overcome such scaling challenges, Clarify employs a linear regression-based feature scaling method. For each feature y, a least-squares line is fitted to correlate each feature value with its corresponding program execution length x (defined as the sum of all feature values of a given execution instance). The feature value is normalized to be the scaled difference between the feature value y and the fitted feature value $f(x)$. The scaling factor is determined such that the variance of each resulting feature is one. We note that for features that have no correlation with program length, the linear regression step will have no effect on the final normalized feature values.

Table 8.5 gives the expected number of correctly classified neighbors for a nearest neighbor search returning five neighbors. Euclidean distance is used.

App.	FP	CSP	CTP	CSRV
latex26	0.75	1.01	0.52	0.93
mpg321	2.22	4.21	1.52	1.40
gcc	2.73	2.67	2.19	0.97
iptables	1.12	1.03	N/A	N/A
iproute2	3.10	2.77	N/A	N/A
mount	2.71	2.40	N/A	N/A

Table 8.5: In the absence of labeled training data, Clarify uses a nearest-neighbor algorithm with linear-regression based feature scaling. This table shows the expected number of correctly classified neighbors for a five-nearest-neighbor search.

For some benchmarks with many classes (the `latex` benchmark in table 8.5 has 27 classes), accuracy of the nearest-neighbor search is somewhat lower. In such cases, a larger number of neighbors should be returned.

8.7 Related work

We first contrast Clarify to several systems that appear similar. Clarify improves error reporting by classifying program behavior, it does not find program bugs [1, 21, 50, 67]. An ambiguous error message or return code might meet the specification for a program (e.g., the `netlink` standard for error reporting). Clarify does not attempt to find the root cause of program faults [17, 83], misconfigurations [63, 102], or program crashes [11, 16, 75, 76]. Its aim is to classify the application behavior to help the developer or end-user get better error reports when these events happen.

The remainder of this subsection compares Clarify with problem diagnosis systems, and systems that classify program behavior. Clarify does help

software problem diagnosis and it classifies program behavior.

8.7.1 Problem diagnosis systems

A group at Microsoft Research correlates low-level system events with error reports to automate problem diagnosis [107, 108], just as Clarify does. They currently focus only on forensic deployments (in our terminology), and on building models from sequences of system calls. Clarify uses control-flow and data from the program, which allows it to deal with errors that involve only user code. Ph [96] also uses sequences of system calls to build a model, though their model detects host intrusions. While system calls are a good representation of certain types of program behavior, many programs make few systems calls (e.g., SPEC). Because every named system call has wrapper functions from user-space libraries, Clarify can detect system calls by detecting function calls to the wrapper functions, giving it a richer input source to determine program behavior.

Statistical bug isolation [66, 67] correlates low-level application behavior with application behavior (bugs) and builds a model, as Clarify does. Statistical bug isolation requires a special compiler to insert invariant checks into the program, while Clarify records a small amount of control-flow and data continuously. Statistical bug isolation samples the invariants it inserts to get good performance. Subsection 8.6.1 demonstrates a sharp loss of accuracy if Clarify uses sampling. Statistical bug isolation must eliminate sub-bug and super-bug predictors; Clarify has an analogous struggle to gain enough train-

ing instances to isolate the program behavior created by the error condition. The systems could be used together to gather statistical data on crashes and provide better error messages for crashes and other misbehaviors.

DIDUCE [50] uses dynamic program invariants to detect program behavioral anomalies. The anomalies can indicate program bugs, but at a performance slowdown of 6–20×. Clarify is much faster and can classify program behavior that is not anomalous.

Stack backtraces are used by many remote diagnostic systems like Dr. Watson [75], Microsoft's online crash analysis [76] and GNOME's bug-buddy [11]. IBM has a system to classify stack backtraces harvested on a crash [16], and the technology has been deployed in their TrapFinder tool. Their motivation is similar to Clarify's—reduce the human effort needed to match problems from different program executions. Clarify diagnoses a wider range of problems than crashes, and it operates on behavior profiles, which are a richer source of data than stack backtraces.

8.7.2 Classifying program behavior

Classifying program behavior has received attention in the software engineering literature. Podgurski *et al.* [83] identify a similar motivation to Clarify and they also investigate `gcc` behavior. Clarify is more accurate (over 85–100% accurate, as compared to 24–96%), and is more of a complete system, designed to address the problem of improving error reporting. Bowring *et al.* [12] models software behavior as Markov models using control flow be-

tween basic blocks and then uses active learning to cluster the models. Markov models use probabilities which make them insensitive to rare events. Clarify needs sensitivity to rare events because rare events often characterize an error behavior—see the sampling results in Subsection 8.6.1. Bowring et. al. evaluate their method on 33 versions of SPACE, which is a very small 6,200 line program.

Liu *et al.* [71] use program behavior graphs as features for a machine-learning model just as Clarify uses data related to program control flow. The number of program behavior graphs grows quickly with program size, and can become computationally intractable even for the small Siemens programs [57] used to evaluate their method.

SimPoint [94] characterizes the phase behavior of applications using basic block execution counts to maintain the accuracy of architectural simulation while executing fewer instructions. The types of program behavior it detects are coarse-grained and occurs over much longer time windows than the errors that Clarify detects. SimPoint can reduce its dataset to 15 dimensions and maintain phase-detection accuracy. Clarify's classifiers must be sensitive to small, localized changes in behavior that form the signature of an error behavior. As seen in Table 8.2, Clarify's representations have tens of thousands of features. We verified that using random projection to reduce the feature count, like SimPoint does, dramatically reduces Clarify's accuracy.

Program paths [5] have been used to analyze runtime program behavior. Path Spectra [85] approximate an execution's behavior with the occurrence

(or frequency) of the individual paths. Spectral differences have been used to identify the portions of a program's execution that differ with different inputs, notably, during Y2K testing [51]. Path Spectra focused on identifying path differences between several program runs, whereas Clarify's novel use of path profiling uses machine learning to identify which paths are *common* to each error class. Clarify's call-site profiling is much more efficient and nearly as accurate as path profiling.

Chapter 9

Inferring Correlations with Costs: A Cost-Sensitive Decision Tree Algorithm

In many real-world problems, collecting features comes at a cost. Statistical software analysis is no exception, and the cost incurred of collecting features comes from the instrumentation overhead required to count method calls, basic block calls, etc. In this chapter, we present a method which learns a decision tree classifier that maximizes both classification accuracy as well as minimizes the costs required to collect the features tested in the tree [32].

9.1 Problem Overview

In the prototypical cost-sensitive classification problem of medical diagnosis, tests are performed sequentially until a diagnosis is made. Classifiers such as decision trees are natural for this problem, as predictions can be made by testing only a small subset of total features (i.e. those features present in the path from the root to the predicted leaf). In this problem, it is acceptable to have very expensive tests present in the decision tree as long as these tests are relatively unlikely to be needed in a typical evaluation of the tree.

In many settings, sequential testing is not feasible. In particular, if

objects to be classified are transient, then they are not available for further testing when diagnosis (i.e. classifier evaluation) is performed. Consider the problem of classifying software errors: the system can be monitored during run-time, but acquiring additional "after the fact" information requires reproducing the error. Error reproduction can be time consuming and costly because oftentimes system errors are non-deterministic or environment-dependent. To efficiently classify software errors, a system must minimize runtime monitoring costs. Equivalently, the cost of the classifier—i.e. the aggregate cost of monitoring needed to construct any feature that can possibly be tested by the classifier—must be minimized.

In this chapter, we present a cost-sensitive decision tree algorithm for forensic classification: the problem of classifying irreproducible events. Here, we assume that all tests (i.e. features) must be acquired before classification; consequently, the classification cost equals the sum of the costs of all features that the classifier may use for testing. We derive our algorithm by expressing the ID3 decision tree algorithm in an information theoretic context; from this, we present a cost-sensitive generalization for the information gain and gain ratio criterion. When used in conjunction with these modified cost-sensitive purity functions, the resulting decision tree algorithm minimizes testing costs under the forensic classification problem while simultaneously maximizing accuracy.

For evaluation, we incorporate our cost-sensitive criterion into the C4.5 decision tree algorithm. We compare our algorithm to existing methods across

various datasets from the UCI machine learning repository, and show that, for a given level of accuracy, our algorithm builds cheaper trees than existing methods. Finally, we apply our algorithm to the Clarify system. We propose a cost model to determine feature costs, and show that, for many programs, computational overhead can be reduced by several orders of magnitude with only a slight ($< 1\%$) decrease in classification accuracy.

9.2 Cost-sensitive decision tree algorithm

We first give an overview of the ID3 decision tree building algorithm. We show that the information gain criteria that ID3 uses to greedily build decision trees can be thought of as a greedy optimization procedure for maximizing an information-theoretic objective function. We then propose an objective for our cost model from which we derive a new cost-sensitive information gain criteria. Finally, we extend our cost-sensitive criteria to the popular gain ratio function that is used with C4.5.

9.2.1 ID3 and cost-sensitive information gain

The ID3 decision tree building method uses a top-down, greedy search procedure and represents the core of Quinlan's highly successful C4.5 decision tree algorithm. For simplicity, we will outline the algorithm as a process of building a tree over a nominal feature space with arbitrarily many classes. However, all methods presented can be easily generalized to continuous attributes.

Given a decision tree with k internal nodes $1, ..., k$, each of which split on features $F^1, ..., F^k$, we will denote the tuple (X^i, y^i) to be the set of (instance, label) pairs that will 'pass through' (for internal nodes), or 'end at' (for leaf nodes) node i when the tree is evaluated. We will define $V(f)$ to be the set of values that feature f takes on, and let $(X^j_{[f=v]}, y^j_{[f=v]})$ denote the set of instances in (X^j, y^j) such that feature f takes on value v. Given some leaf node j, the ID3 algorithm splits on the feature f which maximizes the information gain.

$$Gain(X^j, f) = H(y^j) - \sum_{v \in V(f)} \frac{\left|X^j_{[f=v]}\right|}{|X^j|} H\left(y^j_{[f=v]}\right) \tag{9.2.1}$$

where H is the entropy of a set of class labels:

$$H(y) \;=\; -\sum_{\ell \in Classes} \frac{|y_{[Class=\ell]}|}{|y|} \log \frac{|y_{[Class=\ell]}|}{|y|}.$$

The information gain can be thought of as the expected decrease in entropy caused by splitting on feature f. Furthermore, if we think of the feature f and class labels y^j as random variables over the set of instances, then the information gain is equivalent to the mutual information between f and y^j, which we denote $I(y^j; f)$. Mutual information is a standard information-theoretic measure of the correlation between two random variables [26].

Since the ID3 algorithm builds the tree in a top-down manner, the split at the root node of the tree is selected using $X^1 = X$, the set of all instances used to train the tree. Recursively applying (9.2.1) in terms of $H(y)$, and

re-arranging terms yields:

$$\begin{aligned}\sum_{i \in internal} \frac{|X^i|}{|X|} Gain(X^i, F^i) &= H(y) - \sum_{\ell \in leaf} \frac{|X^\ell|}{|X|} H(y^\ell) \\ &= I(y; p), \end{aligned} \tag{9.2.2}$$

where p is a random variable that gives the class values as predicted by the tree. Thus, maximizing the mutual information between the true and predicted class labels is equivalent to maximizing the weighted sum of the information gain scores at each internal node of the tree. Furthermore, the ID3 algorithm can be viewed as a greedy method to maximize this mutual information.

In an effort to reduce the cost of the features used to build the ID3 decision tree, we propose the following multi-way objective criteria that maximizes the mutual information while minimizing cost:

$$I(y; p) - \gamma \sum_{f \in F} cost(f), \tag{9.2.3}$$

where $F = \cup_{i=1}^{k} F^k$, the set of features used in the tree, *cost* is an arbitrary cost function, and $\gamma \geq 0$ is an adjustable parameter that tunes the tradeoff between maximizing mutual information and minimizing costs.

We optimize this quantity in the same top-down, greedy manner that ID3 operates by maximizing the right hand side of (9.2.2) with respect to node i. We get a new cost-sensitive information gain feature selection criteria of the form:

$$CSG(X^i, f) = \frac{|X^i|}{|X|} Gain(X^i, f) - \gamma \cdot cost(f) \mathbf{1}_{[f \notin F]}. \tag{9.2.4}$$

The indicator function $\mathbf{1}_{[f \notin F]}$ allows for the re-use of features already added to the tree without incurring additional costs. The normalization for the first term can be factored out if the cost term is not present and reduces to the basic ID3 splitting criteria (9.2.1). This normalization results in criteria that are willing to pay for more expensive features at higher levels of the tree, since a larger percentage of the distribution will 'pass through' these nodes. Nodes near the leaves of the tree will be evaluated on a relatively smaller portion of instances, and, consequently, the criteria (9.2.4) will seek cheaper features for such nodes.

9.2.2 Cost-sensitive gain ratio

Quinlan's C4.5 decision tree algorithm [84] uses a modified splitting criteria, called gain ratio, that normalizes the information gain score of splitting on feature f by the entropy of the feature f:

$$H(X, f) = -\sum_{v \in V(f)} \frac{\left|X_{[f=v]}\right|}{|X|} \log \frac{\left|X_{[f=v]}\right|}{|X|}. \tag{9.2.5}$$

Well-balanced splits over few distinct values $V(f)$ will result in a smaller split entropy and thus result in a larger gain ratio score. Although the gain ratio score does not have a direct interpretation in terms of classical information theory, we can still derive a global objective for the criteria. Let $Path(\ell)$ be the set of nodes in the path from the root of the tree to node ℓ. If the ID3 algorithm is used with the gain ratio criteria, then the resulting objective is:

$$H(y) - \sum_{\ell \in leaf} \frac{|X^\ell|}{|X|} \left(\prod_{j \in Path(\ell)} \frac{1}{H(X^j, F^j)} \right) H(y^\ell),$$

where F^j was previously defined to be the feature that node j splits on. This objective is similar to that if the information gain criteria is used, as given by equation (9.2.2). In (9.2.2), the weights of each leaf node are determined from the distribution of the training set (i.e. the probability that a randomly drawn instance will reach a given leaf node). Here, the weights of each leaf node are a function of the gain ratio normalization factors (9.2.5) as well as the training set distribution.

Modifying the gain ratio criteria in terms of our cost-sensitive framework, we have:

$$CSGR(X^i, f) = \frac{|X^i|}{|X|} \left(\prod_{j \in Path(i)} \frac{1}{H(X^j, F^j)} \right) Gain(X^i, f) - \gamma \cdot cost(f) \mathbf{1}_{[f \notin F]}. \tag{9.2.6}$$

Whereas the $CSGain$ criteria (9.2.4) normalizes the $Gain$ term for node j by the probability of an instance arriving at node j, the above criteria normalizes by weights that are a function of both the training set distribution and the split entropies.

9.3 Experiments

To evaluate our method, we incorporate our cost-sensitive criteria (9.2.4) and (9.2.6) into a C4.5 decision tree. The C4.5 algorithm builds the decision tree in the same manner as ID3, but incorporates several post-processing heuristics, including a pruning method that removes statistically insignificant leaf nodes after the tree is built. We found that C4.5 yielded trees with sig-

Author	Proposed Criteria
Nunez [81]	$C(X^i, f) = \frac{2^{Gain(X,f)} - 1}{(Cost(f) + 1)^\gamma}$
Norton [80]	$C(X^i, f) = \frac{Gain(X^i, f)}{Cost(f)^\gamma}$
Tan [99]	$C(X^i, f) = Gain(X^i, f) - \gamma \cdot cost(f)$

Table 9.1: Existing cost-sensitive decision tree building criterion. The method of Tan and Norton have been generalized to incorporate varying cost/accuracy tradeoffs (through the γ parameter).

nificantly higher accuracy than ID3.

