Washington University in St. Louis
Washington University Open Scholarship

Engineering and Applied Science Theses & Dissertations | McKelvey School of Engineering

Winter 12-15-2014

Approximation and Relaxation Approaches for Parallel and Distributed Machine Learning

Stephen Tyree
Washington University in St. Louis

Follow this and additional works at: https://openscholarship.wustl.edu/eng_etds

 Part of the Engineering Commons

Recommended Citation

Tyree, Stephen, "Approximation and Relaxation Approaches for Parallel and Distributed Machine Learning" (2014). *Engineering and Applied Science Theses & Dissertations*. 64.
https://openscholarship.wustl.edu/eng_etds/64

This Dissertation is brought to you for free and open access by the McKelvey School of Engineering at Washington University Open Scholarship. It has been accepted for inclusion in Engineering and Applied Science Theses & Dissertations by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science
Department of Computer Science and Engineering

Dissertation Examination Committee:
Kilian Q. Weinberger, Chair
Kunal Agrawal
Roger Chamberlain
Laurens van der Maaten
Robert Pless

Approximation and Relaxation Approaches for Parallel and Distributed Machine Learning

by
Stephen W. Tyree

A dissertation presented to the
Graduate School of Arts and Sciences
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

December 2014
St. Louis, Missouri

© 2014, Stephen W. Tyree

Contents

List of Figures

List of Tables

Acknowledgments

At first glance, this dissertation may seem to be the result of copious coffee, computer code, and server time. (That there was much coffee is certain.) But more than that, this particular dissertation the result of relationships. To begin, I owe much to the professor who taught a remarkably exciting class in machine learning my first semester (Kilian), therein completely changing my desired research direction. My debt is extended to the colleague (Kunal) he asked to sign on to collaboratively co-advise me. Their commitment to my development and success through thick and thin have brought me to this point.

My lab-mates in Kilian's group have been the best colleagues and friends I could have hoped for in a graduate program. Mimmin, Eddie (Zhixiang), Dor, Matt, Jake, and Wenlin: our collaborations have truly been the highlight of my time here. Austin, David, Mithun, Shaurya, Steve, Zheng, and various others (graduate students, undergraduates, and visitors) that I have befriended along the way: best of luck and godspeed. I must also acknowledge my dissertation committee: thank you for granting me your time and support. I am very grateful to NVIDIA both for the funding and opportunities afforded by the NVIDIA Graduate Fellowship, and for the fruitful collaboration with NVIDIA researcher John Tran.

This long journey to a dissertation began at home, with encouragement and support from my parents, much love from grandparents, and affectionate goads from my two younger brothers. I am proud of each of you and am grateful to have made you proud of me in turn.

From almost the very start, I have been incredibly blessed to have two extended families in St. Louis. International Friends has taught me the bittersweet reality of a world that is at once so very small and so very big. Aigula, Alicia, Allison, Andrea, Arpith, Brian, Carol, Chenyu, Chioma, Chui Li, Dale, Esha, Fernando, Foster, Hyunju, Ifeoma, Josh, Nicolas, Ophelia, Pato, Rachel, Rodger, Seth, Sharleen, Silvana, Stan, and Vernal: thank you for showing and leading me in grace, peace, and love. I can personally attest, "Where morning dawns, where evening fades, You call forth songs of joy." (Psalm 65:8)

My second St. Louis family is New City Fellowship, specifically the Koch-Woodard house church. David, Mandy, Timothy, Christopher, and Karis; Mark, Aubree, Wendy, and Lucy; Matt and Nicole; Sally; Paul and Kathy; and Erin: you are each dear to my heart and bring countless smiles to my face, "For you yourselves have been taught by God to love each other." (1 Thessalonians 4:9) As we walk together, always remember, "He will bring forth your righteousness as the light, and your justice as the noonday." (Psalm 37:6)

I cannot possibly name everyone whose support and companionship I have received while trudging this "road of happy destiny," but I am immensely grateful to have encountered each of you. This work is a testament to your friendship and support.

Stephen W. Tyree

Washington University in St. Louis

December 2014

"For us, there is only the trying. The rest is not our business."

—T. S. Eliot, "East Coker"

ABSTRACT OF THE DISSERTATION

Approximation and Relaxation Approaches for Parallel and Distributed Machine Learning

by
Stephen W. Tyree
Doctor of Philosophy in Computer Science
Washington University in St. Louis, 2014
Professor Kilian Q. Weinberger, Chair

Large scale machine learning requires tradeoffs. Commonly this tradeoff has led practitioners to choose simpler, less powerful models, e.g. linear models, in order to process more training examples in a limited time. In this work, we introduce parallelism to the training of non-linear models by leveraging a different tradeoff—approximation. We demonstrate various techniques by which non-linear models can be made amenable to larger data sets and significantly more training parallelism by strategically introducing approximation in certain optimization steps.

For gradient boosted regression tree ensembles, we replace precise selection of tree splits with a coarse-grained, approximate split selection, yielding both faster sequential training and a significant increase in parallelism, in the distributed setting in particular. For metric learning with nearest neighbor classification, rather than explicitly train a neighborhood structure we leverage the implicit neighborhood structure induced by task-specific random forest classifiers, yielding a highly parallel method for metric learning. For support vector machines, we follow existing work to learn a reduced basis set with extremely high parallelism, particularly on GPUs, via existing linear algebra libraries.

We believe these optimization tradeoffs are widely applicable wherever machine learning is put in practice in large scale settings. By carefully introducing approximation, we also introduce significantly higher parallelism and consequently can process more training examples for more iterations than competing exact methods. While seemingly learning the model with less precision, this tradeoff often yields noticeably higher accuracy under a restricted training time budget.

Chapter 1

Introduction

Large scale machine learning requires tradeoffs. Commonly this tradeoff has led practitioners to choose simpler, less powerful models, e.g. linear models, in order to process more training examples in a limited time. In this work, we introduce parallelism to the training of non-linear models by resorting to a different tradeoff—*approximation*. We demonstrate various techniques by which non-linear models can be made amenable to larger data sets and added parallelism by strategically introducing approximation in certain optimization steps. In practice, the combination of increased non-linear model power and the ability to train on more data counteract the loss of precision due to approximation. The result is higher accuracy in training time which is competitive with less powerful methods.

For gradient boosted regression tree ensembles, we consider trading off the precise selection of tree splits in favor of a coarse-grained, approximate split selection. This opens the door to both faster sequential training and a significant increase in parallelism, in the distributed

setting in particular. While the consequence is a small loss in accuracy, boosted tree ensembles are shown to be highly resilient to this approximation and significantly more parallel as a result. We leverage this method for both large scale regression (e.g. learning to rank) and non-linear metric learning.

For metric learning with nearest neighbor classification, it is common to learn a metric which explicitly produces some desired neighborhood structure. However, this is often both computationally expensive and limited in parallelism. In this work, we opt out entirely from the optimization of an explicit neighborhood structure. Instead, we leverage the implicit neighborhood structure induced by task-specific random forest classifiers. In doing so, we co-opt the highly parallel random forest training procedure to metric learning. The result is an accurate metric for nearest neighbor classification and an additional layer of interpretability for random forest predictions.

For support vector machines, previous work has shown that adopting a carefully selected reduced basis set, instead of learning weights for all support vectors, can maintain high accuracy while significantly reducing training time and model size. We leverage this existing work and demonstrate that this training also presents extremely high parallelism, particularly on GPUs. Further, the method suffers little loss in accuracy under additional approximation by choosing the basis vector set with only minor supervision or even randomly. The result is the fastest publicly-available kernel SVM training code released to date for either multicore or GPU architectures.

We believe these optimization tradeoffs are widely applicable wherever machine learning is put in practice in large scale settings. By carefully introducing approximation, we also introduce significantly higher parallelism and consequently can process more data examples for more iterations than competing exact methods. While seemingly learning the model with less precision, this tradeoff often yields *higher accuracy* under a restricted training time budget.

In the remainder of this chapter, we introduce the general problem of learning predictors from data, along with several common methods for learning non-linear predictors. Finally, we consider a variety of settings for parallel and distributed computing which will be leveraged throughout this dissertation.

1.1 Learning from Data

With many complex tasks, it is remarkably challenging to precisely describe *how* to accomplish the task, e.g. in a computer program. For example, it would be very difficult to write a computer program which takes as input the image of a handwritten digit and returns the value of the digit represented in the image. (Indeed, it is hard to convey the same task to young humans.)

Yet, for many complex problems, it is almost trivial to accumulate *examples* of the successfully completed task. For the handwritten digit recognition task, with a little time it is quite

straightforward to collect numerous examples of handwritten digits with the identity of the digit written in each. Indeed, far more humans are capable of creating or collecting such examples than have sufficient skills to write complex computer code.

With this setup, we no longer want to write a digit recognition program by hand. Rather, if we could leverage a more general, existing program—a pattern recognition method—we could present our collection of task examples and allow the general program to learn the *pattern* behind the examples in order to repeat the task on new examples. Then, the handwritten digit task becomes a data collection and labeling problem rather than a challenging reasoning and programming task!

Supervised learning. Supervised machine learning is concerned with designing pattern recognition methods. We assume we are presented with examples of the inputs to and outputs from a function of unknown form. Examples of functions of interest may include:

- the identity of a handwritten digit in an image;
- whether an email message is or is not spam; and
- the relevance of a document to a web search query string.

In each case, these functions are hard to characterize directly, yet it may be quite simple to gather examples of input/output pairs. For the previously cited examples, one could acquire a set of examples by:

- recruiting several hundred subjects and asking each to write the digits 0-9;
- recording which emails are marked spam by users of a web-based email service; and
- examining click-through rates and human evaluator scores on web search results.

Data and learning problems. In the machine learning context, a set of input and output pairs constitutes a *training set*. A machine learning algorithm attempts to approximate the unknown function underlying the training data, capturing a *model* for the function. When presented with a previously unseen input, referred to as a *test* input, the model is used to predict the function's output.

We assume training data $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ are in the form of n samples from some input distribution. The samples are captured as feature vectors $\mathbf{x}_i \in \mathcal{X} \subseteq \mathbb{R}^d$ in a d-dimensional vector space. We denote the value of feature j of sample i by the notation $[\mathbf{x}_i]_j$. Each sample $\mathbf{x}_i$ is associated with a label $y_i \in \mathcal{Y}$, an observed output of the function to be learned when queried with input $\mathbf{x}_i$. This label y_i indicates the property we would like to automatically predict about unlabeled instances.

When the set of label values is an ordered subset of the real numbers ($\mathcal{Y} \subseteq \mathbb{R}$), we refer to the problem as a *regression* problem. When the set of label values is discrete and unordered ($\mathcal{Y} \in \{1, 2, ..., c\}$), we refer to the discrete labels as *classes* and the problem as a *classification* problem. Classification problems may be binary ($|\mathcal{Y}| = 2$) or multiclass ($|\mathcal{Y}| > 2$), but

multiclass problems can be (and often are) solved as a series of binary classification problems [69, 151].

Learning as optimization. The goal is to learn a function $h : \mathcal{X} \to \mathcal{Y}$ such that $h(\mathbf{x}_i) \approx y_i$. We choose our predictor h from some restricted class of functions $\mathcal{H}$. A common approach is to optimize the predictor h with respect to a cost function $\mathcal{C}(h)$, selecting the function $\hat{h}$ which minimizes the mis-prediction cost on the available training instances:

$$\hat{h} = \operatorname*{argmin}_{h \in \mathcal{H}} \mathcal{C}(h).$$

For many cost functions, this optimization is intractable. This includes, for example, the most straightforward classification error measure, the mis-prediction count:

$$\mathcal{C}(h) = \sum_{i=1}^{n} [h(\mathbf{x}_i) \neq y_i],$$

where $[\cdot]$ denotes the Iverson bracket. Mis-prediction counts are non-continuous in the space of predictor functions, as a small change in predictor output may induce large changes in the cost function. Without a well-behaved error measure, function selection can become a combinatorial optimization problem.

One solution is to optimize h with respect to a well-behaved *surrogate* cost function $\mathcal{C}(h)$—continuous, at least once differentiable, and (ideally) convex. Surrogate cost functions commonly upper bound the desired error measure. With such a function, learning predictors in many function classes is a numerical minimization problem amenable to well-studied first- or second-order mathematical optimization methods, including gradient descent and Newton's method. One typical well-behaved cost function for regression problems is the squared-loss,

$$\mathcal{C}(h) = \sum_{i=1}^{n} (h(\mathbf{x}_i) - y_i)^2.$$

Several cost functions are common for learning classification models, including the logistic, exponential, and hinge loss functions.

1.2 Non-Linear Machine Learning Methods

Machine learning is extremely effective for a wide variety of data analysis and prediction tasks. For many complex problems, *non-linear* models have proven essential to achieving high prediction accuracy. In this context, non-linearity refers to the ability to capture effects which cannot be modeled as a linear combination of features in the input space $\mathcal{X}$. Consider a version of the "XOR-problem" as depicted in Figure 1.1(a). No linear combination of the two features, $[\mathbf{x}]_1$ and $[\mathbf{x}]_2$, can define a rule for separating the two classes. However, as

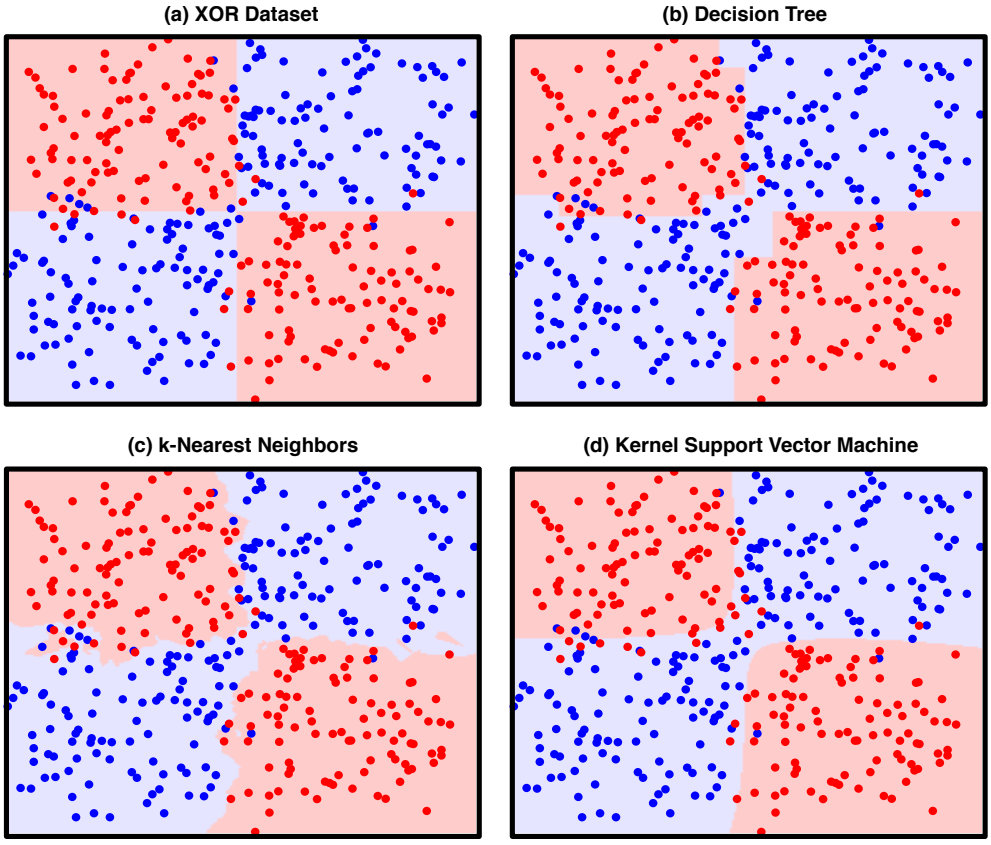

Figure 1.1: A simple noisy "XOR" dataset with two classes (red and blue dots), and the decision boundary (red and blue shading) of (a) the underlying noiseless "XOR" distribution, (b) a decision tree, (c) the k-nearest neighbors decision rule ($k = 3$), and a kernel support vector machine ($C = 1$, $\gamma = 1$).

depicted in Figure 1.1(b-d) and discussed in the following paragraphs, models learned from a variety of non-linear function spaces can easily produce reliable predictors.

Learning non-linear models often presents significant computational challenges, particularly when paired with large-scale training sets. The work in this dissertation encompasses several methods to leverage approximation in concert with parallel and distributed hardware systems

to effectively speed up and scale up machine learning with non-linear models. Here we discuss three categories of non-linear models which will be referenced in subsequent chapters.

Decision trees and tree ensembles. *Decision tree* models operate by recursively partitioning the feature space using axis-aligned splits. This input space partitioning is captured in a binary tree structure. The tree's internal nodes denote single-feature partitioning rules of the form $[\mathbf{x}]_j \leq \theta$, where j is an index in feature vector $\mathbf{x}$ and θ is a split threshold. Leaf nodes in the tree indicate prediction labels, with one constant label value corresponding to each input space partition. Trees are denoted *classification* or *regression* trees based on whether leaf predictions are class labels (or distributions over classes) or real numbers, respectively.

A prediction is made for a sample $\mathbf{x}$ by descending the tree. Following the branching rule at each internal node, the left child node is chosen when the rule evaluates true, otherwise the right child is chosen. Upon reaching a leaf node, the leaf's label is returned as the prediction.

Trees naturally capture non-linear decision boundaries, are invariant to feature scaling, and readily incorporate a variety of input spaces including categorical features. Tree training is typically accomplished via a top-down procedure which greedily chooses splits to minimize prediction error at the next level of the tree. Construction of each level of nodes typically involves a sequential pass over the training samples, considering the effect of each potential split threshold at each node. Since splits are axis-aligned, individual trees lack the ability to

easily capture smooth decision boundaries and, due in part to the greedy training procedure, can easily "overfit" to specific patterns in the training set.

Tree ensemble models use a collection of subtly varying trees to learn powerful and easily configured classifiers for many problems. Merging many trees supports smooth decision boundaries. Fitting the trees to varying views of the training dataset helps to alleviate overfitting. When trained by *gradient boosting*, trees in the ensemble are learned sequentially to correct errors made by previous trees. In *random forests*, a collection of "random" trees are learned from different random samples from the training set and their predictions averaged.

When attempting to scale to large training sets, the sequential training of gradient boosting places the burden of parallelization on the construction of individual trees. Leveraging special properties of boosted tree models, particularly short tree depth and weak assumptions on the accuracy of any individual tree, we present an extremely flexible approximate method to parallelize tree learning in both distributed and multi-core settings. Random forest training is naturally parallel as individual trees are learned on different independent samples from the training set. We leverage parallelism in these methods to perform regression and metric learning (as discussed in the next paragraph) on medium and large scale problems.

Nearest neighbor methods and metric learning. *Nearest neighbor* methods produce arguably the simplest and most interpretable non-linear classification models. Predictions are made by querying a training set for the labeled training instances which are most similar

to the unlabeled test instance, and constructing a prediction from the labels of the similar examples.

For the case of 1-nearest neighbor classification (1NN), the predicted label y_t for a test point $\mathbf{x}_t$ is the label y_i of the nearest training instance $\mathbf{x}_i \in \mathcal{D}$:

$$y_t = \underset{(\mathbf{x}_i, y_i) \in \mathcal{D}}{\operatorname{argmin}} D(\mathbf{x}_t, \mathbf{x}_i),$$

for some distance function $D : \mathcal{X} \times \mathcal{X} \to \mathbb{R}^+$, effectively capturing inverse similarity. Common choices for $D(\cdot)$ are the Euclidean distance for general vector data and the χ^2 distance for histograms or probability-vectors. In the general case of k-nearest neighbor classification (kNN), for some $k \geq 1$, classification predictions are made by a vote among the labels of the k neighbors, while regression predictions are commonly produced by some form of averaging.

The accuracy of nearest neighbor models relies heavily on the choice of similarity metric for determining which neighbors are indeed "nearest" for both the sample input space and the prediction task at hand. *Metric learning* optimizes task-specific similarity metrics to minimize kNN error under the metric. It is often to useful to reframe the metric learning problem as learning a transformation $\phi(\cdot)$ of the input feature space under which a standard metric, such as Euclidean or χ^2, yields good kNN performance:

$$y_t = \underset{(\mathbf{x}_i, y_i) \in \mathcal{D}}{\operatorname{argmin}} D_\phi(\mathbf{x}_t, \mathbf{x}_i) = \underset{(\mathbf{x}_i, y_i) \in \mathcal{D}}{\operatorname{argmin}} D(\phi(\mathbf{x}_t), \phi(\mathbf{x}_i)).$$

In its simplest form, $\phi(\cdot)$ is a linear transformation parameterized by matrix $\mathbf{L}$: $\phi(\mathbf{x}) = \mathbf{Lx}$. Learning linear transformations presents computational advantages, however for some problems the feature space cannot be adequately reshaped by simple linear manipulations. In these cases, non-linear metric learning approaches, often corresponding to learning non-linear transformations, can yield significant increases in accuracy. In this dissertation, we introduce two novel non-linear metric learning approaches, each leveraging a parallel tree ensemble approach for scalable training. Each approach yields highly competitive accuracy with no significant parameters to tune and scalability to medium and large scale training sets.

Kernel support vector machines. *Support vector machines* (SVM) learn a classifier with a large "margin" between a linear hyperplane separating the two classes of samples and the nearest training samples from each class. In the linear formulation, the SVM optimization minimizes the hinge loss with respect to a linear hyperplane parameterized by weights $\mathbf{w}$,

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||_2^2 + C\sum_{i=1}^{n} \max(0, 1 - y_i(\mathbf{w}^\top \mathbf{x}_i + b)).$$

A maximum margin is enforced by L2-regularization on the hyperplane with tradeoff C.

The "kernel trick" provides a natural non-linear extension to SVMs, implicitly learning a hyperplane in a high (or even infinite) dimensional projection of the input space. In the kernel SVM setting, we make a non-linear transformation of the input space, $\mathbf{x} \to \phi(\mathbf{x})$, akin

to the non-linear transformations discussed for metric learning. It is costly (or impossible in the case of projections into infinite-dimensional feature spaces) to explicitly represent the feature transformation and then proceed with SVM training. However, some transformations have corresponding kernel functions, permitting closed-form solutions for inner products in the transformed space, $k(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$. By solving the SVM optimization in the dual [150] or by leveraging the Representer Theorem for primal optimization [47], training samples are accessed only in inner product computations with other training samples, permitting powerful non-linear projections by simply substituting a kernel function. As a result of the kernel substitution, the hyperplane is represented as coefficients on kernel function evaluations between training samples and a test sample. Training samples corresponding to non-zero coefficients are called "support vectors."

Quadratic model complexity and limited parallelism in typical SVM optimization procedures have prevented the application of SVMs to many medium and large datasets. We review the literature of SVM solvers seeking an SVM formulation and optimization method more conducive to parallelism on modern multi-core and GPU hardware. By adopting a previously published sparse SVM approximation, we successfully implement SVM training for multi-core CPUs and GPUs, achieving significant speedup over existing approaches.

1.3 Parallel and Distributed Systems

Training speed is vital for practical machine learning settings. In many real-world applications, models are not simply trained once and reused *ad infinitum*. Rather, models are frequently retrained as new labeled data are acquired. In adversarial settings or systems whose behavior drifts over time, such as spam filtering or web search ranking, retraining may occur at a daily or even hourly frequency to account for the changing environment. Further, development and evaluation of new input features is a common process which also requires model retraining.

Initial model training can exhibit some trivial parallelism when there are hyper-parameters to tune. However, after hyper-parameter selection, a single training phase is typically engaged using the full training set. Additionally, recent developments in Bayesian hyper-parameter tuning [178] limit the need for broad searches over grids or random selections of hyper-parameters. These observations solidify the need for fast, highly parallel training.

In conjunction, trends in computer architecture have been moving toward increasingly parallel hardware. Indeed the major speedups in hardware have been almost exclusively from introducing more parallel cores rather than increasing the processing speed of individual cores. In this dissertation, we explore parallelizations of a variety of non-linear learning methods on a range of platforms.

Thread and process level parallelism is supported in multiprocessing architectures, where multiple processing cores are engaged simultaneously. In *shared memory* architectures, these cores are tightly coupled, having access to the same memory bus, and perhaps residing in the same socket and sharing some level of the cache hierarchy. Communication among processors is extremely fast, but system memory is limited by the amount of RAM configurable in a single machine.

An extreme example of shared memory multiprocessing is found in general-purpose *graphics processing units* (GPUs). GPUs were originally developed for real time rendering of complex visual scenes from underlying 3-D shape models. As such, they have many hundreds of lightweight compute cores. Unlike traditional shared memory multi-core systems, GPUs are optimized for high throughput. GPUs are based on a "same instruction multiple data" (SIMD) architecture, which requires all threads within one block to execute the exact same instructions, whereas multi-core CPUs have much fewer threads with no such restriction. Efficient GPU memory access patterns are restricted to batch memory accesses made cooperatively by multiple threads. Fast thread switching is used hide latency in memory accesses, but requires many simultaneously executing threads. These restrictions can make coding for GPUs, and more importantly optimizing execution performance on GPUs, a significant challenge. Reuse of existing patterns can lighten this development burden and significantly increase both performance and code resiliency across multiple generations of GPU architectures.

In *distributed memory* (or cluster) systems, computing cores are located in physically distinct machines and communication is directed through network connections. In this setting, interaction is distinctly slower and limited by network bandwidth. However the amount of system memory is practically unlimited and no longer bound by what is feasible to install on a single computer. This volume of memory permits tackling of significantly larger problems than are manageable with shared memory systems. However, programmability can suffer as additional considerations are introduced, including data distribution patterns, inter-node communication, and tolerance against node or network failure.

Chapter 2

Parallel Boosted Tree Ensemble Construction

Tree-based classifiers are a popular and powerful set of supervised-learning methods applicable to a wide variety of learning problems. Tree classifiers incorporate natural non-linearity and present few important hyper-parameters, two factors which yield powerful out-of-the-box performance. With ensembling, tree models often reach state-of-the-art accuracy on a variety of problems, including some which are particularly difficult for other methods (e.g. many categorical features or features with widely-varying scales).

Boosting is an ensemble method in which a single strong classifier is iteratively constructed from a sequence of "weak learners." Most commonly the weak learners are depth-limited tree classifiers. The weak learners are trained in sequence, each correcting the prediction errors made by the collection of weak-learners learned previously in the sequence. Boosted tree

classifiers have proven remarkably effective for many industrial scale problems in machine learning, including web search ranking [48] and recommender systems [123].

The sequential nature of boosted tree learning—training one tree at a time on a potentially very large training set—places the full computational burden on the tree learning procedure. Tree learning is expensive as the entire set of training data must be scanned repeatedly during the process of constructing each tree. Further, tree learning in general is non-trivial to parallelize as any parallelization strategy requires some combination of frequent synchronization or repeated data re-distribution.

Surprisingly, while both boosting and tree learning are challenging to parallelize in general, the specific combination presented by boosted regression trees is highly amenable for both parallel and distributed learning. This is due in large part to the strictly limited tree depth imposed commonly imposed in boosted tree learning.

In this chapter, we study methods for speeding up training of boosted tree ensembles by parallelizing the special case of learning depth-limited trees. We first examine the common exact method for split evaluation, which in conjunction with a *feature-wise* data distribution is well-suited for medium scale data. Subsequently, we present a novel approximate method which supports arbitrary data distribution strategies, including a more natural *instance-wise* distribution, and can scale to much larger datasets in distributed cluster and cloud settings. This method uses data compression in the form of histograms to efficiently synchronize split selection among distributed nodes. To our knowledge, this was the first work to explicitly

parallelize regression tree construction specifically tuned for the purpose of gradient boosting. The approximate method demonstrates speedups of up to 40× on 48 shared memory cores and up to 25× on 48 distributed cluster cores, while resulting in no significant loss in accuracy.

Section 2.1 provides an introduction to gradient boosting ensembles and regression tree learning. Section 2.3 details a feature-wise data distribution strategy, particularly in conjunction with exact split selection. Section 2.4 presents an approximate tree learning method supporting an instance-wise data distribution via histogram synchronization. Section 2.5 gives an experimental evaluation.

2.1 Gradient Boosted Regression Trees

In this section we first review gradient boosting [81] as a general meta-learning algorithm for function approximation. We follow this with a description of regression trees [24], tree learning procedures, and considerations for parallelization of the learning procedure.

2.1.1 Notation

We assume the data are in the form of samples $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ consisting of feature vectors $\mathbf{x}_i \in \mathbb{R}^d$ and labels $y_i \in \mathbb{R}$. The notation $[\mathbf{x}_i]_j$ denotes the j^{th} feature of sample $\mathbf{x}_i$. For example, consider a sample corresponding to a user search query and a website which

may or may not closely match the query. (This example of a web search ranking problem will be carried throughout this chapter.) The feature vector $\mathbf{x}_i$ captures characteristics of the query (e.g. language, number of words), the website (e.g. PageRank [146], language, last update time), and both parts jointly (e.g. the number of times the query terms appear in the website). Some of these features may be numerical (e.g. word counts), while others are categorical (e.g. language). In this example, the label of interest y_i is the relevance of document to its query, ranging from "irrelevant" (if $y_i = 0$) to "perfect match" (if $y_i = 4$).

Our goal is to learn a function $h : \mathbb{R}^d \to \mathbb{R}$ such that $h(\mathbf{x}_i) \approx y_i$. In cases where the label set is a continuous or ordered subset of the real numbers, we have a regression learning problem. Otherwise, when labels are drawn from a discrete, unordered set of values, a classification problem results. Continuing the search query example, we seek to learn a regression function on the relevance of queries to documents. At test time, a search engine gathers documents that provide a preliminary match to the query. Subsequently, the engine computes query specific features for this set of documents $\{\mathbf{x}_j\}_{j=1}^m$ and ranks them in decreasing order of their predicted relevance $\{h(\mathbf{x}_j)\}_{j=1}^m$.

A common approach is to optimize the prediction function h with respect to a well-behaved cost function $\mathcal{C}(h)$, selecting the function $\hat{h}$ which minimizes this mis-prediction cost on the available training instances,

$$\hat{h} = \operatorname*{argmin}_h \mathcal{C}(h).$$

One typical cost function for regression problems is the squared-loss,

$$\mathcal{C}(h) = \sum_{i=1}^{n} (h(\mathbf{x}_i) - y_i)^2.$$

We do not restrict ourselves to any particular cost function. Instead we assume that we are provided with a generic cost function $\mathcal{C}(\cdot)$, which is continuous, convex and at least once differentiable. Particularly pertinent to the web search ranking problem are a number of ranking specific cost functions [31, 30].

2.1.2 Gradient Boosting

Gradient boosting [81] is an iterative algorithm to find an additive predictor $h(\cdot)$ which minimizes a cost function $\mathcal{C}(h)$. The additive classifier $h(\cdot) \in \mathcal{H}^T$ is formed from the combination of T predictors from some class of base predictors $\mathcal{H}$. At each iteration t, a new function $g_t(\cdot)$ is added to current predictor $h_t(\cdot)$, such that after T iterations, $h_T(\cdot) = \sum_{t=1}^{T} \alpha_t g_t(\cdot)$, where $\alpha_t > 0$ is some non-negative learning rate. (Often the learning rate is constant, i.e. $\alpha_t = \alpha$ for all iterations t.)