We compare our criteria to three existing methods, given in table 9.1. Nunez [81] proposes a cost-sensitive criteria called the information cost function, which is motivated using an economic argument. Tan [99] proposes a method that is similar to our $CSGain$ criteria. However, this method does not normalize the $Gain$ function. We will see that this normalization results in significantly better results. The criteria given in table 9.1 generalizes Tan's method by adding a cost/accuracy tradeoff parameter γ to the second term. Norton [80] uses a cost-sensitive criteria in his proposed IDX algorithm. We also generalize this algorithm to account for varying cost/accuracy tradeoffs. We note that since the Tan method incorporates the cost factor using an additive term, we have incorporated the cost/accuracy tradeoff parameter γ as a multiplicative factor. The Norton method incorporates costs using a multiplicative factor, so we use an exponential to adjust this tradeoff.

We present our results in terms of cost ratio, which we define as the sum of the costs of the features in the cost-sensitive decision tree, divided by the

total cost of the features in the cost-insensitive tree. We compare our method against existing methods given in table 9.1 using eighteen datasets from the UCI repository [37].

For each dataset, we perform 50 trials of the following test. First, we randomly generate costs for each feature in the dataset from a uniform distribution on [0,1]. Second, for each of our algorithms and for each of the 3 existing algorithms, we identify the value of γ that produces the cheapest tree and that also has a 10-fold cross-validated accuracy within 1% of the baseline, cross-validated cost-insensitive C4.5 tree. We use several values of γ ranging from 10^{-6} to 10^{6}. For each algorithm, we then compute the average cost ratio across all 50 trials. Table 9.2 shows these average ratios for all 5 algorithms.

Our methods outperform the three existing heuristics on sixteen of the eighteen datasets. Furthermore, the methods of Tan and Norton have average costs that are higher than the baseline, cost-insensitive trees. In particular, for the colic dataset, the Norton method has an average cost that is over six times that of the baseline, cost-insensitive tree. However, the Norton method is also the only other method to build cheaper trees than our methods, doing so in two of the eighteen datasets. The poor average performance of the Norton method is a result of its inconsistency.

Among our algorithms, performance is quite similar, with the cost-sensitive information gain building the cheapest trees in eight of the datasets, and the cost-sensitive gain ratio building the cheapest trees in nine of the datasets (there is one tie). We will note that for one of the datasets (iris),

	Cost Ratios				
Dataset	# CSGain	CS Gain Ratio	Nunez	Tan	Norton
anneal	0.583	0.592	0.614	0.646	0.994
audiology	0.964	0.980	0.991	5.650	5.650
balance-scale	0.988	0.988	1.000	1.000	1.000
breast-w	0.647	0.671	0.917	1.106	0.970
colic	0.749	0.712	0.783	3.207	6.460
credit-a	0.394	0.374	0.557	1.015	0.111
dermatology	0.653	0.723	0.611	3.108	2.281
diabetes	0.498	0.541	0.961	0.973	1.123
ecoli	0.908	0.884	0.938	1.011	1.207
hepatitis	0.474	0.417	0.558	1.522	0.536
ionosphere	0.309	0.337	0.490	1.078	0.392
iris	1.366	1.328	1.371	2.153	3.085
labor	0.527	0.503	0.587	1.135	0.487
liver-disorders	0.976	0.972	0.997	1.008	1.013
lung-cancer	0.447	0.456	0.513	15.097	0.464
soybean	0.992	0.953	0.980	1.666	1.712
vehicle	0.653	0.790	0.862	0.936	1.051
zoo	0.524	0.507	0.606	1.045	0.542
average	0.704	0.718	0.765	2.070	1.390

Table 9.2: Average cost ratio for our methods (CSGain and CS Gain Ratio) compared to existing methods. The cost ratio is the cost of the cost-sensitive decision tree normalized by the cost of the baseline, cost-insensitive tree. For a given level of accuracy, trees constructed with the cost-sensitive information gain and cost-sensitive gain ratio criterion tend to build much cheaper trees than existing methods.

all cost-sensitive classifiers resulted in more expensive trees than the cost-insensitive tree. Of course, if such a case arises in practice, one could just use the cost-insensitive tree.

Table 9.2 gives the average costs for trees with accuracy that is within 1% of the baseline accuracy. We found that our algorithms outperform the three baseline methods for other levels of accuracy as well.

9.4 Clarify: forensic classification of confusing software error behavior

In this section, we apply our cost-sensitive decision tree algorithm to the Clarify system that was presented in Chapter 8. Clarify's features are abstractions or *representations* of program control flow, and its classes are error behaviors that are ambiguous or misleading to a program's users. Clarify classifies program error behavior to produce more informative error messages. Clarify monitors various types of program control flow by static analysis and modification of the program binary. When a program produces an error, Clarify uses a classifier to predict the cause of the error from these monitored system forensics. C4.5 decision trees empirically perform very well in this domain.

As a testbed for the Clarify system, we use seven different benchmarks based on the following large, mature programs: `latex` (a typesetting program), `gcc` (GNU C compiler), `mpg321` (mp3 player), `Microsoft Visual FoxPro` (a commercial database management program), `lynx` (a text-based web browser), `apache` (a web server), and `gzprintf` (a function from the popular `zlib` compression library that has a known buffer overflow). For each benchmark, we identified program errors with nondescript, ambiguous, or misleading error handling. For example, such errors include `mpg321` emitting garbled audio resulting from corrupted audio file input—no message is given to the user that any problem has occurred. Benchmarks have 3 (`lynx`) to 9 (`latex`) distinct error cases with 30 (`FoxPro`) to 1,024 (`apache`) instances per error. Table 9.3

	feature counts by type				
Benchmark	FC	CTP-D1	CTP-D2	WEPD	total # features
latex	533	2,517	7,369	10,419	20,838
gcc	2,293	11,164	37,168	50,625	101,250
gzprintf	104	111	183	398	796
apache	605	865	1,584	3,054	6,108
FoxPro	4,724	16,493	42,175	63,392	126,784
lynx	703	2,332	4,366	7,401	14,802
mpg321	272	547	986	1,805	3,610

Table 9.3: Summary of each of the seven benchmarks tested with the Clarify system. Clarify encodes features using several different methods (function counting (FC), call-tree profiling (CTP), and event peak detection (WEPD). The total number of features is quite high and monitoring all features would result in high computational overhead costs.

gives a summary of each of the seven datasets. In the next subsection, we describe the different feature construction methods used.

9.4.1 Feature construction

Clarify uses abstractions of program control flow to monitor program behavior. In addition to the representations described in the previous chapter, we also introduce a new representation method, windowed event peak detection (WEPD) to monitor time-series information about program behavior. WEPD is a technique that captures some time-series information for any program behavior representation, such as CTP or FC. WEPD tracks changes in the frequency of each monitored event, keeping a count of the number of *event peaks*—a period of time in which an event is triggered with relatively high frequency. WEPD computes these peaks by measuring the frequency of each event within a pre-specified time window. Each peak is defined as a local

maximum of the frequency. At a given time t and window size n, the algorithm computes the frequency of a given event to be the number of times the event occurs from time $(t - n + 1)$ to time t; peaks are detected by monitoring the change in frequency. We found a window size of 1,000 events to work well across all benchmarks.

9.4.2 Minimizing overhead costs

The instrumentation inserted into applications to gather data for the Clarify decision tree classifier can have significant computational costs. If Clarify monitored all features it could monitor, the computational overhead of the Clarify system would be high. In a real-world deployment of the Clarify system, users may not be willing to sacrifice too much performance for better error messages. One way to reduce Clarify's computational overhead is to instrument only those features tested in the decision tree. By employing cost-sensitive learning, the amount of instrumentation needed can be reduced even further. Since program instrumentation points must be chosen before the program is executed (i.e. not during prediction), the Clarify classification problem is a forensic problem and is thus well-suited for our algorithm. Feature costs vary greatly in this problem domain: features corresponding to frequently executed functions incur overheads many times larger than features corresponding to rarely called functions.

9.4.3 Cost functions for program instrumentation

As discussed, the feature construction process works by instrumenting the program at a particular granularity, and recording the feature counts as the program executes. All abstractions of behavior representations incur computational overheads proportional to the amount of instrumentation. For the purposes of classification, a decision tree considers a very small portion of the entire feature set. By instrumenting only those features used by the decision tree, the amount of program instrumentation and overhead costs is greatly reduced.

For function counting, instrumentation points are needed only at functions that correspond to nodes in the decision tree. To record function counts, an array of counters is used to track execution for each instrumented function. Let G be the set of monitored functions, and let $E[g]$ be the expected number of times a function g is called in a program's execution. Note that these expectations can be computed from the training set. Then $\sum_{g \in G} E[g]$ gives the expected number of instrumented events per program execution, and will be proportional to overhead cost.

In call-tree profiling (CTP), instrumentation code at the start of each function records function call subtrees. Hence, the cost model counts the execution of all functions that appear within any CTP feature. Given a set of CTP subtrees over a set of functions F, we approximate the overhead cost of instrumenting these subtrees as $\sum_{f \in F} E[f]$. Once a function is part of a CTP feature, including it in a different CTP feature does not add significant

overhead. Therefore, the cost of each feature must be computed in the context of the features that have already been added to the tree at an earlier stage of the algorithm.

The overhead cost of monitoring with event peak detection (WEPD) is minimal. Therefore, our cost model assumes that if a function has already been monitored, any associated WEPD counts can be collected at no additional cost.

9.4.4 Results

In subsection 9.3, we provided strong evidence that our methods build cheaper trees than existing methods. Here, the datasets are much larger (i.e. `gcc` has over 100,000 features) and the computational time needed to train is thus much longer. Instead of providing a comparison of all algorithms, we focus only on our cost-sensitive information gain criteria. We demonstrate that C4.5, modified to use this criteria, can give excellent results for the application at hand.

Figure 9.1 shows the cost/accuracy tradeoff for the `gcc` benchmark. As a baseline, the cost of the trees built using the two best existing methods (as quantified in subsection 9.3) are also plotted. This curve is generated using five-fold cross validation to estimate the classification accuracy of the cost-sensitive decision tree for various values of γ. Among this set of (cost, accuracy) pairs, pareto optimal points are identified to generate the cost/accuracy curve. Since the absolute overhead slowdown is a function of program run-

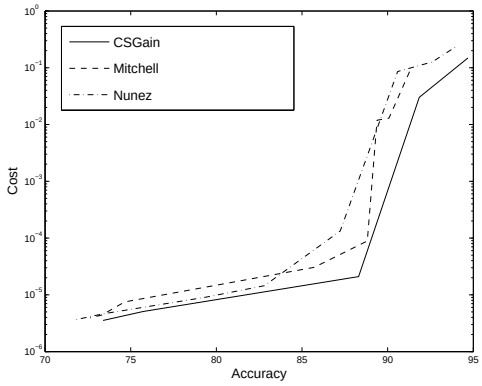


Figure 9.1: Cost/accuracy tradeoff for the `gcc` benchmark. Instrumentation costs can be significantly reduced at only a slight cost of decreasing accuracy.

ning time (which varies greatly from benchmark to benchmark), the costs here are normalized by the total instrumentation slowdown incurred if all available features were instrumented. For example, a cost of .1 corresponds to instrumenting an average of 10% of all function calls in an execution of a program.

Table 9.4 gives decision tree costs for several benchmarks when trained using the baseline, cost-insensitive C4.5 algorithm (using FC and CTP representations), and also when trained using C4.5 with the CSGain criteria (9.2.4) with FC and CTP, and also with the addition of WEPD. This improvement is measured as the cost of the tree divided by the cost of the baseline, cost-insensitive tree (note that this is the inverse of the cost ratio term used in subsection 9.3). For the cost-sensitive algorithms, results are given for trees with accuracy levels that are no less than 1% lower than the cross validated accuracy of the baseline cost-insensitive classifier trained with FC and CTP representation. Our cost-sensitive algorithm yields reduction in execution

Benchmark	Baseline	CSGain w/FC, CTP	CSGain w/FC, CTP, and WEPD
`mpg321`	19.4%	158.3×	188,118.8×
`gcc`	24.2%	1.8×	2.2×
`gzprintf`	20.1%	1.7×	4.7×
`latex`	44.0%	468.1×	244.4×
`foxpro`	3.7%	1,485,943.7×	90,024.3×
`lynx`	1.9%	552.3×	552.3×
`apache`	8.9%	4,684.2×	4,684.2×

Table 9.4: The Baseline column gives the decision tree cost when built with the baseline C4.5 algorithm, using CTP, expressed as a percentage of the total cost of instrumenting all features. The remaining columns provide the speedup ratio (defined as baseline cost / cost) for C4.5 using the cost-sensitive gain criteria (CSGain), using CTP and also using WEPD with CTP. Costs are for trees with accuracy reductions of at most 1% from baseline.

of instrumentation points of up to six orders of magnitude. Although the WEPD representation does not yield better performance for all benchmarks, it gives significant improvement on the two benchmarks that saw only marginal speedup with FC and CTP (`gcc` and `gzprintf`).

9.4.5 Related work

Building classifiers that minimize testing costs has received much attention in the field of medical diagnosis. However, the problem of medical diagnosis is fundamentally different from the forensic classification problem. Our cost model falls under Turney's "Constant Test Cost" case [101], where the cost of testing is independent of the instance. Several cost-sensitive modifications to the information gain criteria have been proposed and tested [77, 80, 81, 99, 100]. In section 9.3, we provided details of these existing methods, and show that our cost-sensitive information gain and gain ratio criteria

result in significantly cheaper trees.

Several cost-sensitive algorithms have been proposed that build decision trees using non-incremental methods, such as a genetic algorithm [100], a "look ahead" heuristic [80], and a heuristic search procedure, AO* [111]. These methods are not considered here, as the training time required is several orders of magnitude larger than a C4.5 based incremental algorithm. In particular, the genetic algorithm proposed in [100] takes an average of 1000 times longer to train than C4.5 For the case study presented in the next section, this would result in training times on the order of weeks.

In this chapter, we have focused on the problem of minimizing test cost while maximizing accuracy. In some settings, it is more appropriate to minimize misclassification costs instead of maximizing accuracy. For the two class problem, Elkan [41] gives a method to minimize misclassification costs given classification probability estimates. Bradford et al. compare pruning algorithms to minimize misclassification costs [14]. As both of these methods act independently of the decision tree growing process, they can be incorporated with our algorithms (although we leave this as future work).

Ling et. al. propose a cost-sensitive decision tree algorithm that optimizes both accuracy and cost. However, the cost insensitive version of their algorithm (i.e. the algorithm run if all feature costs are zero), reduces to a splitting criteria that maximizes accuracy, which is well known to be inferior to the information gain and gain ratio criterion [77, 84]. Consequently, when performing our experiments in section 9.3, for several datasets, we were unable

to generate trees using their method that were within 1% of the cross-validated accuracy of the baseline C4.5 algorithm.