In iteration t, gradient boosting attempts to find the function $g(\cdot)$ such that $\mathcal{C}(h_t + g)$ is minimized,

$$g = \underset{g \in \mathcal{H}}{\operatorname{argmin}}\, \mathcal{C}(h_t + g).$$

By a first-order Taylor approximation, we obtain

$$g \approx \underset{g \in \mathcal{H}}{\operatorname{argmin}} \left[\mathcal{C}(h_t) + \left\langle \frac{\partial \mathcal{C}}{\partial h_t(\cdot)}, g(\cdot) \right\rangle \right].$$

By approximating the inner-product between two functions by summing over the products of known instantiations of the functions, $\langle f(\cdot), g(\cdot) \rangle = \sum_{i=1}^{n} f(\mathbf{x}_i) g(\mathbf{x}_i)$, and dropping the constant term $\mathcal{C}(h_t)$, we obtain

$$g \approx \underset{g \in \mathcal{H}}{\operatorname{argmin}} \left[\sum_{i=1}^{n} \frac{\partial \mathcal{C}}{\partial h_t(\mathbf{x}_i)} g(\mathbf{x}_i) \right]. \tag{2.1}$$

In order to find an appropriate function $g(\cdot)$, we assume the existence of an oracle $\mathcal{O}$. For a given function class $\mathcal{H}$ and a set $\{(\mathbf{x}_i, r_i)\}$ of pairs of instance vectors $\mathbf{x}_i$ and target responses r_i, this oracle returns the function $g \in \mathcal{H}$, that yields the best least squares approximation of the response values (up to some small $\epsilon > 0$):

$$\mathcal{O}(\{(\mathbf{x}_i, r_i)\}) \approx \underset{g \in \mathcal{H}}{\operatorname{argmin}} \sum_{i} (g(\mathbf{x}_i) - r_i)^2. \tag{2.2}$$

We expand the squared term in (2.2) and assume the norm of $g \in \mathcal{H}$ is constant,[1] i.e. $\langle g, g \rangle = c$. The two quadratic terms are constants and therefore independent of g, leaving

$$\mathcal{O}(\{(\mathbf{x}_i, r_i)\}) \approx \underset{g \in \mathcal{H}}{\operatorname{argmin}} \left[\sum_i -r_i \, g(\mathbf{x}_i) \right].$$

The solution of the minimization (2.1) becomes

$$g \approx \mathcal{O}(\{(\mathbf{x}_i, r_i)\}) \text{ where } r_i = -\frac{\partial \mathcal{C}}{\partial h_t(x_i)}.$$

In the case where $\mathcal{C}(\cdot)$ is the squared-loss, $\mathcal{C}(h) = \sum_{i=1}^{n} (h(\mathbf{x}_i) - y_i)^2$, the target assignment is the current residual, $r_i = y_i - h_t(\mathbf{x}_i)$.

Algorithm 1 summarizes gradient boosted regression in pseudo-code. In many domains, including web search ranking and recommender systems, the most successful and practical choice for the oracle $\mathcal{O}(\cdot)$ in (2.2) is the greedy Classification and Regression Tree (CART) algorithm [24] with limited tree depth p. In the following section, we review regression trees and the basic CART algorithm.

[1] We can avoid this restriction on the function class $\mathcal{H}$ with a second-order Taylor expansion in (2.1). We omit the details of this slightly more involved derivation as it does not affect our algorithmic setup. However, we do refer the interested reader to [241].

Algorithm 1 Gradient Boosting

Input: data set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$
Parameters: continuous & differentiable cost function $\mathcal{C}(h)$, learning rate α_t, ensemble size T
Initialization: $r_i = y_i, \ \forall i$
$h(\cdot) = 0$
for $t = 1$ to T **do**
  $g_t \leftarrow \mathcal{O}(\{(\mathbf{x}_i, r_i\})$
  $h(\cdot) \leftarrow h(\cdot) + \alpha_t g_t(\cdot)$
  for $i = 1$ to n **do**
    $r_i \leftarrow -\frac{\partial \mathcal{C}}{\partial h_t(\mathbf{x}_i)}$
  end for
end for
return h

2.1.3 Regression Trees

Decision tree models recursively partition the input feature space, grouping similarly-labeled input samples into the same regions. Beginning with the full feature space at the root node, each internal node in the tree applies a binary axis-aligned split, dividing the feature space into two regions. A full tree of splits results in a set of non-overlapping rectangular regions, with one region corresponding to each "leaf" node in the tree. A full, balanced, binary tree model of depth p results in a partition of the input space into 2^p axis-aligned regions.

Predictions are made by traversing inputs down the tree. Beginning at the root node, an input $\mathbf{x}$ is navigated to either the left or right child by comparing with a split criterion θ on feature j at each internal node. Upon reaching a leaf node, the input is assigned a prediction label.

Partitioning is accomplished by simple decision functions at each branch. Commonly, each decision function is a "decision stump," either a threshold on a single numerical feature or

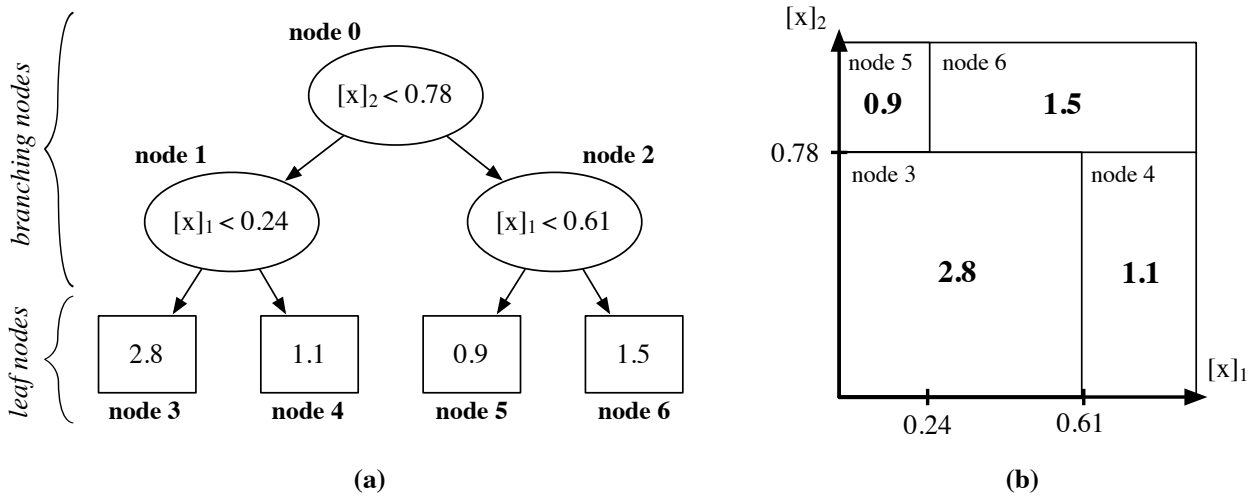

Figure 2.1: A simple regression tree (a), and the corresponding partitioning and labeling of the input feature space, $\mathbf{x} \in [0, 1]^2$ (b).

one category from a single discrete feature. For example, Figure 2.1(a) depicts a regression tree with two levels of branching nodes. The root branch sends instances to the left child node when the value of feature 2, $[\mathbf{x}]_2$, is less than the threshold $\theta_0 = 0.71$, otherwise to the right child node. Similar numerical decision stumps partition on feature 1 in the second level of branches.

After descending the tree, data samples have been partitioned into one of several sets, each set corresponding to a "leaf node" in the tree. Ideally the instances in each set can be reliably characterized by a common constant label prediction. For example, the decision tree in Figure 2.1(a) partitions the input space into four rectangular regions. These regions and the constant predictions made for each are depicted in Figure 2.1(b). For instance, all samples reaching leaf node 3 given the prediction 2.8.

Decision tree training. In supervised decision tree learning, the splits at each branch are chosen to minimize error when making a constant prediction for all samples in each leaf node region. Decision tree models where each output prediction is selected from a continuous range are commonly termed "regression trees," with "classification" trees corresponding to trees with predictions made in a discrete label space.

Learning an optimal decision tree is in general an NP-complete problem, however greedy methods are very effective in practice. Here we detail the CART algorithm [24], a simple and widely-used method for decision tree construction. Throughout, we focus on a discussion of regression tree training while highlighting issues for learning regression trees from large training sets. However, we note that classification trees may be learned by nearly identical methods.

Regression tree construction in CART [24] proceeds by selecting branch splits to greedily minimize label variance in child nodes. Let us consider how to select a split for an arbitrary branching node given training samples $\mathcal{S} \subseteq \mathcal{D}$. (If this node is the root of the tree, $\mathcal{S}$ includes all input samples in the training data, i.e. $\mathcal{S} = \mathcal{D}$.) We wish to select a split (j, θ) on a single feature $[\mathbf{x}]_j$ with split criterion θ. For notational simplicity, we assume θ is a threshold on a numerical feature, corresponding to the binary test $[\mathbf{x}]_j < \theta$, though splits on discrete features may be handled with similar simplicity.

A split (j, θ) induces a partition of the input data into two sets, the first corresponding to the left child of the branching node,

$$\mathcal{A}^{(j,\theta)} = \{(\mathbf{x}_i, y_i) \in \mathcal{S} \,:\, [\mathbf{x}_i]_j < \theta\}\,,$$

and the second corresponding to the right child,

$$\mathcal{B}^{(j,\theta)} = \mathcal{S} - \mathcal{A}^{(j,\theta)} = \{(\mathbf{x}_i, y_i) \in \mathcal{S} \,:\, [\mathbf{x}_i]_j \geq \theta\}\,.$$

The set of candidate split features j is determined by the input dimensionality d, i.e. $j \in \{1, ...d\}$. The set of thresholds θ is seemingly infinite, even for bounded features, however a finite training set only supports meaningful evaluation of a finite set of split thresholds. Given the set of unique values for a feature j, $\mathcal{Q}_j = \{[\mathbf{x}_i]_j \,\forall i\}$, there exist only $|\mathcal{Q}_j| - 1$ unique partitions $\mathcal{A}^{(j,\theta)}$ and $\mathcal{B}^{(j,\theta)}$, as all thresholds between consecutive feature values, $\left\{\theta : \mathcal{Q}_j^{(k)} < \theta < \mathcal{Q}_j^{(k+1)}\right\}$, induce the same partition. It is common in practice to use candidate thresholds chosen to be the values half-way between each consecutive feature value, i.e.

$$\left\{\frac{\mathcal{Q}_j^{(k)} + \mathcal{Q}_j^{(k+1)}}{2}\right\}_{k=1}^{|\mathcal{Q}_j|-1}.$$

Using this construction of candidate thresholds, a full set of candidate splits is

$$\mathcal{P} = \left\{ (j, \theta) : \theta \in \left\{ \frac{\mathcal{Q}_j^{(k)} + \mathcal{Q}_j^{(k+1)}}{2} \right\}_{k=1}^{|\mathcal{Q}_j|-1} \right\}_{j=1}^{d}.$$

Given a set of candidate splits $\mathcal{P}$, we select the split $(\hat{j}, \hat{\theta}) \in \mathcal{P}$ which minimizes the label variance in the two child sets, $\mathcal{A}$ and $\mathcal{B}$,

$$(\hat{j}, \hat{\theta}) = \underset{(j,\theta)\in\mathcal{P}}{\operatorname{argmin}} |\mathcal{A}| \operatorname{var}(\mathcal{A}) + |\mathcal{B}| \operatorname{var}(\mathcal{B}), \tag{2.3}$$

where

$$\operatorname{var}(\mathcal{S}) = \frac{1}{|\mathcal{S}|} \sum_{(\mathbf{x}_i, y_i)\in\mathcal{S}} (y_i - \bar{y}_\mathcal{S})^2 \quad \text{and} \quad \bar{y}_\mathcal{S} = \frac{1}{|\mathcal{S}|} \sum_{(\mathbf{x}_i, y_i)\in\mathcal{S}} y_i.$$

Solving (2.3) once corresponds to branching a single tree node into two child nodes. To build a full regression tree, we begin with the root node and recursively split by (2.3), terminating recursion whenever a child node violates a pre-specified stopping criterion, e.g. maximum tree depth or minimum number of training instances per node ($|\mathcal{S}|$).

A constant prediction is assigned to every terminal (leaf) node. Given a leaf node with input data $\mathcal{S} \subseteq \mathcal{D}$, a common predictor is the average label value of input samples in $\mathcal{S}$, previously denoted $\bar{y}_\mathcal{S}$. It is straight-forward to show that $\bar{y}_\mathcal{S}$ minimizes the squared loss among all

possible constant predictors for a set $\mathcal{S}$:

$$\bar{y}_{\mathcal{S}} = \operatorname*{argmin}_{q \in \mathbb{R}} \sum_{(\mathbf{x}_i, y_i) \in \mathcal{S}} (y_i - q)^2.$$

Dynamic programming for split evaluation. Evaluating the objective function in (2.3) for a single split (j, θ) is $\mathrm{O}\big(n\big)$ complexity as every training sample must be assigned to either the left or right child set. Evaluating a set of splits on a single feature, $\mathcal{P}_j$, can also be accomplished in $\mathrm{O}\big(n\big)$ by a simple dynamic programming schemes [24].

To demonstrate this, we begin by substituting the definition of sample variance into (2.3), expanding the quadratic terms, and dropping the constant term $\sum_{(\mathbf{x}_i, y_i) \in \mathcal{A}} y_i^2 + \sum_{(\mathbf{x}_i, y_i) \in \mathcal{B}} y_i^2$, yielding,

$$\operatorname*{argmin}_{(j,\theta) \in \mathcal{P}} \left(|\mathcal{A}| \bar{y}_{\mathcal{A}}^2 - 2\bar{y}_{\mathcal{A}} \sum_{(\mathbf{x}_i, y_i) \in \mathcal{A}} y_i \right) + \left(|\mathcal{B}| \bar{y}_{\mathcal{B}}^2 - 2\bar{y}_{\mathcal{B}} \sum_{(\mathbf{x}_i, y_i) \in \mathcal{B}} y_i \right).$$

Simplifying further by recalling the definitions of predictors $\bar{y}_{\mathcal{A}}$ and $\bar{y}_{\mathcal{B}}$, we have

$$\operatorname*{argmin}_{(j,\theta) \in \mathcal{P}} -\frac{1}{|\mathcal{A}|} \left(\sum_{(\mathbf{x}_i, y_i) \in \mathcal{A}} y_i \right)^2 - \frac{1}{|\mathcal{B}|} \left(\sum_{(\mathbf{x}_i, y_i) \in \mathcal{B}} y_i \right)^2. \tag{2.4}$$

Rewritten in this way, the optimization depends only two quantities computed from each split-induced set, $\mathcal{A}$ and $\mathcal{B}$: the number of samples in each set, $|\mathcal{A}|$ and $|\mathcal{B}|$; and the sum of the sample labels in each set, $\sum_{i=1}^{|\mathcal{A}|} y_i$ and $\sum_{i=1}^{|\mathcal{B}|} y_i$.

A simple dynamic programming scheme begins with training samples stored feature-wise in memory with each feature sorted independently. (We offer some implementation details of this scheme in Section 2.3.) In a one-time preprocessing step per tree, we count the number of samples, $|\mathcal{S}|$, and the sum of the labels, $\sum_{i=1}^{|\mathcal{S}|} y_i$, for all training samples $\mathcal{S}$ before branching at the root node. To evaluate splits for a feature j, we initialize with all samples taking the right branch, i.e. $\mathcal{A} = \emptyset$ and $\mathcal{B} = \mathcal{S}$. We have $|\mathcal{B}| = |\mathcal{S}|$ and $\sum_{(\mathbf{x}_i, y_i) \in \mathcal{B}} y_i = \sum_{i=1}^{|\mathcal{S}|} y_i$ from preprocessing.

We make a sequential pass through the sorted feature values, $\left\{[\mathbf{x}_{(1)}]_j, [\mathbf{x}_{(2)}]_j, ..., [\mathbf{x}_{(n)}]_j\right\}$. As each sample $[\mathbf{x}_{(k)}]_j$ is figuratively moved from the right side of the branch, $\mathcal{B}$, to the left, $\mathcal{A}$, we update sample counts and label sums for both sets. We stop whenever a new feature value is encountered, i.e. $[\mathbf{x}_{(k)}]_j < [\mathbf{x}_{(k+1)}]_j$. At such points, $\mathcal{A}$ and $\mathcal{B}$ correspond to the sets induced by a split (j, θ), where $[\mathbf{x}_{(k)}]_j < \theta < [\mathbf{x}_{(k+1)}]_j$. We compute the objective in (2.4) using the current label sums and set sizes, recording the current split parameters if the split yields a new minimum objective value.

The remainder of this chapter focuses on repeatedly solving (2.4) in the context of learning many short trees. We consider efficient implementations leveraging either parallel or distributed architectures and computing both exact and approximation solutions. In Section 2.3, we describe a feature-wise parallelization scheme which directly uses the basic dynamic

programming split evaluation described here. In Section 2.4, we introduce a novel approximation scheme for evaluating (2.4) on an instance-wise parallelization of the training samples, a setup very amenable to large scale distributed learning.

2.2 Related Work

Here we present a sample of previous work on parallel methods related to our work. This related work falls into two categories: parallel decision trees and parallelization of boosting.

Parallel decision tree algorithms have been studied for many years, and can be grouped into two main categories: task-parallelism and data-parallelism. Algorithms in the *task-parallelism* category [64, 181] divide a tree into sub-trees, which are constructed on different workers, e.g. after the first node is split, the two remaining sub-trees are constructed on separate workers. There are two downsides of this approach. First, each worker should either have a full working copy of the data or a large amount of data must be communicated to workers after each split. This scheme is infeasible for distributed training with large data sets, especially if the entire data set does not fit in each worker's local memory. Second, small trees, such as those commonly used in boosting approaches, are unlikely to achieve much speedup since they can only utilize as many workers as the number of nodes in a single level of the tree.

The algorithms presented in this chapter fall under the second approach [3], *data-parallelism*, where the training data are divided among different workers. Data can be partitioned by features [79], by samples [166] or both [232]. Distributing by feature requires workers to coordinate which inputs fall into which tree nodes during the construction process, since the individual workers do not have enough information to compute branching decisions using features stored by other workers. This requires communication of $O(n)$ bits for each level of the tree. We detail our approach to this method (and our open-source implementation) in Section 2.3.

Distributing the training data by samples [166] avoids this communication problem. However, in order to obtain an exact solution, all workers are required to aggregate their evaluations of each potential split point [232]. This motivates our approximate instance-wise distributed method. This approach distributes the data by samples, avoiding $O(n)$ communication and allowing significant scaling potential, particularly in distributed settings. We deliberately only approximate the exact split, making use of histograms to synchronize split evaluation across processing nodes, yielding a communication requirement which is *independent* of the data set size.

Two sample-partitioning approaches bear similarities to our work. PLANET [147] selects splits using exact, static histograms constructed in a two stage process. Implemented in the MapReduce framework, PLANET first samples histogram bin boundaries to achieve approximately uniformly-sized bins, then tallies exact data counts and label sums for each

bin. Initially we implemented a similar scheme, but later achieved better accuracy with a single stage process and our dynamic regression-oriented histograms.

Further, unlike PLANET, our implementations specifically avoids use of the MapReduce framework which is ill-suited to iterative computation. In naïve implementations of MapReduce, the internal states of the distributed "mapper" processes are not preserved between iterations. Instead, significant setup and teardown costs are incurred during each re-initialization. While this yields simplicity and robustness to node or link failures, this attribute renders many MapReduce implementations extremely inefficient for highly iterative algorithms such as tree learning. In the case of boosted tree construction, a different MapReduce iteration is required for learning of each level of nodes in each tree in the ensemble. This commonly amounts to several thousand iterations over the course of ensemble training, which in MapReduce could entail reloading the input data from disk at each iteration.

Our approximate instance-wise algorithm is most similar to Ben-Haim and Yom-Tov's work on parallel approximate construction of decision trees for classification [15]. Our histogram methods were largely inspired by their publication. However, our approach differs in several ways. First, we use regression trees instead of classification – requiring us to interpolate label values within histogram bins instead of computing one histogram per label. Further, our method explicitly parallelizes gradient boosted regression trees with a fixed small depth. The communication required for workers to exchange the feature-histograms grows exponentially with the depth of the tree. In our experiments, for trees with depth $p \geq 15$ (consisting of over

65,535 tree nodes), we saw a slowdown (instead of speedup) due to increased communication. This drastically reduces the benefit of parallelization of full decision or regression trees on large data sets, since the required tree depth grows with increasing data set size. In contrast, our framework deliberately fixes tree-depth to a small value (e.g. p between 4 and 6 with 63 to 255 tree-nodes). Unlike other tree methods, boosting addresses larger data sets by opting for additional boosting iterations rather than deeper trees, precisely fitting our approach. We will show that our approach obtains *more* speed-up on larger data sets, as the parallel scan of the data to construct histograms takes a larger fraction of the overall running time.

Most of the previous work on parallelizing boosting focuses on parallel construction of the weak learners [147] or on the original AdaBoost algorithm [127, 203] instead of gradient boosting. MultiBoost [213] combines bagging with AdaBoost, which can be performed in parallel, but inherits AdaBoost's sensitivity to noise.

2.3 Feature-wise Distribution

Parallelizing by features is perhaps the simplest approach for large scale tree learning [79]. In this setting, features in the training set are partitioned among the available processing nodes. As each processing node has access to an entire feature across all training samples, we can directly implement the efficient dynamic programming approach for split evaluation (2.4) described in Section 2.1.3.

Parallel execution proceeds as follows: Each processing node k loads and sorts a subset of the training features, $\{[\mathbf{x}_i]_j\}$ for $j \in \mathcal{J}_k$. Tree construction operates breadth first, as every processor k independently computes the best branch parameters (j_k, θ_k) for the root node based on its locally-stored features $\mathcal{J}_k$. The best local splits are exchanged among the processors and the split with minimal global cost is selected and added to each processor's local copy of the tree.

Before proceeding to compute the next level of branching nodes, the previous splits must be applied to the training data, navigating each training sample to either the left or right child of its current node. Due to the feature-wise data distribution, only one processor will have on hand the globally-best splitting feature for each new branch, necessitating another round of communication. We assign every training instance a single bit in a n-length bit-vector. Processors with a local copy of a splitting feature will set the bit for a training sample to 1 if the sample should be assigned to the right child, 0 otherwise. After this assignment vector is distributed, each local processor updates its local node assignments and proceeds to expand the next layer of the tree.

Implementing the exact split evaluation requires $\mathrm{O}\big(n \log n\big)$ preprocessing time, including sorting each feature. In practice, this time is roughly equivalent to learning a few small boosting trees. Further, it requires $2n$ memory, as the sample index must be stored alongside each feature value *for every feature* to navigate samples through each new branching node and identify the label value for each sample.

Advantages and Drawbacks. A feature-wise data distribution, as described here, has the distinct advantage of supporting both exact *and* approximate evaluation of splits. The following section describes an instance-wise distribution method, but efficient evaluation in that setting requires approximations for most problems. Meanwhile, the approximation scheme may also be applied in the feature-wise setting for added speedup.

The drawbacks of a feature-wise distribution are apparent in three areas: feature storage, communication, and robustness. First, data must be stored (or redistributed) by feature, which in many cases is the transpose of the most natural format (storing each sample vector contiguously) and could require significant communication to achieve in a distributed storage setting. Second, while communication is constant with respect to tree depth and the number of features, it scales linearly with the number of training samples (requiring one bit per sample). Third, implementations may be vulnerable to two difficulties specific to the distributed setting. High variance may be observed in processing time observed among different features if features vary significantly in the number of candidate splits. This results in a problem termed the "curse of the last reducer," where most processing nodes must idle while a small number of nodes complete their computations. Further, feature-wise distributions are significantly affected by compute node or communication link failures, potentially meaning the loss of entire features from training.

Finally, electing for exact split evaluations also present drawbacks when applied to large training sets. Preprocessing (with $\mathrm{O}\big(n \log n\big)$ complexity for sorting) and additional storage

are required to accommodate the necessary feature sorting. Further, while the training procedure can linearly scan feature values in memory, simultaneously a random memory access is needed to access the label and tree node index for the sample corresponding to each feature value. Unpredictable memory accesses such as these are extremely unfriendly to the caching schemes in modern memory architectures.

2.4 Instance-wise Distribution with Histograms

In this section, we introduce and analyze a novel method for learning gradient boosted regression trees. The method is motivated by the distributed data setting, in which individual data instances are partitioned across nodes in a cluster or cloud computing system. Learning in this setting is challenging as no individual computing node has access to all data instances, or even all instances for a particular feature.

The method introduced here is inspired by Ben-Haim and Yom-Tov's work on parallel construction of decision trees for classification [15]. We use adaptive histograms to summarize local data distributions during tree construction. This method is optimized to learn depth-limited regression trees on large datasets, making the method ideal for learning boosted regression trees in large-scale settings, such as those found in learning web ranking functions.

In our approach, the algorithm works incrementally, constructing one layer of the regression tree at a time. The data are partitioned among a set of worker nodes/processors whose efforts are organized by a master node.[2] At each step, the workers compress their portion of the data into small histograms and send these histograms to the master. The master aggregates the histograms and uses them to approximate the tree split optimization in (2.4) and compute the next layer in the regression tree. It then communicates this layer to the workers, allowing them to compute the histograms for constructing subsequent layers. Construction stops when a predefined depth is reached.

This master-worker approach with bounded communication has several advantages. First, it can be generalized to numerous platforms and implementation schemes: multicores and shared memory machines (e.g. OpenMP, MPI), clusters (e.g. MPI, MapReduce) and clouds (e.g. Amazon Elastic MapReduce [4]) with relatively little effort. Second, the data samples are partitioned among workers instance-wise and each worker only accesses its own partition of the data. Third, the amount of communication between processors is independent of the size of the distributed training set and is tunable, allowing an increase in the compression ratio, possibly at the expense of accuracy. In the distributed memory setting, this allows practically unlimited scaling in the size of the training set.

While adapted from [15], where a similar method is demonstrated for learning single, full-depth classification trees, this approach is a very natural fit for gradient boosting for two

[2]While useful conceptually, the role of the master node may be distributed among the workers in actual implementation.

reasons. First, the communication between processors at each step is proportional to the number of leaves in the current layer. Therefore, it grows exponentially with the tree depth, which is a significant drawback for full decision trees. However, regression trees used for boosting are typically very small. Second, while inaccuracies from approximate splits may be detrimental to a single tree model, in the boosting setting, the minor inaccuracies can be compensated for through a relatively small increase in the number of boosting iterations or by slightly deeper trees (which are still much too small for inter-processor communication to have a noticeable effect).

2.4.1 Setting

In this section, we describe our approach for parallelizing the construction of gradient boosted regression trees using an *instance-wise* data distribution and *approximation* of split selection. In this approach, the boosting still occurs sequentially, as we parallelize the construction of individual trees. Two key insights enable our parallelization. First, in order to evaluate a potential split point during tree construction we only need cumulative statistics about all data left and right of the split, but not the individual data instances themselves. Second, boosting does not require the weak learners to be particularly accurate. A small reduction in accuracy of the regression trees can potentially be compensated for by using more trees without affecting the accuracy of the cumulative boosted regressor.

In our method, we have a master processor and P workers. Workers may be separate compute nodes (each with one or more cores) in a distributed cluster or cloud environment or individual cores on a shared memory processor. We assume that data are partitioned by instance into P disjoint subsets, each possibly stored in a different physical location. Each worker p has access one of these subsets, $\mathcal{D}_p$, such that $\bigcup_{p=1}^{P} \mathcal{D}_p = \mathcal{D}$, $\mathcal{D}_p \cap \mathcal{D}_q = \emptyset$ for $p \neq q$ and $|\mathcal{D}_p| \approx |\mathcal{D}|/P$.

The master processor guides construction of a regression tree layer by layer. We proceed layer-wise (rather than node-wise, for instance) to optimize memory access patterns. When learning balanced trees by layer, each training instance is assigned to exactly one node and instances may be scanned in-order and in-place during the data compression phase, merely noting the node containing each instance. This procedure avoids repeatedly copying to coalesce instances by tree node or making scattered memory accesses to read just the instances belonging to a single node.

At each iteration, a new layer is constructed as follows: Each worker compresses its share of the data using histograms (as in [15]) and sends them to the master processor. The histograms capture the distribution of labels at each current tree node under the ordering of each feature. The master collects and merges the histograms, using them to approximate the best splits for each branching node, thereby constructing a new layer. The master sends the splits for this new layer (features j and thresholds θ) to each worker, and the process repeats as workers construct histograms for the new layer.

The communication consists entirely of the workers sending histograms to the master and the master sending the splits for a new layer of the tree to the workers. The amount of communication is related only to the number of nodes in the current layer, the dimensionality of the feature vectors, and granularity of the histograms (number of bins)—and is independent of the number of training instances. The size of communication does increase with the depth of the tree, but since the depth of the regression trees for gradient boosting is bounded and very small, the amount of communication is also bounded and reasonable.

To explain the details of this algorithm, we first identify the cumulative statistics that are sufficient for regression tree training. Second, we describe how we can construct histograms with a single pass over the data and use them to approximate these cumulative statistics and, consequently, approximate the best tree splits. Finally, we describe the algorithms that run on the master and workers and how we overlap computation and communication to achieve good parallel performance.

2.4.2 Cumulative Statistics

We wish to build trees from compressed summaries of distributed data. In other words, we wish to evaluate the splitting criterion (2.3) using cumulative statistics about the data set. To begin, we have a subset $\mathcal{S} \subseteq \mathcal{D}$ of training inputs and a branching node which dictates a split (j, θ) on feature j with split criterion θ (assumed for notational convenience to be a real-valued threshold). This partitions the input set $\mathcal{S}$ into disjoint sets $\mathcal{A}^{(j,\theta)} =$

$\{(\mathbf{x}_i, y_i) \in \mathcal{S} : [\mathbf{x}_i]_j < \theta\}$ and $\mathcal{B}^{(j,\theta)} = \mathcal{S} - \mathcal{A}^{(j,\theta)} = \{(\mathbf{x}_i, y_i) \in \mathcal{S} : [\mathbf{x}_i]_j \geq \theta\}$. For convenience, we will generally refer to this sets simply as $\mathcal{A}$ and $\mathcal{B}$, with the parameters of the split considered only implicitly.

Let $\ell_{\mathcal{S}}$ denote the sum of all labels and $m_{\mathcal{S}}$ the number of inputs within a set of inputs $\mathcal{S}$:

$$\ell_{\mathcal{S}} = \sum_{(\mathbf{x}_i, y_i) \in \mathcal{S}} y_i \quad \text{and} \quad m_{\mathcal{S}} = |\mathcal{S}|. \tag{2.5}$$

With this notation, the constant least squares predictors $\bar{y}_{\mathcal{A}}$ and $\bar{y}_{\mathcal{B}}$, for the left and right subsets respectively, can be expressed as

$$\bar{y}_{\mathcal{A}} = \frac{\ell_{\mathcal{A}}}{m_{\mathcal{A}}} \text{ and: } \bar{y}_{\mathcal{B}} = \frac{\ell_{\mathcal{A}}}{m_{\mathcal{A}}} = \frac{\ell_{\mathcal{S}} - \ell_{\mathcal{A}}}{m_{\mathcal{S}} - m_{\mathcal{A}}}. \tag{2.6}$$

We apply the notation from (2.5) and (2.6) to simplified split evaluation of (2.4):

$$(j^*, \theta^*) = \underset{(j,\theta) \in \mathcal{P}}{\operatorname{argmin}} - \frac{(\ell_{\mathcal{A}^{(j,\theta)}})^2}{m_{\mathcal{A}^{(j,\theta)}}} - \frac{(\ell_{\mathcal{S}} - \ell_{\mathcal{A}^{(j,\theta)}})^2}{m_{\mathcal{S}} - m_{\mathcal{A}^{(j,\theta)}}}. \tag{2.7}$$

Since $\ell_{\mathcal{S}}$ and $m_{\mathcal{S}}$ are constants for a set $\mathcal{S}$, in order to evaluate a split point on $\mathcal{S}$, we only require the values $\ell_{\mathcal{A}^{(j,\theta)}}$ and $m_{\mathcal{A}^{(j,\theta)}}$ on the set $\mathcal{A}^{(j,\theta)}$.

Here we deviate from the previous exact approach which relied on the linear time dynamic programming evaluation with sorted features. Given a procedure to efficiently estimate $\ell_{\mathcal{A}^{(j,\theta)}}$ and $m_{\mathcal{A}^{(j,\theta)}}$ for *arbitrary* splits (j, θ), we need not exhaustively consider all possible

splits $\mathcal{P}$ (at a considerable computational savings for many features) nor require a scan of *sorted* feature values. In the following sections, we introduce a novel histogram-based method for compressing training samples and efficiently estimating the cumulative statistics for evaluation of arbitrary splits.

2.4.3 Histograms

The traditional GBRT algorithm, as described in Section 2.1.3, spends the majority of its computation time evaluating split points during the creation of regression trees. We speed up and parallelize this process by summarizing label and feature-value distributions using histograms. Here we describe how a single split is selected using histogram summaries of the raw input data.

Ben-Haim and Yom-Tov [15] introduce a parallel histogram-based decision tree algorithm for classification. A histogram $\mathcal{H}_j$ summarizes the j^{th} feature of a data set $\mathcal{S}$. The histogram is a set of b tuples $\mathcal{H}_j = \{(p_1, m_1), ..., (p_b, m_b)\}$, where each tuple (p_k, m_k) summarizes a bin $\mathcal{B}_k \subseteq \mathcal{S}$ containing $m_k = |\mathcal{B}_k|$ inputs around a bin center $p_k = \frac{1}{m_k} \sum_{(\mathbf{x}_i, y_i) \in \mathcal{B}_k} [\mathbf{x}_i]_j$. In this original setting, each processor summarizes its data by generating one histogram per label.

Unlike [15], we are working in a regression setting, so we cannot have a different histogram for each label. Instead, our histograms contain triples, (p_k, m_k, r_k), where $r_k = \sum_{(\mathbf{x}_i, y_i) \in \mathcal{B}_k} y_i$ is the cumulative label value of the k^{th} bin and p_k and m_k as defined previously.

Construction. A histogram $\mathcal{H}_j$ can be built over a data set $\mathcal{S}$ in a single pass. For each input $(\mathbf{x}_i, y_i) \in \mathcal{S}$, a new bin $([\mathbf{x}_i]_j, 1, y_i)$ is added to the histogram. If the size of the histogram exceeds a predefined maximum value b^* then the two nearest bins, $\mathcal{B}_{k_1}$ and $\mathcal{B}_{k_2}$ where

$$k_1, k_2 = \underset{k_1', k_2'}{\operatorname{argmin}} |p_{k_1'} - p_{k_2'}|, \tag{2.8}$$

are merged and replaced by the bin

$$\left(\frac{m_{k_1} p_{k_1} + m_{k_2} p_{k_2}}{m_i + m_{k_2}},\, m_{k_1} + m_{k_2},\, r_{k_1} + r_{k_2} \right). \tag{2.9}$$

Entire histograms may be merged by sequentially inserting the bins of one histogram into the other and applying the same merging rule. By this method, distributed subsets of a data set may be compressed in parallel, followed by a constant time merging operation to capture the entire data set.

Interpolation. Given the compressed information from the merged histogram

$$\mathcal{H}_j = \{(p_k, m_k, r_k)\}_{k=1}^{b},$$

we would like to approximate the values needed to evaluate a split threshold (j, θ) as defined in (2.7). These values are the number of points "left" of the split, $m_{\mathcal{A}}$, and the sum of the labels "left" of the split, $\ell_{\mathcal{A}}$. If $\theta = p_k$ for some $1 \leq k \leq b$, i.e., we are evaluating a split

exactly at a centroid of a bin, we assume the points and label "mass" are evenly distributed the bin centroid. In other words, half of the points, $m_k/2$ and half of the total label sum, $r_k/2$, lie on each side of p_k. The approximations then become:

$$m_{p_k} \approx \sum_{k'=1}^{k-1} m_{k'} + \frac{m_k}{2} \quad \text{and} \quad \ell_{p_j} \approx \sum_{k'=1}^{k-1} r_{k'} + \frac{r_k}{2}. \tag{2.10}$$

If a candidate split threshold θ is not at the center of a bin, *i.e.* $p_k < \theta < p_{k+1}$, we interpolate the values of $m_{\mathcal{A}}$ and $\ell_{\mathcal{A}}$. Let us consider $\ell_{\mathcal{A}}$ first. As we already have ℓ_{p_k}, all we need to compute is the remaining relevance $\Delta = \ell_{\mathcal{A}} - \ell_{p_k} = \sum_{p_k \le [\mathbf{x}_i]_j < \theta} r_k$, so $\ell_{\mathcal{A}} = \ell_{p_k} + \Delta$. Following our assumption that the points around bins k and $k+1$ are evenly distributed, there is a total relevance of $R = \frac{r_k + r_{k+1}}{2}$ between the bin centers p_k and p_{k+1}. We assume this total relevance is evenly distributed within the area under the histogram curve between $[p_k, p_{k+1}]$. Let $a(\theta) = \int_{p_k}^{\theta} h(x)\partial x$ be the area under the curve within $[p_k, \theta]$. The sum of relevance within $[p_k, \theta]$ is then proportional to $a(\theta)/a(p_{k+1})$. We use the trapezoid method to approximate the integral $a(\theta)$ and interpolate $\ell_{\mathcal{A}}$:

$$\begin{aligned} \ell_{\mathcal{A}} &= \ell_{p_k} + \frac{a_r(\theta)}{a_r(p_{k+1})} R, \qquad (2.11) \\ \text{where:} & \\ a_r(\theta) &\approx \frac{(r_k + r_\theta)(\theta - p_k)}{2} \text{ and} \\ r_\theta &= r_k + \frac{r_{k+1} - r_k}{p_{k+1} - p_k}(\theta - p_k). \end{aligned}$$

The interpolation of $m_{\mathcal{A}}$ is analogous to (2.11), except that m_x are substituted for all r_x.

Now that we can interpolate cumulative statistics $\ell_{\mathcal{A}}$ and $m_{\mathcal{A}}$ from histograms for arbitrary split points θ, there are potentially infinite number of candidate split points. We select the set of candidate split points $\mathcal{P}$ positioned uniformly on the distribution of $[\mathbf{x}_i]_j \in \mathcal{S}$. These uniformly distributed points may be estimated by the Uniform procedure described in [15].

The use of histograms (even on a single CPU) speeds up the GBRT training time significantly, as it alleviates both the need to sort the features and the need to identify and evaluate all possible split thresholds.

In addition to an inherent speedup in tree construction, histograms allow the construction of regression trees to proceed in parallel. Worker nodes compress distributed subsets of the data into small histograms. These, when merged, represent an approximate, compressed view of an entire data set, and can be used to compute the split points. We now explain our distributed algorithm in more detail.

Distributed GBRT. A layout in pseudo-code for the master and worker are depicted in Algorithms 2 and 3. As mentioned above, the data are partitioned into P sets, one for each worker. The workers are responsible for constructing histograms and the master is responsible for merging the histograms from all workers and finding the best split points. Initially, the tree consists of a single root node. At each step, the master finds the best split for all the current leaf nodes, generating a new layer of branching nodes, and sends

Algorithm 2 Parallel CART Master

```
Parameter: maximum depth p, number of workers P
T ← ∅: initialize tree as empty node
while depth(T) < p do
  for j = 1 to d do
    for l = 1 to breadth(T) do
      {H_jl}: instantiate an empty histogram for each feature j and leaf l
      for k = 1 to P do
        receive({H_jkl}): initiate non-blocking receive for each worker k's histograms
      end for
    end for
  end for
  while ∃ incomplete({H_jkl}) do
    waitany({H_jkl}): wait for some receive to complete
    H_jl ← merge(H_jl, H_jkl): merge received histogram
  end while
  for l = 1 to breadth(T) do
    T ← T ∪ split({H_jl}): update best splits for each leaf l
  end for
  broadcast(T): send next layer of leaves to each worker
end while
return tree T
```

this new layer to the workers. The workers first evaluate each data point in their partition on the new branches, navigating it to the correct leaf node. The workers then initialize an empty histogram per feature per leaf and summarize their respective data partition in these histograms. The master collects and merges these histograms to create another layer of branches.

The number of histograms generated per iteration is proportional to the number of features times the number of leaves, $O(d \times 2^{p-1})$, where p is the current depth of the tree. As p increases, this communication requirement may become significant and overwhelm the gains made from parallelization. However, for small depth trees and sufficiently large data sets, we achieve drastic speedups since the majority of computation time is spent by the workers

Algorithm 3 Parallel CART Worker

Input: data set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$
Parameter: maximum depth p
$T \leftarrow \emptyset$: initialize tree
while depth(T) < p **do**
  $\{v_i = T(\mathbf{x}_i) : (\mathbf{x}_i, y_i) \in \mathcal{D}\}$: navigate training data $\mathcal{D}$ to leaf nodes v
  for $j = 1$ to d **do**
    $\{\mathcal{H}_{jk} : k \in \{1, ..., \text{breadth}(T)\}\}$: instantiate an empty histogram for each leaf k
    for $(\mathbf{x}_i, y_i) \in \mathcal{D}$ at leaf v_i **do**
      merge($\mathcal{H}_{jv_i}$, $([\mathbf{x}_i]_j, 1, y_i)$)
    end for
    send($\{\mathcal{H}_{jk}\}$): initiate non-blocking send for histograms for feature j and all leaves k
  end for
  $T \leftarrow$receive(T): receive next layer of tree from master
end while

in compressing the data. In addition, notice that the number of histograms does not depend on the size of the data set. Communication volume is tunable by the compression parameter b^* which sets the maximum number of bins in each histogram. At a possible sacrifice of accuracy, smaller histograms may be used to limit communication.

When using such master/worker message passing algorithms, it is important to consider two objectives in order to achieve good performance. First, the computation and communication should be overlapped so processors do not block useful computation while waiting for communication. Second, the number of messages sent should not be excessive since initializing communication carries some fixed cost. Our implementation is designed to carefully balance these two objectives. In order to overlap communication and computation, the workers compute histograms feature by feature, sending these histograms to the master while they move on to the next feature. In order to allow this with just one pass over the data, we store our data feature-wise in local memory, that is, the values of a particular feature for

all instances are stored contiguously. To avoid generating unnecessarily small messages, all the histograms corresponding to a particular feature, across all leaves, are sent as a single message. That is, instead of sending one message per histogram, the worker generates one message per feature. Therefore, the number of messages does not increase with depth, even though the size of messages does increase.

Advantages and Drawbacks. We consider three properties of the instance-wise distributed method which lend themselves to scalability in training on large datasets. First, while communication increases exponentially with tree depth, it is constant with respect to the number of training samples. In the GBRT setting, tree depth is constant and small, while training sets in distributed settings may be very large, allowing significant scaling.

Second, the instance-wise distribution strategy is both a more natural configuration for distributed data and enables significantly more robustness than a feature-wise distribution. In the event of sporadic failure of compute nodes or communication links, training may proceed on the remaining processors. The failures represent the loss of only a small fraction of the training samples, rather than the omission of entire features. This also reduces the impact of the "curse of the last reducer" effect since processors are unlikely to have significantly varying workloads, assuming roughly uniformly-sampled data sets at each processor. Further, it should be noted that the approximation strategy can easily be applied data samples which

stored feature-wise or with a mixture of feature- and instance-wise distribution, making this approach significantly more flexible than previous methods.

Third, in addition to the elimination of the feature-wise sorting preprocessing and the additional memory usage entailed therein, training samples may be compressed in a single in-order linear pass without the need for out-of-order accesses to sample labels and node indices, as encountered during the dynamic programming procedure for computing exact splits. These sequential memory access patterns are well suited to the optimizations present in modern memory hierarchies.

The primary disadvantage of this procedure is the approximation itself, which may lead to sub-optimal splits and weaker regression trees. As noted previously, this could be a significant disadvantage for learning single decision trees. However, in the gradient boosted ensemble setting, sub-optimal splits are mitigated in part by the boosting procedure, as errors are corrected with each iteration. With subtle modifications, including marginal increases in the number of boosting iterations or tree depth, we demonstrate no significant impact on accuracy due to the approximation.

2.5 Experimental Results

In this section, we describe the empirical evaluation of our parallel and distributed GBRT methods using two publicly available web search ranking data sets. We see impressive

speedups on both shared memory and distributed memory machines. In addition, we found that, while the individual regression trees are weaker using our approximate instance-wise method (as expected), with appropriate parameter settings, the final gradient-boosted ensembles lose little or no accuracy compared to those trained with an exact implementation.

2.5.1 Web Search Ranking

We focus our evaluation on problems from the web search ranking domain due to the widespread success of gradient boosted tree ensembles in this area as well as the availability of several large-scale, real-world data sets. Here we provide a brief background on web search ranking from a machine learning perspective.

Document retrieval was traditionally based on manually designed ranking functions. However, in recent years, web search ranking has been recognized as a supervised machine learning problem [28, 241], where each query-document pair is represented by a high-dimensional feature vector and its label indicates the document's degree of relevance to the query.

Fueled in part by the publication of real-world data sets from large corporate search engines [134, 48], machine learned web search ranking has become one of the great success stories of machine learning. Researchers have applied many different learning paradigms to web-search ranking data, including neural networks [28], support vector machines [116], random forests [22, 142] and gradient boosted regression trees [241]. Among these various

approaches, gradient boosted regression trees arguably define the current state-of-the-art: In the Yahoo Labs *Learning to Rank Challenge 2010* [48], the largest web-search ranking competition to date, *all* eight winning teams (out of a total of 1055) used approaches that incorporated GBRT.

A web ranking data set consists of a set of web documents and user queries. Each query-document pair is represented with a set of features which are generated using properties of both the query and the document. In addition, each pair is labeled, indicating how relevant the document is to the query. Using this data, the goal is to learn a regressor so that given a new query we can return the most relevant documents in decreasing order of predicted relevance. (Our algorithmic setup is not affected by the number of queries. Therefore, to simplify notation, we assume that all documents belong to a single query throughout the following sections. However, the techniques work for training sets with multiple queries, as are our evaluation data sets, and for boosting cost functions which optimize ranking within a query.)

For an instance $(\mathbf{x}_i, y_i)$, the label y_i indicates how relevant document is to its query, ranging from "irrelevant" (if $y_i = 0$) to "perfect match" (if $y_i = 4$). A document is represented by a d dimensional vector of features $\mathbf{x}_i$ computed from the document and the query. This vector typically consists of three parts:

Query-feature vector consists of features that depend only on the query and have the same value across all the documents in the document set. Examples of such features are the number of terms in the query, whether or not the query is the name of a person, etc.

Document-feature vector consists of features that depend only on the document and have the same value across all the queries in the query set. Examples include the number of inbound links pointing to the document, the amount of anchor-text (in bytes) for the document, the language of the document, etc.

Query-document feature vector consists of features that depend on the relationship between the query and the document. Examples are the number of times each term in the query appears in the document, the number of times each term in the query appears in the anchor-text of the document, etc.

Our goal is to learn a regressor $h : \mathbb{R}^d \to \mathbb{R}$ such that $h(\mathbf{x}_i) \approx y_i$. At test time, the search engine ranks the documents $\{\mathbf{x}_j\}_{j=1}^m$ of a new query in decreasing order of their predicted relevance $\{h(\mathbf{x}_j)\}_{j=1}^m$. The quality of a particular predictor $h(\cdot)$ is measured by specialized ranking metrics. The most commonly used metrics are Expected Reciprocal Rank (ERR) [45], which is based on a simple probabilistic model of user behavior, and Normalized Discounted Cumulative Gain (NDCG@k) [114], which heavily emphasizes accuracy on the k leading results. (we use NDCG@10 throughout the evaluation, which we denote simply as NDCG.) However, these metrics can be non-convex, non-differentiable or even non-continuous. Although some recent work [235, 184, 46] has focused on optimizing these ranking metrics

directly, the more common approach is to optimize a well-behaved surrogate cost function $\mathcal{C}(h)$ instead, assuming that this cost function mimics the behavior of these other metrics.

In general, the cost functions $\mathcal{C}$ can be put into three categories of ranking: pointwise [81], pairwise [104, 241] and listwise [37]. In pointwise settings the regressor attempts to approximate the label y_i of a document $\mathbf{x}_i$ directly, *i.e.* $h(\mathbf{x}_i) \approx y_i$. A typical loss function is the squared-loss,

$$\mathcal{C}(h) = \sum_{i=1}^{n} (h(\mathbf{x}_i) - y_i)^2.$$

The pairwise setting is a relaxation of pointwise functions, where pairs of points are considered. It is no longer important to approximate each relevance score exactly, rather the partial order of any two documents should be preserved. An example is the cost function of GBRANK [241],

$$\mathcal{C}(h) = \sum_{(i,j)\in\mathcal{Q}} \max(0, 1 - (h(\mathbf{x}_i) - h(\mathbf{x}_j)))^2,$$

where $\mathcal{Q}$ is the preference set of all document pairs (i, j) belonging to the same query, where i should be preferred over j. Listwise approaches [37] are similar to the pairwise approach, but focus on all the documents that belong to a particular query and tend to have slightly more complicated cost functions. Related research [130] also focuses on breaking the ranking problem into multiple binary classification tasks.

	Yahoo LTRC		*MSLR MQ2008 Folds*				
Training Set	Set 1	Set 2	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
Features	700	700	136	136	136	136	136
Documents	473,134	34,815	723,412	716,683	719,111	718,768	722,602
Queries	19,944	1266	6000	6000	6000	6000	6000
Avg. Doc per Query	22.7	26.5	119.6	118.4	118.9	118.8	119.4
Test Set	Set 1	Set 2	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
Documents	165,660	103,174	241,521	241,988	239,093	242,331	235,259
Queries	6983	3798	6000	6000	6000	6000	6000
Avg. Doc per Query	22.7	26.2	119.8	120.0	118.5	120.2	116.6

Table 2.1: Statistics of the Yahoo and Microsoft Learning to Rank data sets.

2.5.2 Data Sets

For our empirical evaluation, we use the two data sets from Yahoo! Inc.'s *Learning to Rank Challenge 2010* [48], and the five folds of Microsoft's LETOR [134] dataset. Each of these sets come with predefined training, validation and test sets. Table 2.1 summarizes the statistics of these data sets. The training sets range in size from $\sim$35,000 to $\sim$725,000 samples and incorporate between 136 and 700 features.

2.5.3 Experimental Setup

We conducted experiments on a parallel shared memory machine and a distributed memory cluster. The shared memory machine is an AMD Opteron 1U-A1403 48-core SMP machine with four sockets containing AMD Opteron 6168 Magny-Cours processors. The distributed memory cluster consists of 8-core, Nehalem based computing nodes running at 2.73GHz.

Each node has 24GB of RAM. For our experiments, we used up to 6 of these nodes (for a total of 48 cores).

We implemented the algorithm using MPI [177], which has the advantage of supporting efficient operation in both shared memory and distributed systems with the same library. We make the code available[3] under an open source license. We compare accuracy against the exact GBRT implementation[4] described in [142].

We opt not to use the somewhat simpler Map/Reduce framework [147] as the framework is often not well suited to highly iterative computations. In most implementations, "mapper" processes are instantiated and destroyed between iterations, which in our case would require expensive reading of the training set between each level of the tree. Rather, with MPI we are able to maintain the internal state of each process throughout training.

For simplicity, we used the squared-loss as our cost-function $\mathcal{C}(\cdot)$ in all experiments, though our methods are easily adaptable to other continuous and differentiable loss functions. Our algorithm has four parameters: The depth of the regression trees p, the number of boosting iterations m, the step-size α and the maximum number of bins b in the histograms. We perform experiments on the sensitivity of these parameters on both Yahoo Set 1 and 2, as these span two different ranges of data set sizes (Set 1 is almost one order of magnitude larger than Set 2).

[3] `http://research.engineering.wustl.edu/~tyrees/`
[4] `http://research.engineering.wustl.edu/~amohan/`

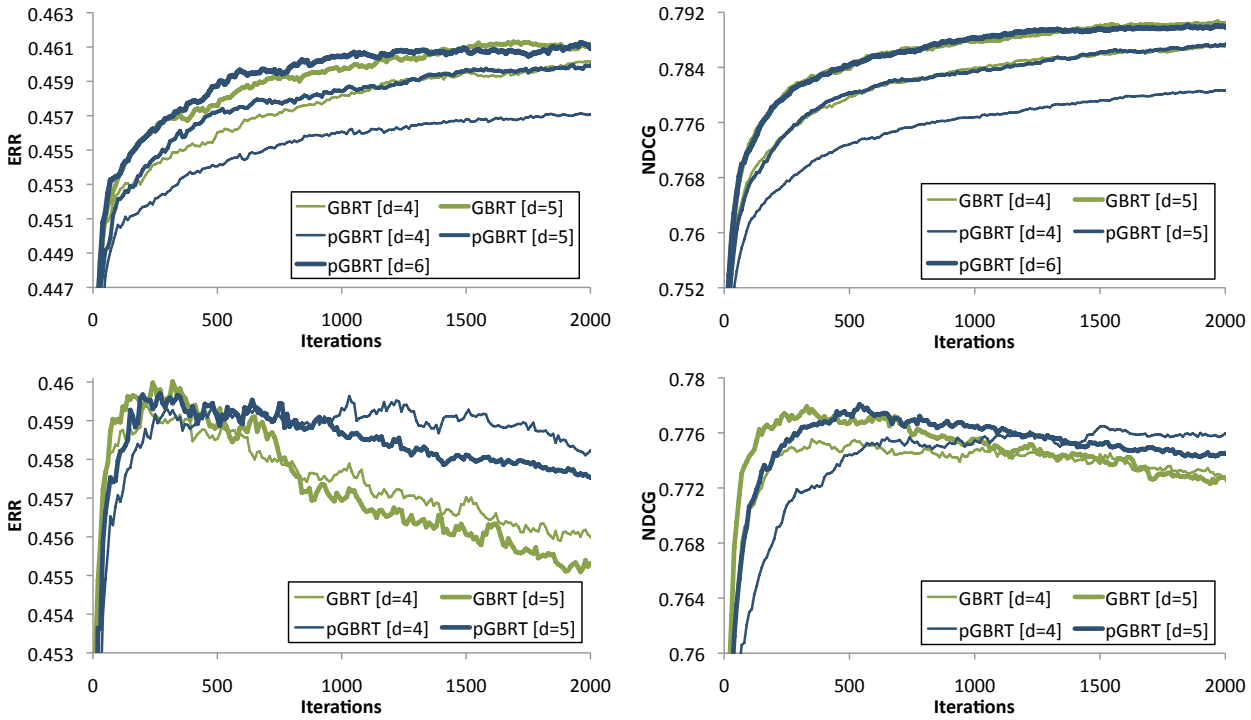

Figure 2.2: ERR and NDCG for Yahoo Set 1 (*top*) and Yahoo Set 2 (*bottom*) on approximate parallel (pGBRT) and exact (GBRT) implementations with various tree depths p. The NDCG plot for Set 1 *(top right)* shows nicely that pGBRT with a tree depth of $p+1$ leads to results similar to the exact algorithm with depth p.

2.5.4 Prediction Accuracy

As a first step, we investigate how much the ranking performance, measured in ERR [45] and NDCG [114], is impacted by approximate construction of the regression trees. Figure 2.2 shows the ERR and NDCG of the approximate, instance-wise implementation ("pGBRT") and of the exact algorithm ("GBRT") as a function of the number of boosting iterations on

ERR	Yahoo LTRC		MSLR MQ2008 Folds				
method	Set 1	Set 2	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
GBRT (p=4)	0.461	0.458	0.361	0.358	0.355	0.367	0.373
pGBRT (p=4)	0.458	0.459	0.346	0.341	0.342	0.343	0.357
pGBRT (p=5)	0.460	0.460	0.355	0.348	0.355	0.353	0.367
pGBRT (p=6)	0.461	0.460	0.355	0.354	0.357	0.363	0.367

NDCG	Yahoo LTRC		MSLR MQ2008 Folds				
method	Set 1	Set 2	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
GBRT (p=4)	0.789	0.765	0.495	0.493	0.484	0.498	0.500
pGBRT (p=4)	0.782	0.743	0.474	0.469	0.466	0.473	0.479
pGBRT (p=5)	0.785	0.754	0.483	0.479	0.479	0.484	0.491
pGBRT (p=6)	0.785	0.760	0.486	0.484	0.482	0.491	0.495

Table 2.2: Results in ERR and NDCG on the Yahoo and Microsoft data sets. The number of boosting iterations is selected with the validation data set. On both Yahoo sets, pGBRT matches the result of GBRT with $p = 4$ when the tree depth is increased. For the Microsoft sets, the ranking results tend to be slightly lower.

the Yahoo Set 1 and 2 under varying tree depths. For the approximate parallel implementation, we used $b = 25$ bins for Set 2 and $b = 50$ for the much larger Set 1. The step-size was set to $\alpha = 0.06$ in both cases.

As expected, the histogram approximation reduces the accuracy of the weak learners. Consequently, with equal depth and iterations, pGBRT has lower ERR and NDCG than the exact GBRT (higher scores are better). However, we can compensate for this effect by either running additional iterations or increasing the depth of the regression trees. In fact, it is remarkable that on Set 1 (Figure 2.2) the NDCG curves of pGBRT with $p = 6$ and $p = 5$ align almost perfectly with the curves of GBRT with $p = 5$ and $p = 4$, respectively. For Set 2 the lines are mostly shifted by approximately 200 iterations. We will see that additional

computation required by either of these approaches (increasing p or m) is more than compensated for by the increase in parallel or distributed performance afforded by the histogram method. (For small depths $p \leq 10$ – while the computation is dominated by computation – the running time increases roughly linearly with increasing p. On the Yahoo Set 1, training pGBRT with $m = 6000$ trees on 16 CPUs and depth $p = 5$ was only a factor 1.34 slower than $p = 4$ and a depth of $p = 6$ slowed the training time down by a factor of 1.75.)

Table 2.2 shows the test set results on all data sets for GBRT with $p = 4$ and pGBRT for $p = 4, 5, 6$. The number of trees m was picked with the help of the corresponding validation data sets. As the table shows, for all data sets, the difference between pGBRT with $p = 6$ and GBRT with $p = 4$ is in the third significant digit for all data sets. For the two Yahoo data sets, pGBRT provides slightly better accuracy, while for the Microsoft data sets, the exact algorithm is slightly better. The Microsoft data sets were run with parameters $\alpha = 0.1$, $b = 100$ and $m \leq 5000$. We increased the number of bins since the data set is larger.

We also evaluated the sensitivity of the algorithm to the number of histogram bins b and the number of processors. Figure 2.3 shows several runs ($\alpha = 0.05, p = 5$) with varying numbers of histogram bins assessed by quality measures which have been scaled by the best observed accuracy for each measure. For each run we report the best result as selected on the validation data set. We see that while the prediction accuracy increases slightly as the number of bins increases, it converges quickly at about 15 bins, and the differences thereafter are insignificant. In a similar experiment (not shown) we measured prediction

accuracy while varying the number of processors. Despite more histograms being inexactly merged with each additional processor, we did not observe any noticeable drop in accuracy as the number of processors increased.

To demonstrate that our algorithm is competitive with the state-of-the-art, we selected the parameters m, p, and b by cross-validation on the validation sets of both Yahoo data sets (for simplicity we fixed $\alpha = 0.06$, as GBRT is known to be relatively insensitive to the step-size). This yielded test set ERR of 0.4614 on Set 1 ($m = 3926$, $p = 7$, $b = 100$) and 0.4596 on Set 2 ($m = 3000$, $p = 5$, $b = 50$). Both results are almost identical to the best results of the exact GBRT algorithm (under slightly different optimized settings). The ERR score of Set 1 would have placed our result 15^{th} (and 14^{th} for Set 2) on the leaderboard of the 2010 Yahoo Learning to Rank Challenge[5] out of a total of 1055 competing teams. This result – despite our simple squared-loss cost function – is only 1.4% below the top scoring team, which used an ensemble of specialized predictors and fine-tuned cost functions that explicitly approximate the ERR metric [31].

2.5.5 Performance and Speedup

For performance measurements, we trained pGBRT for $m = 250$ trees of depth $p = 5$ using histograms with $b = 25$ bins. Figure 2.4 shows the speedup of our pGBRT algorithm on both the Yahoo and the LETOR Fold 1 while running on the shared memory machine. For

[5] `http://learningtorankchallenge.yahoo.com/ leaderboard.php`

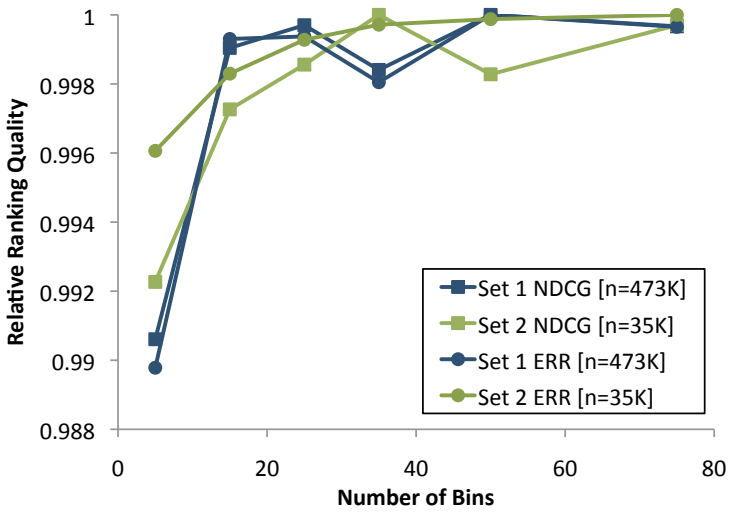

Figure 2.3: Ranking performance of pGBRT on the Yahoo Set 1 and 2 as a function of varying number of histogram bins b. With $b \geq 20$ both metrics are less than a factor 0.004 away from the best value.

the smaller data set (Set 2), we achieve speedup of up to 10× on 13 cores. For the larger data set (Set 1), we achieve much higher speedups, up to 33× on 41 processors, reducing the training time on Set 1 from over 11 hours to merely 21 minutes. On the Microsoft data (almost twice as many samples as Yahoo Set 1), we see the speedup of up to 42× on 48 cores, and there is potential for more speedup on more cores since the curve has yet to asymptote. We see more speedup on the Microsoft data since it has fewer features (requiring less communication per iteration) and more documents (increasing the fraction of time spent on histogram construction). While Yahoo Set 1 and LETOR are among the largest publicly available data sets, proprietary data sets are much larger, and we would expect further speedup with larger scales.

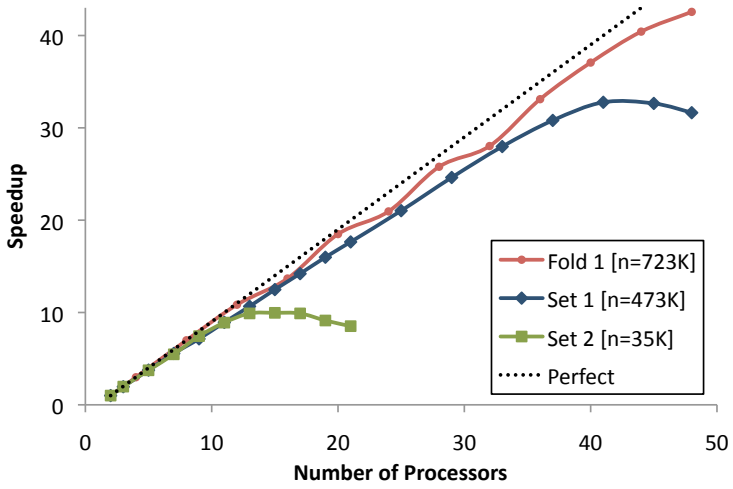

Figure 2.4: The speedups of pGBRT on a multicore shared memory machine as a function of cpu cores. The speedup increases with data set size and is almost perfectly linear for Yahoo Set 1 and the Microsoft LETOR data set. The latter set could potentially obtain even higher speedup with more cores.

Figure 2.5 shows the speedup of our parallel GBRT on both the Yahoo and the LETOR datasets while running on the cluster. As expected, the speedup is smaller on this distributed memory setup due to communication latency. However, we still see speedup of about 20× with Yahoo! Set 1 and about 25× with the Microsoft data on 32 cores, after which point the performance flattens out.[6] This result demonstrates the generality of our parallelization methods in that the same strategy (and even the same code) can provide impressive speedups on a variety of parallel machines.

All speed-up results are reported relative to the sequential pGBRT version (1 helper CPU). We do not report speedup compared to the exact algorithm, since this codebase uses different data structures and timing results might not be representative. In general, however, the

[6]We see some performance irregularities (in the form of zigzags) for the LETOR data set. We suspect these are due to caching and memory bandwidth effects.