Integrating machine learning with program understanding is an active area of current reasearch. Systems that analyze root cause errors in distributed systems [8, 19, 24] and systems that find bugs using dynamic predicates [17, 50, 67] may both benefit from cost-sensitive learning to decrease overhead monitoring costs.

Chapter 10

Improved Software Unit Test Ordering via Metric Learning

This chapter describes an algorithm for unit test prioritization [30] that uses clustering, nearest neighbor searches, along with a metric learning component, to efficiently search and execute large unit test suites.

10.1 Introduction

Unit tests are small functional software tests that verify correctness of a particular software component. These tests are designed as a check for developers to determine if he or she has introduced a bug during the course of development. Popular unit testing environments such as Java's Junit testing framework permit developers to easily run unit tests after code changes have been made, but before the changes are made public by checking into a source code repository. Frequent unit testing allows developers to identify errors immediately after they have been introduced, before the code is ever committed to a code repository. Many companies qualify each code change against a set of unit tests before allowing a code check-in to complete.

As software projects grow in size, so do the number of unit tests. For

example, the Apache commons collections library [68] has over 12,000 unit tests. Similarly, the unit test suite associated with the Lucene open source search engine [42] takes over 9 minutes to complete. Large unit tests suites are necessary to ensure adequate testing coverage and to reduce program errors. However, large test suites do not necessarily identify newly introduced bugs quickly, and the long time it takes to run the suites make unit tests an inefficient part of code development.

This chapter addresses the problem of test ordering. Code check-ins are usually localized to a relatively small set of changes within the code base. The goal of test ordering is to predict which unit tests are more likely to fail given a set of changes to the code. Intuitively, changes that are confined to a particular program class or module may not affect the correctness of tests in unrelated sections of the program. For example, if an engineer makes a set of changes to a bi-directional hash map class in the Apache commons collections library, it is unlikely that these changes will affect tests that check sorting properties for the library's tree-based data structures. An effective ordering can expose failing tests more quickly than standard ad-hoc orderings, enabling engineers to immediately investigate, understand, and fix bugs that were recently introduced.

Developing an effective test ordering strategy presents several challenges. Popular object-oriented programming languages, such as Java, support highly factored code that uses inheritance extensively. Changes made to abstract classes with many layers of sub-classes can impact correctness for

software components throughout several layers of a system. For example, the Apache commons collections library has 19 sub-classes and four levels of inheritance. Complicating matters, good programming practice promotes code reuse, yet resuing software components often results in the violation of such abstraction boundaries. For example, it is not unusualy for utility and other library functions to be called from varied components throughout a large system. Consequently, code changes that are highly localized to a specific system component may cause failures in seemingly unrelated components.

This chapter proposes a statistical test ordering method that leverages state-of-the-art machine learning algorithms. Our methods represent each unit test via a vector representation that measures counts of each method or basic block from a particular execution. This representation allows us to efficiently compute the similarity (or difference) between two tests in a programming language-independent manner. This is in contrast to existing approaches that rely on language-specific correctness models, which generally only operate under strict language assumptions and experimentally have not been shown to scale to large modern programs [4, 52]. Finally, almost all of the computation required by our algorithm can be performed offline, enabling our test selection process to select test orderings with little to no computational overhead.

Our algorithm consists of two components that work by utilizing an unsupervised learning approach to cluster tests, followed by nearest neighbor algorithms to quickly search and execute the space of possible failing tests. The clustering step groups the set of all unit tests by their similarity. The

purpose of the clustering step is to divide the space of unit tests into smaller groups so that the initial set of executed tests are fairly diverse, testing a variety of possible bug types. Once a single failing test is found, there often exist other similar nearby failing tests. For example, if the `put` functionality for a hash map is broken, many tests in the hash map test suite will fail. To take advantage of this behavior, the second component of the algorithm is a nearest neighbor search that quickly locates additional failing tests once an initial failure has been discovered.

Underlying both the nearest neighbor component and the clustering component is the distance or similarity function used to measured the similarity between unit tests. To further improve the quality of our algorithm for a specific program or benchmark, we employ distance metric learning algorithms to fine-tune domain models of test execution similarity. Standard distance measures such as the Euclidean distance between two vectors give equal weighting to each feature (i.e. method or basic block) being compared. Metric learning utilizes information from previously failing runs to learn program-specific metrics that give more or less weight to appropriate features.

Experimentally, we compare our approach to existing heuristics with respect to several metrics across three large and mature Java benchmarks: the Apache commons collections library, the Lucene search engine, and the Hibernate framework. Overall, we show that our methods can greatly reduce the required number of unit tests executed, providing reductions in the number of executed tests by up to 99.2%, with an average reduction of 76%. We show

that across all criteria, including the time required to discover an initial failure and time required to discover all failing tests, our methods are highly efficient for ordering unit tests.

10.2 Software Unit Testing

Figure 10.1 shows a simplified version of the typical software development cycle. The cycle starts at the left with the developer checking a bug database or feature request list. In the second step of the cycle (on top), the developer makes appropriate changes to the code base in order to implement the required features, bug fixes, etc. Once changes have been made, the final step is the code check-in process (on right). A standard component of modern quality assurance is that of validating code changes via small, functional tests commonly referred to as unit tests. Many companies or projects require that all code changes must be verified against a set of unit tests *before* the changes are checked into the code repository. Such measures are quite effective in reducing the number of bugs introduced to the software system.

Unfortunately, as software repositories grow in size, so do the number of unit tests, and large testing suites can take a long time to execute. Unit testing bogs down the development cycle as engineers wait for the tests to verify a lack of regessions before code is checked in. The goal of unit test ordering is to develop a policy which orders unit tests to prioritize those that are most likely to fail. If a bug has been introduced to the code base, having a unit test fail at the start of testing will make an engineer aware of the problem

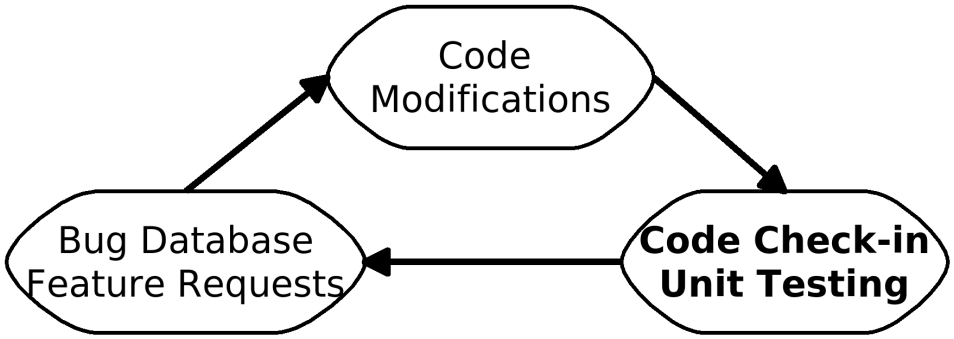


Figure 10.1: A simplified view of the software development cycle. Many software projects and their associated software repositories have a very large number of unit tests. Running these tests poses a bottleneck to this cycle.

quickly, allowing him or her to resolve issues in a more streamlined manner.

In motivating the problem of test ordering, let us first consider the shortcomings of a few simple heuristics. Consider the `put` method from the Apache commons collections bi-directional hash map class. Apache commons collections [68] is a relatively large open source software project that offers additional data structures over what is provided by the standard Java development kit. A bi-directional hash map is a map in which objects can be retrieved by either the key of the map or by the value. The specification for the bi-directional hash map requires that the `put` operation returns the key object inserted into the map. This is a possible point of confusion, however, since the bi-directional hashmap is a symmetric data structure. Consider the bug shown in Figure 10.2 where the value object returned (incorrectly) instead of the key object. The commons collections library has over 12,000 unit tests, and this bug causes eight of them to fail.

```
public Object put(Object key, Object value) {
    if (maps[0].containsKey(key)) {
        maps[1].remove(maps[0].get(key));
    }
    if (maps[1].containsKey(value)) {
        maps[0].remove(maps[1].get(value));
    }
    final Object keyObj = maps[0].put(key, value);
    final Object valObj = maps[1].put(value, key);
    // BUG! keyObj should be returned
    return valObj;
}
```

Figure 10.2: An example bug injected into the Apache Commons Collections library. The value object is incorrectly returned when inserting into a bi-directional hash map. The Commons Collection library has over 12,000 unit tests, and this change causes 8 of them to fail.

In this example, the set of changes to the code base to introduce the bug are quite minor and are restricted to only a single method. One possible ordering heuristic is to restrict testing to only those tests that exercise changed functionality. Tests that do not call the `put` method will be unaffected by this change and can be safely ignored. While this heuristic will reduce the set of tests considerably for rarely called methods, it is not as effective for core methods. In particular, the `put` method is called by over 1,500 unit tests.

Unit tests are typically organized by "suites"—a set of tests that are related to a particular class or namespace. If code changes are entirely contained within a single class or restricted to a set of methods within a single test suite, another heuristic is to confine testing only to relevant suites. This approach is limited in that a set of changes applied to a particular class or

module can result in incorrect functionality in other modules. For example, the `put` method considered here results in failing runs across three different unit test suites, because it is used outside of the module in which it is defined.

10.3 Test Ordering

In this section, we describe our approach to the test ordering problem. Before doing so, we first consider a set of goals and objectives. One of the primary motivations for the test ordering problem is to reduce unit testing time for *large* software repositories. Therefore, an effective test ordering method must be able to scale to handle thousands or more tests over repositories with hundreds of thousands or more lines of code. A second consideration is platform independence and an avoidance of strong programming language assumptions. Large modern software projects are often times composed of components written in various languages. Further, a test ordering policy should not rely on specific language assumptions. For example, modern enterprise Java programming relies heavily on the use of reflection through open source packages such as the Hibernate framework [55]. Dynamic code generation and the use of reflection present problems for static-analysis-based methods.

Figure 10.3 shows the workflow for our test ordering framework. Our test ordering algorithm represents tests by instrumenting the unmodified code base, measuring run-time behavior, and storing relevant statistics via *program feature vectors*. These vectors are computed before any code modifications are

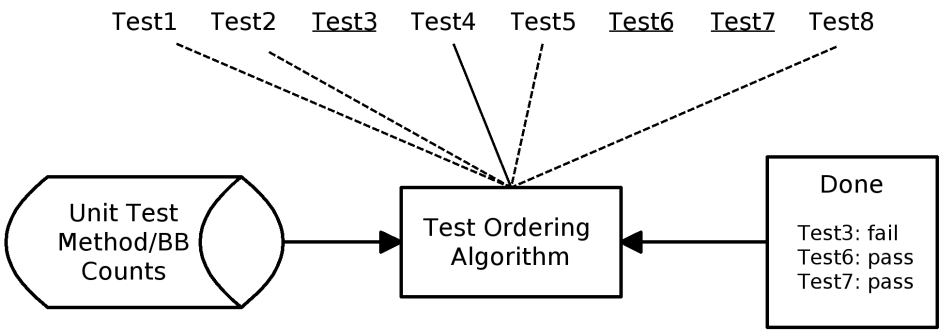


Figure 10.3: Our test ordering algorithm chooses the most effective test to run from a set of unit tests. Offline, all unit tests are executed on an instrumented version of the code present in a respository, and the results are stored in the BB Counts database (on left). A developer then modifies the code and before checking in, needs to order unit tests to find problems quickly. Our algorithm does the ordering, choosing among all tests with dotted lines (highlighting the current choice of Test4). Tests that have run are underlined and appear in the Completed report on the right. In addition to unit test vectors, the algorithm also incorporates information regarding which tests have passed and which have failed.

made in an offline training phase and are stored in a test vector database. In this paper, our program feature vectors contain either method or basic block counts; the length of the vector is equal to the number of methods (basic blocks), and value of the i^{th} position in the vector is equal to the number of times the method (basic block) i is executed for the given unit test. When code is modified, our test ordering algorithm determines test execution order by analyzing the vectors stored in the database. Furthermore, tests are executed sequentially, and our test ordering algorithm also exploits information about the failures or successes of completed tests.

10.3.1 Statistically Motivated Ordering Algorithm

Our test ordering algorithm analyzes unit test run-time behavior. We describe two such features that can be used by our system, method counting and basic block counting. Method counting works by instrumenting each method in a program with a counter. When the method is entered, the counter is incremented, and each unit test is thus represented by the set of methods exercised in the test along with the number of times each of these methods is called. A basic block is defined as a set of instructions with a single entry point, one exit point, and no jump instructions contained within the set. Basic blocks are typically the basic unit to which compiler optimizations are applied, and are the nodes of a control flow graph. Basic blocks are finer-grained than methods, and monitoring tests at the basic block level represents a finer resolution and provides richer representations. Our methods can incorporate any

type of program feature, including program paths or method call-sites.

Our algorithm takes a statistical approach in analyzing these program features vectors to create test orderings. Given a set of d program features, $f_1, \ldots, f_d$, each unit test is represented as a d-dimensional program feature vector $\boldsymbol{x}$ where entry j of this vector corresponds to the count for the j^{th} feature. Underlying all of our algorithms is a distance function used to compare two tests. One such measure is the squared Euclidean distance:

$$d(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{d} (\boldsymbol{x}_i - \boldsymbol{y}_i)^2 . \tag{10.3.1}$$

As we will see, distance measures are used throughout our methods, and our algorithm largely relies on the fact that this distance reflects the domain-specific notion of similarity for the program at hand.

Our algorithm is designed to optimize two different metrics: time to first failure, and time to all failures. Time to first failure measures the number of tests run until the first failing test is executed. In terms of the software development cycle outlined in Figure 10.1, a small time to first failure allows an engineer to immediately realize a bug has been introduced. If relatively minor changes have been made to the code base, the engineer may be able to immediately troubleshoot the problem without the need to fully investigate the failing unit test. For more complex bugs, having a larger set of bug symptoms—i.e. identifying multiple failing unit tests—may be helpful in troubleshooting the problem. Here, our second criteria, time to all failures, measures how many tests need to be run before all failing tests are discovered.

Algorithm 4 Test ordering via clustering and nearest neighbor searches

Require: $X = \{\boldsymbol{x}_1, \ldots \boldsymbol{x}_n\}$: a set of d-dimensional vectors representing each of n unit tests

```
1: Cluster X into k clusters C_1, ..., C_k using the k-means algorithm
2: current_cluster ← 0
3: mode ← cluster
4: repeat
5:   if mode = cluster then
6:     current_cluster ← current_cluster mod k
7:     x ← random instance from Cluster C_current_cluster
8:     result ← Execute x
9:     if result = fail then
10:      mode ← nearest_neighbor
11:    end if
12:    current_cluster ← current_cluster + 1
13:  else if mode = nearest_neighbor then
14:    {x_i1, ..., x_ic} ← c nearest neighbors within cluster C_current_cluster to
       last failing test
15:    nn_failures ← false
16:    for all x ∈ {x_i1, ..., x_ic} do
17:      result ← Execute x
18:      if result = fail then
19:        nn_failures ← true
20:        break
21:      end if
22:    end for
23:    if nn_failures = false then
24:      mode ← cluster
25:    end if
26:  end if
27: until All tests have been executed
```

Our method is given in Algorithm 4 and consists of two distinct components: a clustering component, and a nearest neighbor component. The clustering component works by forming groups of similar instances. Clusters are formed using the k-means algorithm [**?**]. The assumption here is that tests within each cluster will tend to have similar failing behavior. That is, if a single test in the cluster fails, then others within the cluster are also likely to fail. The clustering stage of the algorithm works by iterating through each cluster, and selecting and running a test randomly from each cluster. This procedure ensures that the initial set of executed tests are sufficiently diverse. In practice, cluster sizes tend to have a large variation, as some clusters may have only a few tests while others have hundreds or more. Because of this, clustering is much more effective in finding initial failing tests than a policy in which tests are chosen uniformly at random without clustering. The clustering component can be viewed as one that optimizes the time to first failure, as its exploratory nature is well suited for quickly finding a single failing run. In section 10.6, we provide experimental evidence supporting our claim.

Once the clustering-based sampling process discovers a failing test, the algorithm then performs a nearest neighbor search to identify additional tests that may fail. The purpose of the nearest neighbor component is to find additional failing runs that are similar to ones discovered during the clustering step. One option is to randomly run additional tests from within the failing test's cluster. However, the nearest neighbor component provides a more accurate means of identifying similar unit tests. The component works by

identifying the set of c nearest neighbors to the failing test, and each of these tests are then executed. When a single test fails, other similar tests are also likely to fail, and the nearest neighbor component is well-suited for finding all failing runs, a quality that minimizes the time to all failures. Much like the k-means clustering component, the nearest neighbor component also compares examples using the Euclidean distance.

10.3.2 Algorithm Visualization

To provide additional intuition behind our method, Figure 10.4 visualizes method counts for a set of unit tests in the Apache commons collections library. The visualization is achieved through the use of principal components analysis (PCA) [**?**]. Here, the original space of over four thousand features (i.e. method counts) are projected into two dimensions. PCA is well-suited for visualizing the relative distances between pairs of points. The values of the PCA components (i.e., each of the two dimensions in Figure 10.4) do not have well-defined meanings in terms of the original method or basic block counts, and we omit axes labels to stress this. Failing instances are shown with a red 'x', while non-failing instances are denoted by blue dots.

The set of tests shown in Figure 10.4 group naturally into several clusters. Furthermore, we can see that most failures are localized within a handful of clusters. Consider Algorithm 4 in the context of this data set. Initially, the data is clustered into a set of k clusters, which would include several elongated clusters on the far right hand side, many clusters towards the center,

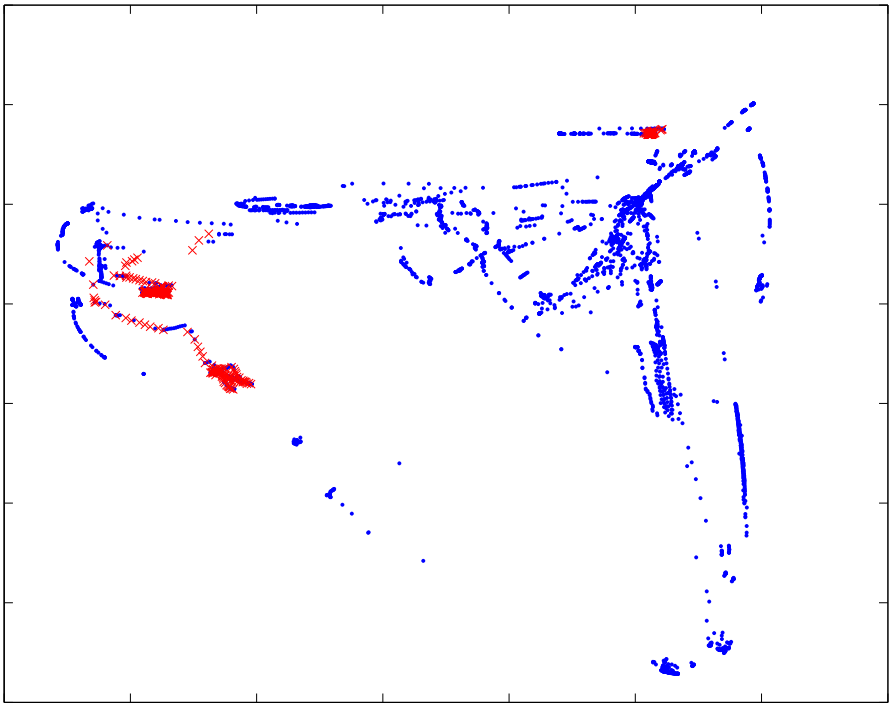


Figure 10.4: Visualization of unit tests in the Apache Commons Collections library. Failing instances are shown as red x's. This figure shows that unit tests tend to cluster well, and that failing tests tend to be highly localized.

and several more elongated clusters stretching towards the left of the figure. The algorithm then iteratively chooses a random instance from each cluster until a failing test is discovered. Assume that this first failing test is found within the cluster on the top right-hand side of the plot. As we can see in line 9 of the algorithm, once this failing run is executed, the algorithm switches to nearest neighbor mode. Here, we can see that there are several other failing instances in close proximity to one another, and the nearest neighbor mode of the algorithm will quickly discover these similar executions. Once the remaining failing executions are discovered in this cluster, the algorithm will then return back to its exploratory clustering mode in line 24, and will continue running tests from other clusters until one of the failing tests on the left side of the plot are discovered.

While some of the clusters are nicely rounded in Figure 10.4, others are long and skinny. This is partly due to distortions incurred by PCA in the visualization process. Nevertheless, long and skinny clusters pose problems for the k-means clustering algorithm that uses Euclidean distance. In the next section, we consider the problem of learning distance function tailored to a specific program or benchmark, enabling more effective cluster identification and more accurate nearest neighbor searches.

10.4 Improved Distance Metrics for Comparing Tests

The nearest neighbor and clustering components used in Algorithm 4 rely on an underlying distance function. The success of the algorithm depends

on the assumption that the distance function reflects the domain-specific requirements for the problem. The standard Euclidean distance shown in equation (10.3.1) computes the difference between two points as the sum of the squares of the difference between each pair of features. A key point here is that each feature is given equal weighting in this sum, and that each component in the sum is computed independently. This is at odds with our unit testing domain, as there are certain methods that are much more likely to cause failing tests than others. For example, a `put` method in an abstract hash map class which is inherited by numerous sub-classes is much more critical than a `toString` method.

In this section, we use distance metric learning algorithms to learn a distance function specifically tailored to each individual program or benchmark. Metric learning algorithms, like any machine learning algorithm, are trained using labelled training constraints. For the unit testing domain at hand, constraints are inferred from bugs that were previously introduced into the code. Every time an engineer introduces a bug that causes one or more unit tests to fail, this information is stored for use by the metric learning algorithm. As more bugs are introduced, more training data is available for the algorithm, resulting in potentially better distance functions.

The metric learning algorithm trains on constraints that relate pairs of objects (i.e. unit tests): two tests can be constrained to be similar (i.e. the distance between them is smaller), or two tests can be constrained to be dissimilar (the distance between them is larger). Recall that in motivating

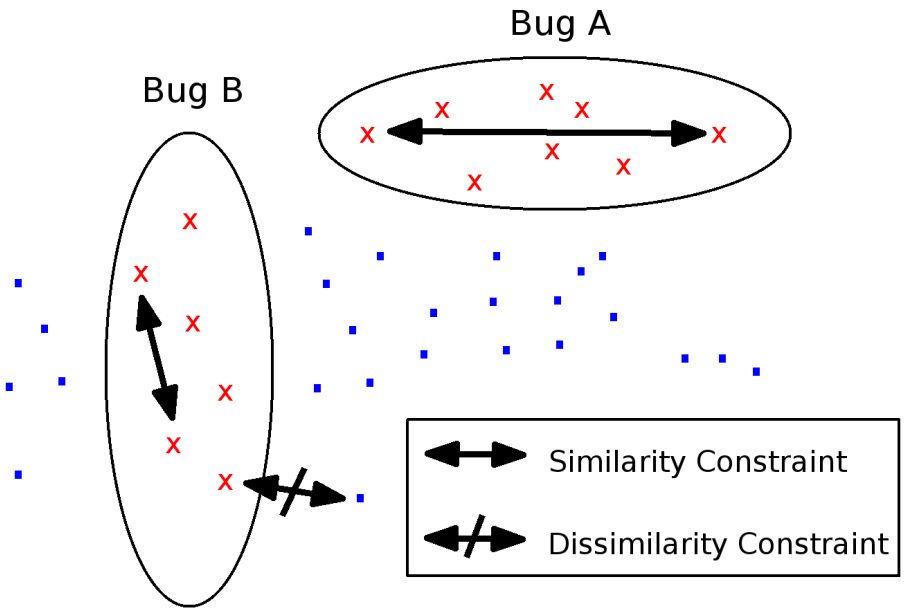


Figure 10.5: Our distance metric learning component learns a program-specific distance function by analyzing test failing behavior of previously realized bugs. Here, red x's denote failing tests, and blue dots represent tests that succeeded. Two tests that both failed as a result of the same bug are constrained similar. Conversely, tests are constrained dissimilar if one test failed and another succeeded for a given bug.

Algorithm 4, we assumed that failing instances are quite similar. We can see this in Figure 10.4, as most of the failing tests are grouped together relatively tightly and have small distances between them. Ensuring small distances between failing tests enables the nearest neighbor component to locate additional failing tests once a single failing tests has been identified. Conversely, Algorithm 4 also assumes that the distance between failing and non-failing tests will generally be larger. This aids the clustering algorithm in finding a good set of clusters from which tests can be efficiently explored and sampled.

We infer training constraints from introduced bugs as shown in Figure 10.5. To form similarity constraints, pairs of tests that both failed as result of the same bug are constrained similar. Figure 10.5 shows two such similarity constraints between pairs of failing tests for two different bugs, Bug A and Bug B. Conversely, we form dissimilarity constraints by selecting a test that failed and one that succeeded as a result of the same bug. In Figure 10.5, we see one such dissimilarity constraint between a failing test for Bug B and another test that succeeded when Bug B was present.

Constraints are generated for all previously realized bugs, and the resulting constraint set will enforce the fact that tests that fail together because of the same bug (i.e. tests that fail as a result of the same set of code changes) will tend to have smaller distances than pairs of tests in which one test passed and the other failed.

The metric learning algorithm we use is based on the Information Theoretic Metric Learning (ITML) algorithm [31]. Experimentally, ITML has been shown to yield very good results in various applications, including statistical software analysis domains [29, 31]. ITML works by learning a Mahalanobis distance function. This function generalizes the standard squared Euclidean distance and is parametrized by a $(d \times d)$ matrix A. The Mahalanobis distance between $\boldsymbol{x}$ and $\boldsymbol{y}$ is given by

$$\begin{aligned} d_A(\boldsymbol{x}, \boldsymbol{y}) &= \sum_{i=1}^{d} \sum_{j=1}^{d} A_{ij}(x_i - y_i)^2 \\ &= (\boldsymbol{x} - \boldsymbol{y})^T A (\boldsymbol{x} - \boldsymbol{y}). \end{aligned} \tag{10.4.1}$$

The distance reduces to the squared Euclidean distance when A is the identity matrix.

ITML learns a distance metric from a set of pairs of examples that are constrained to be either similar or dissimilar. If two examples are constrained similar, then their resulting distance should be somewhat small. Examples that are constrained dissimilar should result in distances which are somewhat larger. Given a set of similarity constraints S and dissimilar constraints D, ITML optimizes a Mahalanobis distance parametrized by A:

$$\begin{aligned} \min_{A} \quad & D_{\ell \mathrm{d}}(A|I) \\ \text{subject to} \quad & d_A(\boldsymbol{x}, \boldsymbol{y}) \leq u \qquad (i,j) \in S, \\ & d_A(\boldsymbol{x}, \boldsymbol{y}) \geq \ell \qquad (i,j) \in D. \end{aligned} \tag{10.4.2}$$

The objective function used by ITML is the *log-determinant* (LogDet) divergence and which is shown to have maximum entropy properties [31]. The similarity constraints S constrain the learned distance between two points $\boldsymbol{x}$ and $\boldsymbol{y}$ to be at must some upper bound u. The dissimilar constraints D constrain two have distance at least some lower bound ℓ. Details on how to determine the parameters u and ℓ can be found in other work [31]. One drawback of ITML is the fact that the learned matrix A is $(d \times d)$, which requires estimating a number of parameters which is quadratic in the number of features. Large, modern programs have upwards of thousands of methods or tens of thousands of basic blocks. Estimating a Mahalanobis matrix for such programs would require optimizing millions of parameters, a difficult task for

any machine learning or statistical method. In the next section, we present a recently proposed variant of ITML which learns a Mahalanobis distance parametrized by a linear number of features.

10.4.1 Learning Metrics for Software Tests

Here, we present technical details of our metric learning algorithm. One challenge with learning Mahalanobis distances in domains where the number of features is large is the quadratic number of values needed to parametrize the distance function d_A as shown in equation (10.4.1). It is not unusual for programs to have upwards of thousands of methods or tens of thousands of basic blocks. In such a case, learning a Mahalanobis distance matrix with more than a million values presents several problems. First, learning such a large matrix using existing metric learning methods is computationally very expensive and can take days or even weeks. Second, metric learning, like any machine learning method, can be viewed as a statistical inference procedure. Estimating upwards of a million parameters can require a large amount of training data (i.e. many previous bugs) for even the most robust of methods. Finally, computing the distance between two points using a Mahalanobis matrix as shown in equation (10.4.1) is quadratic in the number of features and hence cannot scale to large problems.

To overcome such difficulties, we use a recently proposed metric learning method, HD^{2}LR , which learns a Mahalanobis matrix with a relatively low number of parameters [29]. Let I be the identity matrix and let V be a $(d \times k)$

matrix. Instead of learning a full, $(d \times d)$ matrix A with $O(d^2)$ parameters, the HD^{2}LR metric learning algorithm learns a matrix of the form $I + A_\ell$, where $A_\ell = VV^T$ is a low-rank matrix. The Mahalanobis distance with $I + A_\ell$ equals

$$\begin{aligned} d_{I+A_\ell}(\boldsymbol{x}, \boldsymbol{y}) &= (\boldsymbol{x} - \boldsymbol{y})^T (I + A_\ell)(\boldsymbol{x} - \boldsymbol{y}) \\ &= (\boldsymbol{x} - \boldsymbol{y})^T (\boldsymbol{x} - \boldsymbol{y}) + (\boldsymbol{x} - \boldsymbol{y})^T A_\ell (\boldsymbol{x} - \boldsymbol{y}) \\ &= d_I(\boldsymbol{x}, \boldsymbol{y}) + d_{A_\ell}(\boldsymbol{x}, \boldsymbol{y}). \end{aligned} \tag{10.4.3}$$

Intuitively, this metric can be viewed as having two distinct components. The first component that corresponds to the identity matrix compares each feature element-wise using the standard squared Euclidean distance. The second component is a low-rank component that works by projecting data from the original d-dimensional feature space to a lower k-dimensional space. Instead of comparing features in the original space, the low-rank component computes the Euclidean distance between two tests using this lower dimensional representation. The projection matrix V is conceptually similar to standard dimensionality reduction methods such as principal components analysis (recall that PCA was used for visualization purposes in Figure 10.4). However, while PCA is a totally unsupervised technique, we learn the "best" possible matrix V given the test data.