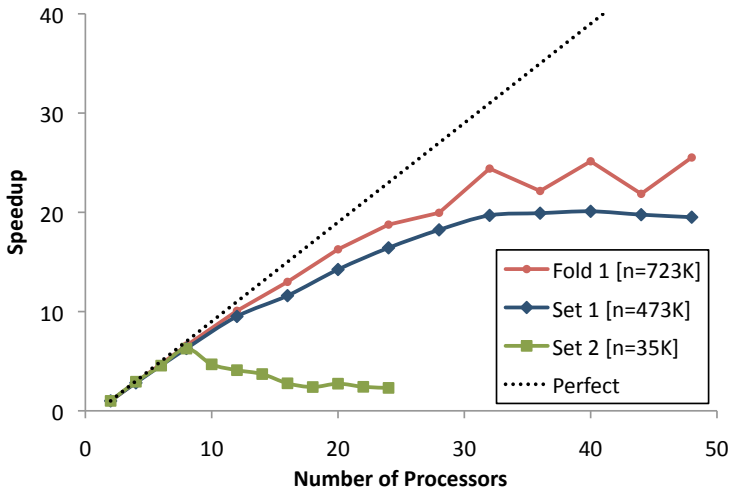

Figure 2.5: The speedups of pGBRT on a distributed memory cluster as a function of CPU cores. We observe up to 25× speedups in this distributed setting for the Microsoft LETOR data set.

speed-up with respect to the 1-CPU pGBRT runs understate the speedup over the exact algorithm, since the exact algorithm available to us is considerably slower than pGBRT, even on a single processor. For comparison, "sequentially" (1 helper CPU), our approximate parallel algorithm completes execution in 3178s on Yahoo Set 2 and 43,189s on Set 1, both with depth $p = 5$. Even with a smaller depth $p = 4$, the exact GBRT implementation takes 5940s on Set 2 and 259,613s on Set 1. Particularly for Set 1, the exact algorithm is about 6× slower than the approximate algorithm even when running with a smaller depth.[7]

[7]Exact implementations with clever bookkeeping [88] may be faster – however, no large-scale implementations were openly available. We expect, nonetheless, that our algorithm can be optimized to perform, on a single CPU, comparably or faster than optimized exact implementations, since it does not require feature sorting and evaluations significantly fewer candidate splits.

2.6 Discussion

We have presented parallel algorithms for training gradient boosted regression trees. To our knowledge, this is the first work that explicitly parallelizes the construction of regression trees for the purpose of gradient boosting. Our approach utilizes the facts that gradient boosting is known to be robust to the classification accuracy of the weak learners and that regression trees are of strictly limited depth. We have shown that our approximate approach provides impressive (almost linear) speedups on several large-scale web-search data sets without any significant sacrifice in accuracy.

Our approximate instance-wise method applies to both multicore shared-memory systems and distributed setups in clusters and clouds (e.g. Amazon EC2) using the same implementation. The distributed setup makes our method particularly attractive to real-world settings with very large data sets. Since each processor only needs enough physical memory for its partition of the training set, and communication is strictly bounded, this allows the training of machine-learned ranking functions on web-scale data sets with standard, off-the-shelf computer hardware and readily available cloud computing solutions (e.g. Amazon EC2). Further, unlike feature-wise parallelizations, the histogram binning method method can be applied to data samples distributed by instance, feature, or both without requiring redistribution. As such, we can now reasonably take our algorithm to the training data, rather than demanding that the data first be reshaped or regrouped to accommodate the algorithm.

This work can potentially extend in several directions. First, when scalability is more paramount than training time, workers could be situated in more loosely connected cloud environments, rather than dedicated clusters. Latency would increase, but added flexibility and cost-effectiveness could result. Second, a more aggressive speed/accuracy tradeoff can be pursued in the computation of the splits based on stochastic approximations of the histograms or static histogram binning strategies. Third, more efficient use of communication bandwidth could result from local split evaluations, wherein histograms for only the most promising splits are shared among distributed nodes. Further, if distributed merging were adopted in place of the master/worker paradigm, partially merged histograms could be dropped or held up if they seem unlikely to yield the best split. Such strategies could be most beneficial in cluster and cloud environments.

Given the current trend toward multicore processors, parallel computing and larger data sets, we expect our algorithm to increase in both relevance and utility in the foreseeable future.

Chapter 3

Parallel Non-Linear Metric Learning with Boosted Tree Ensembles

Similarity metrics are critical to achieving high accuracy with nearest neighbor classifiers and other distance-based machine learning approaches. Much work has been done in learning *linear* feature transformations to improve similarity metrics. Limited work in *non-linear* metric learning has shown noticeable improvements in accuracy for many problems, but the solutions are often difficult to train for medium and large scale datasets.

In this chapter, we examine two novel methods for learning non-linear similarity metrics. Both methods are built on parallel tree ensembles, inheriting the natural nonlinearity of tree models. The first technique, presented in Section 3.2, is a natural extension to the popular Large Margin Nearest Neighbor (LMNN) metric using gradient boosted regression trees for non-linearity. This method leverages the parallel gradient boosted tree learning speedups

detailed in the previous chapter. The second method, described in Section 3.3, extracts a metric from a trained tree ensemble model, in this case a random forest model, quantifying similarity in predictions made by the individual trees in the random forest. By eschewing an explicit neighborhood optimization in favor of an implicit notion of similarity, this method adapts an existing, highly parallel supervised learning technique to the purpose of metric learning.

3.1 Introduction

Defining similarity between examples is a fundamental problem in machine learning, underlying numerous learning methods including nearest neighbor classification and clustering. If an algorithm could perfectly determine whether two examples were semantically similar or dissimilar, many subsequent machine learning tasks would become trivial (*i.e.*, a nearest neighbor classifier will achieve theoretically optimal results).

Similarity is commonly addressed by computing a distance between examples in a feature vector space and using the distance as a measure of *dissimilarity*. A common choice for dissimilarity measure is an uninformed norm, such as the Euclidean distance, where $D_{ij} = ||\mathbf{x}_i - \mathbf{x}_j||_2$ is the Euclidean dissimilarity between examples $\mathbf{x}_i$ and $\mathbf{x}_j$. Although convenient and easy-to-use, the "true" semantic meaning of similarity is both data and task dependent and therefore may not be easily captured by such uninformed distance metrics.

One *data-dependent* factor that may prevent an uninformed norm from rendering a useful measure of similarity is feature scaling. Suppose a feature corresponds to the height of an individual. If the feature is changed from units of meters to millimeters then the effect of differences in height on dissimilarity between individuals is significantly more pronounced.

Furthermore, similarity is often *task-dependent*. Consider the problem of clustering a set of text documents. One analyst may desire to group documents by characteristics of the writing, e.g. fiction or nonfiction; prose, poetry or play; verbose or succinct style; comedic or dramatic approach. Another may wish to cluster by topic, e.g. love, war, philosophy or science. Given the orthogonal nature of their respective tasks, each should use a different metric to measure document similarity, despite operating on the same data.

The prevalence of similarity-based machine learning techniques motivated a surge of research in *metric learning*. Much of this work focuses on learning Mahalanobis metrics [66, 90, 92, 171, 217, 224]. The Mahalanobis metric can be viewed as a generalized Euclidean metric,

$$D_{ij} = D_{\mathbf{L}}(\mathbf{x}_i, \mathbf{x}_j) = \|\mathbf{L}(\mathbf{x}_i - \mathbf{x}_j)\|_2, \tag{3.1}$$

parameterized by a linear transformation matrix $\mathbf{L} \in \mathbb{R}^{d \times d}$. Mahalanobis metric learning optimizes the matrix $\mathbf{L}$ such that the distance D_{ij} under the metric better represents similarity in the target domain. One significant benefit of Mahalanobis metrics is the linear

transformation $\mathbf{x} \rightarrow \mathbf{L}\mathbf{x}$ yields an explicit feature representation under which the Euclidean distance corresponds to the learned metric.

The resulting Mahalanobis-based methods greatly improve the performance of metric dependent algorithms and have gained popularity in many research areas and applications both within and beyond machine learning. Reasons for this success include the out-of-the-box usability and robustness of several popular methods, computationally efficient solvers to learn these linear metrics, and the explicit representation which results.

However, linear transformations are limited by their inability to capture complex *non-linear* relationships within the data. Numerous non-linear approaches to metric learning have been proposed [54, 113, 196, 216]. These have demonstrated improved accuracy for supervised similarity-based tasks. However, existing non-linear approaches have not managed to replicate the success of the popular linear approaches. Although more expressive, the corresponding optimization problems are often expensive to solve and plagued by sensitivity to many hyper-parameters.

Approaches based on kernels [113, 196] and locally-linear metrics [216] only provide distances, while lacking a naturally corresponding data representation—making the algorithms inapplicable as generic pre-processing tools in many applications. Ideally, we would like to develop easy-to-use black-box algorithms that learn new data representations for the use of established metrics.

In this chapter, we introduce two novel non-linear metric learning approaches. The approaches robustly learn non-linear similarity measures using tree ensembles. In part by leveraging the approaches of the previous chapter, these methods are highly parallel and fast to train and evaluate on a variety of hardware platforms. Further, as tree-based methods, they enjoy natural non-linearity, insensitivity to feature scaling, and straightforward handling of discrete features.

The first non-linear approach is an extension to the popular Large Margin Nearest Neighbors (LMNN) framework [217]. Gradient boosted LMNN (GB-LMNN) employs a non-linear mapping combined with a traditional Euclidean distance function. It is a natural extension of LMNN from linear to non-linear mappings. By training the non-linear transformation directly in function space with gradient boosted regression trees (GBRT) [84] the resulting algorithm inherits the positive aspects of GBRT—robustness against overfitting, speed, and parallelism in both training [201] and evaluation.

The second approach extracts a measure of similarity from a trained random forest of decision trees. This method meets two objectives. First, it produces a highly competitive similarity measure which often replicates the high classification accuracy of the random forest while training with trivial parallelism. Further, it renders random forest classifiers more interpretable by returning "certificates" with each test prediction — training instances treated similarly to the test instance by the random forest.

3.2 Nonlinear LMNN with Gradient Boosting

In this section, we introduce a novel non-linear extension to the Large Margin Nearest Neighbors (LMNN) framework [217]. Gradient boosted LMNN (GB-LMNN) learns a non-linear mapping in combination with a traditional Euclidean distance function. It is a natural extension of LMNN from linear to non-linear mappings. By training the non-linear transformation directly in function space with gradient-boosted regression trees (GBRT), [84] the resulting algorithm inherits the positive aspects of GBRT—its insensitivity to hyper-parameters, robustness against overfitting, speed and natural parallelism in both training [201] and evaluation.

GB-LMNN scales naturally to medium-sized data sets, can be optimized using standard techniques and only introduces a single additional hyper-parameter. Its efficacy is demonstrated on several real-world data sets. We observe that GB-LMNN (with default settings) achieves state-of-the-art k-nearest neighbor classification errors with high consistency across all of our evaluation data sets. For learning tasks where non-linearity is not required, it reduces to LMNN as a special case. On more complex data sets it reliably improves over linear metrics and matches or out-performs previous work on non-linear metric learning.

3.2.1 Background

Let $\{(\mathbf{x}_i, y_i)\}_{i=1}^n \subseteq \mathbb{R}^d \times \mathcal{Y}$ be labeled training data with discrete labels $\mathcal{Y} = \{1, \ldots, c\}$. The k-nearest neighbors (kNN) [60] classification rule relies heavily on the underlying metric, since a test input is classified by a majority vote among the labels of its k nearest neighbors. Performance of kNN on classification tasks is therefore a good indicator of the quality of the metric in use.

Large margin nearest neighbors (LMNN) [216, 217] is an algorithm to learn a Mahalanobis metric specifically to improve the classification error of kNN. The Mahalanobis metric can be viewed as a straightforward generalization of the Euclidean metric,

$$D_{\mathbf{L}}(\mathbf{x}_i, \mathbf{x}_j) = \|\mathbf{L}(\mathbf{x}_i - \mathbf{x}_j)\|_2, \tag{3.2}$$

parameterized by a matrix $\mathbf{L} \in \mathbb{R}^{d \times d}$, which in the case of LMNN is learned such that the linear transformation $\mathbf{x} \to \mathbf{L}\mathbf{x}$ better represents similarity in the target domain. In the remainder of this section we briefly review the necessary terminology and basic framework behind LMNN and deferring to [217] for more details.

Local neighborhoods. LMNN identifies two types of neighbor relationships between an input $\mathbf{x}_i$ and other inputs $\mathbf{x}_j$ in the data set: As a first step, k dedicated *target neighbors* are identified for each $\mathbf{x}_i$ prior to learning. These are the inputs which *should ideally* be the actual nearest neighbors after applying the transformation (we use the notation $j \rightsquigarrow i$ to

indicate that $\mathbf{x}_j$ is a target neighbor of $\mathbf{x}_i$). A common heuristic for choosing target neighbors for a given $\mathbf{x}_i$ is picking the k closest inputs $\mathbf{x}_j$ (according to the Euclidean distance) which share the same class, i.e. $y_i = y_j$.

The second type of neighbors are *impostors*. These are inputs that *should not* be among the k-nearest neighbors of $\mathbf{x}_i$ — defined to be all inputs from a *different* class that are within the local neighborhood, i.e. among the k inputs nearest to $\mathbf{x}_i$.

LMNN optimization. The LMNN objective has two terms, one for each neighbor relationship: First, it reduces the distance between an instance and its target neighbors, thus pulling them closer and making the input's local neighborhood smaller. Second, it moves *impostor neighbors* (i.e. differently labeled inputs) farther away so that the distances to impostors should exceed the distances to target neighbors by a large margin. Weinberger et. al [217] combine these two objectives into a single unconstrained optimization problem:

$$\min_{\mathbf{L}} \quad \sum_i \sum_{j:j\rightsquigarrow i} \underbrace{D_{\mathbf{L}}(\mathbf{x}_i, \mathbf{x}_j)^2}_{\textit{pull} \text{ target neighbor } \mathbf{x}_j \text{ closer}} \quad + \ \mu \quad \underbrace{\sum_{k \ : \ y_i \neq y_k} \left[\ 1 + D_{\mathbf{L}}(\mathbf{x}_i, \mathbf{x}_j)^2 - D_{\mathbf{L}}(\mathbf{x}_i, \mathbf{x}_k)^2 \ \right]_+}_{\textit{push} \text{ impostor } \mathbf{x}_k \text{ away, beyond target neighbor } \mathbf{x}_j \text{ by a large margin } \ell} . \tag{3.3}$$

The parameter μ defines a trade-off between the two objectives and $[x]_+$ is defined as the hinge-loss $[x]_+ = \max(0, x)$. The optimization (3.3) can be transformed into a semidefinite program (SDP) [217] for which a global solution can be found efficiently. The large margin in (3.3) is set to 1 as its exact value only impacts the scale of $\mathbf{L}$ and not the resulting kNN classifier.

Dimensionality reduction. As an extension to the original LMNN formulation, [196, 216] show that with $\mathbf{L} \in \mathbb{R}^{r \times d}$ with $r < d$, LMNN learns a projection into a lower-dimensional space $\mathbb{R}^r$ that still represents domain specific similarities. While this low-rank constraint breaks the convexity of the optimization problem, significant speed-ups [216] can be obtained when the kNN classifier is applied in the r-dimensional space — especially when combined with special-purpose data structures [233].

3.2.2 Related Work

There have been some previous attempts to generalize learning linear distances to nonlinear metrics. A nonlinear mapping $\mathbf{x} \rightarrow \phi(\mathbf{x})$ can be implemented with kernels [51, 85, 113, 196]. These extensions have the advantages of maintaining computational tractability as convex optimization problems. However, they do not learn an explicit representation of the data and their utility is limited by the sizes of kernel matrices. Weinberger et. al [216] propose M^2-LMNN, a locally linear extension to LMNN. They partition the space into multiple regions, and jointly learn a separate metric for each region—however, these local metrics do not give rise to a global metric and distances between inputs from different regions are not well-defined.

Neural network-based approaches offer the flexibility of learning arbitrarily complex nonlinear mappings [54]. However, they often demand high computational expense, not only in

parameter fitting but also in model selection and hyper-parameter tuning. Of particular relevance to our GB-LMNN work is the use of boosting ensembles to learn distances between bit-vectors [8, 167]. Note that their goals are to preserve distances computed by locality sensitive hashing to enable fast search and retrieval. Ours are very different: we alter the distances *discriminatively* to minimize classification error.

3.2.3 Non-linear Transformations with Gradient Boosting

Affine transformations preserve collinearity and ratios of distances along lines — *i.e.*, inputs on a straight line remain on a straight line and their relative distances are preserved. This can be too restrictive for data where similarities change locally (*e.g.*, because similar data lie on non-linear sub-manifolds). Chopra et al. [54] pioneered non-linear metric learning, using convolutional neural networks to learn embeddings for face-verification tasks. Inspired by their work, we propose to optimize the LMNN objective (3.3) directly in function space with gradient boosted CART trees [84]. Combining the learned transformation $\phi(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}^d$ with a Euclidean distance function has the capability to capture highly non-linear similarity relations. It can be optimized using standard techniques, naturally scales to large data sets, while only introducing a single additional hyper-parameter in comparison with LMNN.

Generalized LMNN. To generalize the LMNN objective (3.3) to a non-linear transformation $\mathbf{x} \to \phi(\cdot)$, we denote the Euclidean distance after the transformation as

$$D_\phi(\mathbf{x}_i, \mathbf{x}_j) = \|\phi(\mathbf{x}_i) - \phi(\mathbf{x}_j)\|_2, \tag{3.4}$$

which satisfies all properties of a well-defined pseudo-metric in the original input space. To optimize the LMNN objective directly with respect to D_ϕ, we substitute D_ϕ for D_L in (3.3). The resulting unconstrained loss function becomes

$$\mathcal{L}(\phi) = \sum_i \sum_{j:\ j \rightsquigarrow i} \|\phi(\mathbf{x}_i) - \phi(\mathbf{x}_j)\|_2^2 \ + \ \mu \sum_{k:\ y_i \neq y_k} \big[\ 1 + \|\phi(\mathbf{x}_i) - \phi(\mathbf{x}_j)\|_2^2 - \|\phi(\mathbf{x}_i) - \phi(\mathbf{x}_k)\|_2^2 \big]_+ . \tag{3.5}$$

In its most general form, with an unspecified mapping ϕ, (3.5) unifies most of the existing variations of LMNN metric learning. The original linear LMNN mapping [217] is a special case where $\phi(\mathbf{x}) = \mathbf{L}\mathbf{x}$. Kernelized versions [51, 85, 196] are captured by $\phi(\mathbf{x}) = \mathbf{L}\psi(\mathbf{x})$, producing the kernel $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j) = \psi(\mathbf{x}_i)^\top \mathbf{L}^\top \mathbf{L} \psi(\mathbf{x}_j)$. The embedding of Globerson and Roweis [91] corresponds to the most expressive mapping function $\phi(\mathbf{x}_i) = \mathbf{z}_i$, where each input $\mathbf{x}_i$ is transformed independently to a new location $\mathbf{z}_i$ to satisfy similarity constraints — without extension to out-of-sample data.

GB-LMNN. The previous examples vary widely in expressiveness, scalability, and generalization, largely as a consequence of the mapping function ϕ. It is important to find the right

non-linear form for ϕ, and we believe an elegant solution lies in gradient boosted regression trees.

Our method, termed GB-LMNN, learns a global non-linear mapping. The construction of the mapping, an ensemble of multivariate regression trees selected by gradient boosting [84], minimizes the general LMNN objective (3.5) directly in function space. Formally, the GB-LMNN transformation is an additive function $\phi = \phi_0 + \alpha \sum_{t=1}^{T} h_t$ initialized by ϕ_0 and constructed by iteratively adding regression trees h_t of limited depth p [24], each weighted by a learning rate α. Individually, the trees are weak learners and are capable of learning only simple functions, but additively they form powerful ensembles with good generalization to out-of-sample data. In iteration t, the tree h_t is selected greedily to best minimize the objective upon its addition to the ensemble,

$$\phi_t(\cdot) = \phi_{t-1}(\cdot) + \alpha h_t(\cdot), \quad \text{where } h_t \approx \operatorname*{argmin}_{h \in \mathcal{T}^p} \mathcal{L}(\phi_{t-1} + \alpha h). \tag{3.6}$$

Here, $\mathcal{T}^p$ denotes the set of all regression trees of depth p. The (approximately) optimal tree h_t is found by a first-order Taylor approximation of $\mathcal{L}$. This makes the optimization akin to a steepest descent step in function space, where h_t is selected to approximate the negative gradient g_t of the objective $\mathcal{L}(\phi_{t-1})$ with respect to the transformation learned at the previous iteration ϕ_{t-1}. Since we learn an approximation of g_t as a function of the training data, sub-gradients are computed with respect to each training input $\mathbf{x}_i$, and approximated

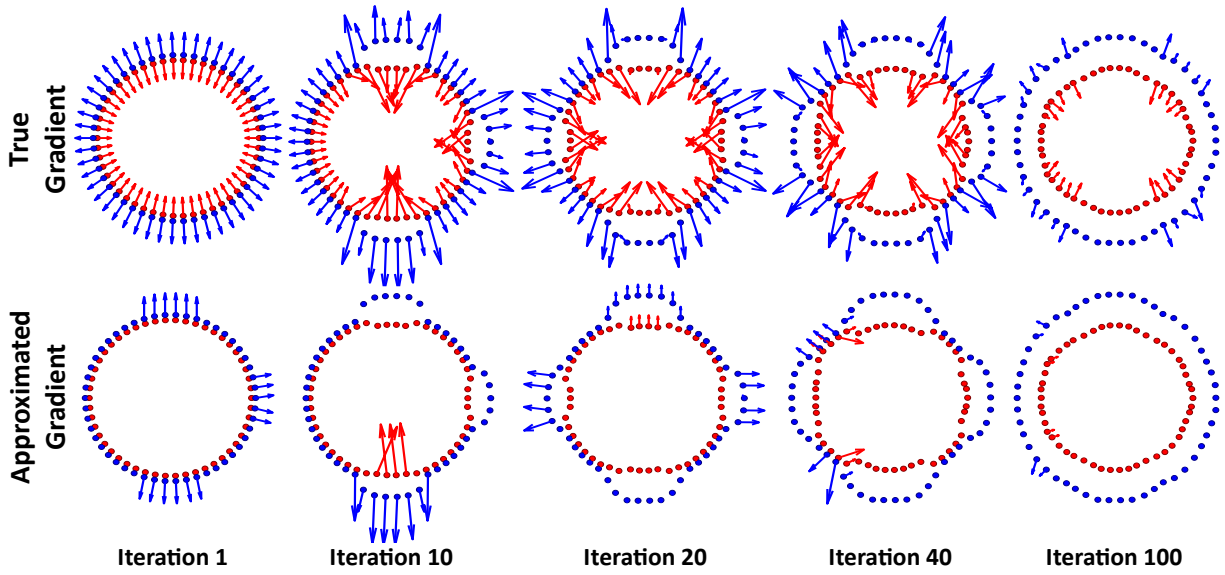

Figure 3.1: GB-LMNN illustrated on a toy data set sampled from two concentric circles of different classes (blue and red dots). The figure depicts the true gradient (*top row*) with respect to each input and its least squares approximation (*bottom row*) with a multi-variate regression tree (depth, $p=4$).

by the tree $h_t(\cdot)$ in the least-squared sense,

$$h_t(\cdot) = \underset{h \in \mathcal{T}^p}{\operatorname{argmin}} \sum_{i=1}^{n} (g_t(\mathbf{x}_i) - h_t(\mathbf{x}_i))^2, \text{ where: } g_t(\mathbf{x}_i) = \frac{\partial \mathcal{L}(\phi_{t-1})}{\partial \phi_{t-1}(\mathbf{x}_i)}. \tag{3.7}$$

Intuitively, at each iteration, the tree $h_t(\cdot)$ of depth p splits the input space into 2^p axis-aligned regions. All inputs that fall into one region are translated by a constant vector — consequently, the inputs in different regions are shifted in different directions. We learn the trees greedily with a modified version of the public-domain CART implementation pGBRT [201][8].

[8]We slightly modified the implementation described in Chapter 2 to optimize multi-variate regression by learning trees with vector outputs.

Optimization details. Since (3.5) is non-convex with respect to ϕ, we initialize with a linear transformation. In this case, we learned an initial transformation with LMNN, $\phi_0 = \mathbf{Lx}$, making our method a non-linear refinement of LMNN. The only additional hyperparameter to the optimization is the maximum tree depth p to which the algorithm is not particularly sensitive (we set $p=6$).[9]

Figure 3.1 depicts a simple toy-example with concentric circles of inputs from two different classes. By design, the inputs are sampled such that the nearest neighbor for any given input is from the other class. A linear transformation is incapable of separating the two classes. However GB-LMNN produces a mapping with the desired separation. The figure illustrates the actual gradient (top row) and its learned approximation (bottom row). The limited-depth regression trees are unable to capture the gradient for all inputs in a single iteration. But by greedily focusing on inputs with the largest gradients or groups of inputs with the most easily encoded gradients, the gradient boosting process additively constructs the transformation function. At iteration 100, corresponding to a boosted ensemble with 100 trees, the gradients with respect to most inputs vanish. This indicates that a local minimum of $\mathcal{L}(\phi)$ is almost reached. We observe that inputs from the two classes are separated by a large margin.

Dimensionality reduction. Like linear LMNN, it is possible to learn a non-linear transformation to a lower dimensional space, $\phi(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}^r$, $r \leq d$. Initialization is made with

[9]Here, we set the step-size, a common hyper-parameter across all variations of LMNN, to $\alpha = 0.01$.

the rectangular matrix output of the dimensionality-reduced LMNN transformation, $\phi_0 = \mathbf{Lx}$ with $\mathbf{L} \in \mathbb{R}^{r \times d}$. Training proceeds by learning trees with r- rather than d-dimensional outputs.

3.2.4 Experimental Results

We evaluate our non-linear metric learning algorithm against several competitive methods. The effectiveness of learned metrics is assessed by kNN classification error. The open-source implementation of GB-LMNN is available the most recent version of LMNN at `http://www.cse.wustl.edu/~kilian/code/lmnn/lmnn.html`.

We compare the non-linear global metric learned by GB-LMNN to three linear metrics: the Euclidean metric and metrics learned by LMNN [217] and Information-Theoretic Metric Learning (ITML) [66]. Both optimize similar discriminative loss functions. We also compare to the metrics learned by Multi-Metric LMNN (M^2-LMNN) [216]. M^2-LMNN learns $|\mathcal{Y}|$ linear metrics, one for each input label.

We evaluate these methods and GB-LMNN on several medium-sized data sets: *ISOLET*, *USPS* and *Letters* from the UCI repository [78]. ISOLET and USPS have predefined test sets, otherwise results are averaged over 5 train/test splits (80%/20%). A hold-out set of 25% of the training set[10] is used to assign hyper-parameters and to determine feature

[10]In the case of *ISOLET*, which consists of audio signals of spoken letters by different individuals, the hold-out set consisted of one speaker.

	isolet	usps	letters	dslr	webcam	amazon	caltech
	$n=7797$ $d=617$	$n=9298$ $d=256$	$n=20000$ $d=16$	$n=157$ $d=800$	$n=295$ $d=800$	$n=958$ $d=800$	$n=1123$ $d=800$
Euclidean	8.4±0.0	6.2	6.0±0.2	60.6±3.1	43.8±1.7	33.7±0.7	53.8±1.3
ITML	5.3±0.0	5.7	6.0±0.2	**25.0±3.0**	**12.4±1.6**	31.6±1.2	52.2±2.1
LMNN	**1.5±0.1**	2.6	3.8±0.3	28.9±1.6	15.8±3.0	31.8±1.4	**50.9±1.4**
M^2-LMNN	**1.4±0.1**	**2.5**	3.8±0.2	27.4±2.1	15.7±3.2	**31.2±1.1**	51.5±1.5
GB-LMNN	**1.4±0.0**	**2.5**	**1.9±0.1**	**22.9±2.7**	**12.4±0.9**	**29.6±1.7**	**49.8±1.0**

Table 3.1: kNN classification error (in %, ± standard error where applicable) for linear and nonlinear metric learning methods. Best results up to one standard error in **bold**. Datasets are ordered by increasing number of training examples.

pre-processing (*i.e.*, feature-wise normalization). We set $k=3$ for kNN classification, following [217].

Table 3.1 reports the means and standard errors of each approach (standard error is omitted for data with pre-defined test sets), with numbers in bold font indicating the best results up to one standard error.

On all three datasets, GB-LMNN outperforms methods of learning linear metrics. This shows the benefit of learning nonlinear metrics. On *Letters*, GB-LMNN outperforms the second-best method M^2-LMNN by significant margins. On the other two, GB-LMNN is as good as M^2-LMNN.

We also apply GB-LMNN to four smaller datasets with *histogram* data. The results are displayed on the right side of the table. These datasets are popularly used in computer vision for object recognition [159]. Data instances are 800-bin histograms of visual codebook entries. There are ten common categories to the four datasets and we use them for multiway classification with kNN.

None of the methods evaluated here is specifically adapted to histogram features. Especially linear models, such as LMNN and ITML, are expected to fumble over the intricate similarities that such data types may encode. As shown in the table, GB-LMNN consistently outperforms the linear methods and M^2-LMNN.

		isolet	usps	dslr	webcam	amazon	caltech
$r=10$	PCA	26.2	10.1	**42.8±3.7**	**32.7±1.6**	49.1±2.2	63.8±1.1
	LMNN	4.2±0.1	6.0	56.1±2.0	38.1±2.5	**43.6±4.6**	**54.6±2.1**
	M^2-LMNN	4.3±0.2	**5.2**	56.1±2.0	38.4±2.7	**42.8±1.7**	**55.0±2.2**
	GB-LMNN	**3.7±0.0**	5.3	46.7±7.4	34.6±2.6	**41.6±2.7**	**55.8±2.1**
$r=20$	PCA	15.1	6.6	**46.1±3.7**	**27.3±1.7**	43.9±1.1	59.9±0.5
	LMNN	**2.1±0.1**	3.8	53.3±2.8	34.0±2.9	39.9±1.5	55.4±1.7
	M^2-LMNN	**2.1±0.2**	**3.3**	53.3±2.8	34.3±2.6	40.3±1.3	55.5±1.5
	GB-LMNN	**2.0±0.1**	3.8	50.0±3.4	33.0±2.8	**38.7±0.8**	**53.7±1.3**
$r=40$	PCA	11.0	6.0	**46.7±3.0**	**29.2±2.2**	43.1±1.6	57.7±0.5
	LMNN	1.5±0.0	3.2	51.7±0.7	36.8±2.0	**39.4±1.0**	56.1±1.5
	M^2-LMNN	**1.2±0.1**	3.2	51.7±0.7	36.2±1.3	**39.4±1.3**	56.1±1.6
	GB-LMNN	1.4±0.1	**2.9**	50.0±2.1	31.7±1.3	**39.3±1.3**	**53.3±1.4**
$r=80$	PCA	9.4	6.1	**39.4±1.8**	39.4±1.8	46.0±1.1	69.4±3.9
	LMNN	**1.6±0.1**	3.2	51.1±2.4	36.5±2.8	43.4±0.9	60.3±0.8
	M^2-LMNN	**1.6±0.0**	**1.8**	51.1±2.4	35.9±2.7	43.4±1.0	**54.4±1.5**
	GB-LMNN	**1.6±0.1**	2.4	50.0±1.9	**27.3±3.5**	**41.1±1.2**	**54.1±1.3**

Table 3.2: kNN classification error (in %, ± standard error where applicable) with dimensionality reduction to output dimensionality r. Best results up to one standard error in **bold**.

Dimensionality reduction. GB-LMNN capable of performing dimensionality reduction. We compare with three dimensionality reduction methods (PCA, LMNN, and M^2-LMNN) on the histogram datasets and the larger UCI datasets. Each dataset is reduced to an output dimensionality of $r = 10, 20, 40, 80$ features. As we can see from the results in Table 3.2, it is fair to say that GB-LMNN performs comparably with LMNN and M^2-LMNN (We do

not apply dimensionality reduction to *Letters* as it already lies in a low-dimensional space ($d = 16$).)