Mathematically, the HD^{2}LR metric learning algorithm solves the following optimization problem, for similarity constraints S and dissimilarity

constraints D:

$$\begin{aligned} \min_{A} \quad & D_{\ell \mathrm{d}}(A|I) \\ \text{subject to} \quad & d_A(\boldsymbol{x}, \boldsymbol{y}) \leq u \qquad (i,j) \in S, \\ & d_A(\boldsymbol{x}, \boldsymbol{y}) \geq \ell \qquad (i,j) \in D, \\ & A = I + A_\ell. \end{aligned} \tag{10.4.4}$$

Here, the function $D_{\ell \mathrm{d}}(A|I)$ denotes the *log-determinant* (LogDet) divergence between the matrices A and I. The LogDet divergence can be viewed as a measure of distance between $(d \times d)$ positive semi-definite matrices, and is given by

$$D_{\ell \mathrm{d}}(A|I) = \mathrm{tr}(A) - \log |A| - d,$$

where $|X|$ denotes the determinant of the matrix X. The goal of problem (10.4.4) is to minimize the LogDet divergence from identity matrix I that parametrizes the squared Euclidean distance to the matrix A that parametrizes the learned Mahalanobis distance. The matrix A is found such that the given set of similarity and dissimilarity constraints are satisfied. The problem can also be modified to allow constraints to be violated by introducing slack variables (for a cost in the objective). This is useful in cases where constraints are incorrect or disagree with one another. Computationally, HD^{2}LR can be optimized using an iterative dual coordinate descent method. Details of this method can be found in [29]. Finally, the algorithm requires that the range space of the target matrix A_ℓ be determined beforehand. In this paper, we use the method of principal components analysis to parametrize this basis.

The identity component I of the HD^{2}LR method computes the difference between features in a pairwise fashion, giving each feature equal weighting. In terms of the software domain considered here in this paper, this translates into the assumption that each method (or basic block) is given equal weighting in computing the difference between two tests. In reality, however, some methods are much more important than others. For example, a heavily used method in an abstract class is generally much more critical to a program's functionality than a rarely used I/O method in a class with no descendants.

To overcome such limitations, we employee an additional metric learning step which learns a diagonal matrix so that features can be compared with different weightings. To learn such a diagonal matrix, we use a similar framework as that used by the HD^{2}LR algorithm, except that the objective function uses an L_2 (sum of squares) loss instead of the LogDet measure used in problem 10.4.4. This diagonal Mahalanobis distance function with matrix D is learned first, followed by a procedure which runs the HD^{2}LR algorithm. However, instead of learning a matrix of the form $I + A_\ell$, we modify the HD^{2}LR algorithm to instead learn a matrix of the form $D + A_\ell$. The result is a Mahalanobis distance function parametrized by a diagonal plus low-rank matrix.

10.5 Computational Issues

The goal of the test ordering problem is to quickly identify failing tests without having to run the entire test suite. If the time taken by the test or-

dering algorithm exceeds the time required to run the test suite, then there is no benefit to ordering the tests. The various components of our algorithm, in particular the metric learning component, do take more time to run than the test suites considered in the results section. In terms of absolute computational time, learning metrics for the benchmarks presented in section 10.6 took between 12 and 75 minutes, while the clustering and nearest neighbor components required 2 to 5 minutes each to complete.

Fortunately, almost all of the computation required by our methods can be run offline, before any code changes are made. Our method analyzes instrumented code collected from a baseline run of the test suite in which all tests succeed. This is performed before any code changes are made. Therefore, unit tests can be clustered offline. Further, for each test, its nearest neighbors can be pre-computed and stored in an ordered list.

The metric learning component is trained from bugs realized in the past. Given a version of the code base at some state of revision s, the metric learning component is trained on bugs that occurred in previous versions of the code r. For each such bug, the metric learning component forms similarity and dissimilarity constraints by analyzing $fail(r)$, the set of failed tests associated with revision r, and $succeed(r)$, the set of unit tests that succeeded for revision r. While the revision r analyzed by the metric learning component should be similar to s, they do not have to be identical, as the learned distance function is trained over the program features in s. Specifically, the feature weights associated with the learned metric correspond to methods or basic blocks in

revision s, which may or may not correspond to methods or basic blocks present in r. However, revision r should be sufficiently similar to s in order to capture program semantics contained in s. In section 10.6, we provide experimental evidence that a metric for revision s can be effectively learned using earlier program revisions.

Finally, since co-failure patterns can be mined offline to form the similarity and dissimilarity constraints, the distance metric learning component can be run offline as well. Thus, all three components can be run offline, reducing the required computation needed by Algorithm 4 to a set of simple look-ups to a cluster table and a set of nearest neighbor lists.

Benchmark	Classes	LOC	Methods	Tests
Commons Collections	422	111,027	4,047	12,936
Lucene Search Engine	376	47,110	2,867	739
Hibernate Framework	973	133,675	11,226	799

Figure 10.6: An overview of our three benchmarks with respect to the number of classes, lines of code (LOC), and the number of methods, basic blocks and tests. Apache commons collections has over 12,000 tests, and the Lucene test suites take over nine minutes to finish. Hibernate is unique in that it relies heavily on reflection and dynamic code generation, two practices that are challenging for static-analysis based methods.

10.6 Results

In validating our algorithms, we provide an overview of the benchmarks used, describe our methodologies, and present quantitative results for our algorithm in the context of both hand-generated and real-world bugs.

10.6.1 Benchmarks

In this section, we present experimental results of our algorithm applied to three large open source projects: the Apache commons collections library [68], the Lucene search engine [42], and Hibernate, a framework for persisting Java classes to a relational database [55]. The size of these benchmarks in terms of the number of classes, lines of code, methods, and basic blocks is outlined in Figure 10.6.

Apache commons collections is a library that provides many rich data structures that are not offered in the standard Java development kit [68]. The library offers several classes for bags (sets in which duplicates are allowed), bi-directional maps (a map in which keys can be found from values), along with a variety of other data structures. Development for this project is quite mature, with check-ins dating back to 2001. Testing for this project is also very complete, with over 12,000 unit tests.

The Lucene search engine is a high performance text search engine library [42]. Lucene's code base offers a diverse set of algorithms, including efficient data structures to map words to documents (i.e. an inverted index), fast numerical methods for performing dot product computations between query words and documents, and a highly optimized caching layer that is tightly integrated with various components of the program. Although Lucene has fewer tests than Apache commons, the tests take much longer to complete, as much of the tested search functionality is computationally quite expensive.

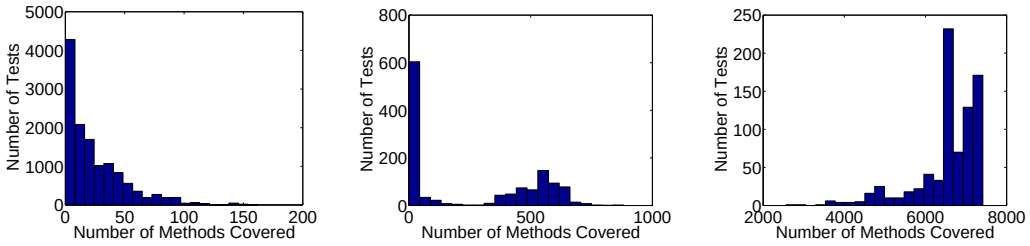


Figure 10.7: Histograms showing the number of unique methods called per unit test, for each of the three benchmarks. Overall, we see that the Commons library has relatively short unit tests, whereas Hibernate's unit tests tend to be much longer.

Finally, the Hibernate framework provides a transparent persistence layer that maps Java objects to standard relational database models such as SQL [55]. Hibernate achieves this through the use of reflection and byte-code engineering which work by modifying Java class files during program run-time. Reflection and dynamic code generation are notoriously difficult to analyze by static-analysis based methods. Hibernate also employs a sophisticated and relatively complex transactions layer which can roll back database writes in the event of conflict.

Figure 10.7 shows a histogram for the average number of unique methods called by each test. Overall, we see that unit tests for the Commons library tend to be relatively short, as over 80% of all tests cover fewer than 50 methods. Tests in the Lucene benchmark tend to be larger, with a median around 500 methods per test. Finally, the unit tests in the Hibernate framework tend to cover thousands or more methods per each test.

10.6.2 Methodology

For each of these three benchmarks, we manually introduce a variety of bugs into the code base. These bugs include relatively simple issues such as of off-by-one errors, null pointer initialization errors, and simple variable swaps (i.e. reading integer x1 instead of x2). We also introduced a variety of logical errors that are specific to each benchmark. For example, the Lucene search engine supports both conjunctive or disjunctive queries. One introduced bug involves changing a clause `BooleanClause.Occur.SHOULD` to `BooleanClause.Occur.MUST`.

We instrument these benchmarks using either method counters or basic block counters. This is achieved through static instrumentation of the Java bytecode using the BCEL package [69]. As discussed in Section 10.5, method or basic block counts are first collected in an offline fashion from all unit tests run on an unmodified version of the code. As shown in Figure 10.3, our test ordering algorithm analyzes these offline counts in determining its test orderings.

The metric learning component optimizes our algorithm's distance function for version r of the code by analyzing previous versions of the code r'. To incorporate metric learning, we adopt a cross-validated, "leave one bug out" approach. Given a bug b_j, we learn a metric by analyzing all other bugs except b_j. This simulates the scenario in which bug b_j is introduced *after* all other bugs have been introduced. This learned metric is then used as a basis for running our test ordering algorithm. We emphasize that the metrics we learn

Benchmark	Bugs	Constraints	Tests
Commons Collections	6	626	12,936
Lucene Search Engine	9	1587	739
Hibernate Framework	6	758	799

Table 10.1: The number of constraints used to train metrics for each benchmark.

are specific to each benchmark and are therefore trained only on other bugs within the same benchmark.

In section 10.4, we describe how pairwise distance constraints are inferred between unit tests from a set of bugs and their associated test failures. Recall that for each bug b_j, two tests are constrained to be similar if they both failed, and two tests are constrained dissimilar if one failed and the other did not. Since the total number of unique constraints can grow quadratically in the number of tests, we limit each bug to at most 100 similarity constraints and 100 dissimilarity constraints, for a maximum of 200 constraints per bug. However, for bugs with a fewer number of failed tests (in our case, bugs with fewer than 11 failed tests), the number of unique similarity constraints available will be less than 100. Table 10.1 shows the total number of constraints used for training our metrics. Finally, Algorithm 4 requires that the number of clusters k and the nearest neighbor search size c be determined as parameters for the algorithm. Across all experiments, we set $k = 50$ and $c = 20$. Although not presented here, we found our algorithms to be robust for various values of k and c.

Most of the bugs considered in our evaluation are constrained to at most a few modified methods. As a simple optimization, we eliminate all tests that do not exercise methods that have been changed since tests were last run. Whether or not a unit test exercises a given method is not determined statically but is determined dynamically—is the method count for the test is larger then zero (i.e. the method was called at least once).

10.6.3 Baseline Methods

We compare our test ordering methods to several existing baseline methods. For every ordering algorithm, only tests that exercise changed methods are run.

- **Random**: Tests are evaluated in random order.
- **Total Method / Basic Block Coverage**: This heuristic [86] counts the number of distinct methods (or basic blocks) executed for each test. Tests are then executed to maximize basic block coverage. This heuristic is motivated by standard test coverage metrics which seek to maximize the number of methods or basic blocks covered by the test set.
- **Additional Method Coverage**: Tests are greedily run to maximize the total number of covered methods (or basic blocks) among the set of tests run so far [86, 97].

To provide additional insight into our methods, we also consider two sub-methods which are derived from our full test ordering algorithm. Our full

test ordering algorithm uses clustering, nearest neighbor search and a machine learning distance metric (**C+NN+ML**).

- **Nearest Neighbor Only (NN)**: Randomly choose tests until a failing instance is found. Once a failing test is found, execute its t nearest neighbors. This method differs from our proposed method in that failing tests here are initially located by random sampling whereas our method guides initial sampling via clustering.

- **Clustering Only (C)**: Like our proposed method, this method finds initial failing tests via clustering. However, once a failing instance is found, the method randomly executes all tests within the failing instance's cluster. This method differs from our proposed method in that the nearest neighbor component is replaced by a random component.

- **Nearest Neighbor & Clustering without Metric Learning (C+NN)**: Here, Algorithm 4 is run using the baseline Euclidean distance without using the metric learning component.

10.6.4 Performance

Here, we provide an overview of the performance of our algorithms as compared to that of existing methods. All results presented represent the average performance over 20 runs. For all algorithms, ties are borken arbitrarily and runs of our algorithm are initialized with different random seeds.

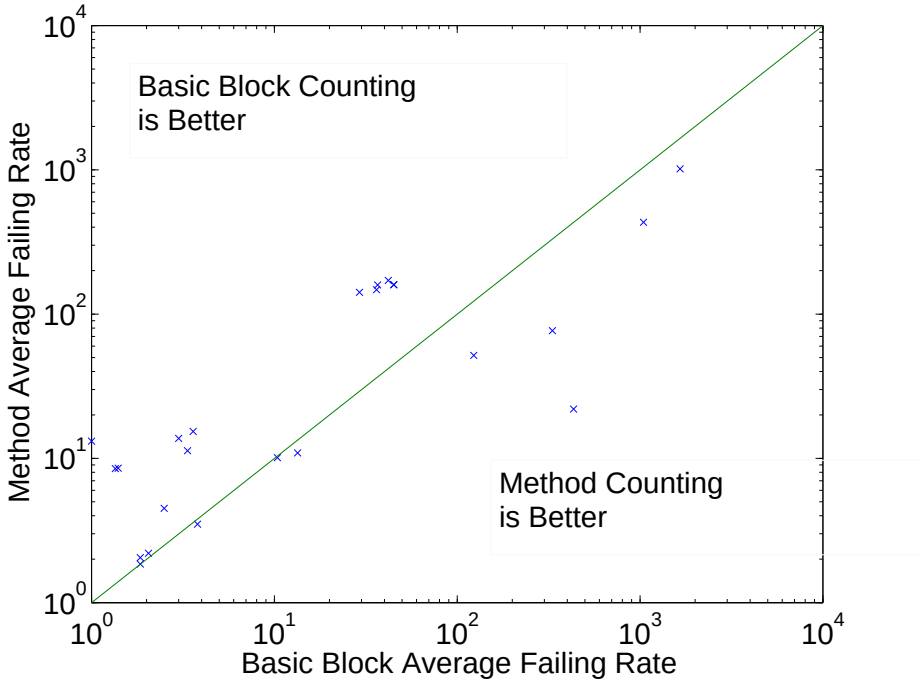


Figure 10.8: A scatter plot comparing the time to all failures if tests are instrumented at the method level compared instrumentation at the basic block level.

First, we consider the two instrumentation variants that we used in conjunction with our two methods: basic block instrumentation and method counting. Figure 10.8 gives a scatter plot comparing the time to all failures for our algorithm (without metric learning) when run using each of the two instrumentation methods. Each point represents a single bug, with its x value denoting the failure rate if method counts are used and its y value giving the rate if basic blocks are used. Overall, basic block counting outperforms method counting in 15 of the 23 total bugs. However, in terms of total tests executed, method counting requires an average of 110.2 tests per bug, whereas basic block counting takes an average of 161.3 tests, or an additional 51.1 tests.

Overall, in figure 10.8, we can see that basic block counting performs slightly better for bugs with lower time to all failures (i.e. bugs in which all failing tests were discovered after at most 100 test executions). For "harder" bugs with higher time to all failures, method counting tends to provide slightly lower failure rates. This is most likely due to the fact that basic block representations have a larger number of features than method counts and also tend to be extremely sparse. In general, as dimensionality and sparsity increases, the problem of clustering becomes more difficult.

Even though most of the computation required by our algorithm can be performed offline, the amount of time required to collect and process basic block representations as compared to method counting is significantly larger. Since the performance of the two methods is somewhat similar, we restrict our attention to method counting.

Bug Name (# Failures)	C+NN+ML	C+NN	C Only	NN Only	Additional	Total	Random
Apache Commons							
AbstractDualBidiMap.hashCode (339)	8 / **874**	**2** / 881	**2** / 938	3 / 885	**2** / 937	3 / 898	5 / 937
AbstractDualBidiMap.inverse_typo (860)	**1** / 1235	2 / 1244	3 / 1295	2 / **1233**	2 / 1294	**1** / 1277	**1** / 1294
AbstractDualBidiMap.put (20)	81 / 1434	115 / 1395	110 / 1499	**64** / 1512	117 / 1665	288 / **909**	**64** / 1660
AbstractHashedMap.hash (5)	140 / **178**	169 / 361	160 / 728	134 / 190	**8** / 645	406 / 496	107 / 699
AbstractHashedMap.put (613)	**1** / **934**	2 / 938	3 / 990	2 / 945	4 / 989	53 / 990	2 / 989
NullComparator.compare (3)	**107** / **111**	208 / 312	187 / 786	199 / 184	387 / 827	931 / 933	239 / 641
Lucene							
BooleanClause.getSoSWeights (14)	**6** / **112**	14 / 125	9 / 129	8 / 115	12 / 124	11 / 123	10 / 124
BooleanClause.isProhibited (28)	3 / **125**	4 / 147	**2** / 175	4 / 142	9 / 173	41 / 177	6 / 173
DefaultSimilarity.tf (11)	45 / 294	139 / 287	141 / 336	155 / 259	64 / 311	**14** / **200**	34 / 319
FastCharStream.refill (19)	10 / **122**	5 / 130	**3** / 166	8 / 148	8 / 162	29 / 158	8 / 158
HitQueue.lessThan (34)	**2** / 109	5 / 114	**2** / 129	4 / **105**	5 / 128	11 / 132	4 / 131
QueryParser.addClause2 (7)	**23** / 106	25 / **77**	22 / 109	36 / 100	25 / 154	31 / 157	22 / 148
QueryParser.getRankQuery (3)	3 / **6**	**2** / **6**	3 / 7	3 / **6**	4 / 9	**2** / 9	**2** / 8
Token.setTermBuffer (49)	5 / 88	2 / 96	**1** / 101	2 / **87**	3 / 100	3 / 101	2 / 99
Hibernate							
AbstractCollectionPersister.writeKey (6)	**4** / **300**	24 / 500	42 / 765	139 / 516	133 / 717	202 / 797	154 / 656
AbstractPersistentCollection.isdirty (13)	19 / **363**	13 / 439	**11** / 788	52 / 415	44 / 744	241 / 780	39 / 725
ActionQueue.executeActions (16)	25 / 330	**17** / **272**	25 / 779	53 / 552	43 / 742	251 / 789	35 / 761
CacheEntry.assemble (8)	**7** / **532**	22 / 559	16 / 757	76 / 549	47 / 711	252 / 791	121 / 678
Collections.prepareUpdate (13)	20 / 387	11 / **297**	**3** / 750	89 / 431	41 / 725	241 / 780	50 / 720
DeleteListener.deleteEntity (9)	134 / 499	57 / 442	**53** / 677	122 / 434	66 / 717	249 / **339**	110 / 755

Figure 10.9: X/Y where X is the time to first failure and Y is the time to all failures. Results are given for each bug in our three benchmarks across seven methods: Our full test ordering algorithm with metric learning (C+NN+ML), our method without the metric learning component (C+NN), a method that only uses the clustering component (C Only), a method that uses only nearest neighbor searches (N Only), two existing methods (Additional and Total), and a baseline method that runs tests in random order. Overall, our full algorithm C+NN+ML is competitive with our outperforms all other methods.

Figure 10.9 shows results for each bug for all three benchmarks: Apache commons collections, Lucene, and Hibernate. Results are displayed as 'X/Y', where X is the time to first failure and Y is the time to all failures. The best results are bolded. Overall, our full algorithm with metric learning (C+NN+ML) is either the best or is competitive with the best time to first failures and time to all failures. Comparing our algorithm with metric learning to the same algorithm without metric learning (which uses squared Euclidean distance), we see that using a learned metric improves results dramatically. For example, in terms of time to all failures, using a learned metric provided two-fold improvement for the bug `AbstractHashedMap.hash` in the Commons library and a 1.7× improvement for `AbstractCollectionPersister.writeKey` in the Hibernate benchmark.

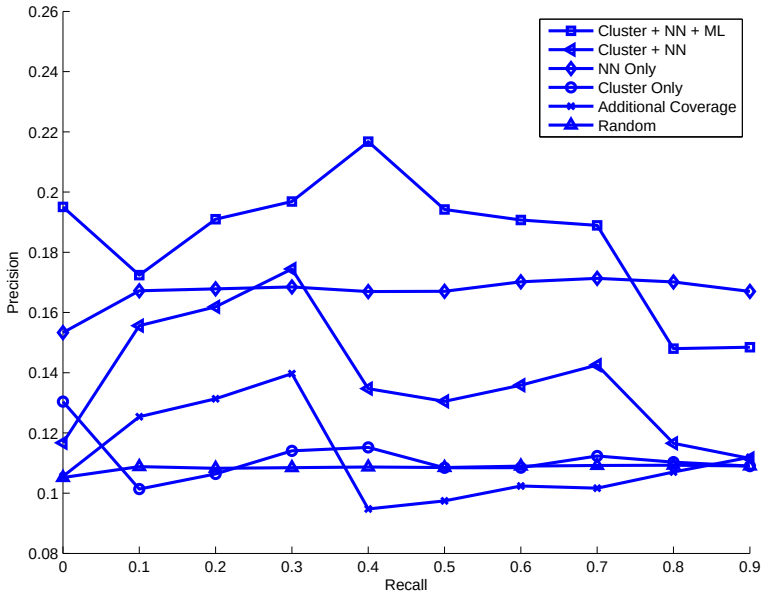
Comparing the algorithm which performs only clustering and no nearest neighbor searches (C Only) to that of the algorithm that performs nearest neighbor searches but chooses initial points at random (NN Only), we see that time to first failure tends to be lower for the clustering algorithm, whereas time to all failures tends to be lower for the nearest neighbor algorithm. This reinforces our algorithm motivations: the clustering component is effective in quickly sampling and locating initial failing tests, and the nearest neighbor component can quickly locate additional failing once a single failing test has been identified.

Compared to the two existing methods, Additional and Total, our method provides lower time to all failures for five out of the six bugs in Apache

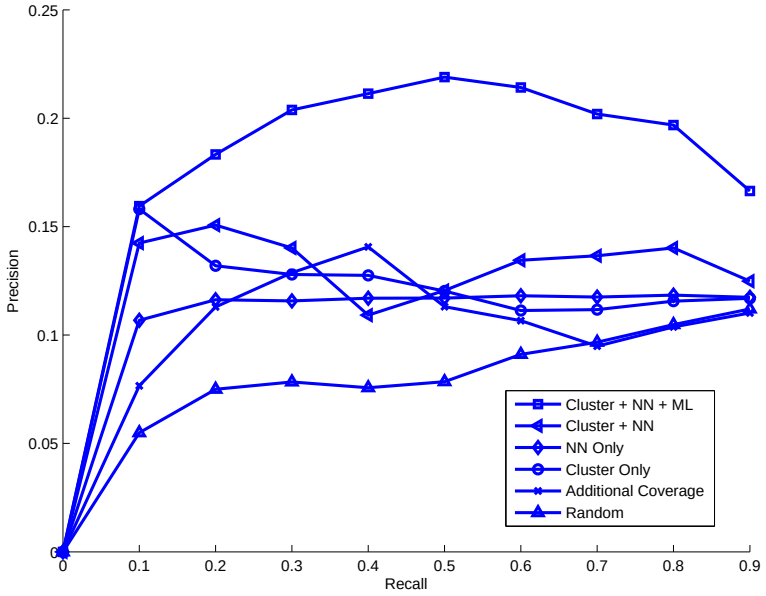
Commons, seven out of eight in Lucene, and five out of six bugs in the Hibernate framework.

In terms of absolute numbers, the reduction in the number of executed tests is considerable. For example, the Lucene benchmark has 739 total unit tests, yet our algorithm requires that at most 294 of these be run. In all cases except one, fewer than 125 total tests need to be executed in order to discover all failed tests. This represents a savings of 83%. Reduction in the number of tests is not quite as good for the Hibernate benchmark. Hibernate has 799 tests, and our algorithm reduces the number of required test runs by an order of two or more. Finally, the Commons Collections benchmark has over 12,000 tests, and our algorithm realizes significant gains here. In particular, all three tests associated with the NullComparator.compare are discovered after only 111 tests, a savings of 99.2%.

The metrics considered in Figure 10.9 are a function of when the first failing test is discovered (time to first failure) and when the last failing test is discovered (time to all failures). We now provide a comparison of our methods in the context of how quickly each of the failing tests are discovered. To do this, we appeal to two standard information retrieval metrics: precision and recall [3]. Precision measures how effective the ordering algorithm is at discovering failing tests as compared to executing passing tests. It is defined as the number of failing tests run as compared to the total number of tests run.



(a) Apache Commons Collections



(b) Lucene

Figure 10.10: Recall-precision curves for the Commons and Lucene benchmarks. Overall, our full algorithm with metric learning achieves noticeably higher precision than all other methods.

$$precision = \frac{\text{Number of Failing Tests Run}}{\text{Total Number of Tests Run}}.$$

Recall measures how effective the ordering algorithm is in discovering all failing tests. It is defined as the fraction of total failing tests that have been run, compared to the total number of failing tests in the test suite:

$$recall = \frac{\text{Number of Failing Tests Run}}{\text{Total Number of Failing Tests}}.$$

Figure 10.10 shows precision-recall curves for the Lucene benchmark and the Commons benchmark. The precision-recall curve is a standard evaluation metric used in information retrieval. For each recall level, the curve gives the associated precision value. An ideal test ordering policy would achieve 100% precision for all recall values, e.g., at the time when the algorithm has found 50% of all test failures, all previously executed failing tests were also failures, resulting in 100% precision. The curves represent an average of each of the bugs shown in Figure 10.9. Overall, our full algorithm with metric learning significantly outperforms existing methods. In particular, for a recall value of 0.5, the precision of our full method is 0.22, which is almost twice as high as that of the next best method (NN Only, 0.124) for Lucene.

Finally, we note that in many information retrieval systems, the precision recall curve is typically monotonically decreasing. Our plots are not. This is due to the fact that identifying an initial failing run is harder than identifying additional failing runs.

10.6.5 Real-World Bugs

The bugs analyzed and tested in the previous section are introduced by hand. In this section, we present results from our system applied to bugs harvested from real-world software repositories. The purpose of unit testing is to execute unit tests before checking code into the repository. Therefore, bugs that are checked into software repositories are generally the result of inadequate test coverage and do not cause any existing unit tests to fail. The key challenge in finding real-world bugs is not just to locate bugs in software repositories, but to identify bugs that also have failing unit tests.

In practice, unit tests are almost always run by developers before code is checked in. Therefore if a bug b is checked into the repository at some revision number r, it is generally a result of incomplete test coverage for the test set $T(r)$ at revision r. That is, $T(r)$ does not specifically test cases and behavior associated with the bug. When bug b is fixed in some later revision s ($s > r$), engineers oftentimes write a few unit tests to demonstrate that their bug fix corrects previously incorrect functionality. Here, the test set $T(s)$ for revision s is generally a superset of $T(r)$.

Given this pattern, we employ the following strategy to discover real-world bugs. For each revision r, we run tests $T(s)$ from revisions s that occur up to six revisions after r. The goal here is to check if r contains bugs that were later realized, fixed, and tested in revision s. We performed this procedure over the Apache commons collections library for approximately 8,000 revisions dating back to 2001. In total, we found 28 bugs such that:

1. Revision s occurs after revision r.

2. All tests $T(s)$ succeed when run against the source code from s, and all tests in $T(r)$ succeed against revision r's source.

3. Tests $T(s)$ compiles against the source of revision r and at least one test fails in $T(s)$ when run against r's source.

We note that item (2) is required since occasionally unit tests exhibiting buggy behavior are checked in *before* bugs are fixed, resulting in checked-in revisions with failing unit tests. Finally, since the code changes from r to s may involve functionality other than bug fixes, we manually revert any classes in s back to that found in revision r that are unrelated to the bug.

Of the 28 bugs, 20 resulted in only one or two failing tests. Results for the remaining tests (those with more than two failing tests) are shown in Figure 10.11. Our introduced bugs were assumed to be time-independent, and a "leave one bug out" approach was taken in order to train the distance metrics. Here, each real-world bug has a distinct date associated with it (i.e. the date corresponding to the revision in which the bug was checked-in). Therefore, we train each bug at revision r using bugs only in prior revisions s ($s < r$).

Compared to the introduced bugs presented in the previous section, the bugs considered in this section have a more local influence with respect to the unit tests. Recall that as a pre-processing step for all methods considered, we eliminate any test that does not exercise changed or modified code. In

Rev (Failures)	C+NN+ML	C+NN	C Only	NN Only	Additional	Total	Random
131786 (10)	**112** / 997	118 / **838**	115 / 1314	165 / 1281	273 / 1614	1364 / 1710	255 / 1608
131840 (5)	16 / **44**	36 / 169	27 / 428	59 / 87	**13** / 60	380 / 388	72 / 360
131791 (3)	**2** / **3**	**2** / **3**	**2** / **3**	3/4	3 / 7	4 / 7	3 / 4
131838 (5)	3 / 7	3 / 7	3 / 7	**2** / **6**	3 / 9	3 / 8	**2** / 8
209680 (8)	4 / 21	**1** / **9**	**1** / 24	3 / 11	4 / 23	8 / 22	3 / 20
209683 (8)	7 / 18	**6** / **15**	11 / 47	8 / 17	17 / 55	8 / 51	**6** / 54
418637 (4)	2 / 6	2 / **4**	2 / **4**	**1** / **4**	**1** / 6	**1** / 5	2 / **4**
428130 (3)	**1** / 8	2 / 8	2 / **5**	6 / 11	12 / 21	13 / 15	7 / 13

Figure 10.11: Results from real-world bugs, given as X/Y, where X is the time to first failure and Y is the time to all failures. Overall, our full algorithm with clustering, nearest neighbor searches, and metric learning, outperforms all other methods.