Hyperparameter Sensitivity. One of the most compelling aspects of the GB-LMNN method is that it introduces only a single new hyper-parameter to the LMNN framework, regression tree depth p. In the previous experiments, we use a fixed tree depth $p = 6$. Here we explicitly examine its effect on the learned metric. Figure 3.2 compares depths $4-7$ for several of the datasets evaluated previously. The figure depicts the ratio of kNN classification error for each depth setting to the kNN error of linear LMNN. GB-LMNN appears to be largely insensitive to tree depth within this range.

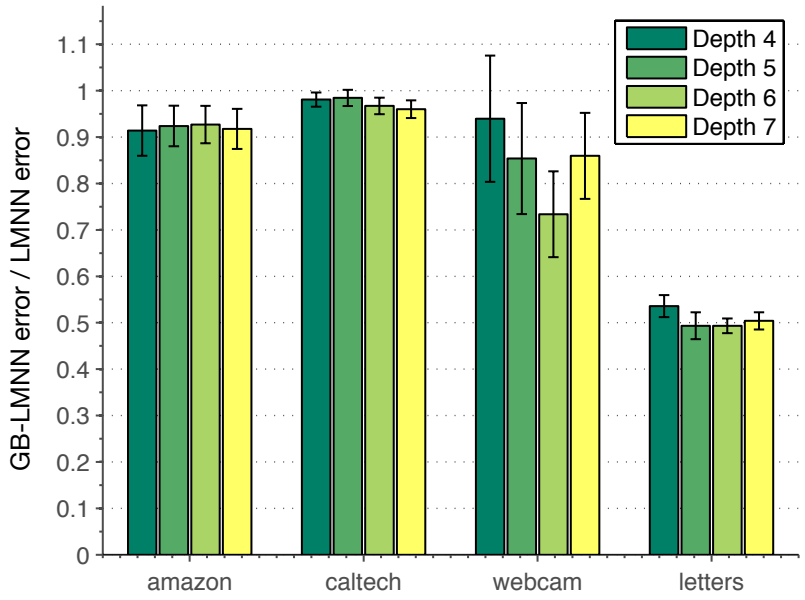

Figure 3.2: Sensitivity of GB-LMNN to the tree depth parameter. Bar height represents the ratio between GB-LMNN error and LMNN error (lower is better).

3.2.5 Discussion

In this section, we introduced GB-LMNN, a non-linear extension to LMNN. This method leverages the parallel GBRT training methods developed in the previous chapter to yield a fast, scalable non-linear metric learning algorithm. GB-LMNN significantly improves over the original (linear) LMNN metric and matches or out-performs existing non-linear algorithms.

The high consistency with which GB-LMNN obtains state-of-the-art results across diverse data sets is very encouraging. In fact, the use of ensembles of CART trees [24] not only inherits all positive aspects of gradient boosting (robustness, speed and insensitivity to hyper-parameters) but is also a natural match for metric learning. Each tree splits the space into different regions and translates inputs within these regions along different directions. In contrast to prior work, such as M^2-LMNN [216], this splitting is fully automated, results in new (discriminatively learned) Euclidean representations of the data and gives rise to well-defined pseudo-metrics.

3.3 Random Forest Ensemble Metrics

In this section, we consider the application of another tree ensemble method—random forests—to metric learning. This work extracts a similarity metric from a trained random

forest classifier. For distance-based methods (e.g. nearest neighbor classification and clustering), the metric produces a powerful non-linear similarity measure which is also highly parallel both to learn and to evaluate. Additionally, from the random forest perspective, this metric yields predictions with similar accuracy to those of random forests but with straightforward interpretability.

Interpretability. As machine learning makes its way into more and more high-impact applications and decision-making processes, interpretability is becoming an important criterion when selecting a machine-learning algorithm. In many domains, among several options which each produce highly accurate predictions, the method that yields the most insight into its decision will likely be most valuable to practitioners. Interpretability allows the user to understand the reasons for a prediction. Interpretable predictions are particularly paramount for hard problems, where even the best machine learning methods may frequently be inaccurate.

Consider an example from a medical domain. Suppose a system is designed to predict when a hospital patient may require transfer to intensive care [9]. The system may accurately convey advanced warning of a decline in the patient's condition, but this prediction may be of little value to the physician if not accompanied with insights that can lead to an appropriate treatment plan. However, if the system were to return the profiles of similar

patients while highlighting the relevant predictive attributes, the physician would be more informed to make relevant interventions to forestall the decline in condition.

Nearest neighbor classifiers (kNN; [60]) provide this type of interpretability with unparalleled simplicity. The kNN decision rule classifies an instance by a majority vote of the k most similar labeled instances. These k neighbors can be returned alongside the predicted label as "certificates", efficiently explaining the decision.

Metric learning. The k neighbors are chosen as the closest instances under some metric, often defaulting to the Euclidean metric. Achieving high accuracy from a kNN predictor requires this metric to reflect the underlying semantic similarity for the problem at hand. When instances are described by a diverse set of features, there arise issues of feature scaling, discrete features, and non-linear feature interactions.

In the last decade, learning metrics for kNN classification has yielded substantial improvements in kNN accuracy for many problems. Much of this work has focused on learning Mahalanobis metrics [92, 66, 217], corresponding to learning a *linear* transformation of the input feature space. Linear metrics are often insufficient for problems with complex feature interactions, motivating the development of many non-linear metrics [163, 160, 51, 85, 113, 119]. Although successful in accuracy, many of these non-linear approaches often lack the same compelling simplicity of corresponding Mahalanobis metric learning algorithms. Further, they tend to easily overfit [216] or be sensitive to kernel or hyper-parameter selection [119, 196].

While work on metric learning has significantly improved the state-of-the-art for kNN classification, its accuracy still often lags behind, for instance, ensemble classifiers such as random forests (RF; [22]). RFs are invariant to feature scaling, naturally handle discrete features, and learn non-linear feature interactions to yield very accurate classifiers. Moreover, RFs are easy to train as they have no significant hyperparameters and are very robust against overfitting. This leaves practitioners with the dilemma of choosing between the interpretability of kNN and the higher accuracy and usability of methods like RF.

In this section, we propose a (pseudo-)metric designed to leverage both—replicating the high accuracy and ease to train of RFs, while yielding interpretable predictions under the kNN decision rule. From a practical standpoint our Random Forest Ensemble Metric (RFEM) is extremely straightforward: distances between instances are measured by the average similarity of their predictions as made by each tree in the RF. Yet in this simple design lies a powerful metric. In our evaluation on a diverse group of *eleven* datasets, RFEM achieves the best accuracy on *eight*, beating existing (non-)linear metric-learning algorithms. RFEM inherits the benefits of RFs: unparalleled insensitivity to hyperparameters, invariance to feature scaling, straightforward handling of discrete features, natural non-linearity, and no requirement for a "seed" metric to set target neighbors (as required by *e.g.* [119]). We take advantage of specific RFEM properties to prove bounds for fast neighbor search.

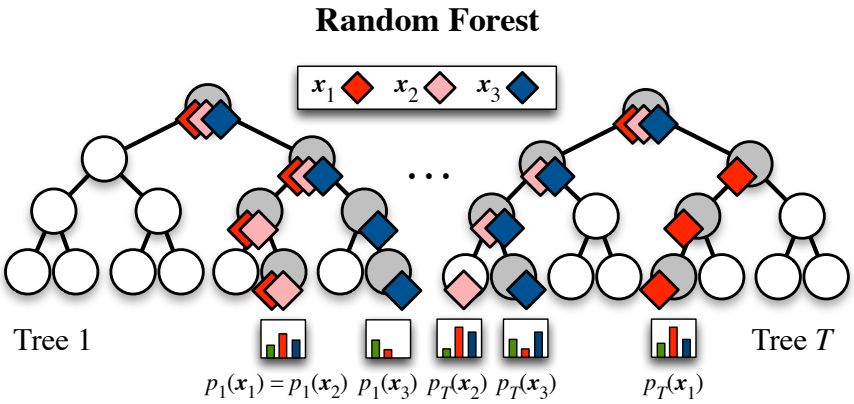

Figure 3.3: Random forest schematic showing posterior label predictions (histograms $p_t(\mathbf{x}_i)$) made at each tree t.

3.3.1 Background and Related Work

This section provides some background notation and terminology. We first review random forests – the ensemble classifier we use for our metric. Next, we briefly review a number of histogram distances that we will use.

Notation. Training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ consist of n vectors of dimensionality d, $\mathbf{x}_i \in \mathbb{R}^d$, where scalar $[\mathbf{x}_i]_j$ is the jth feature of the i-th instance. The features $[\mathbf{x}_i]_j$ may be real-valued, $[\mathbf{x}_i]_j \in \mathbb{R}$, or discrete, $[\mathbf{x}_i]_j \in \{1, \ldots, f\}$. Labels corresponding to each instance are selected from a set of classes $y_i \in \mathcal{Y} = \{1, \ldots, c\}$.

Random Forests. A random forest [22] is an ensemble of T decision trees [24]. To construct a decision tree, the input space is repeatedly partitioned using axis-aligned feature threshold splits at each node. The feature threshold split at a node is chosen to maximize

the *purity*—the fraction of inputs with the same label—of that node's children. Eventually, inputs reach a leaf node when either some specified maximum depth or full purity is achieved.

During training, the t^{th} random forest tree is learned independently on a "bootstrapped" training set, which consists of a set of n instances subsampled with replacement from the original training set [212]. Each split is picked greedily, but is restricted to a reduced set of e candidate features, uniformly selected from all d available features. The randomness yields a forest of subtly varying trees with strong generalization and robustness to overfitting. Figure 3.3 depicts two trees in a random forest. Random forests have only two hyper-parameters, the number of trees T and the number of features selected for each split, e. In general T should always be set as large as permitted (subject to CPU and runtime constraints). A good "rule of thumb" is to set $e = \lceil \sqrt{d} \rceil$, which we follow throughout.

Given an instance $\mathbf{x}$, the t^{th} tree in a random forest returns a probability distribution $p_t(y|\mathbf{x})$ over the label y of an instance $\mathbf{x}$. This distribution is simply the histogram of the labels of training samples in the leaf node to which $\mathbf{x}$ is assigned. For full trees, the leaf nodes are (nearly) pure, so these distributions have very low variance. For limited-depth trees, leaves are much less pure and the distributions have higher variance.

The random forest itself can be considered to return a probability distribution $P(y|\mathbf{x})$ over the label of a test point by averaging over the distribution of each tree,

$$P(y|\mathbf{x}) = \frac{1}{T}\sum_{t=1}^{T} p_t(y|\mathbf{x}). \tag{3.8}$$

A classification decision can then be rendered using the mode of this distribution,

$$H(\mathbf{x}) = \operatorname*{argmax}_y \; P(y|\mathbf{x}).$$

Histogram Distances. Histogram distances will form a useful tool to compare and contrast the predicted distributions returned by the random forest trees. Consider two histograms $\mathbf{p}$ and $\mathbf{q}$, each with c bins. In our case, since they represent probability distributions, histograms values are nonnegative and sum to one:

$$\forall k : p_k \geq 0, \forall k : q_k \geq 0, \text{ and } \sum_{k=1}^{c} p_k = \sum_{k=1}^{c} q_k = 1.$$

The *squared Euclidean distance* is a simple metric between two histograms, which grants equal impact to all pairwise differences between bins:

$$\mathrm{SE}(\mathbf{p}, \mathbf{q}) = \frac{1}{2}\sum_{k=1}^{c}(p_k - q_k)^2.$$

The *chi-squared (χ^2) distance* [149] assigns higher importance to differences in low-probability entries than to differences in high-probability entries:

$$\chi^2(\mathbf{p}, \mathbf{q}) = \frac{1}{2} \sum_{k=1}^{c} \frac{(p_k - q_k)^2}{p_k + q_k}.$$

The *earth mover's distance (EMD)* [136, 157, 62] measures the minimum amount of probability mass that must be redistributed among the bins of one histogram $\mathbf{p}$ to yield another histogram $\mathbf{q}$ and is solvable as a linear program.

The *cosine distance* measures the cosine of the angle between two vectors:

$$\cos(\mathbf{p}, \mathbf{q}) = \frac{\mathbf{p} \cdot \mathbf{q}}{||\mathbf{p}||\,||\mathbf{q}||}.$$

The *Hamming distance* counts the number of elements that differ between two vectors:

$$\text{Hamming}(\mathbf{p}, \mathbf{q}) = \sum_{k=1}^{c} [p_k \neq q_k],$$

where $[\cdot]$ denotes the Iverson bracket.

Related Work. Much prior work has focused on learning metrics for kNN classification. This work has commonly focused on learning global linear metrics corresponding to a Mahalanobis distance. This is equivalent to learning a linear transformation $\mathbf{x} \rightarrow \mathbf{L}\mathbf{x}$ such

that distances in the transformed space better represent similarity in the problem domain. Information-theoretic metric learning (ITML; [66]) learns a linear metric constrained by an input set of similar and dissimilar points. Large-margin nearest neighbors (LMNN; [217]) minimizes the number of training points which have points of another class among their nearest neighbors after transformation.

Learning nonlinear metrics, often equivalent to learning the transformation $\mathbf{x} \rightarrow \phi(\mathbf{x})$, has also been a recent topic of research. Much work has utilized kernels for learning global nonlinear metrics [51, 85, 113, 196], however these do not scale well in the size of the training set. [119] propose to optimize the LMNN objective with a non-linear transformation $\phi(\mathbf{x})$ of gradient boosted regression trees. This method still requires a "seed" metric to identify target neighbors. [102] use kernel density estimation on each feature independently to learn a discriminative representation that gives rise to a non-linear metric.

Two existing approaches utilize random forests for learning similarity. [23] propose computing proximity in a random forest (RFP) by counting the number of leaf nodes in common between two points across all trees in the forest. Random forest distance (RFD; [226]) trains a regression forest on a set of similar and dissimilar instances to return a similarity value for a pair of inputs.

3.3.2 Methods

While often highly accurate, random forest predictions lack the interpretability of nearest-neighbor classifiers, where the resulting neighbors may be informative about the reasons underlying the classification. Here, we propose a simple metric arising naturally from a trained random forest. When deployed in a nearest neighbor classifier, the metric can achieve the same high accuracy as a random forest—sometimes even outperform it—while yielding interpretable "certificates" in the form of the neighbors selected.

We use a simple intuition to design this metric. The individual trees in a random forest recursively partition the space into regions that are predictive of the same label—thus are semantically similar. Each tree splits in slightly different ways and *the individual tree distributions are therefore much more nuanced than their averaged prediction.* Consider two instances $\mathbf{x}$ and $\mathbf{z}$ which may or may not be semantically similar. Each individual tree t within a random forest classifier will output predictions, $p_t(y|\mathbf{x})$ and $p_t(y|\mathbf{z})$. A histogram distance between these two distributions, $d_H(p_t(y|\mathbf{x}), p_t(y|\mathbf{z}))$, reflects *some aspects* of semantic dissimilarity captured by this particular tree.

Analogous to the random forest classifier, which averages the predictions of all trees, we define the Random Forest Ensemble Metric (RFEM) as the average *histogram distance* of

all trees in the ensemble,

$$D_T(\mathbf{x}, \mathbf{z}) = \frac{1}{T} \sum_{t=1}^{T} d_H(p_t(y|\mathbf{x}), p_t(y|\mathbf{z})). \tag{3.9}$$

In the limiting case, two points $\mathbf{x}$ and $\mathbf{z}$ fall into the same leaf node in each tree, and have identical label predictions. This is captured in the aforementioned random forest proximity distance [23]. However, the random forest proximity distance is unnecessarily restrictive in requiring each sequence of decision-stump splits to be in exact agreement.

Instead of holding so tightly to the outcomes of the brittle splits made at each node, RFEM examines the local *leaf neighborhoods* carved out by each random tree. Each leaf corresponds to a d-dimensional "box" in the space of the training set. Each box contains a set of training instances that are assigned the same label distribution $p_t(y|\mathbf{x})$ — in other words, they share predictive similarity. RFEM considers two points to be similar when their leaf neighborhoods consistently share a similar structure, regardless of whether they ever shared the same leaf node.

As such, the metric leverages both the predictive power of the random forest and the variability and imperfection of the individual trees in the forest. Intuitively, if a pair of instances are similar under the metric, they are likely to share very similar random forest predictions, allowing the metric to capture the semantics of the random forest decision boundary. (We formalize this notion later for binary classification.)

However, the converse does not necessarily hold – sharing similar random forest label distributions $P(y|\mathbf{x})$ does not guarantee similar treatment in individual trees. If there are multiple distinct modes in a class, a tree is likely to agree on predictions for instances *within* a mode, yielding consistently similar label distributions $p_t(y|\mathbf{x})$. But between modes, the individual trees may predict different distributions, which only match the other modes of that class on average. This gives RFEM the ability to find structure underlying the random forest predictions.

We should note that whether the distance returned by RFEM is strictly a (pseudo-)metric depends on the choice of histogram distance. Of the five histogram distances introduced in the previous section, only three (χ^2, EMD, and Hamming) yield (pseudo-)metrics for RFEM. The remaining two, squared Euclidean and cosine, violate the triangle inequality. However, all four are examined in our experimental evaluation presented in Section 3.3.3 and perform well.

Computational Complexity. The training complexity for RFEM matches that of random forest. Training a random forest tree on n training instances and feature sampling parameter e is $O(neT \log n)$, assuming roughly balanced trees of depth $O(\log n)$ [220].

At test time, computing the RFEM distance requires averaging over the label distributions distances of T trees, each computed by evaluating a decision stump at $O(\log n)$ nodes per tree. Assuming the distributions for the training set have been precomputed and stored,

the computation of the RFEM distance for an instance pair is $O(cT \log n)$, where c is the number of classes.

Trees may be pruned by depth (to a single-depth limit set across the random forest to minimize out-of-bag error) to reduce evaluation time. Since each tree in an random forest may be learned or evaluated independently of the others, both training and testing is embarrassingly parallel.

Special Case: Binary Problems with Squared Euclidean Distance. Let $p_t(y|\mathbf{x})$ be the label distribution predicted by the t^{th} tree for instance $\mathbf{x}$. When the labels are binary, $y \in \{-1, 1\}$, $p_t(y|\mathbf{x})$ can be described entirely by a single value, *viz.* $p_t(y = 1|\mathbf{x}) = 1 - p_t(y = -1|\mathbf{x})$, which we abbreviate to $p_t(1|\mathbf{x})$. Here, the random forest prediction $P(1|\mathbf{x})$ is:

$$P(1|\mathbf{x}) = \frac{1}{T} \sum_{t=1}^{T} p_t(1|\mathbf{x}).$$

Rewriting Equation (3.9) with the squared Euclidean histogram distance, we obtain:

$$D(\mathbf{x}, \mathbf{z}) = \frac{1}{T} \sum_{t=1}^{T} \left(p_t(1|\mathbf{x}) - p_t(1|\mathbf{z}) \right)^2 .$$

This is equivalent to computing the squared Euclidean distance between $\phi(\mathbf{x})$ and $\phi(\mathbf{z})$ after mapping $\mathbf{x}$ and $\mathbf{z}$ by:

$$\mathbf{x} \rightarrow \phi(\mathbf{x}) = \begin{pmatrix} p_1(1|\mathbf{x}) \\ p_2(1|\mathbf{x}) \\ \vdots \\ p_T(1|\mathbf{x}) \end{pmatrix}. \tag{3.10}$$

Under this special case, we show a useful property of RFEM. If two instances $\mathbf{x}$ and $\mathbf{z}$ are close under RFEM, *i.e.* if $D(\mathbf{x}, \mathbf{z})$ is small, their random forest predictions $P(1|\mathbf{x})$ and $P(1|\mathbf{z})$ are guaranteed to be very similar. To show this, we make use of the explicit mapping of each instance, $\phi(\mathbf{x})$ and $\phi(\mathbf{z})$.

Theorem 1. *If two instances* $\mathbf{x}$ *and* $\mathbf{z}$ *are nearby under the binary random forest metric, then the squared difference between their random forest predictions is bounded. Formally, if* $D(\mathbf{x}, \mathbf{z}) < \delta$*, then* $\left(P(1|\mathbf{x}) - P(1|\mathbf{z})\right)^2 < \delta$*.*

Proof. If we map $\mathbf{x} \rightarrow \phi(\mathbf{x})$ and $\mathbf{z} \rightarrow \phi(\mathbf{z})$ as described previously and define $\Delta = \phi(\mathbf{x}) - \phi(\mathbf{z})$, then we can rewrite the theorem condition as $D(\mathbf{x}, \mathbf{z}) = \frac{1}{T} \|\Delta\|_2^2 < \delta$. In addition, given our definition of $P(1|\mathbf{x})$, $(P(1|\mathbf{x}) - P(1|\mathbf{z}))^2 = \frac{1}{T^2} \left(\sum_{t=1}^{T} (p_t(1|\mathbf{x}) - p_t(1|\mathbf{z})) \right)^2$. This is just $\frac{1}{T^2} \left(\sum_{t=1}^{T} \Delta_t \right)^2$, or more compactly, $\frac{1}{T^2} (\Delta^\top \mathbf{1})^2$. By the Cauchy-Schwarz inequality, $\frac{1}{T^2} \left| \Delta^\top \mathbf{1} \right|^2 \leq \frac{1}{T^2} \|\Delta\|_2^2 \|\mathbf{1}\|_2^2$, and therefore $\frac{1}{T^2} (\Delta^\top \mathbf{1})^2 \leq \frac{1}{T} \|\Delta\|_2^2$. Since $\frac{1}{T} \|\Delta\|_2^2 < \delta$, it follows that $\frac{1}{T^2} (\Delta^\top \mathbf{1})^2 < \delta$. Therefore, $\left(P(1|\mathbf{x}) - P(1|\mathbf{z})\right)^2 < \delta$. □

This theorem implies that whenever $\phi(\mathbf{x})$ and $\phi(\mathbf{z})$ are very close, it is likely that the random forest would make the same prediction for both $\mathbf{x}$ and $\mathbf{z}$. (kNN using this RFEM, however, will classify $\mathbf{x}$ using the true label of $\mathbf{z}$ rather than the random forest prediction of $\mathbf{z}$, giving kNN the opportunity to correct random forest errors in some cases.)

Importantly, the converse of the theorem is not guaranteed to be true. This means that the random forest prediction for $\mathbf{z}$ and $\mathbf{x}$ can be similar, but the instances can be far apart after mapping $\mathbf{x} \to \phi(\mathbf{x})$. If $p(\mathbf{x}|y)$ is multimodal and this structure is detected by the random forest, RFEM can maintain the multimodal structure. This property lends interpretability to our method, as it has the ability to distinguish clusters within the same class.

Fast Neighbor Search. It is common practice to speed up kNN neighbor search at test time using data structures such as k-d trees [16], ball trees [133], or vantage point trees [233]. RFEM with the squared Euclidean histogram distance induces a new explicit representation as given in Equation (3.10). In this new high-dimensional space, the squared Euclidean distance (divided by the number of trees) is equal to the RFEM distance. This space can be directly used in kNN search data structures.

However, its high dimensionality—$O(Tc)$ dimensions, where c denotes the number of classes—may limit the performance of these structures, which are known to perform best in low-dimensional spaces. In this case, PCA may yield a suitable low-dimensional representation.

For metric histogram distances that do not induce an explicit representation $\phi(\mathbf{x})$ such as χ^2 and EMD, vantage-point trees may be used as these require only pairwise distances.

In addition to these traditional methods for speeding up kNN search, we explore two methods based on the structure of RFEM itself that allow us to avoid distance computations when finding nearest neighbors under our metric.

Random Forest Prediction Bound. In binary classification, a simple bound to prune away distant instances arises from the contrapositive of Theorem 1. The theorem states that if the RFEM distance between two instances is small, the squared difference between the random forest label distributions, denoted $P_{\mathbf{xz}}$, must also be small. The contrapositive of this theorem is:

$$P_{\mathbf{xz}} = (P(1|\mathbf{x}) - P(1|\mathbf{z}))^2 > \delta \implies D(\mathbf{x}, \mathbf{z}) > \delta. \tag{3.11}$$

Suppose we compute the RFEM distance between a test instance $\mathbf{x}$ and a training instance $\mathbf{v}$, $D(\mathbf{x}, \mathbf{v})$. It follows directly from the contrapositive that:

$$P_{\mathbf{xz}} > D(\mathbf{x}, \mathbf{v}) \implies D(\mathbf{x}, \mathbf{z}) > D(\mathbf{x}, \mathbf{v}). \tag{3.12}$$

As a result, after computing $D(\mathbf{x}, \mathbf{v})$, we do not need to compute the RFEM distance between $\mathbf{x}$ and any training point $\mathbf{z}$ for which $P_{\mathbf{xz}} > D(\mathbf{x}, \mathbf{v})$. Hence, when computing the distance from $\mathbf{x}$ to each training point, we need not consider training points $\mathbf{z}$ for which $P_{\mathbf{xz}} > D^*(\mathbf{x})$,

where $D^*(\mathbf{x})$ is the smallest distance to a training point we have encountered so far. This procedure extends naturally to the case where $j > 1$ by letting $D^*(\mathbf{x})$ be the j^{th} smallest distance we have computed so far, rather than the minimum.

The procedure described above is efficient because $P_{\mathbf{xz}}$ can be computed very efficiently. For training instances $\mathbf{z}$, the random forest prediction $p(1|\mathbf{z})$ can be precomputed offline. Computing $P(1|\mathbf{x})$ for a test instance $\mathbf{x}$ requires $O(T)$ time and must be done only once. Given $P(1|\mathbf{x})$ and $P(1|\mathbf{z})$, computing $P_{\mathbf{xz}}$ requires only $O(1)$ time. Computing $P_{\mathbf{xz}}$ for all training data thus requires $O(T+n)$ time, which is asymptotically faster than the $O(Tn)$ required to compute $D(\mathbf{x}, \mathbf{z})$ for each training instance $\mathbf{z}$.

Hoeffding Bound. A probabilistic view suggests that RFEM is the expectation of the distance between two points under the distribution of trees in the random forest. The trees (and their predictions) are conditionally independent given the training data. Assuming we have partially evaluated an RFEM distance between two instances, *i.e.* we have computed the sample mean distance over t trees drawn *i.i.d.* from the ensemble of trees: $D_t(\mathbf{x}, \mathbf{z}) = \frac{1}{t}\sum_{i=1}^{t}(p_i(1|\mathbf{x}) - p_i(1|\mathbf{z}))^2$. Exploiting the fact that histogram distances are bounded, $0 \leq D(\mathbf{x}_i, \mathbf{x}_j) \leq \alpha$, we can use Hoeffding's inequality to bound the probability that the sample mean distance D_t is more than ϵ away from the expectation of the RFEM distance:

$$P\left(\|D_t - \mathbb{E}[D]\| \geq \epsilon\right) \leq 2\exp\left(-\frac{2t\epsilon^2}{\alpha}\right).$$

Rearranging terms, we can state with confidence $1 - p$ that:

$$\epsilon \leq \sqrt{-\frac{\alpha \log(p/2)}{2t}}. \tag{3.13}$$

Using confidence intervals on all distances, we greedily select distances to refine further until either all but k points have been ruled out as potential nearest neighbors (to some confidence) or all T trees have been exhausted.

It is worth pointing out that the Hoeffding bound and the Random Forest Prediction bound are complementary: the Random Forest Prediction bound is useful for one-shot filtering of the training set with very limited computation, whereas the Hoeffding bound provides a way to gradually refine the distances.

3.3.3 Experimental Results

We perform classification experiments with kNN classifiers using our RFEM metric on eleven data sets, comparing its performance to a range of competing metric learning techniques as well with random forests. Below, we describe the data sets used in our experiments, the setup of our comparative experiments, and the results of our experiments.

Datasets. The left three columns of Table 3.3 describe the number of training instances n, the number of features d, and the number of classes c for each dataset. The SCENE

15[1] dataset comprises of photographs of fifteen types of natural scenes to be distinguished. Input data consist of GIST and HOG features extracted from the photographs. Yale Faces (YFACES) is a set of grayscale images of faces for a 38 subjects under varying lighting conditions. SPLICE[2] involves the identification of splice junctions in DNA snippets with discrete features, while DNA[3] is another version of the same problem with binary features. CHESS[2] contains chess endgame settings that are either winnable or unwinnable for the white player. SPAM[2] comprises the recognition of spam emails based on a small number of features, such as the frequencies of certain words and characters. ISOLET[2] is a collection of sound recordings of phonemes uttered by 150 subjects; the aim is to identify which phoneme was uttered. ADULT[2] aims to predict whether a person's annual income exceeds $50,000 based on census data. YAHOO LTRC (Set 2)[4] is a set of web search query-document pairs in which document relevance is predicted based on query-document features. (Yahoo LTRC was made into a classification task by assigning a binary label indicating if the relevance was greater than or equal to 3.) W8A[3] comprises the categorization of web pages based on keyword attributes. MNIST[5] is a collection images of handwritten digit with the task of recognizing the depicted digit. The MNIST data set was preprocessed using PCA, preserving the first 300 principal components.

[1] http://tinyurl.com/uiuc-cvr
[2] http://tinyurl.com/uci-ml-data
[3] http://tinyurl.com/libsvm-data
[4] http://tinyurl.com/yahoo-ltrc
[5] http://tinyurl.com/mnist-data

	Characteristics			Linear ML			Non-Linear ML		RF ML	
Data set	n	d	c	Euclidean	ITML	LMNN	KLMNN	GBLMNN	RFD	RFEM
SCENE 15	1500	6812	15	41.5	–	41.5	50.2	–	38.1	**25.0**
DNA	1400	180	3	25.4	18.5	**6.3**	11.2	**5.7**	11.4	**5.2**
YFACES	1962	2016	38	30.2	26.7	6.9	96.3	6.5	**3.7**	**2.4**
SPLICE	2552	60	3	32.0	29.5	30.9	28.1	14.7	7.1	**3.6**
CHESS	2557	36	2	7.8	3.9	2.2	6.6	2.0	3.8	**0.5**
SPAM	3681	57	2	17.2	31.3	9.0	22.1	8.0	7.1	**5.1**
ISOLET	6238	617	26	11.2	21.4	**4.7**	10.8	**4.7**	21.6	9.6
ADULT	32562	123	2	20.5	20.7	20.1	–	20.4	**19.3**	**19.4**
YAHOO	34815	700	2	6.8	6.9	6.0	–	6.0	4.7	**3.9**
W8A	49749	300	2	2.1	1.6	2.1	–	2.1	**0.9**	2.1
MNIST	60000	784	10	2.4	4.2	**2.1**	–	**2.1**	10.1	4.0

Table 3.3: *Left side:* Number of training instances (n), features (d), and classes (c) of the data sets. *Right side:* Test error (%) of kNN classifiers using RFEM compared with six alternative metrics. The lowest errors up to $p=0.05$ (binomial) significance are **boldfaced**.

Experimental Setup. In each of our metric-learning experiments, we measure the generalization error of the 1-nearest neighbor classifiers. We compare the performance of RFEM with that of three linear and three non-linear distances. The three linear (Mahalanobis) metrics are (1) a *Euclidean* distance metric; (2) a metric learned by *ITML* [66]; and (3) a metric learned using *LMNN* [217]. The three non-linear metrics are: (1) a linear metric in a kernel space learned by *KLMNN*[11] [51]; (2) the *GB-LMNN* metric [119] described in Section 3.2; and (3) the random forest distance (*RFD*; [226]), which use a regression forest to predict the similarity between a pair of instances.

In a one set of experiments, we compare RFEM directly with random forest predictions and further evaluate RFEM using five histogram distances: squared Euclidean, χ^2, earth mover's distance (EMD), cosine, and Hamming. With the Hamming distance, RFEM is equivalent

[11]For scalability reasons, KLMNN was only run on datasets with fewer than $10,000$ training instances.

to the proximity metric proposed by [23], where pairwise distance is defined as the frequency at which the pair lands in different leaf nodes in the random forest.