Figure 10.9, the code changes associated with each bug affect upwards of one thousand unit tests. This is due to the fact that bugs were added to code that composed core functionality of the system or library (i.e. the tt Hashed-Map.put method). In contrast, the bugs discovered in this real-world case study generally effect non-central classes whose functionality has few or no dependencies. For example, the bug in revision 131,791 involved the incorrect implementation of an `ExtendedProperties` class. This class extends Java's standard `Properties` class to allow property values to be appended instead of overwritten. As a result, the code changes associated with each of the real-world bugs affected relatively few unit tests. For six of the eight bugs in Figure 10.11, the pre-processing step was able to eliminate a large fraction (greater than 99%) all unit tests. However, among the two bugs in which the majority of tests cannot be trivially eliminated, our algorithm provides the lowest time to first failure (revision 131,786) and time to all failures (revision 131,840) among all methods considered.

10.7 Threats to Validity

The purpose of unit testing is to discover and fix bugs before code is checked in. As part of the software development process, most bugs exist for a relatively short period of time immediately after code has been written, and are fixed before the code is checked in to the repository. As a result, no traces of these bugs can be inferred from repository check-in logs.

In section 10.6, we introduced bugs that were systematically generated by hand. In creating these bugs, we put forth our best efforts to introduce bugs that are representative of common bug forms—i.e. null pointer errors, off-by-one errors, logical errors, etc. However, without an extensive case study, it is impossible to determine whether these bugs are completely reflective of the true distribution of bugs relevant to our framework.

In evaluating our system over real-world bugs, we proposed a methodology to mine the bugs from existing repositories. A fundamental challenge here is the fact that bugs that are checked into the repository tend to be those which are not tested by any unit tests. While we discovered several real-world bugs (that were subsequently fixed) in the Apache commons collections library, we noted that these bugs were somewhat different from the bugs that we introduced by hand. Specifically, these real-world bugs are biased in that they effect relatively non-central functionality to the software or library.

10.8 Related Work

Closest to our work is that of Rothermel et al. [87]. Here, several test selection methods are proposed in which tests are represented by their set (but not counts) of methods or basic blocks executed. They propose three methods: total coverage, additional coverage, and fault-exposing-potential prioritization. Total coverage prioritization orders tests based on those that have maximal coverage. Depending on representation, coverage is defined as either the unique number of basic blocks exercised or the unique number of methods exercised (i.e. the size of each execution's set). Additional coverage is a greedy heuristic in which tests are sequentially chosen to maximize the total number of features covered so far. Given a set of executed tests T that cover a set of features F, additional coverage selects the test $x \notin T$ that covers features f such that the size of the set $F \cup f$ is maximal. Finally, two fault-exposing-potential (FEP) prioritization methods are presented which try to estimate failure probabilities using a statistical technique called PIE (propagation, infection, and execution). These methods are computationally more expensive than the total and additional coverage methods, yet experimentally do not perform significantly better. Further, the FEP methods were not considered in a follow-up case study of larger Java programs [38] and are not considered here. Our distance metric learning algorithm also can be viewed as a statistical inference procedure [31].

Related to the problem of test set ordering is that of test set selection. Here, the goal is to find a subset of tests given a set of code changes such that

tests not included in this subset are *guaranteed* to execute correctly. Tests that will provably succeed do not need to be run. The hope in the test selection problem is to efficiently prune away a large fraction of tests which can be proven will succeed, reducing time needed to execute the test suite. Work by Harrold et al. [52] proposes a Java-specific model that seeks to identify a minimal set of affected tests from a set of code changes; a method is presented which analysis the Java inter-class graph to safely eliminate tests. Ball [4] also presents a similar edge-detection algorithm which uses control-flow-based regression test selection (CRTS).

Overall, test selection has several limitations. First, effective test selection models are programming-language specific and often require additional assumptions. For example, the method presented in [52] does not allow the use of reflection, which is becoming increasingly common in enterprise Java development platforms such as our Hibernate benchmark. Experimentally, the subset discovered by test selection algorithms oftentimes represents the majority (i.e. greater than 50%) of all tests [52]. Finally, test selection methods are computationally much more expensive than the ordering methods presented here and consequently have not been shown to scale to large modern programs. Even though the test ordering algorithms presented in this chapter have no provable guarantees, our methods are programming language independent, require no assumptions, and experimentally quite effective, scaling to large modern programs.

Finally, our algorithms learn domain-specific distance function models

through a novel application of metric learning, and the HD^{2}LR method used is particularly well suited for the high-dimensional problems that arise in the problem of statistical software representation. One alternative to metric learning is to incorporate supervision into the clustering process via semi-supervised clustering algorithms [10]. However, our learned metrics can be used for new test cases that are later written, whereas semi-supervised clustering algorithms only learn relations between existing tests.

Chapter 11

Conclusions and Future Work

In this thesis, we proposed models for analyzing correlations, algorithms for learning correlations, and novel correlation mining applications. In comparing two sets of correlations, we proposed the use of the LogDet divergence and motivated its usage through connections with information theory and statistics. To learn correlations, we developed models, formulations, and algorithms for learning correlation matrices for parametrization a Mahalanobis distance function. Finally, we considered two statistical software analysis applications: software error reporting and unit testing prioritization.

The primary tool used by the methods and algorithms presented in this thesis is the LogDet divergence. We argued that the LogDet divergence is a natural measure here, showing connections between LogDet and other popular measures from information theory and statistics, namely, the Wishart distribution between covariance matrices and the relative entropy between Gaussian distributions. In Chapter 4, we provided an example comparing the LogDet divergence with the Frobenius divergence, a standard measure that compares correlations via the sum of squares of the element-wise difference. As future work, one could consider other divergences between correlation matrices, for

example, the Von Neumann divergences presented in [61]. In general, matrix divergences are not well-studied, and the LogDet, Frobenius, and the Von Neumann divergence can are all special cases of a class of distance measures called Bregman divergences. In deriving our LogDet algorithms, existing work on Bregman based methods was heavily exploited to provide computational efficiency. However, an investigation of other non-Bregman based measures comparing correlations would be very interesting.

The correlation clustering algorithm presented in Chapter 3 is a direct corollary from the connection between the LogDet divergence and the relative entropy between Gaussian distributions. We showed that this relative entropy measure can be expressed as a convex combination of two distances measures: the LogDet divergence between covariances, and a Mahalanobis distance between means. Experimentally, we evaluated our algorithm across sensor network data as well as a novel statistical software application that analyzed method calling behavior in the `latex` program. As future work, it would be interesting to investigate an in-depth study of the algorithm's use for program debugging and analysis. One challenge here is identifying correlations and other dependencies which are unexpected. For example, it is not surprising that the calling behavior between two methods that are called sequentially in a program will be highly correlated. However, inferring correlations between two methods in seemingly unrelated components of a large-scale system can provide a great deal of insight. A solution here would require use of static-analysis methods to complement the dynamic-based methods used in

statistical software analysis.

Our Information-theoretic metric learning (ITML) formulation uses pairwise feature correlations to compute distances between objects. We modeled the problem via a LogDet objective, which resulted in a maximum-entropy formulation and efficient Bregman-based algorithms. Experimentally, we showed that the baseline ITML method can learn high quality metrics for relatively low dimensional problems.

In the presence of high dimensionality, ITML is not scalable, as it optimizes over all pairs between features, a quantity that grows quadratically in the dimensionality. Statistical software analysis data sets have upwards of 100,000 features, and running ITML over such data would require inference over billions of parameters. To this end, we presented two methods for learning low parameter structured metrics. We developed two different structured metrics, one which is parametrized by a low-rank matrix, and another parametrized by a diagonal plus low-rank matrix. We analyzed these two measures in the context of precision and recall, showing that the latter method is better suited for problems which require both high recall as well as high precision. Experimentally, we validated these methods across several high dimensional applications, including text, statistical software analysis, and collaborative filtering.

One limitation of our structured metrics is the fact that the low-rank basis must be specified and is not learned by our method. As we saw in the experimental results, the quality of the learned metric is very much a function of the method used to determine this basis. In particular, using a

basis derived from class label information resulted in higher quality metrics than using an unsupervised principal components analysis method. Thus, there is room for improvement in our algorithm if this basis can also be learned by our methods. One complication here is the non-convexity that arises when optimizing over rank constraints. Recall that our metric learning problems are all formulated as convex problems, which result in simple and efficient algorithms. Optimizing over an arbitrary, unspecified basis will result in a highly non-convex problem.

Another Mahalanobis matrix form not considered here is that which is constrained to be sparse. As a simple heuristic, one possibility here is to use a decision tree to determine sparsity patterns. Decision trees classify by predicating a small number of available features. In the context of sparse correlation matrices, correlations which are not tested by the tree can take on zero values. Furthermore, when used in conjunction with the cost-sensitive decision tree methods presented in Chapter 9, correlations could be inferred via features that are inexpensive to collect.

In Chapter 8, we presented Clarify, a system which uses machine learning to provide improved error messaging. Statistical software analysis is a relatively new area, and the Clarify system is one of the most complete statistical software analysis systems developed to date. In presenting the system, we overviewed several fundamental challenges in statistical software analysis, including the high dimensionality of the data and the cost of monitoring and collecting program features. Experimentally, we evaluated Clarify on sev-

eral very large, real-world benchmarks. For example, we showed that for the `latex` system, we can provide better error messaging over 81 error classes with accuracy upwards of 95%. Additionally, we presented the model of nearest neighbor software support, which provides better messaging when data is semi-supervised or completely unsupervised. In developing our metric learning algorithms, we provided several experiments showing that this component can be greatly improved by learning a program-specific distance function via our algorithms.

Addressing the issue of program feature costs, we developed a cost-sensitive decision tree algorithm. Here, a decision tree is optimized for not only the accuracy of the resulting classifier, but also the cost of the features tested throughout the tree. We showed that the overheads of the Clarify system can be greatly reduced by using our algorithms. As discussed, combining decision tree learning with sparse metric learning would be an interesting direction for future work.

Finally, in Chapter 10, we developed a unit test ordering algorithm for reducing the time needed to run unit test suites. By using a clustering and nearest neighbor component, our method efficiently searches and executes the space of possible failing tests. We showed that by incorporating our metric learning methods, the resulting test ordering algorithms can reduce the time required to run all unit tests by up to 99.8%. As future work, it would be interesting to develop a fully functional system to deploy for use in popular development frameworks.

Appendix

Appendix 1

An Overview of Various Classification Methods Applied to the Clarify System

Here, we provide further evaluation of different classification algorithms, and gives empirical justification that decision trees are best suited for Clarify's classification problem.

Among the classifiers we test for the Clarify system are: Quinlan's C4.5 decision tree, the AdaBoostM1 algorithm, using decision stumps as a base learner, and a Support Vector Machine. Unless otherwise noted, all classifiers are either implemented in, or part of, the WEKA machine learning package [104]. Because all of the Clarify benchmarks are multi-class problems, we use a standard one-versus-all approach [40] for multi-class predictions with the boosting algorithm. The SVM we use that is part of the WEKA package uses pairwise coupling [54]. We also tested a multi-class SVM that is part of the SVM Struct package [60]. Preliminary experiments yielded relatively low cross validated accuracy (78.3% for FoxPro), and training time needed to perform each experiment was on the order of days. Consequently, full results of this SVM are not evaluated. We experimented with various kernels, including the linear, radial basis function, gaussian, and degree 2 polynomial kernels.

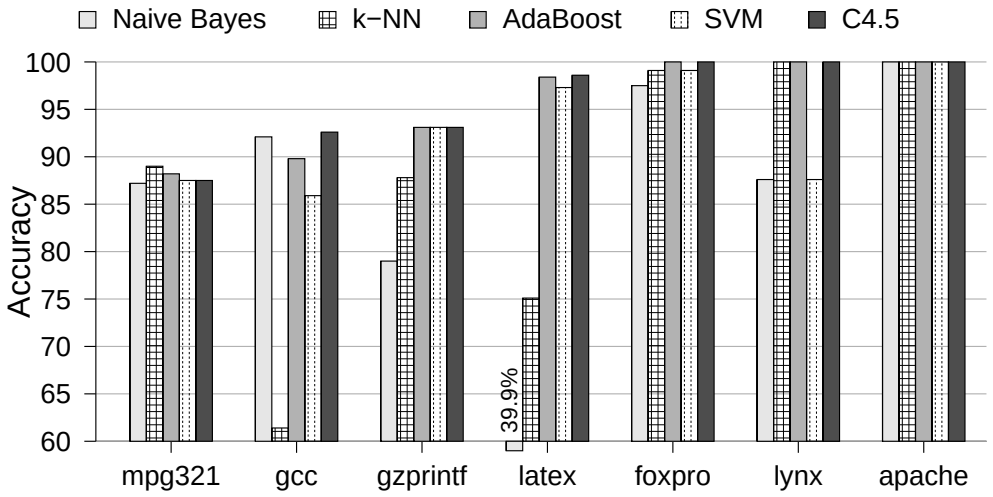


Figure 1.1: Cross validated classification accuracy, for different classifiers across a range of benchmarks. All benchmarks use the CTP behavior representation.

Overall, the linear kernel reliably yields the highest classification accuracy. We optimize the regularization parameter for the slack variables in the SVM formulation across many values of different orders, using cross validation. For AdaBoost, we found that using stronger base learners such as C4.5 reduced accuracy. As a baseline for comparison, we also consider the naive Bayes classifier, and the k-nearest neighbor classifier, where k is optimized using cross validation.

Figure 1.1 shows the cross validated accuracy for each of the classifiers described above using call-tree profiling of depth bounds 0, 1, and 2 (CTP).

Surprisingly, for several benchmarks, the support vector machine is not competitive with either the decision tree or boosted decision stumps. The

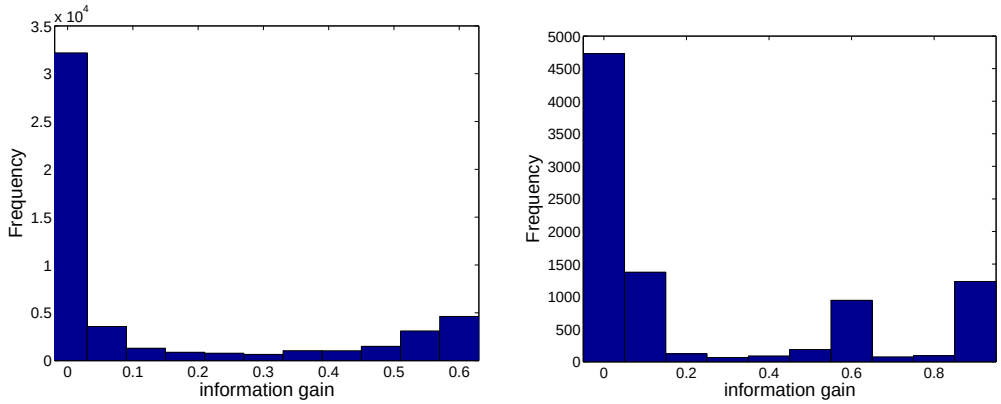


Figure 1.2: Histogram of the information gain scores for call-tree profiling features in the `gcc` and `lynx` benchmarks. The histogram shows a strong multimodal distribution: many features have no discriminative power (and thus have information gain scores close to zero), while the remaining features have relatively high information gain.

cause of most software errors occur in small, isolated subsections of code, so the remaining majority of the codebase and the features representing it provide no useful context for a given error. Figure 1.2 gives a histogram of the information gain scores (a popular feature selection method) for the call-tree profiling features in the `gcc` and `lynx` benchmarks. The distributions are markedly bimodal (or even trimodal, for `lynx`): many features are non-discriminative and have information gain scores very close to zero, while the rest of the features are quite discriminative, having relatively high information gain scores. We also found similar results in the other benchmarks, although the trend was less pronounced for benchmarks such as `latex`, where the SVM is more competitive with C4.5.

To demonstrate that poor SVM performance is the result of a high

volume of non-discriminative features, we performed the following experiment using the `gcc` benchmark, with CTP representation. First, a decision tree is trained using all features, and a linear SVM is then trained using only those features present in the decision tree. The 5-fold cross validated accuracy of this classifier is similar to that of just using the decision tree (92.6% vs. 92.2%, respectively), while the SVM trained using all features achieved accuracy of only 86.0%. We found that other feature selection methods, such as choosing features located around or above the discriminative peak of the bimodal distribution, significantly improved accuracy of the SVM for `lynx`, but led to decreased accuracy for `gcc`.

With the exception of AdaBoost, the C4.5 decision tree outperforms all classifiers. Decision trees have the advantage that they can be easily interpreted: it is possible for a software engineer to validate the classifier based on knowledge of program structure. While decision tree structure tends to have a natural fit with program structure, the continuous nature of the other machine learning algorithms we considered do not lend themselves to the construction of easy to understand hypotheses.

Bibliography

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, Bolton Landing, NY, Oct. 2003.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performacne counters with flow and context sensitive profiling. In *PLDI '97*, pages 4–16, June 1997.

[3] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[4] T. Ball. On the limit of control flow analysis for regression test selection. In *ACM International Symposium on Software Testing and Analysis*, pages 134–142, 1998.

[5] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO*, 1996.

[6] A. Banerjee, X. Guo, and H. Wang. On the optimality of conditional expectation as a bregman predictor. In *IEEE Transactions on Information Theory*, volume 51, pages 2664–2669, July 2005.

[7] A. Banerjee, S. Merugu, I. Dhillon, and S. Ghosh. Clustering with Bregman divergences. In *Siam International Conference on Data Mining*, pages 234–245, 2004.

[8] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, 2004.

[9] R. Barrett, E. Haber, E. Kandogan, P. P. Maglio, M. Prabaker, and L. A. Takayama. Field studies of computer system administrators: Analysis of system management tools and practices. In *ACM CSCW (Computer-supported Cooperative Work)*, 2004.

[10] S. Basu, M. Bilenko, and R. J. Mooney. A Probabilistic Framework for Semi-Supervised Clustering. In *Proc. 10th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 59–68, 2004.

[11] J. Berkman. *Bug-buddy — GNOME bug-reporting utility*, 2004.

[12] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *ISSTA*, Jul 2004.

[13] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, March 2004.

[14] J. Bradford, C. Kunz, R. Kohavi, C. Brunk, and C. Brodley. Pruning decision trees with misclassification costs. In *European Conference on Machine Learning*, 1998.

[15] L. Bregman. The relaxation method finding the common point of convex sets and its application to the solutions of problems in convex programming. In *USSR Comp. of Mathematics and Mathematical Physics*, volume 7, pages 200–217, 1967.

[16] M. Brodie, Sheng Ma, G. Lohman, L. Mignet, N. Modani, M. Wilding, J. Champlin, and P. Sohn. Quickly finding known software problems via automated symptom matching. In *ICAC'05*, pages 101–110, 2005.

[17] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE*, 2004.

[18] Y. Censor and S. A. Zenios. *Parallel Optimization: Theory, Algorithms, and Applications*. Oxford University Press, 1997.

[19] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, 2002.

[20] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[21] Trishul M. Chilimbi and Vinod Ganapathy. Heapmd: Identifying heap-based bugs using anomaly detection. In *ASPLOS '06*, 2006.

[22] S. Chopra, R. Hadsell, and Y. LeCun. Learning a Similarity Metric Discriminatively, with Application to Face Verification. In *IEEE Conf. on Computer Vision and Pattern Recognition*, 2005.

[23] Latex Error Classes. *http://www.cs.utexas.edu/users/habals/clarify/latex_errors.html*, 2006.

[24] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP*, 2005.

[25] Michael Collins, Robert E. Schapire, and Yoram Singer. Logistic regression, adaboost and bregman distances. In *Computational Learing Theory*, pages 158–169, 2000.

[26] T. M. Cover and J. A. Thomas. *Elements of information theory*. Wiley Series in Telecommunications, 1991.

[27] K. Crammer, J. Keshet, and Y. Singer. Kernel Design Using Boosting. In *Advances in Neural Information Processing Systems (NIPS)*, 2002.

[28] J. V. Davis and I. S. Dhillon. Differential Entropic Clustering of Multivariate Gaussians. In *Advances in Neural Information Processing Systems (NIPS)*, 2006.

[29] J. V. Davis and I. S. Dhillon. Structured Metric Learning for High Dimensional Problems. In *Knowledge Discovery and Data Mining*, August 2008.

[30] J. V. Davis and I. S. Dhillon. Unit Test Selection. In *Submitted for publication*, February 2008.

[31] J. V. Davis, B. Kulis, P. Jain, S. Sra, and I. S. Dhillon. Information-theoretic Metric Learning. In *Int. Conf. on Machine Learning (ICML)*, June 2007.

[32] Jason V. Davis, Jungwoo Ha, Christopher J. Rossbach, Hany E. Ramadan, and Emmett Witchel. Cost-sensitive decision tree learning for forensic classification. In *ECML*, 2006.

[33] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[34] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-based approximate querying in sensor networks. In *International Journal of Very Large Data Bases*, 2005.

[35] I. Dhillon and Y. Guan. Information theoretic clustering of sparse co-occurrence data. In *IEEE International Conference on Data Mining*, 2003.

[36] I. Dhillon, S. Mallela, and R. Kumar. A divisive information-theoretic feature clustering algorithm for text classification. In *Journal of Machine Learning Research*, volume 3, pages 1265–1287, 2003.

[37] C.L. Blake D.J. Newman, S. Hettich and C.J. Merz. UCI repository of machine learning databases, 1998.

[38] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a junit testing environment. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 113–124, 2004.

[39] B. E. Dom. An information-theoretic external cluster-validity measure. Technical Report 10219, IBM, 2001.

[40] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. John Wiley and Sons, Inc., 2001.

[41] C. Elkan. The foundations of cost-sensitive learning. In *International joint conference on artifical intelligence*, 2001.

[42] Apache Lucene Search Engine. *http://lucene.apache.org/java/docs/*, 2008.

[43] RTI for G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, NIST, 2002.

[44] E. Gabrilovich and Shaul Markovitch. Text categorization with many redundant features: using aggressive feature selection to make svms competitive with c4.5. In *ICML*, 2004.

[45] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, 1979.

[46] A. Globerson and S. Roweis. Metric Learning by Collapsing Classes. In *Advances in Neural Information Processing Systems (NIPS)*, 2005.

[47] J. Goldberger, S. Roweis, G. Hinton, and R. Salakhutdinov. Neighbourhood Component Analysis. In *Advances in Neural Information Processing Systems (NIPS)*, 2004.

[48] G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, second edition, 1989.

[49] J. Ha, C. Rossbach, J. Davis, I. Roy, D. Chen, H. Ramad an, and E. Witchel. Improved Error Reporting for Software that Uses Black Box Components. In *Programming Language Design and Implementation (PLDI)*, 2007.

[50] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.

[51] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between fault-revealing test behavior and differences in program spectra. In *Journal of Software Testing, Verification and Reliability, vol 10, no 3*, 2000.

[52] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, and Steven Spoon. Regression test selection for java software. In *OOPSLA*, 2001.

[53] T. Hastie and R. Tibshirani. Discriminant adaptive nearest neighbor classification. *Pattern Analysis and Machine Intelligence*, 18:607–616, 1996.

[54] T. Hastie, R. Tibshirani, and J. H. Friedman. *The elements of statistical learning*. Springer, 2001.

[55] Hibernate. *http://www.hibernate.org/*, 2008.

[56] J. Humphreys and V. Turner. On-demand enterprises and utility computing: A current market assessment and outlook. Technical report, IDC, Jul 2004.

[57] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*, 1994.

[58] W. James and C. Stein. Estimation with quadratic loss. In *Proc. Fourth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 361–379. University of California Press, 1961.

[59] Jim Keniston and Prasanna S Panchamukhi. *Kernel Probes (Kprobes)*, 2006.

[60] Thorsten Joachims. Text categorization with suport vector machines: Learning with many relevant features. In *ECML '98: Proceedings of the 10th European Conference on Machine Learning*, pages 137–142, London, UK, 1998. Springer-Verlag.

[61] B. Kulis, M. Sustik, and I. S. Dhillon. Learning Low-rank Kernel Matrices. In *Int. Conf. on Machine Learning (ICML)*, 2006.

[62] Ruby Programming Language. *www.ruby-lang.org/*, 2008.

[63] N. Lao, J. Wen, W. Ma, and Y. Wang. Combining high level symptom descriptions and low level state information for configuration fault diagnosis. In *LISA*, 2004.

[64] G. Lebanon. Metric Learning for Text Documents. *Pattern Analysis and Machine Intelligence*, 28(4):497–508, 2006.

[65] E. L. Lehmann and G. Casella. *Theory of Point Estimation*. Springer, second edition, 2003.

[66] B. Liblit, A. Aiken, A.X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[67] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[68] Apache Commons Collections Library. *http://commons.apache.org/collections/*, 2008.

[69] Byte Code Engineering Library(BCEL). *http://bcel.sourceforge.net/*, 2008.

[70] Chih-Jen Lin. *LIBSVM – A Library for Support Vector Machines*, 2005.

[71] C. Liu, X. Yang, H.Yu, J. Han, and P. S. Yu. Mining behavior graphs for "backtrace" of noncrashing bugs. In *Proc. of 2005 SIAM Int. Conf. on Data Mining (SDM05)*, 2005.

[72] S. Madden. Intel lab data. http://berkeley.intel-research.net/labdata, 2004.

[73] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, 1999.

[74] K. V. Mardia, J. T. Kent, and J. M. Bibby. *Multivariate Analysis*. Academic Press, 1979.

[75] Microsoft Corporation. *Dr. Watson Overview*, 2002.

[76] Microsoft Corporation. *Online Crash Analysis*, 2004.

[77] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Boston, 1997.

[78] R.J. Muirhead. *Aspects of Multivariate Statistical Theory*. Wiley, NY, 1982.

[79] T. Myrvoll and F. Soong. On divergence based clustering of normal distributions and its application to HMM adaptation. In *Eurospeech*, pages 1517–1520, 2003.

[80] S.W. Norton. Generating better decision trees. In *International joint conference on artifical intelligence*, 1989.

[81] M. Nunez. The use of background knowledge in decision tree induction. In *Machine Learning*, 1991.

[82] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *International Conference on Measurement and Modeling of Computer Systems (SIGMetrics extended abstract)*, June 2003.

[83] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *ICSE*, 2003.

[84] R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers, 1992.

[85] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In M. Jazayeri and H. Schauer, editors, *ESEC/FSE 97*, pages 432–449. Springer–Verlag, 1997.

[86] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Test case prioritization: An empirical study. In *Proceedings of the International Conference on Software Maintenance*, 1999.

[87] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Test case prioritization. In *IEEE Transactions of Software Engineering*, volume 27, pages 929–948, 2001.

[88] Rubber. *http://www.pps.jussieu.fr/ beffara/soft/rubber*, 2007.

[89] M. Schutz and T. Joachims. Learning a Distance Metric from Relative Comparisons. In *Advances in Neural Information Processing Systems (NIPS)*, 2003.

[90] Classic3 Data Set. *ftp.cs.cornell.edu/pub/smart*, 2008.

[91] CMU 20-Newsgroups Data Set. *http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-20/www/data/news20.html*, 2008.

[92] S. Shalev-Shwartz, Y. Singer, and A. Y. Ng. Online and Batch Learning of Pseudo-Metrics. In *Int. Conf. on Machine Learning (ICML)*, 2004.

[93] N. Shental, T. Hertz, D. Weinshall, and M. Pavel. Adjustment learning and relevant component analysis. In *Proc. of European Conf. Computer Vision*, Copenhagen, DK, 2002.

[94] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, Oct 2002.

[95] Y. Singer and M. Warmuth. Batch and on-line parameter estimation of Gaussian mixtures based on the joint entropy. In *Neural Information Processing Systems*, 1998.

[96] A. Somayaji and S. Forrest. Automated response using system-call delays. In *Proceedings of 9th Usenix Security Symposium*, August 2000.

[97] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–106, New York, NY, USA, 2002. ACM.

[98] SvmMulticlass. *http://svmlight.joachims.org*, 2008.

[99] M. Tan and J. Schlimmer. Csl: A cost-sensitive learning system for sensing and grasping objects. In *IEEE International Conference on Robotics and Automation*, 1993.

[100] P. Turney. Cost-sensitive classification: Empirical evaluation of a hybrid genetic decision tree induction algorithm. In *Journal of artificial intelligence research*, 1995.

[101] P. Turney. Types of costs in inductive concept learning. In *ICML*, 2000.

[102] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI*, 2004.

[103] K. Q. Weinberger, J. Blitzer, and L. K. Saul. Distance Metric Learning for Large Margin Nearest Neighbor Classification. In *Advances in Neural Information Processing Systems (NIPS)*, 2005.

[104] I. Witten and E. Frank. *Data Mining: Practical machine learning tools with Java implementations*. Morgan Kaufmann, San Francisco, 2000.

[105] E. P. Xing, A. Y. Ng, M. I. Jordan, and S. Russell. Distance metric learning with application to clustering with side-information. In *Advances in Neural Information Processing Systems (NIPS)*, volume 14, 2002.

[106] T. Yoshimura, T. Masuko, K. Tokuda, T. Kobayashi, and T. Kitamura. Speaker interpolation in HMM-based speech synthesis. In *European Conference on Speech Communication and Technology*, 1997.

[107] C. Yuan, N. Lao, J. Wen, J. Li, Z. Zhang, Y. Wang, and W. Ma. Automated known problem diagnosis with event traces. *MSR-TR-2005-81*, 2005.

[108] C. Yuan, N. Lao, J. Wen, J. Li, Z. Zhang, Y. Wang, and W. Ma. Automated known problem diagnosis with event traces. In *EuroSys*, 2006.

[109] A. Zheng, M. Jordan, B. Liblit, and A. Aiken. Statistical debugging of sampled programs. In *Neural Information Processing Systems*, 2004.

[110] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI*, 2006.

[111] V. Zubek and T. Dieterich. Pruning improves heuristic search for cost-sensitive learning. In *Internation Conference on Machine Learning*, 2002.

Vita

Jason Davis was born in Austin, TX and grew up in Potomac, MD, where he went to the Landon School. He completed his undergraduate degree at Cornell University with a bachelor of science in computer science in 2003. Continuing his education, he began work on his Ph.D. at the University of Texas at Austin, where he worked on machine learning and data mining problems. In 2007, he won a best paper award at the International Conference on Machine Learning. He is a reviewer for the Journal of Machine Learning and has served on the program committee for the Knowledge Discovery in Data Mining conference.

Permanent address: Dept. of Computer Sciences
University of Texas at Austin
Austin, TX 78712

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.