When training random forests, the tree depths were limited by enforcing a minimum number of training instances per leaf. This number was set via cross-validation on out-of-bag training error and enforced uniformly across the entire forest. All random forests were trained with $T=250$ trees and $e=\lceil\sqrt{d}\rceil$ randomly sampled features (per split).

Results. In Table 3.3, we present the results of our experiments comparing kNN classifiers using RFEM with the six other metrics. In the table, the best performance on each data set up to $p=0.05$ binomial significance is **boldfaced**. We make two observations on the results presented in Table 3.3. First, we observe that for most datasets, the use of a non-linear metric substantially decreases the generalization error of the kNN classifiers. Linear metrics are among the top performers (within the $p=0.05$ significance level) on only four of the eleven data sets.

Second, we observe the strong performance of RFEM: it obtains best results (up to significance) on nine of the eleven data sets considered. On several data sets—including the challenging Scene 15 task—the performance improvement obtained by RFEM is very substantial. The strong performance of RFEM comes at little effort: RFEM has no sensitive hyperparameters and is "embarrassingly" parallel at both training and test time.

	RFEM Distances					
Dataset	SqEu	χ^2	EMD	Cos	Ham	RF
SCENE15	**25.0**	26.4	35.6	27.2	28.7	26.4
DNA	**5.2**	**5.9**	**6.1**	**6.1**	**6.0**	**5.8**
YFACES	**2.4**	**2.7**	**3.1**	**2.7**	**2.4**	**3.3**
SPLICE	**3.6**	**3.3**	**3.4**	**3.3**	**3.1**	**2.8**
CHESS	**0.5**	**0.5**	**0.5**	**0.5**	**0.8**	1.1
SPAM	**5.1**	**5.1**	**4.8**	**4.8**	**5.7**	**5.1**
ISOLET	9.6	**7.4**	17.3	8.9	10.0	9.7
ADULT	19.4	19.2	18.5	19.7	19.8	**15.2**
YAHOO	**3.9**	**3.9**	**4.0**	4.2	**4.1**	**3.9**
W8A	2.1	2.1	2.1	2.1	2.1	**0.8**
MNIST	4.0	4.1	10.3	4.1	**3.7**	4.2

Table 3.4: Test error (%) with RFEM using five different histogram distances: Squared Euclidean (SqEu), χ^2, Earth Mover's Distance (EMD), Cosine (Cos), and Hamming (Ham); and random forests (RF). The lowest error up to statistical significance ($p = 5\%$ binomial significance test) is indicated in **bold**.

Table 3.4 presents the generalization errors obtained by random forests and by RFEM with five different histogram distances: squared Euclidean, χ^2, EMD, cosine, and Hamming distance. The results show that the performance of RFEM is quite insensitive to the choice of histogram distance. Interestingly, the squared Euclidean distance performs very well on both binary and multi-class problems. This is beneficial since the squared Euclidean distance yields an explicit representation (for binary classification) and is conducive to traditional kNN speedup methods.

Table 3.4 also shows that kNN classifiers using RFEM perform on par with the corresponding random forests on eight of eleven datasets. On a few of the data sets, RFEM even *corrects some mistakes* made by the random forest predictor, which lowers the generalization error

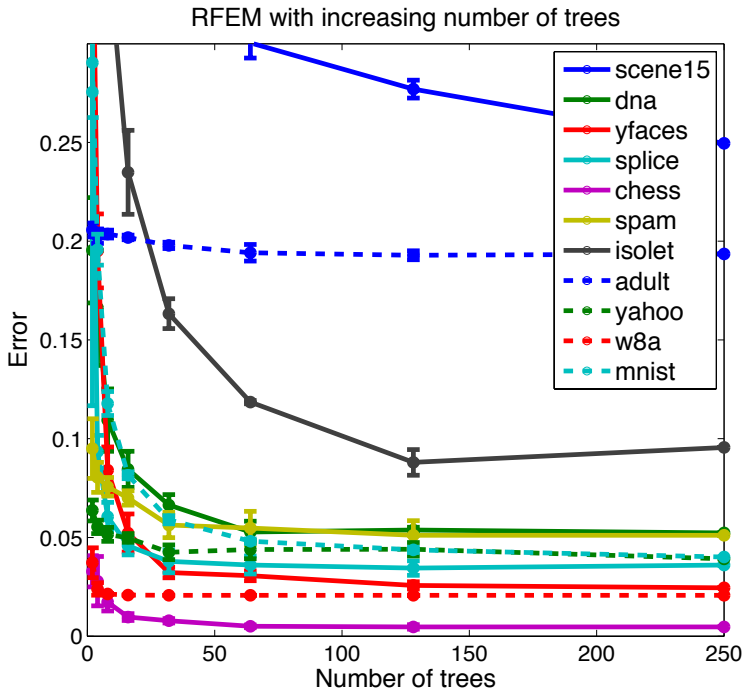

Figure 3.4: Test error of RFEM with a varying number of trees (mean and standard deviation over 5 runs).

of the predictor. The results imply that RFEM lends interpretability to a random forest classifier without affecting the classifier's strong generalization.

Figure 3.4 shows the test error of RFEM (with squared Euclidean distance) on all eleven data sets while varying the number of trees in the random forests. The figure shows that for most datasets, the performance of RFEM converges fairly quickly: 128 trees often suffice to obtain good predictions.

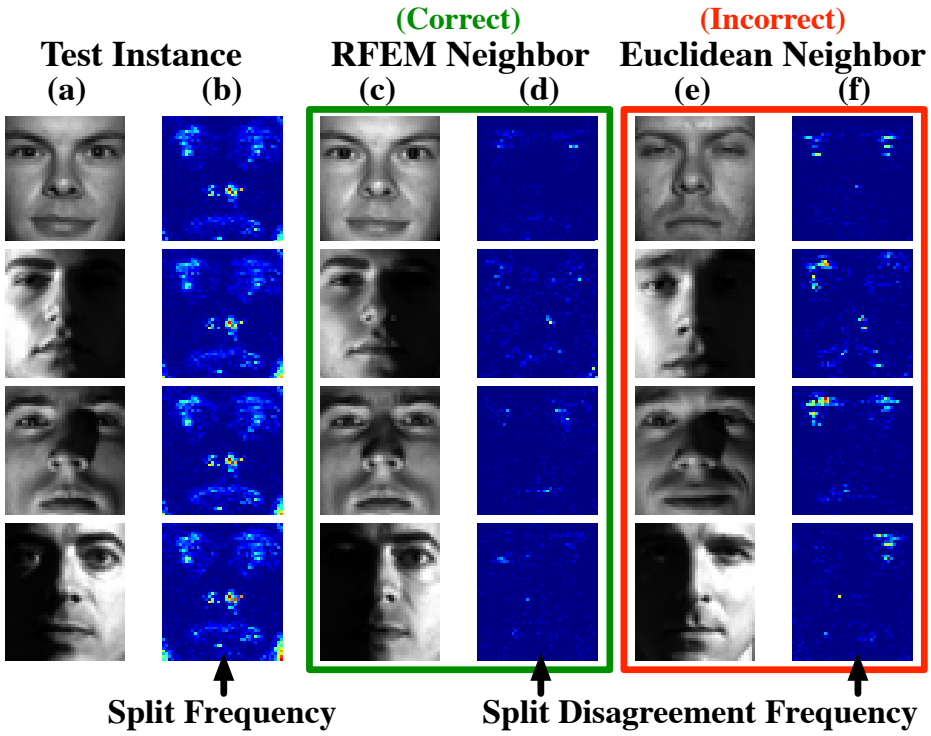

Figure 3.5: RFEM and Euclidean metrics applied to Yale Faces test instances. Masks show splitting features encountered by each test instance during tree traversal (b) and the disagreement of each candidate neighbor (d and f) on those feature splits.

Characteristics of RFEM. We performed three additional experiments to investigate: (1) to what extent RFEM identifies task-relevant features, (2) how RFEM generalizes to unseen classes, and (3) to what extent RFEM preserves intra-class structure.

Task-Relevant Features. Figure 3.5 shows four Yale Faces test instances (column a) which were correctly classified by RFEM (c) but incorrectly classified using Euclidean similarity (e). The split frequency mask (b) shows splitting features encountered by the test instances during random forest tree traversal. These demonstrate which pixels are deemed

YFACES Unseen Classes			
Euclidean	ITML	LMNN	RFEM
21.3 (3.4)	33.1 (16.0)	4.5 (1.1)	**2.2 (0.6)**

Table 3.5: Mean (and standard deviation of) test error on classes held out during training of kNN classifiers using four metrics (in %), averaged over five random sets of unseen classes. The lowest error up to ($p=0.05$ binomial) significance is **boldfaced**.

to be discriminative by the random forest. The split disagreement frequency masks (d and f) show the frequency of disagreeing with the test point on the splits encountered. The Euclidean neighbors show a higher rate of disagreement than the RFEM neighbors. The disagreements are centered on face-specific regions rather than the lighting differences which confuse the Euclidean metric.

Generalization to Unseen Classes. Table 3.5 presents the performance of RFEM on a face-verification task on the Yale Faces data set: as test data, we only use individuals who were not present the training set. In this task, the random forest itself is useless for prediction as it was only trained to identify the individuals in the training set. The results in the table show that, in contrast, RFEM has captured general properties for distinguishing faces, which leads to a very strong performance compared to the competing methods on the verification task. This result highlights the potential of RFEM for one-shot learning [76].

Preservation of Intra-Class Structure. To investigate whether RFEM preserves multimodal intra-class structure, we visualized four of our (test) data sets using t-SNE [204] in

Figure 3.6. The figure presents the visualizations obtained from running t-SNE using Euclidean distance (left), LMNN distance (middle), and RFEM distance (right). The results show that RFEM obtains better class separation on all four data sets. In particular, the Isolet visualization highlights the ability of metric learning to improve class separation while maintaining intra-class structure: ellipses in the figure highlight a few data regions whose proximity is maintained in LMNN and RFEM, while the class separation is improved.

3.3.4 Discussion

In this section we presented a novel (pseudo-)metric, called Random Forest Ensemble Metric (RFEM), that allows kNN classifiers to take advantage of the compelling aspects of random forests. These include high accuracy, the ability to deal with discrete and continuous inputs, robustness to hyper-parameters, invariance to feature scaling, and robustness against overfitting. Most importantly, with RFEM, kNN maintains its ability to provide interpretable decisions.

Theoretical Properties. We derived theoretical results that link the similarity of two instances under the RFEM metric to the similarity of the ensemble prediction for both instances, and we have derived two bounds (an exact and an approximate one) to speed up the nearest neighbor classification using our method, making it practical on large data sets.

Using experiments, we demonstrate that RFEM performs well (providing high accuracy and interpretable results) on a variety of classification tasks.

Simplicity. The most striking aspect of RFEM is its simplicity. This is in strong contrast to much more involved prior work on non-linear metric learning [119, 196]—yet, we show that RFEM clearly outperforms these more complicated algorithms with impressive consistency across many diverse data sets. This demonstrates that, although prior work might be more interesting from an algorithmic point of view and may involve more challenging optimization problems, sometimes the best results can be achieved with simple algorithms. Because of its simplicity, we believe that RFEM may become a useful technique in the toolchest of machine-learning practitioners.

While our treatment of the RFEM has focused on obtaining metrics from trained random forests, it should be noted this approach can be used with any classifier ensemble in which the individual experts output a distribution over classes (such as logistic regressors, discriminative RBMs, and naive Bayes classifiers). Such extensions of our metric-learning approach may prove fruitful as future work.

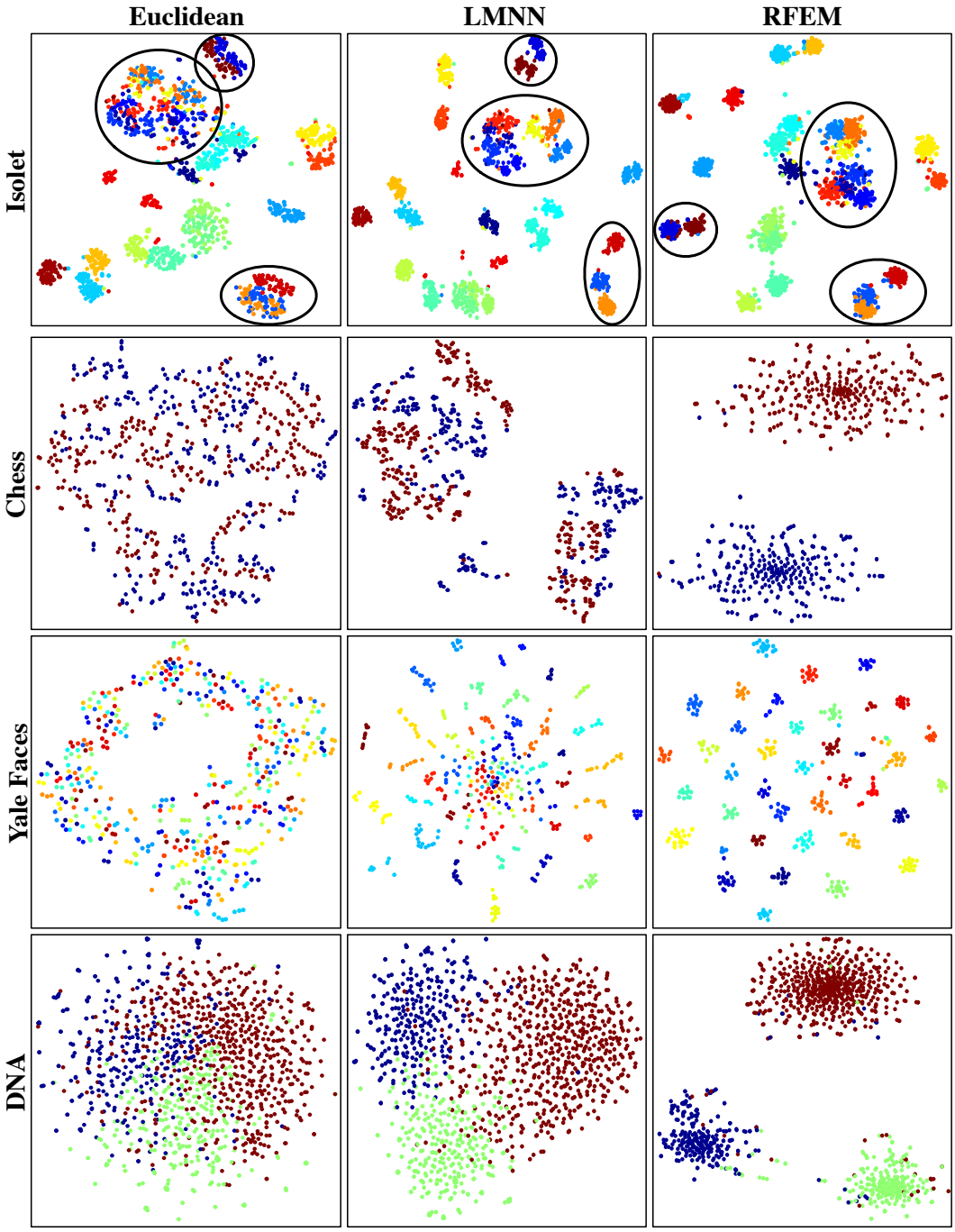

Figure 3.6: t-SNE visualization of four test data sets using Euclidean distance, LMNN distance, and RFEM distance. In the Isolet figures, circles highlight regions where metric learning improves class separation while preserving intra-class structure.

Chapter 4

Multi-Platform Parallelism for Support Vector Machines

In this chapter, we discuss the problem of learning kernel support vector machine (SVM) models across a range of parallel systems, from highly multi-threaded GPUs to shared memory multi-core systems. Despite the popularity of SVMs in practice, parallel and distributed implementations of SVM training software have yet to find widespread adoption.

We discuss some of the difficulties that SVMs have presented to parallelization. These challenges include limited inherent parallelism in the most popular solver methods and the tendency to produce tedious parallel code. These difficulties are most pronounced in GPU implementations, where coders must operate in a restrictive, highly multi-threaded environment.

This motivates a different approach to constructing a parallel SVM solver. In contrast to existing "explicitly" parallel approaches, we adopt an "implicitly" parallel approach built on an *approximate* reformulation of the SVM training problem. We adopt the SVM formulation of Keerthi et al. [121]. By leveraging an approximate reduced basis set in exchange for the full space of support vectors, this approach expresses an SVM solver almost entirely in matrix multiplication and other dense linear algebra operations. These dense linear algebra functions yield a high degree of parallelism, and finely-tuned libraries exist for a wide range of platforms, including multi-cores and GPUs. Furthermore, as hardware evolves rapidly—as is the case with NVIDIA GPUs—software written with these libraries evolves correspondingly. Meanwhile, explicitly parallel approaches often require tedious updates or thorough re-writes to take advantage of new features or to fit changing paradigms. We demonstrate that adopting the right SVM reformulation yields significant parallelism and robust implementations on both shared memory and GPU systems using the same code-base.

4.1 Introduction

Kernel support vector machines are among the most established machine learning algorithms. SVMs capture complex, nonlinear decision boundaries with good generalization to previously unseen data. Numerous specialized solvers exist [43, 117, 176], which take advantage of the sparseness inherent in the optimization and are known to be effective on a large variety of classification problems.

Recently, trends in computer architecture have been moving toward increasingly parallel hardware. Most CPUs feature multiple cores, and general purpose graphics processing units (GPUs) can execute thousands of parallel threads on their hundreds of throughput-optimized cores. Both parallel frameworks offer enormous raw power, and have the potential to provide huge speedups. However, to utilize each type of parallel thread effectively, algorithms must be carefully decomposed and optimized in fundamentally different ways. For example, GPUs are based on a "same instruction multiple data" (SIMD) architecture, which requires all threads within one block to execute the exact same instructions, whereas multi-core CPUs have much fewer threads with no such restriction.

On a high level, there are two different approaches to parallelizing algorithms, which we term here the *explicit* approach and the *implicit* approach, respectively. In the explicit approach, an algorithm is parallelized by hand—that is, the programmer finds the independent components of the algorithm which can be run in parallel and encodes this parallelism using some appropriate explicitly parallel language or library such as OpenMP (for multicores), MPI (for clusters), CUDA or OpenCL (for GPUs). In the implicit approach, the algorithm is expressed as a series of operations which are known to be highly parallel and for which highly optimized parallel libraries already exist for most platforms. Examples include libraries for dense linear algebra operations—such as PLASMA [2] and Intel's MKL [112] for multicores; MAGMA [2], Jacket [1], and CuBLAS [145] for GPUs—and PDE solvers such as PETSc [10].

Both approaches have advantages and disadvantages. The explicit approach can be applied to most algorithms; therefore, in particular, it can probably be applied to the exact algorithm of one's choice. However, it often requires a significant engineering effort and a fine-tuned tradeoff between parallel work and induced overhead—which needs to be calibrated specifically for any particular algorithm and parallel architecture.

The implicit approach is only applicable if the algorithm in question can be formulated as operations of some well-optimized library (in our case, linear algebra operations), which may not always be possible or may require approximation or relaxation of the problem, potentially leading to a loss in accuracy. If it is possible, however, the implicit approach has two advantages. First, since researchers and engineers have carefully designed and optimized these linear algebra libraries for peak performance [2, 145], they typically provide great speedups as long as they are called on sufficiently large problems. Therefore, if we can express an algorithm in terms of linear algebra operations of large-enough granularity, implicit algorithms can provide great parallel speedups, often more so than explicit algorithms. Second, these libraries are maintained and ported to new hardware as it becomes available; therefore, there is no need to rewrite an implicit algorithm for each new generation. In light of these two options, we investigate the following question: *Given recent changes in hardware design, is there a reformulation or approximation of kernel SVM training which yields efficient implicit parallelism?*

To our knowledge, all existing (competitive) parallel SVM implementations for multi-core or GPU systems [7, 39, 41, 43, 59] use the explicit parallelization approach on dual decomposition methods, such as the SMO algorithm [150]. Although implicit parallelization comes naturally for *e.g.* deep neural nets [124], it does not initially fit the SVM formulation and until this work there were no comparable SVM implementations of implicit parallelization. However, there exist at least three publications that reduce the kernel SVM optimization to dense linear algebra operations. Sha et al. [165] introduce a multiplicative update rule for the exact SVM optimization problem, which uses large matrix-vector multiplications in each iteration. Chapelle [47] proposes a primal formulation for the least squares hinge loss [183] which results in matrix-matrix and matrix-vector operations, and Keerthi et al. [120] approximates this approach by restricting the support vectors to a smaller subset (for reduced test-time complexity).

One advantage of the implicitly parallel approach is that, if done correctly, the algorithm spends almost all of its execution time in highly optimized routines and very little time in the remainder of the program, which therefore can be written in a high level language like `MATLAB` or `Python`. This enables us to implement implicit parallel versions of all three approaches, which naturally work on both multi-core and GPU systems, by linking against appropriate algebra libraries [112, 145].

We apply an empirical approach and compare the various implementations with each other on several medium-sized classification data sets on GPU and multi-core architectures and

arrive at an interesting conclusion: Although the multiplicative update rule [165] and the primal optimization [47] do not scale to our data set sizes due to their quadratic memory complexity, Keerthi's [121] sparse primal optimization appears to be an excellent compromise. Our `MATLAB` implementation tends to consistently outperform all highly optimized explicitly parallel algorithms and generally suffers no or little decrease in accuracy due to the problem relaxations.

In this chapter we make two contributions: 1. We provide the first detailed empirical analysis of both explicit and implicit SVM parallelization for multi-core CPUs and GPU architectures; 2. We observe that implicit parallelization unveiled by approximation can be a far more efficient approach. We believe that these insights are valuable to the machine learning community, which has so far focused almost entirely on explicit parallelism, and encourage further research into implicit approaches to parallelism.

Throughout this chapter we type vectors in bold ($\mathbf{x}_i$), scalars in regular (C or b), matrices in capital bold ($\mathbf{K}$) and sets in cursive ($\mathcal{J}$) font. Specific entries in vectors or matrices are scalars and follow the corresponding convention, *i.e.* the i^{th}, j^{th} entry of matrix $\mathbf{K}$ is written as K_{ij} and the i^{th} dimension of vector $\mathbf{x}$ is x_i. In contrast, depending on the context, $\mathbf{x}_i$ refers to the i^{th} vector within some ordered set $\mathbf{x}_1, \ldots, \mathbf{x}_n$ and $\mathbf{k}_i$ refers to the i^{th} column in a matrix $\mathbf{K}$.

Kernelized SVMs. When training a support vector machine, we are given a training dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ of feature vectors $\mathbf{x}_i \in \mathbb{R}^d$ with class labels $y_i \in \{-1, +1\}$. The goal of the optimization is to find a maximum margin hyperplane separating the two classes. (Binary classifiers can easily be extended to multiclass settings through pairwise coupling or similar approaches [164].) The primal formulation of the SVM optimization problem [57] learns a hyperplane parameterized by weight vector $\mathbf{w}$ with a scalar offset b:

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n} \max(0, 1 - y_i(\mathbf{w}^\top \mathbf{x}_i + b)). \tag{4.1}$$

The simple linear case can be solved very efficiently with special purpose algorithms [74]. In this chapter we focus on non-linear SVMs, which map the inputs into a new feature space $\mathbf{x}_i \rightarrow \phi(\mathbf{x}_i)$ prior to optimizing, where $\phi(\mathbf{x}_i)$ is a nonlinear feature-space transformation of $\mathbf{x}_i$. This mapping is generally to a higher (possibly infinite) dimensional representation. As inputs are only accessed through pairwise inner products in the dual formulation of the optimization, the mapping can be computed implicitly with the *kernel-trick* [164] through a positive semi-definite kernel function $k(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$. The (dual) optimization to find the large-margin hyperplane becomes

$$\max_{C \geq \alpha_i \geq 0} -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j) + \sum_{i=1}^{n} \alpha_i, \tag{4.2}$$

where a Lagrange multiplier variable α_i corresponds to each training input. At the end of the optimization, only some variables α_i are nonzero, which are referred to as *support vectors*.

(For convenience, henceforth, we omit the bias term b, which can be solved for in a straightforward fashion from the solution of (4.2) [164]. Throughout this chapter we will speak primarily on the Radial Basis Function (RBF) kernel: $k(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma||\mathbf{x}_i - \mathbf{x}_j||^2}$. Its explicit feature representation $\phi(\mathbf{x}_i)$ is infinite dimensional and can at best be approximated [153] for explicit use in the primal formulation. The RBF kernel is particularly interesting because of its universal approximation properties [164] and its wide-spread application. However, this work extends beyond the RBF kernel to any kernel without a corresponding low-dimensional explicit feature representation $\phi(\mathbf{x}_i)$.

Although solving the SVM optimization in the dual formulation (4.2) avoids the explicit feature computation $\phi(\mathbf{x}_i)$, it is still significantly slower than solving the linear formulation. In particular, it requires either precomputing the kernel matrix $\mathbf{K}$ where $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$, requiring $O(n^2)$ space, or recomputing $k(\mathbf{x}_i, \mathbf{x}_j)$ as it is needed, with space or time complexity that is too great for ever increasing data set sizes. This motivates the adoption of SVM-specific optimization procedures.

4.2 Parallelism for GPUs and Multicores

4.2.1 Explicitly Parallel SVM Optimization

To our knowledge, all competitive implementations of parallel SVMs (for multi-core CPUs or GPU architectures) are based on explicit parallelization of dual decomposition approaches. Dual decomposition methods, which include Sequential Minimal Optimization (SMO) [150], are among the most efficient sequential algorithms for solving the dual formulation. They operate on a small *working set* of Lagrange multiplier variables in each iteration, holding others constant. For example, in each iteration, SMO heuristically selects two dual variables, α_i and α_j, and optimizes them analytically. LibSVM, a very popular tool for training SVMs, implements a variant of this method [43]. In general, any small number of dual variables may be optimized at once with working set size representing a tradeoff between work per iteration and number of iterations required. Explicit parallelization approaches parallelize the computation within each iteration as well as parallelizing kernel computations. Due to their fine grained iterative nature, these approaches are not a natural fit for highly parallel hardware. Nevertheless, there exist a variety of implementations that parallelize the individual iterations and the kernel computations on GPUs and multi-core architectures. A common theme among explicitly parallel methods is high code complexity, making it hard to verify correctness or port the code to new or updated hardware platforms.

Multi-core. There are several parallel implementations of dual decomposition-based SVM solvers targeted toward multi-cores. Some methods attempt to extract existing parallelism from SMO-based approaches [67, 71], including a simple modification to LibSVM that computes kernel matrix entries in parallel with OpenMP. Other approaches attempt some restructuring of the problem. Increasing the working set size (originally two variables in SMO) exposes additional parallelism, as several dual variables are optimized at each iteration [27, 70, 236], as does optimizing over nested working sets [239]. Another common approach is to partition the training set, optimize over the partitions in parallel, and combine the resulting solutions [35, 56, 95, 101, 238]. We were only able to obtain source code for two of these methods — namely LibSVM with OpenMP and PSVM[243]. We only report the results of the former, since the latter was not designed for multi-core CPUs and consumed an infeasible amount of memory for medium-scale datasets. However, a comparison of published training times (with consideration of the various architectures) makes us believe that most other approaches are comparable or (more often) less competitive in practice.

GPU. Likewise, all previous attempts to accelerate the training of kernelized SVMs on GPUs have been direct implementations of a dual decomposition method such as SMO. GPU SVM [41] offloads computation of kernel matrix rows to the GPU using the CUBLAS library and computes KKT condition updates on the GPU with explicitly parallelized routines. A similar approach and results were demonstrated by CUSVM [39]. GTSVM [59] takes the strategy of increasing the working set size of dual variables to 16 to better utilize GPU

resources. The method features built-in support for both multi-class SVMs and sparse training vectors. GTSVM achieves the best previously published kernel SVM training times of which we are aware. Other GPU implementations include solvers especially optimized for multi-class problems [144] and a specialized implementation in `R` [231]. The most successful GPU implementations of dual decomposition generally require algorithmic modifications and significant customized CUDA code just to leverage the full capability of the GPU. We seek an algorithm that lends itself more gracefully to a parallel implementation.

4.2.2 Implicitly Parallel SVM Optimization

As an alternative to explicitly parallelized SMO-type optimization methods, we also investigate algorithms that are amenable to implicit parallelization; that is, algorithms where the majority of the work can be expressed in few iterations with dense linear algebra computations, which can then be performed using optimized libraries. We identify three reformulations of the SVM problem that lend themselves to this approach, while noting that none of these methods were explicitly developed for increased parallelism. (It is important to point out that in all formulations in this section, the linear algebra computations are dense irrespective of the sparsity of the data, as they operate on the dense kernel matrix, *e.g.* computing Hessian updates.)

A key factor in the implicit approach is that it can readily engage approximation—making a reformulation or relaxation of the SVM optimization in (4.2). While this can impact

accuracy and memory efficiency, compared with decomposition methods, we will show that it also has the capacity to unlock significant parallelism.

Multiplicative update. Sha et al.[165] propose the multiplicative update rule, which updates all dual variables α_i in each iteration, to solve the dual optimization (4.2). This approach relies on matrix-vector multiplication which can be readily parallelized; the authors remark in their original publication that the algorithm could potentially be used for parallel implementations. While our implementation demonstrated some speedups when linked against parallel libraries, the method was ultimately not competitive (and is not included in our experimental section) for two reasons: 1. The entire kernel matrix must be stored in memory at all times, which renders the method infeasible for typical medium-sized data sets; and 2. the convergence rate of the multiplicative update is too slow in practice, requiring too many iterations.

Primal optimization. Chapelle introduces a method for solving a kernel SVM optimization problem in the primal [47]. The SVM classifier can be expressed as $h(\mathbf{x}) = \mathbf{w}^\top \phi(\mathbf{x}) + b$, where $\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \phi(\mathbf{x}_i)$ (and with bias b). After the transformation $\mathbf{x} \to \phi(\mathbf{x})$, solving (4.1) with respect to $\mathbf{w}$ directly is impractical or impossible, due to the high (possibly infinite) dimensionality of $\phi(\mathbf{x})$. However, after a change of variable, with $\beta_i = \alpha_i y_i$ and $\boldsymbol{\beta} \in \mathbb{R}^n$,

(4.1) can be rewritten as follows:

$$\min_{\boldsymbol{\beta},b} \frac{1}{2}\boldsymbol{\beta}^\top \mathbf{K}\boldsymbol{\beta} + \frac{C}{2}\sum_{i=1}^{n} \max(0, 1 - y_i(\boldsymbol{\beta}^\top \mathbf{k}_i + b))^2 \tag{4.3}$$

where $\mathbf{k}_i$ is the kernel matrix row corresponding to the i^{th} training example. Notice that there are two relaxations: 1. the β_i are unconstrained, in contrast to α_i in (4.2), which must satisfy $0 \leq \alpha_i \leq C$; and 2. the squared hinge loss is used in place of the more common absolute hinge loss. These changes allow the use of second order optimization methods. In particular, Newton's method yields very fast convergence with computations expressed as dense linear algebra operations. As noted in [47], the squared hinge loss leads to almost identical results as the absolute hinge loss—a claim that we confirm in our experimental results. Similar to the multiplicative approach, this method requires the computation of the entire kernel matrix, which renders it impractical for larger data sets. We therefore do not include it in our experimental results section, which focuses on data sets with prohibitively large sizes.

Sparse primal optimization. Keerthi et al. propose a method to reduce the complexity of Chapelle's primal approach by restricting the support vectors to some subset of *basis vectors* $\mathcal{J} \subset \{1, \ldots, n\}$ so that $j \notin \mathcal{J} \Rightarrow \beta_j = 0$. With the approximation, equation (4.3) then becomes:

$$\min_{\boldsymbol{\beta},b} \frac{1}{2}\boldsymbol{\beta}^\top \mathbf{K}_{\mathcal{J}\mathcal{J}}\boldsymbol{\beta} + \frac{C}{2}\sum_{i=1}^{n} \max(0, 1 - y_i(\boldsymbol{\beta}^\top \mathbf{k}_{\mathcal{J}i} + b))^2. \tag{4.4}$$

Here, $\boldsymbol{\beta}$ has been restricted to contain only those β_j with $j \in \mathcal{J}$. $\mathbf{K}_{\mathcal{J}\mathcal{J}}$ is the kernel matrix between only basis vectors, and $\mathbf{k}_{\mathcal{J}i}$ is the kernel row of the i^{th} training example with all basis vectors (i.e., the vector $\mathbf{k}(\mathbf{x}_k, \mathbf{x}_i)$ for each $k \in \mathcal{J}$).

As the set $\mathcal{J}$ is originally unknown, Keerthi et al. propose to grow $\mathcal{J}$ with a heuristic. Initially, $\mathcal{J}$ is empty and the algorithm then has two distinct stages that are cycled. *Basis vector selection:* A small subset of the training set is randomly sampled, and then a heuristic is used to estimate the reduction in loss from adding each input to $\mathcal{J}$. The highest scoring point is then greedily added to $\mathcal{J}$ to get $\mathcal{J}'$. *Reoptimization:* After a certain number of basis vectors have been added to $\mathcal{J}'$, (4.4) is optimized using $\mathcal{J}'$ as the basis vector set. This whole process of gradually selecting basis vectors and then re-optimizing repeats until a stopping criterion is met. The resulting algorithm performs only a few iterations in total, each of which make use of intensive linear algebra computation. This method still requires the kernel matrix of basis vectors with all training examples, requiring $O(|\mathcal{J}|n)$ space. In practice, $|\mathcal{J}| \ll n$; however, this may still be a concern, particularly on GPUs where memory availability is more limited than RAM.

4.2.3 Implementing Sparse-Primal SVM

In this section, we provide details of SP-SVM, focusing on the aspects that relate to parallel implementation. In particular, we observe which steps are expressed using linear algebra

operations and can be easily accelerated using modern parallel machines such as GPUs and CPUs.

Newton Optimization. Suppose we begin with a pre-chosen set of basis vectors $\mathcal{J}$. The optimization (4.4) may be solved by Newton steps on $\boldsymbol{\beta}$. Let indices

$$\mathcal{I} = \left\{ i : 1 - y_i \boldsymbol{\beta}^\top \mathbf{K}_{\mathcal{J}i} \geq 0 \right\} \subseteq \mathcal{D}$$

correspond to training instances which contribute to the loss in the objective function. These are the points that violate the margin, whether they fall on the wrong side of the hyperplane or are merely within a unit distance. Note that the bias term b is omitted from the following equations for convenience, but may be easily included as shown in [47] and has been incorporated in our implementation.

Gradient: Taking the gradient of (4.4) with respect to $\boldsymbol{\beta}$, we obtain the vector $\mathbf{g} \in \mathbb{R}^{|\mathcal{J}|}$,

$$\mathbf{g} = \mathbf{K}_{\mathcal{J}\mathcal{J}}\boldsymbol{\beta} - C\mathbf{K}_{\mathcal{J}\mathcal{I}}(\mathbf{y}_\mathcal{I} - \boldsymbol{\beta}^\top \mathbf{K}_{\mathcal{J}\mathcal{I}}). \tag{4.5}$$

The gradient is the sum of two matrix-vector products. The first of these products, the multiplication $\mathbf{K}_{\mathcal{J}\mathcal{J}}\boldsymbol{\beta}$ between a $|\mathcal{J}| \times |\mathcal{J}|$ matrix and a vector of length $|\mathcal{J}|$, captures the regularization of $\boldsymbol{\beta}$. The second multiplication accounts for margin violations.

Observations for Optimization: While the matrix-vector multiplication of the gradient calculation is a linear algebra operation, we do not offload it to the GPU since each matrix row is used just once, and the memory access latency and the transfer latency from DRAM to GPU memory can not be effectively hidden. However, on the CPU, we do use linear algebra libraries for the efficient execution of this step.

Hessian: Differentiating again with respect to $\boldsymbol{\beta}$, we have the Hessian $\mathbf{H} \in \mathbb{R}^{|\mathcal{J}|\times|\mathcal{J}|}$, where

$$\mathbf{H} = \mathbf{K}_{\mathcal{J}\mathcal{J}} + C\mathbf{K}_{\mathcal{J}\mathcal{I}}\mathbf{K}_{\mathcal{J}\mathcal{I}}{}^{\top}. \tag{4.6}$$

Typically, for smooth, unconstrained minimization problems on CPUs, it is more efficient to use quasi-newton methods [26, 242], which approximate the Hessian matrix and avoid its costly computation at the price of slower convergence. In our case, as we can express the Hessian matrix entirely in terms of dense matrix matrix multiplications (4.6), we can obtain massive speedups through parallel hardware, and the exact Newton method becomes very attractive.

The Hessian is the sum of two $|\mathcal{J}| \times |\mathcal{J}|$ matrices. The first matrix $\mathbf{K}_{\mathcal{J}\mathcal{J}}$ captures partial second derivatives of the regularization term. The second matrix is the outer product matrix between a $|\mathcal{J}| \times |\mathcal{I}|$ kernel submatrix and itself.

Observations for Optimization: In general, matrix multiplication is very naturally implemented on parallel machines, especially GPUs. However, the matrix multiplication in (4.6)

presents a special case as $\mathbf{K}_{\mathcal{J}\mathcal{I}}$ quickly grows too large to store on the GPU. To compute on the GPU, the matrix must be transferred from the CPU and computed on the GPU in blocks. A naive implementation in which blocks of the kernel are *synchronously* transferred to the GPU between computations yields little speedup over computing directly on the CPU. While the block computations themselves are noticeably faster on the GPU, the transfer latency between CPU RAM and GPU global memory is significant. However, we make use of multiple *asynchronous* streams of CPU-to-GPU memory transfers of small blocks with interleaved matrix multiplication.

Iterative update. To solve for weights $\boldsymbol{\beta}$, we take Newton steps of the form

$$\boldsymbol{\beta}' = \boldsymbol{\beta} - \mathbf{H}^{-1}\mathbf{g}. \tag{4.7}$$

In practice, very high accuracy can be achieved with a set of basis vectors that is much smaller than the set of training instances, $|\mathcal{J}| \ll n$. This renders the Newton optimization tractable, even on large problems. Computing $\mathbf{H}$ and $\mathbf{g}$ is $O(n|\mathcal{J}|)$, while $\mathbf{H}^{-1}\mathbf{g}$ is $O(|\mathcal{J}|^3)$. The significant costs are incurred while computing the Hessian and gradient. Both require $|\mathcal{J}|$ rows of the kernel matrix $\mathbf{K}$, and the Hessian involves the costly matrix multiplication $\mathbf{K}_{\mathcal{J}\mathcal{I}}\mathbf{K}_{\mathcal{J}\mathcal{I}}^{\top}$.

Generally only a few Newton steps are required for convergence, between which margin violations $\mathcal{I}$ and gradient $\mathbf{g}$ must be recomputed, since they are functions of $\boldsymbol{\beta}$. Since we

are using quadratic penalization of errors in (4.4), the Hessian is constant in $\boldsymbol{\beta}$ and changes only in response to changes in the set of loss incurring inputs $\mathcal{I}$. Consequently, only small incremental updates to the Hessian are required between Newton steps.

Adding Basis Vectors. Solving the optimization in practice without an *a priori* set of basis vectors requires a two phase approach. Beginning with an empty basis set $\mathcal{J}^0$, basis vectors are added greedily, $\mathcal{J}^{k+1} = \mathcal{J}^k + \{c_{k+1}\}$. To select each basis vector, a candidate set $\mathcal{C}$, of some small size e.g. 10, is randomly sampled from the training examples. Each new basis vector is chosen from a candidate set to minimize the objective (4.4) assuming the existing weights $\boldsymbol{\beta} \in \mathbb{R}^k$ are held constant, which can be computed in closed form. The candidate with the largest estimated improvement is chosen.

Then all weights (now $\boldsymbol{\beta} \in \mathbb{R}^{k+1}$) are optimized by Newton steps to minimize (4.4) over the increased basis set $\mathcal{J}^{k+1}$. The cycle repeats until a maximum number of basis vectors is reached or a stopping criterion is met.

As the size of the basis set increases, the contribution of each additional vector becomes smaller. Given also that re-optimizing $\boldsymbol{\beta}$ with Newton steps becomes more expensive as the basis vector set grows, in practice, an increasing number of new basis vectors are selected between subsequent re-optimization steps.

It should be noted that it is not necessary to recompute the Hessian from scratch between steps. After a set $\tilde{\mathcal{J}}$ of new basis vectors is selected, we must account for the changes in

the $\mathbf{K}_{\mathcal{J}\mathcal{J}}$ and $\mathbf{K}_{\mathcal{J}\mathcal{I}}\mathbf{K}_{\mathcal{J}\mathcal{I}}^{\top}$ terms in the Hessian computation (4.6), since $\mathcal{J}$ must change to incorporate $\tilde{\mathcal{J}}$. It suffices to compute an update to the Hessian based on the new basis vectors chosen and the updated set of error vectors $\mathcal{I}$. More details about both the Hessian update and the basis vector selection heuristics are found in [121].

Observations for Optimizations: There are two ways in which we optimize the candidate selection process. First, the bulk of the time in the scoring process for each candidate is in computing the kernel rows $\mathbf{K}_{\mathcal{C}\mathcal{I}}$ for each candidate $\mathbf{x}_{\mathcal{C}}$ and points violating the margin $\mathbf{x}_{\mathcal{I}}$. The kernel computation itself can be easily performed using existing highly-optimized code for linear algebra operations on the GPU or CPU. Second, each submatrix $\mathbf{K}_{\mathcal{C}\mathcal{I}}$ is narrow—just a few rows—but matrix operations generally scale better for larger matrices. By grouping several candidate sets (between Newton reoptimizations, error vectors $\mathcal{I}$ are held constant), several smaller operations are consolidated into one larger operation with more square dimensions for better efficiency, especially on GPUs.

Implementation Details. When originally proposed, the sparse primal method lacked a well-defined stopping criterion, instead relying on a user-specified maximum number of basis vectors. In SP-SVM, we add basis vectors until the average decrease in training error per additional basis vector is below a fixed threshold, in this case 10^{-5}.

In the subsequent evaluation of SP-SVM, we implement SP-SVM in MATLAB. For linear algebra operations on multicores, we use a combination of built-in MATLAB linear algebra

methods and Intel MKL. For linear algebra operations on the GPU, we use Jacket[12], a MATLAB toolkit for accelerating computations on GPUs. Additionally, we incorporate the freely available C++/CUDA package CUBLAS [145] in cases where Jacket proves to be inefficient or lacks desired functionality. We have subsequently released an optimized C++ version of SP-SVM, called *WU-SVM*, for both multicore and GPU architectures. It is available for download at `http://tinyurl.com/wu-svm`.

4.3 Experimental Results

This section presents an empirical evaluation of several of the algorithms described in sections 4.2.1 and 4.2.2 on two modern parallel architectures: multi-core CPUs (MC) and graphics processing units (GPUs). Running time and accuracy statistics on seven datasets show the benefits and drawbacks of the approaches included in our evaluation.

Hardware. Experiments are run on a 12-core machine with Intel Xeon X5650 processors at 2.67 GHz with hyperthreading enabled and 96 GB of RAM. The attached NVIDIA Tesla C2075 graphics card contains 448 cores and 6 GB of global memory.

[12] `http://www.accelereyes.com/jacket`

Data Set		Method	Test Error (%)	Training Time	Speedup
Adult	SC	LibSVM	14.9	1m 6s	1×
7MB	MC	**LibSVM**	14.9	**10.5s**	**18**×
$n=31562,\ d=123$	MC	SP-SVM	14.8	15.2s	13×
$C=1, \gamma=0.05$ [59]	GPU	GPU SVM	14.9	6s	32×
	GPU	**GTSVM**	14.8	**1s**	**190**×
	GPU	SP-SVM	14.8	11.3s	17×
Covertype/Forest	SC	LibSVM	13.9	5h 1m 19s	1×
96MB	MC	LibSVM	13.9	1h 5m 46s	5×
$n=522911,\ d=54$	MC	**SP-SVM**	13.7	**10m 10s**	**29**×
$C=3, \gamma=1$ [59]	GPU	GPU SVM	13.9	7m 32s	40×
	GPU	GTSVM	36.8	5m 15s	57×
	GPU	**SP-SVM**	13.8	**4m 38s**	**65**×
KDDCup99	SC	LibSVM	7.4	3h 0m 29s	1×
970MB	MC	LibSVM	7.4	26m 37s	7×
$n=4898431,\ d=127$	MC	**SP-SVM**	7.9	**56s**	**193**×
$C=10^6, \gamma=0.137$ [199]	GPU	GPU SVM	–	–	–
	GPU	GTSVM	19.9	1h 15m 39s	2×
	GPU	SP-SVM	–	–	–
MITFaces	SC	LibSVM	5.6†	34m 22s	1×
1.3GB	MC	**LibSVM**	5.6†	**4m 8s**	**8**×
$n=489410,\ d=361$	MC	SP-SVM	7.4†	20s	103×
$C=20, \gamma=0.02$ [199]	GPU	**GPU SVM**	5.7†	**33s**	**61**×
	GPU	GTSVM	5.6†	1m 34s	22×
	GPU	SP-SVM	7.4†	10s	200×
FD	SC	LibSVM	1.4	2h 6m 50s	1×
1.3GB	MC	LibSVM	1.4	27m 54s	5×
$n=200000^*,\ d=900$	MC	**SP-SVM**	1.5	**1m 22s**	**92**×
$C=10, \gamma=1$	GPU	GPU SVM	1.4	6m 20s	20×
	GPU	GTSVM	1.5	2m 26s	52×
	GPU	**SP-SVM**	1.5	**29s**	**262**×
Epsilon	SC	LibSVM	10.9	19h 12m 27s	1×
2.4GB	MC	LibSVM	–	–	–
$n=160000^*,\ d=2000$	MC	**SP-SVM**	10.8	**8m 10s**	**141**×
$C=1, \gamma=0.125$	GPU	GPU SVM	10.9	29m 1s	40×
	GPU	GTSVM	10.9	4m 33s	253×
	GPU	**SP-SVM**	10.8	**1m 55s**	**601**×
MNIST8M (24GB)	SC	LibSVM	1.0	12d 15h 21m 31s	1×
$n=8100000,\ d=784$	MC	**LibSVM**	1.0	**1d 23h 12m 8s**	**6**×
$C=1000, \gamma=0.006$ [135]	MC	SP-SVM	1.4	2h 37m 50s	115×

Table 4.1: Comparison of test error, training time, and speedup of kernelized SVM training methods. The first column indicates dataset file size, number of instances, dimensionality, and SVM hyperparameters C and γ (from cited values, otherwise derived by cross-validation using GTSVM). Results for SP-SVM are the average of five runs with randomly sampled candidate sets (see text for standard deviations). Row colors indicate architecture: single-core (SC), multi-core (MC), GPU. Red font color indicates poor test error rate. **Bold typeface** indicates the best timing results for each dataset and architecture. Symbol † indicates accuracy metric is (1−AUC)%. Symbol − indicates a data set/method pair that was unable to be run, as explained in the text.

Methods evaluated. The *single-threaded* CPU baseline method is LibSVM [43], a popular implementation of SMO, which we use as the baseline for classification accuracy. On *multi-cores* we evaluate a modified version of LibSVM which performs kernel computations in parallel with OpenMP[13]. Further, we evaluate our implementation of SP-SVM in `MATLAB` with Intel MKL BLAS functions for matrix operations. For the *GPU* settings, we compare two explicitly parallel GPU adaptations of dual decomposition: GPU SVM [41], an adaptation of LibSVM for GPUs, and GTSVM [59]. We also include the implicitly parallel `MATLAB` implementation of SP-SVM, linked against the appropriate libraries for GPU linear algebra computations. With the exception of SP-SVM, all implementations are written in `C/C++` by the authors of the respective publications.

Datasets. We evaluate all methods on several medium scale data sets, each involving classification tasks. Medium scale datasets are chosen because parallel runtimes with small datasets tend to be dominated by overhead while large-scale datasets generally require an exorbitant amount of system memory. The datasets are as follows: Adult[14]—an annual income prediction task (greater or less than $50K) based on census data; Covertype/Forest[15]—a tree cover prediction task based on geographical and climate features (predicting class 2 versus the rest); KDDCup99[16]—a classification task for intrusion detection in network traffic; MIT-Faces[17]—a face detection task from raw images (with accuracy presented in (1-AUC)% due to

[13] `http://www.csie.ntu.edu.tw/~cjlin/libsvm/faq.html`
[14] `http://archive.ics.uci.edu/ml/datasets/Adult`
[15] `http://archive.ics.uci.edu/ml/datasets/Covertype`
[16] `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`
[17] `http://c2inet.sce.ntu.edu.sg/ivor/cvm.html`

an extreme class imbalance); Epsilon[18]—a synthetic classification task from the 2008 PASCAL Large Scale Learning Challenge; FD[18]—another face detection task (without heavy class imbalance); and MNIST8M[19]—a multiclass handwritten digit recognition task based on label invariant transformations of images from the MNIST data set. We use the one-versus-one classifier approach to multi-class classification, as also adopted by LibSVM [43].

Features for the datasets Adult, Covertype/Forest, KDDCup99, MITFaces, and MNIST8M are scaled to $[0, 1]$ before training. In addition, we subsample two of the largest data sets, Epsilon and FD, uniformly at random from 400,000 to 160,000 and 5,469,800 to 200,000 respectively for two reasons. First, single core algorithms require prohibitively long training times on the full sets. Second, on GPUs, if the data does not fit into GPU memory the running time is dominated by memory transfer, which is not the focus of this study.

Hyper-parameters. The left column of Table 4.1 provides details of the size and dimensionality of each data set. In addition, it also indicates the regularization parameter C and inverse Gaussian kernel width γ used throughout the experiments. These parameters are derived from cited works for most datasets, as indicated in the table. For Epsilon and FD, a thorough cross-validation grid search was conducted using GTSVM as it is an exact implementation and tends to behave identically to LibSVM in terms of hyper parameters but does not have the large time requirement of cross validating with LibSVM. This approach does a slight disservice to SP-SVM, however it may be viewed as a fair compromise as LibSVM

[18] `http://largescale.ml.tu-berlin.de/instructions/`
[19] `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html`

is the gold standard and our main focus is the speedup. Throughout all experiments with SP-SVM we set the stopping criterion to $\epsilon = 5 \times 10^{-6}$.

Evaluation. Table 4.1 shows test error, training time, and speedup versus single-core LibSVM for all methods on each of the seven data sets. The training times omit both loading data from disk and computing test predictions for all methods. As MNIST8M is multi-class, the times reported are the accumulative time for each one-versus-one classifier trained individually.[20]

Since SP-SVM deploys a heuristic based on random sampling of basis vectors, we computed five runs for each setting and report the average runtime and test error. Standard deviations on SP-SVM test error are less than 0.001 for all datasets except for the multicore implementation on KDDCup99 (0.0023). Similarly, standard deviations for SP-SVM training time are on the order of seconds for each run. (For increased readability, we omit them from the table.)

Not all algorithms converge on all data sets. GTSVM is the only GPU method that runs on KDDCup99 (which is 90% sparse). GPU SVM and SP-SVM both store the inputs in dense format on the GPU, which exceed its memory. The dense MNIST8M data is too large for all

[20]Shared memory computers, such as multi-core CPUs and GPUs, are arguably less suited for this kind of multi-class classification, since one-versus-one classifiers are "embarrassingly parallel" for problems with many classes and can be solved on (cheaper) distributed memory machines (clusters) with near-perfect speedup.

GPU algorithms.[21] Also, LibSVM with OpenMP failed to converge on Epsilon in less time than single-core LibSVM.

Accuracy. For most datasets and methods, test errors are remarkably consistent, even between exact and approximate methods. However there are a few notable exceptions, highlighted in red in Table 4.1. GTSVM fails on Covertype/Forest and we hypothesize that this anomaly may be due to a floating point precision error as the method converges when run on smaller subsets of the training data. On KDDCup99, GTSVM obtains an error rate of 19.9%, which is not significantly better than a constant predicting the most common class (no GPU method in our evaluation could successfully learn from this data). SP-SVM performs slightly worse on KDDCup99 (7.9% vs. 7.4%) and noticeably worse on MITFaces (7.4% 1-AUC vs. 5.6%) and MNIST8M (1.4% vs. 1.0%). The approximation error may be more pronounced on MITFaces due to the large class imbalance (a few additional false positives have a strong effect on the final area under the curve) and also for MNIST8M, where the approximation error is being aggregated across the many (45) one-versus-one classifiers.

Speedup. The most basic method of speedup is LibSVM on multicores. This involves a trivial change directly to the source of LibSVM, allowing for the use of OpenMP parallel for-loops in kernel computations. Because kernel computations account for such a significant

[21] As GPU memory sizes grow, this limitation will become less important. In addition, GPUs and CPUs might eventually converge on using a single memory space. For sparse data sets one might also consider special purpose libraries, such as CUSPARSE (`https://developer.nvidia.com/cusparse`), for the kernel computation.

portion of LibSVM's runtime, this baseline improvement results in a $5-8\times$ speedup on twelve cores.

GPU SVM achieves $20-40\times$ speedups over single-core LibSVM by performing kernel computations and KKT condition updates directly on the GPU. GTSVM achieves the largest speedups among the dual decomposition methods, by also increasing the working set size to 16 (compared to 2 used by LibSVM and GPU SVM), leading to $2.5-6.5\times$ speedup over GPU SVM, and $2-250\times$ speedup over LibSVM. This highlights the correlation between speedup and the amount of handcrafted parallelism that is included in the algorithm design for the explicit parallel approaches.

In comparison to single core LibSVM, SP-SVM achieves $13\times$ to $193\times$ speedup on multi-cores, and $17\times$ up to $601\times$ speedup on GPUs. On both architectures, the speedup of SP-SVM tends to increase with data set size, which reflects the increasing time spent inside parallelized library operations. The smallest speedup for both architectures is on the smallest data set, Adult—however, by a mere $11s$ or $15s$ compared to the fastest algorithm (GTSVM). It is surprising just how effective the parallelism derived from the dense linear algebra in SP-SVM proves to be on both architectures. SP-SVM is particularly effective on GPUs where it outperforms all other GPU methods by $1.5\times$ to $5\times$ on all but Adult, and achieves a $1.3-4.3\times$ speedup over multi-core SP-SVM. However even on multi-cores, SP-SVM outperforms GPU SVM and GTSVM significantly on MITFaces and FD. SP-SVM requires only 11 minutes on

average across all binary classification datasets, compared to the several hours often required by LibSVM.

4.4 Discussion

One trend clearly follows from our study: massive speedups are possible when the parallelism of modern hardware is leveraged. Explicit parallelization is by far the most common approach to SVM parallelization. Our results demonstrate the significant benefits of implicit parallelization. By adopting an appropriate SVM relaxation, we leverage existing, highly efficient libraries for parallel linear algebra. This approximation, by heuristically or randomly choosing a reduced basis vector set, unlocks dramatic parallel speedups with very limited loss of accuracy.

We believe that the community can benefit from our findings in two ways: First, practitioners obtain an easy to use implementation of SP-SVM with unprecedented training speed which can be readily used on or adapted to most platforms with BLAS compatible libraries. Second, researchers working on parallel machine learning algorithms may reconsider spending days in agony on C/C++ programming of parallel code and may instead focus on designing reformulations or approximations to their algorithms which rely more heavily on dense linear algebra routines. Similar to relaxing optimization problems into convexity, as has been common practice for years, we predict that relaxations into implicit parallelization may

become increasingly important as multi-cores and GPUs establish themselves as the common computing platforms.

This suggests general principles that can help us in both deciding which specific algorithms should be targets for parallelization and designing new algorithms specifically for parallel architectures such as GPUs. First, *coarse-grained iterative algorithms*—algorithms that perform a large amount of computation on each step and then take a smaller number of steps, are generally better than fine-grained iterative solutions.

This hints at what may be an important principle for the design of algorithms meant for machines with a very large number of cores. There are often many ways of solving any given machine learning problem. For years, researchers have been focusing on sequential performance and developing algorithms that minimize the amount of *work*—roughly, the total number of instructions executed by the program. As hardware is developed with 100's or 1000's of cores, we must consider another parameter, namely the *critical path length*—roughly, the longest chain of dependent instructions. When we consider machines with 100's or 1000's of cores, we must design algorithms that minimize the critical path length even at the cost of increasing the work by some small factor.

Chapter 5

Conclusions

Improving the feasibility of learning large-scale non-linear machine learning models is an area of active research. Ever-increasing parallel and distributed computing resources hold the promise of enhanced scalability. However, efficiently transferring sequential implementations of popular non-linear methods to parallel platforms has not proven to be a straightforward task. Rather, the result is parallel implementations with tedious code, limited speedup, and difficult transferability to new or updated platforms.

In this work, we have taken an approach driven by approximation to achieve significant parallel speedup and scalability for three classes of non-linear models. We replace exact methods with strategically relaxed algorithms for boosted tree learning, metric learning, and support vector machines. Our parallel and distributed implementations demonstrate scalability to larger datasets and faster execution time than previously possible. Further,

the code produced readily bridges multiple parallel platforms and leverages existing, highly-optimized parallel libraries, improving both performance and transferability.

Approximation will be an important trend moving forward, as parallelism, scalability, and model power remain vitally important attributes of novel machine learning algorithms. With larger training sets and ubiquitous parallel computing resources, exact optimizations will continue to give way to the strategic use of approximation and randomness in the non-essential sections of learning procedures, trading precision for more data and greater model complexity.

References

[1] AccelerEyes. Jacket. `http://www.accelereyes.com/jacket`.

[2] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[3] N. Amado, J. Gama, and F. Silva. Parallel implementation of decision tree learning algorithms. *Progress in Artificial Intelligence*, pages 34–52, 2001.

[4] Amazon. Amazon Elastic MapReduce, October 2014. `http://aws.amazon.com/elasticmapreduce/`.

[5] Nima Asadi, Jimmy Lin, and Arjen P. de Vries. Runtime optimizations for prediction with tree-based models. *IEEE Transactions on Knowledge and Data Engineering*, 99(1):1, 2013.

[6] A. Asuncion, M. Welling, P. Smyth, and Y. W. Teh. On smoothing and inference for topic models. In *Proceedings of the International Conference on Uncertainty in Artificial Intelligence*, 2009.

[7] A. Athanasopoulos, A. Dimou, V. Mezaris, and I. Kompatsiaris. GPU acceleration for support vector machines. In *WIAMIS 2011, Delft, Netherlands*, 2011.

[8] B. Babenko, S. Branson, and S. Belongie. Similarity metrics for categorization: from monolithic to category specific. In *ICCV '09*, pages 293–300. IEEE, 2009.

[9] T.C. Bailey, Y. Chen, Y. Mao, C. Lu, G. Hackmann, S.T. Micek, K.M. Heard, K.M. Faulkner, and M.H. Kollef. A trial of a real-time alert for clinical deterioration in patients hospitalized on general medical wards. *Journal of Hospital Medicine*, 2013.

[10] Satish Balay, Jed Brown, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.4, Argonne National Laboratory, 2013.

[11] Jonathan Baxter. Theoretical models of learning to learn. In S. Thrun and L. Pratt, editors, *Learning to learn*, pages 71–94. Kluwer Academic Publishers, 1998.

[12] Jonathan Baxter. A model of inductive bias learning. *Journal of Artificial Intelligence Research*, 12:149–198, 2000.

[13] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *Computer Vision–ECCV 2006*, pages 404–417, 2006.

[14] A. Beck and M. Teboulle. Mirror descent and nonlinear projected subgradient methods for convex optimization. *Operations Research Letters*, 31(3):167–175, 2003.

[15] Y. Ben-Haim and E. Yom-Tov. A streaming parallel decision tree algorithm. *The Journal of Machine Learning Research*, 11:849–872, 2010.

[16] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[17] T. L. Berg, A. C. Berg, J. Edwards, M. Maire, R. White, Y. W. Teh, E. Learned-Miller, and D. A. Forsyth. Names and faces. Submitted, 2007.

[18] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *The Journal of Machine Learning Research*, 3:993–1022, 2003.

[19] Antoine Bordes, Léon Bottou, and Patrick Gallinari. Sgd-qn: Careful quasi-newton stochastic gradient descent. *The Journal of Machine Learning Research*, 10:1737–1754, 2009.

[20] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Compstat*, volume 2010, pages 177–186, 2010.

[21] L. Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[22] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[23] L. Breiman and A. Cutler. Random forests. `http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm#prox`.

[24] Leo Breiman, Jerome Friedman, Charles J. Stone, and R. A. Olshen. *Classification and regression trees*. Chapman & Hall/CRC, 1984.

[25] J. Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. *Advances in neural information processing systems (NIPS)*, 1990.

[26] CG Broyden. Quasi-newton methods. *Numerical Methods for Unconslrained Optimization, W. Murray, Ed. New York: Academic I*, 972, 1970.

[27] Dominik Brugger. Parallel support vector machines. 2007. `http://tobias-lib.uni-tuebingen.de/volltexte/2007/2768/pdf/tech_21.pdf`.

[28] Chris Burges, Tal Shaked, Erin Renshaw, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Internation Conference on Machine Learning*, pages 89–96, 2005.

[29] Christopher J. C. Burges, Robert Ragno, and Quoc V. Le. Learning to rank with nonsmooth cost functions. In Bernhard Schölkopf, John C. Platt, Thomas Hoffman, Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 193–200. MIT Press, 2006.

[30] Christopher J.C. Burges, Krysta M. Svore, Paul N. Bennett, Andrzej Pastusiak, and Qiang Wu. Learning to rank using an ensemble of lambda-gradient models. *Journal of Machine Learning Research, Workshop and Conference Proceedings*, 14:25–35, 2011.

[31] C.J.C. Burges. From RankNet to LambdaRank to LambdaMART: An Overview. 2010.

[32] J. F. Cai, W. S. Lee, and Y. W. Teh. Improving word sense disambiguation using topic features. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-coNLL)*, pages 1015–1023, 2007.

[33] J. F. Cai, W. S. Lee, and Y. W. Teh. NUS-ML: Improving word sense disambiguation using topic features. In *Proceedings of the International Workshop on Semantic Evaluations*, volume 4, 2007.

[34] P.B. Callahan and S.R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM (JACM)*, 42(1):67–90, 1995.

[35] Li Juan Cao, SS Keerthi, Chong-Jin Ong, JQ Zhang, Uvaraj Periyathamby, Xiu Ju Fu, and HP Lee. Parallel sequential minimal optimization for the training of support vector machines. *Neural Networks, IEEE Transactions on*, 17(4):1039–1049, 2006.

[36] Y. Cao, J. Xu, T.Y. Liu, H. Li, Y. Huang, and H.W. Hon. Adapting ranking SVM to document retrieval. In *Proc. 29th Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 186–193, 2006.

[37] Zhe Cao and Tie-Yan Liu. Learning to rank: From pairwise approach to listwise approach. In *Proceedings of the 24th International Conference on Machine Learning*, pages 129–136, 2007.

[38] D. Caragea, A. Silvescu, and V. Honavar. A framework for learning from distributed data using sufficient statistics and its application to learning decision trees. *International Journal of Hybrid Intelligent Systems*, 1:80–89, 2004.

[39] A. Carpenter. CUSVM: A CUDA implementation of support vector classification and regression. 2009.

[40] Rich Caruana. Multitask learning. In *Machine Learning*, pages 41–75, 1997.

[41] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th International Conference on Machine Learning*, pages 104–111. ACM, 2008.

[42] C.-C. Chang. A boosting approach for supervised mahalanobis distance metric learning. *Pattern Recognition*, 45(2):844–862, 2012. cited By (since 1996) 0.

[43] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

[44] Hong Chang and Dit-Yan Yeung. Kernel-based distance metric learning for content-based image retrieval. *Image Vision Comput.*, 25(5):695–703, May 2007.

[45] O. Chapelle, D. Metlzer, Y. Zhang, and P. Grinspan. Expected reciprocal rank for graded relevance. In *Proceeding of the 18th ACM Conference on Information and Knowledge Management*, pages 621–630. ACM, 2009.

[46] O. Chapelle and M. Wu. Gradient descent optimization of smoothed information retrieval metrics. *Information Retrieval Journal*, Special Issue on Learning to Rank for Information Retrieval, 2010. to appear.

[47] Olivier Chapelle. Training a support vector machine in the primal. *Neural Computation*, 19:1155–1178, 2007.

[48] Olivier Chapelle and Yi Chang. Yahoo! Learning to Rank Challenge overview. *Journal of Machine Learning Research, Workshop and Conference Proceedings*, 14:1–24, 2011.

[49] Olivier Chapelle and Sathiya S. Keerthi. Efficient algorithms for ranking with SVMs. *Information Retrieval*, 2009. to appear.

[50] Olivier Chapelle and Mingrui Wu. Gradient descent optimization of smoothed information retrieval metrics. *Information Retrieval*, 2009. to appear.

[51] Ratthachat Chatpatanasiri, Teesid Korsrilabutr, Pasakorn Tangchanachaianan, and Boonserm Kijsirikul. A new kernelization framework for mahalanobis distance learning algorithms. *Neurocomputing*, 73:1570 – 1579, 2010.

[52] Depin Chen, Yan Xiong, Jun Yan, Gui-Rong Xue, Gang Wang, and Zheng Chen. Knowledge transfer for cross domain learning to rank. *Information Retrieval*, 2009.

[53] H. L. Chieu, W. S. Lee, and Y. W. Teh. Cooled and relaxed survey propagation for MRFs. In *Advances in Neural Information Processing Systems*, volume 20, 2008.

[54] S. Chopra, R. Hadsell, and Y. LeCun. Learning a similarity metric discriminatively, with application to face verification. In *CVPR '05*, pages 539–546. IEEE, 2005.

[55] R. Collobert and J. Weston. A unified architecture for NLP: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM New York, NY, USA, 2008.

[56] Ronan Collobert, Samy Bengio, and Yoshua Bengio. A parallel mixture of svms for very large scale problems. *Neural computation*, 14(5):1105–1114, 2002.

[57] Corinna Cortes and Vladimir Vapnik. Support-vector networks. In *Machine Learning*, pages 273–297, 1995.

[58] David Cossock and Tong Zhang. Subset ranking using regression. In *Proc. Conf. on Learning Theory*, pages 605–619, 2006.

[59] A. Cotter, N. Srebro, and J. Keshet. A GPU-tailored approach for training kernelized svms. In *Proceedings of the 17th ACM SIGKDD*, pages 805–813. ACM, 2011.

[60] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[61] O.G. Cula and K.J. Dana. 3D texture recognition using bidirectional feature histograms. *International Journal of Computer Vision*, 59(1):33–60, 2004.

[62] M. Cuturi and D. Avis. Ground metric learning. *arXiv preprint, arXiv:1110.2306*, 2011.

[63] W. Dai, Q. Yang, G.R. Xue, and Y. Yu. Boosting for transfer learning. In *Proceedings of the 24th international conference on Machine learning*, pages 193–200. ACM, 2007.

[64] J. Darlington, Y. Guo, J. Sutiwaraphun, and H. To. Parallel induction algorithms for data mining. *Advances in Intelligent Data Analysis Reasoning about Data*, pages 437–445, 1997.

[65] Jason V. Davis, Brian Kulis, Prateek Jain, Suvrit Sra, and Inderjit S. Dhillon. *Information Theoretic Metric Learning.* UT, Austin, http://www.cs.utexas.edu/users/pjain/itml/.

[66] J.V. Davis, B. Kulis, P. Jain, S. Sra, and I.S. Dhillon. Information-theoretic metric learning. In *ICML '07*, pages 209–216. ACM, 2007.

[67] Jian-xiong Dong, Adam Krzyżak, and Ching Y. Suen. A fast parallel optimization for training support vector machine. In Petra Perner and Azriel Rosenfeld, editors, *Machine Learning and Data Mining in Pattern Recognition*, volume 2734 of *Lecture Notes in Computer Science*, pages 96–105. Springer Berlin Heidelberg, 2003.

[68] F. Doshi, K. T. Miller, J. Van Gael, and Y. W. Teh. Variational inference for the Indian buffet process. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, volume 12, 2009.

[69] Kai-Bo Duan and Sathiya Keerthi. Which is the best multiclass svm method? an empirical study. *Multiple Classifier Systems*, pages 732–760, 2005.

[70] Tatjana Eitrich and Bruno Lang. *Efficient implementation of serial and parallel support vector machine training with a multi-parameter kernel for large-scale data mining.* World Academy of Science, Engineering and Technology, 2005. `http://www.waset.org/journals/waset/v11/v11-130.pdf`.

[71] Tatjana Eitrich and Bruno Lang. Data mining with parallel support vector machines for classification. *Advances in Information Systems*, pages 197–206, 2006.

[72] T. Evgeniou, C.A. Micchelli, and M. Pontil. Learning multiple tasks with kernel methods. *Journal of Machine Learning Research*, 6(1):615, 2006.

[73] Theodoros Evgeniou and Massimiliano Pontil. Regularized multi–task learning. In *KDD*, pages 109–117, 2004.

[74] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, June 2008.

[75] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137. ACM, 2004.

[76] L. Fei-Fei, R. Fergus, and P. Perona. One-shot learning of object categories. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(4):594–611, 2006.

[77] Vojtěch Franc and Soeren Sonnenburg. Optimized cutting plane algorithm for support vector machines. In *Proceedings of the 25th International Conference on Machine learning*, pages 320–327. ACM, 2008.

[78] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[79] A.A. Freitas and S.H. Lavington. *Mining very large databases with parallel processing.* Springer, 1998.

[80] Yoav Freund, Raj Iyer, Robert E. Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *J. Mach. Learn. Res.*, 4:933–969, 2003.

[81] J. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2001.

[82] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, 28:2000, 1998.

[83] Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics and Data Analysis*, 38:367–378, 1999.

[84] J.H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, pages 1189–1232, 2001.

[85] C. Galleguillos, B. McFee, S. Belongie, and G. Lanckriet. Multi-class object localization by combining local contextual interactions. *CVPR '10*, pages 113–120, 2010.

[86] J. Gao, Q. Wu, C. Burges, K. Svore, Y. Su, N. Khan, S. Shah, and H. Zhou. Model adaptation via model interpolation and boosting for web search ranking. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 505–513. Association for Computational Linguistics, 2009.

[87] J. Gasthaus, F. Wood, D. Görür, and Y. W. Teh. Dependent Dirichlet process spike sorting. In *Advances in Neural Information Processing Systems*, volume 21, 2009.

[88] J. Gehrke, R. Ramakrishnan, and V. Ganti. RainForest: A framework for fast decision tree construction of large datasets. *Data Mining and Knowledge Discovery*, 4(2):127–162, 2000.

[89] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Data Bases*, pages 518–529, 1999.

[90] A. Globerson and S. Roweis. Metric learning by collapsing classes. In *NIPS '06*, pages 451–458. MIT Press, 2006.

[91] A. Globerson and S. Roweis. Visualizing pairwise similarity via semidefinite programming. In *AISTATS '07*, pages 139–146, 2007.

[92] J. Goldberger, G.E. Hinton, S.T. Roweis, and R. Salakhutdinov. Neighbourhood components analysis. In *NIPS*, pages 513–520. MIT Press, 2004.

[93] R. Gopalan, R. Li, and R. Chellappa. Domain adaptation for object recognition: An unsupervised approach. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 999–1006. IEEE, 2011.

[94] D. Görür and Y. W. Teh. An efficient sequential Monte Carlo algorithm for coalescent clustering. In *Advances in Neural Information Processing Systems*, volume 21, 2009.

[95] Hans Peter Graf, Eric Cosatto, Leon Bottou, Igor Dourdanovic, and Vladimir Vapnik. Parallel support vector machines: The cascade svm. *Advances in neural information processing systems*, 17:521–528, 2004.

[96] Håkan Grahn, Niklas Lavesson, Mikael Hellborg Lapajne, and Daniel Slat. Cudarf: A cuda-based implementation of random forests. In *IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*, pages 95–101, 2011.

[97] G. Griffin, A. Holub, and P. Perona. Caltech-256 object category dataset. 2007.

[98] Shalini Gupta. Efficient object detection on gpus using mb-lbp features and random forests. Talk at GPU Technology Conference, 2013.

[99] G. R. Haffari and Y. W. Teh. Hierarchical Dirichlet trees for information retrieval. In *Proceedings of the Annual Meeting of the North American Association for Computational Linguistics and the Human Language Technology Conference*, 2009.

[100] J. Hafner, H.S. Sawhney, W. Equitz, M. Flickner, and W. Niblack. Efficient color histogram indexing for quadratic form distance functions. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 17(7):729–736, 1995.

[101] Tamir Hazan, Amit Man, and Amnon Shashua. A parallel decomposition solver for svm: Distributed dual ascend using fenchel duality. In *CVPR 2008*, pages 1–8. IEEE, 2008.

[102] Yujie He, Wenlin Chen, and Yixin Chen. Kernel density metric learning. In *Proceedings of IEEE International Conference on Data Mining*, 2013.

[103] K. A. Heller, Y. W. Teh, and D. Görür. Infinite hierarchical hidden Markov models. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, volume 12, 2009.

[104] Ralf Herbrich, Thore Graepel, and Klaus Obermayer. *Large margin rank boundaries for ordinal regression*, pages 115–132. MIT Press, Cambridge, MA, 2000.

[105] Tom Heskes. Empirical bayes for learning to learn. In *Proceedings of ICML*, pages 367–374. Morgan Kaufmann, 2000.

[106] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems, 1952.

[107] G. E. Hinton, S. Osindero, and Y. W. Teh. A fast learning algorithm for deep belief networks. *Neural Computation*, 18(7):1527–1554, 2006.

[108] G. E. Hinton, S. Osindero, M. Welling, and Y. W. Teh. Unsupervised discovery of non-linear structure using contrastive backpropagation. *Cognitive Science*, 30(4), 2006.

[109] G.E. Hinton and S.T. Roweis. Stochastic neighbor embedding. In *NIPS*, pages 833–840. MIT Press, 2002.

[110] M. Hoffman, D. Blei, and P. Cook. Easy as CBA: A simple probabilistic model for tagging music. In *ISMIR '09*, pages 369–374, 2009.

[111] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[112] Intel. Intel math kernel library. `http://software.intel.com/en-us/intel-mkl/`.

[113] P. Jain, B. Kulis, J.V. Davis, and I.S. Dhillon. Metric and kernel learning using a linear transformation. *Journal of Machine Learning Research*, 13:519–547, 03 2012.

[114] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

[115] K. Jarvelin and J. Kekalainen. IR evaluation methods for retrieving highly relevant documents. In *ACM Special Interest Group in Information Retrieval (SIGIR)*, pages 41–48. New York: ACM, 2002.

[116] T. Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 2002.

[117] Thorsten Joachims. Advances in kernel methods. chapter Making large-scale support vector machine learning practical, pages 169–184. MIT Press, Cambridge, MA, USA, 1999.

[118] I.T. Jolliffe and MyiLibrary. *Principal component analysis*, volume 2. Wiley Online Library, 2002.

[119] D. Kedem, S. Tyree, K.Q. Weinberger, F. Sha, and G. Lanckriet. Non-linear metric learning. In *NIPS*, pages 2582–2590, 2012.

[120] S Sathiya Keerthi and Dennis DeCoste. A modified finite Newton method for fast solution of large scale linear SVMs. *Journal of Machine Learning Research*, 6(1):341, 2006.

[121] S.S. Keerthi, O. Chapelle, and D. DeCoste. Building support vector machines with reduced classifier complexity. *Journal of Machine Learning Research*, 7(7):1493–1515, 2006.

[122] George S Kimeldorf and Grace Wahba. A correspondence between bayesian estimation on stochastic processes and smoothing by splines. *The Annals of Mathematical Statistics*, pages 495–502, 1970.

[123] Yehuda Koren. The bellkor solution to the netflix grand prize. *Netflix prize documentation*, 81, 2009.

[124] Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1106–1114, 2012.

[125] K. Kurihara, M. Welling, and Y. W. Teh. Collapsed variational Dirichlet process mixture models. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 20, 2007.

[126] G. Latouche, V. Ramaswami, and VG Kulkarni. Introduction to matrix analytic methods in stochastic modeling. *Journal of Applied Mathematics and Stochastic Analysis*, 12(4):435–436, 1999.

[127] A. Lazarevic and Z. Obradovic. Boosting algorithms for parallel and distributed learning. *Distributed and Parallel Databases*, 11(2):203–229, 2002.

[128] G. Lebanon. Metric learning for text documents. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(4):497–508, 2006.

[129] W. S. Lee, X. Zhang, and Y. W. Teh. Semi-supervised learning in reproducing kernel Hilbert spaces using local invariances. Technical Report TRB3/06, School of Computing, National University of Singapore, 2006.

[130] P. Li, C. Burges, and Q. Wu. McRank: Learning to rank using multiple classification and gradient boosting. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 21*, pages 897–904. MIT Press, Cambridge, MA, 2008.

[131] Yisheng Liao, Alex Rubinsteyn, Russell Power, and Jinyang Li. Learning random forests on the gpu.

[132] Y. J. Lim and Y. W. Teh. Variational Bayesian approach to movie rating prediction. In *Proceedings of KDD Cup and Workshop*, 2007.

[133] T. Liu, A.W. Moore, A. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. *Advances in neural information processing systems (NIPS)*, 2004.

[134] T.Y. Liu, J. Xu, T. Qin, W. Xiong, and H. Li. Letor: Benchmark dataset for research on learning to rank for information retrieval. In *Proceedings of SIGIR 2007 Workshop on Learning to Rank for Information Retrieval*, pages 3–10, 2007.

[135] Gaëlle Loosli, Stéphane Canu, and Léon Bottou. Training invariant support vector machines using selective sampling. In Léon Bottou, Olivier Chapelle, Dennis DeCoste, and Jason Weston, editors, *Large Scale Kernel Machines*, pages 301–320. MIT Press, Cambridge, MA., 2007.

[136] C.L. Mallows. A note on asymptotic joint normality. *The Annals of Mathematical Statistics*, pages 508–515, 1972.

[137] Olvi L Mangasarian. A finite Newton method for classification. *Optimization Methods and Software*, 17(5):913–929, 2002.

[138] L. Mason, J. Baxter, P. Bartlett, and M. Frean. Boosting algorithms as gradient descent in function space. In *Neural information processing systems*, volume 12, pages 512–518, 2000.

[139] B. McFee and G. Lanckriet. Metric learning to rank. In *27th International Conference on Machine Learning, Haifa, Israel*. Citeseer, 2010.

[140] S. Merler, B. Caprile, and C. Furlanello. Parallelizing AdaBoost by weights dynamics. *Computational statistics & data analysis*, 51(5):2487–2498, 2007.

[141] K. Mikolajczyk and C. Schmid. Indexing based on scale invariant interest points. In *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, volume 1, pages 525–531. IEEE, 2001.

[142] Ananth Mohan, Zheng Chen, and Kilian Q. Weinberger. Web-search ranking with initialized gradient boosted regression trees. *Journal of Machine Learning Research, Workshop and Conference Proceedings*, 14:77–89, 2011.

[143] A.M. Mood, F.A. Graybill, and D.C. Boes. *Introduction in the theory of statistics*. McGraw-Hill International Book Company, 1963.

[144] Multisvm. Multiclass SVM in CUDA, November 2009. `http://code.google.com/p/multisvm/`.

[145] NVIDIA. CUDA basic linear algebra subroutines, 2012. `http://tinyurl.com/cublas`.

[146] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[147] B. Panda, J. Herbach, S. Basu, and R.J. Bayardo. Planet: Massively parallel learning of tree ensembles with mapreduce. *Proceedings of the Very Large Database Endowment*, 2(2):1426–1437, 2009.

[148] Dmitry Pavlov and Cliff Brunk. Bagboo: Bagging the gradient boosting. Talk at Workshop on Websearch Ranking at the 27th International Conference on Machine Learning, 2010.

[149] O. Pele and M. Werman. The quadratic-chi histogram distance family. *ECCV '10*, pages 749–762, 2010.

[150] John C. Platt. Fast Training of Support Vector Machines Using Sequential Minimal Optimization. 1998.

[151] John C Platt, Nello Cristianini, and John Shawe-Taylor. Large margin dags for multiclass classification. In *nips*, volume 12, pages 547–553, 1999.

[152] G. Quon, Y. W. Teh, E. Chan, M. Brudno, T. Hughes, and Q. D. Morris. A mixture model for the evolution of gene expression in non-homogeneous datasets. In *Advances in Neural Information Processing Systems*, volume 21, 2009.

[153] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. *Advances in neural information processing systems*, 20:1177–1184, 2007.

[154] Stephen Robertson, Hugo Zaragoza, and Michael Taylor. Simple bm25 extension to multiple weighted fields. In *CIKM '04: Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 42–49, New York, NY, USA, 2004. ACM.

[155] Saharon Rosset, Ji Zhu, Trevor Hastie, and Robert Schapire. Boosting as a regularized path to a maximum margin classifier. *Journal of Machine Learning Research*, 5:941–973, 2004.

[156] D. M. Roy and Y. W. Teh. The Mondrian process. In *Advances in Neural Information Processing Systems*, volume 21, 2009.

[157] Y. Rubner, C. Tomasi, and L.J. Guibas. The earth mover's distance as a metric for image retrieval. *International Journal of Computer Vision*, 40(2):99–121, 2000.

[158] C. Rudin and R.E. Schapire. Margin-based Ranking and an Equivalence between AdaBoost and RankBoost. *The Journal of Machine Learning Research*, 10:2193–2232, 2009.

[159] K. Saenko, B. Kulis, M. Fritz, and T. Darrell. Adapting visual category models to new domains. *Computer Vision–ECCV 2010*, pages 213–226, 2010.

[160] R. Salakhutdinov and G. Hinton. Learning a nonlinear embedding by preserving class neighbourhood structure. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, volume 11, 2007.

[161] R.E. Schapire. A brief introduction to boosting. In *IJCAI*, volume 16, pages 1401–1406, 1999.

[162] R.E. Schapire, Y. Freund, P. Bartlett, and W.S. Lee. Boosting the margin: A new explanation for the effectiveness of voting methods. *The annals of statistics*, 26(5):1651–1686, 1998.

[163] B. Schölkopf. The kernel trick for distances. *NIPS*, pages 301–307, 2001.

[164] B. Schölkopf and A.J. Smola. *Learning with kernels: Support vector machines, regularization, optimization, and beyond*. MIT press, 2001.

[165] Fei Sha, Yuanqing Lin, Lawrence K. Saul, and Daniel D. Lee. Multiplicative updates for nonnegative quadratic programming. *Neural Computation*, 19(8):2004–2031, 2007.

[166] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proceedings of the International Conference on Very Large Data Bases*, pages 544–555, 1996.

[167] G. Shakhnarovich. *Learning task-specific similarity*. PhD thesis, Massachusetts Institute of Technology, 2005.

[168] G. Shakhnarovich, P. Viola, and T. Darrell. Fast pose estimation with parameter-sensitive hashing. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 750–757. IEEE, 2003.

[169] Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro. Pegasos: Primal estimated sub-gradient solver for SVM. In *Proceedings of the 24th international conference on Machine learning*, pages 807–814. ACM, 2007.

[170] Toby Sharp. Implementing decision trees and forests on a gpu. In *ECCV 2008*, pages 595–608. Springer, 2008.

[171] N. Shental, T. Hertz, D. Weinshall, and M. Pavel. Adjustment learning and relevant component analysis. In *ECCV '02*, volume 4, pages 776–792. Springer-Verlag, 2002.

[172] Pannaga Shivaswamy, Olivier Chapelle, Kilian Q. Weinberger, Srinivas Vadrevu, Ya Zhang, and Belle Tseng. Multi-task boosting applied to web search ranking. In *AISTATS*, 2010. submitted.

[173] J Shotton, D Robertson, and T Sharp. Efficient implementation of decision forests. In *Decision Forests for Computer Vision and Medical Image Analysis*, pages 313–332. Springer, 2013.

[174] Jamie Shotton, Toby Sharp, Pushmeet Kohli, Sebastian Nowozin, John Winn, and Antonio Criminisi. Decision jungles: Compact and rich models for classification. In *Advances in Neural Information Processing Systems*, pages 234–242, 2013.

[175] Daniel Slat and Mikael Hellborg Lapajne. Random forests for CUDA GPUs. 2010.

[176] Alexander J Smola, SVN Vishwanathan, and Quoc Le. Bundle methods for machine learning. *Advances in neural information processing systems*, 20:1377–1384, 2008.

[177] M. Snir. *MPI–the Complete Reference: The MPI core*. The MIT Press, 1998.

[178] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959, 2012.

[179] B. Sriperumbudur, O. Lang, and G. Lanckriet. Metric embedding for kernel classification rules. pages 1008–1015, 2008.

[180] B.K. Sriperumbudur, A. Gretton, K. Fukumizu, B. Schölkopf, and G.R.G. Lanckriet. Hilbert space embeddings and metrics on probability measures. *The Journal of Machine Learning Research*, 11:1517–1561, 2010.

[181] A. Srivastava, E.H. Han, V. Kumar, and V. Singh. Parallel formulations of decision-tree classification algorithms. *High Performance Data Mining*, pages 237–261, 2002.

[182] M. Stricker and M. Orengo. Similarity of color images. In *Storage and Retrieval for Image and Video Databases*, volume 2420, pages 381–392, 1995.

[183] Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[184] M. Taylor, J. Guiver, S. Robertson, and T. Minka. SoftRank: optimizing non-smooth rank metrics. In *Proc. 1st ACM Int'l Conf. on Web Search and Data Mining*, pages 77–86, 2008.

[185] Y. W. Teh. A Bayesian interpretation of interpolated Kneser-Ney. Technical Report TRA2/06, School of Computing, National University of Singapore, 2006.

[186] Y. W. Teh. A hierarchical Bayesian language model based on Pitman-Yor processes. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 985–992, 2006.

[187] Y. W. Teh. Dirichlet processes. Submitted to Encyclopedia of Machine Learning, 2007.

[188] Y. W. Teh, H. Daumé III, and D. M. Roy. Bayesian agglomerative clustering with coalescents. In *Advances in Neural Information Processing Systems*, volume 20, 2008.

[189] Y. W. Teh, D. Görür, and Z. Ghahramani. Stick-breaking construction for the Indian buffet process. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, volume 11, 2007.

[190] Y. W. Teh and M. I. Jordan. Hierarchical Bayesian nonparametric models with applications. In N. Hjort, C. Holmes, P. Müller, and S. Walker, editors, *To appear in Bayesian Nonparametrics: Principles and Practice*. Cambridge University Press, 2010.

[191] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei. Hierarchical Dirichlet processes. *Journal of the American Statistical Association*, 101(476):1566–1581, 2006.

[192] Y. W. Teh, K. Kurihara, and M. Welling. Collapsed variational inference for HDP. In *Advances in Neural Information Processing Systems*, volume 20, 2008.

[193] Y. W. Teh, D. Newman, and M. Welling. A collapsed variational Bayesian inference algorithm for latent Dirichlet allocation. In *Advances in Neural Information Processing Systems*, volume 19, 2007.

[194] Y. W. Teh, M. Seeger, and M. I. Jordan. Semiparametric latent factor models. In *Proceedings of the International Workshop on Artificial Intelligence and Statistics*, volume 10, 2005.

[195] Igor V. Tetko, David J. Livingstone, and Alexander I. Luik. Neural network studies, 1. comparison of overfitting and overtraining. *Journal of Chemical Information and Computer Sciences*, 35(5), 1995.

[196] L. Torresani and K. Lee. Large margin component analysis. *Advances in neural information processing systems (NIPS)*, 19:1385, 2007.

[197] M.F. Tsai, T.Y. Liu, T. Qin, H.H. Chen, and W.Y. Ma. FRank: a ranking method with fidelity loss. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 383–390. ACM, 2007.

[198] Ivor W Tsang, Andras Kocsor, and James T Kwok. Simpler core vector machines with enclosing balls. In *Proceedings of the 24th international conference on Machine learning*, pages 911–918. ACM, 2007.

[199] Ivor W Tsang, James T Kwok, and Pak-Ming Cheung. Core vector machines: Fast svm training on very large data sets. *Journal of Machine Learning Research*, 6(1):363, 2006.

[200] T. Tuytelaars and K. Mikolajczyk. Local invariant feature detectors: a survey. *Foundations and Trends® in Computer Graphics and Vision*, 3(3):177–280, 2008.

[201] Stephen Tyree, Kilian Q. Weinberger, Kunal Agrawal, and Jennifer Paykin. Parallel boosted regression trees for web search ranking. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 387–396, New York, NY, USA, 2011. ACM.

[202] J.K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, 1991.

[203] N.T.V. Uyen and T.C. Chung. A new framework for distributed boosting algorithm. *Future Generation Communication and Networking*, 1:420–423, 2007.

[204] L. van der Maaten and G. Hinton. Visualizing data using t-SNE. *JMLR*, 9(2579-2605):85, 2008.

[205] Brian Van Essen, Chris Macaraeg, Maya Gokhale, and Ryan Prenger. Accelerating a random forest classifier: Multi-core, gp-gpu, or fpga? In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 232–239. IEEE, 2012.

[206] J. Van Gael, Y. Saatci, Y. W. Teh, and Z. Ghahramani. Beam sampling for the infinite hidden Markov model. In *Proceedings of the International Conference on Machine Learning*, volume 25, 2008.

[207] J. Van Gael, Y. W. Teh, and Z. Ghahramani. The infinite factorial hidden Markov model. In *Advances in Neural Information Processing Systems*, volume 21, 2009.

[208] V. Vapnik. Structure of statistical learning theory. *Computational Learning and Probabilistic Reasoning*, page 3, 1996.

[209] M. Varma and A. Zisserman. A statistical approach to material classification using image patch exemplars. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(11):2032–2047, 2009.

[210] Jun Wang, Huyen T Do, Adam Woznica, and Alexandros Kalousis. Metric learning with multiple kernels. In *Advances in Neural Information Processing Systems*, pages 1170–1178, 2011.

[211] X. Wang, C. Zhang, and Z. Zhang. Boosted multi-task learning for face verification with applications to web image and video search. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Patter Recognition*, 2009.

[212] L. Wasserman. *All of statistics: a concise course in statistical inference*. Springer, 2004.

[213] G.I. Webb. Multiboosting: A technique for combining boosting and wagging. *Machine learning*, 40(2):159–196, 2000.

[214] K. Q. Weinberger, A. Dasgupta, J. Attenberg, J. Langford, and A. Smola. Feature hashing for large scale multitask learning. In *ICML*, 2009.

[215] K.Q. Weinberger, J. Blitzer, and L. Saul. Distance metric learning for large margin nearest neighbor classification. In *NIPS '06*, pages 1473–1480. 2006.

[216] K.Q. Weinberger and L.K. Saul. Fast solvers and efficient implementations for distance metric learning. In *Proceedings of the 25th international conference on Machine learning*, pages 1160–1167. ACM, 2008.

[217] K.Q. Weinberger and L.K. Saul. Distance metric learning for large margin nearest neighbor classification. *The Journal of Machine Learning Research*, 10:207–244, 2009.

[218] M. Welling, T. Minka, and Y. W. Teh. Structured region graphs: Morphing EP into GBP. In *Proceedings of the International Conference on Uncertainty in Artificial Intelligence*, volume 21, 2005.

[219] M. Welling, Y. W. Teh, and H. J. Kappen. Hybrid Variational/Gibbs collapsed inference in topic models. In *Proceedings of the International Conference on Uncertainty in Artificial Intelligence*, volume 24, 2008.

[220] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2005.

[221] F. Wood, C. Archambeau, J. Gasthaus, L. F. James, and Y. W. Teh. A stochastic memoizer for sequence data. In *Proceedings of the International Conference on Machine Learning*, 2009.

[222] F. Wood and Y. W. Teh. A hierarchical nonparametric Bayesian approach to statistical language model domain adaptation. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, volume 12, 2009.

[223] Qiang Wu, Christopher J.C. Burges, Krysta M. Svore, and Jianfeng Gao. Adapting boosting for information retrieval measures. *Information Retrieval*, 2009.

[224] E. P. Xing, A. Y. Ng, M. I. Jordan, and S. Russell. Distance metric learning, with application to clustering with side-information. In *NIPS '02*, pages 505–512. MIT Press, 2002.

[225] E. P. Xing, K.-A. Sohn, M. I. Jordan, and Y. W. Teh. Bayesian multi-population haplotype inference via a hierarchical Dirichlet process mixture. In *Proceedings of the International Conference on Machine Learning*, volume 23, 2006.

[226] C. Xiong, D. Johnson, R. Xu, and J.J. Corso. Random forests for metric learning with implicit pairwise position dependence. In *SIGKDD*, pages 958–966. ACM, 2012.

[227] Jun Xu. A boosting algorithm for information retrieval. In *In Proceedings of SIGIR '07*, 2007.

[228] Zhixiang Xu, Matt Kusner, Minmin Chen, and Kilian Q. Weinberger. Cost-sensitive tree of classifiers. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 133–141. JMLR Workshop and Conference Proceedings, 2013.

[229] Yandex. Internet mathematics competition. 2009.

[230] Liu Yang, Rong Jin, L. Mummert, R. Sukthankar, A. Goode, Bin Zheng, S.C.H. Hoi, and M. Satyanarayanan. A boosting framework for visuality-preserving distance metric learning and its application to medical image retrieval. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(1):30 –44, jan. 2010.

[231] Chi Yau. Support vector machine with GPU, 2013.

[232] J. Ye, J.H. Chow, J. Chen, and Z. Zheng. Stochastic gradient boosted distributed decision trees. In *CIKM '09: Proceeding of the 18th ACM Conference on Information and Knowledge Management*, pages 2061–2064. ACM, 2009.

[233] P.N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *ACM-SIAM Symposium on Discrete Algorithms '93*, pages 311–321, 1993.

[234] C. Yu and DB Skillicorn. Parallelizing boosting and bagging. 2001.

[235] Y. Yue, T. Finley, F. Radlinski, and T. Joachims. A support vector method for optimizing average precision. In *Proc. 30th Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 271–278, 2007.

[236] Luca Zanni, Thomas Serafini, and Gaetano Zanghirati. Parallel software for training large scale support vector machines on multiprocessor systems. *The Journal of Machine Learning Research*, 7:1467–1492, 2006.

[237] J. Zhang, M. Marszalek, S. Lazebnik, and C. Schmid. Local features and kernels for classification of texture and object categories: A comprehensive study. In *Computer Vision and Pattern Recognition Workshop, 2006. CVPRW'06. Conference on*, pages 13–13. Ieee, 2006.

[238] Jian-Pei Zhang, Zhong-Wei Li, and Jing Yang. A parallel svm training algorithm on large-scale classification problems. In *Machine Learning and Cybernetics, 2005.*

Proceedings of 2005 International Conference on, volume 3, pages 1637 –1641 Vol. 3, aug. 2005.

[239] Hai-xiang Zhao, Frédéric Magoules, et al. Parallel support vector machines on multi-core and multiprocessor systems. In *Proceedings of the IASTED International Conference Artificial Intelligence and Applications (AIA 2011)*, pages 75–81, 2011.

[240] Zhaohui Zheng, Hongyuan Zha, Keke Chen, and Gordon Sun. A regression framework for learning ranking functions using relative relevance judgments. In *In Proc. of SIGIR*, 2007.

[241] Zhaohui Zheng, Hongyuan Zha, Tong Zhang, Olivier Chapelle, Keke Chen, and Gordon Sun. A general boosting method and its application to learning ranking functions for web search. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 1697–1704, 2008.

[242] Ciyou Zhu, Richard H Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software (TOMS)*, 23(4):550–560, 1997.

[243] Kaihua Zhu, Hao Wang, Hongjie Bai, Jian Li, Zhihuan Qiu, Hang Cui, and Edward Y. Chang. Parallelizing support vector machines on distributed computers. In J.c. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 257–264. MIT Press, Cambridge, MA, 2007.

[244] Zeyuan Allen Zhu, Weizhu Chen, Gang Wang, Chenguang Zhu, and Zheng Chen. P-packSVM: Parallel primal gradient descent kernel SVM. In *ICDM'09*, pages 677–686. IEEE, 2009.