SCALING UP MACHINE LEARNING

Parallel and Distributed Approaches

EDITED BY
RON BEKKERMAN
MIKHAIL BILENKO
JOHN LANGFORD

CAMBRIDGE

Scaling Up Machine Learning

Parallel and Distributed Approaches

This book comprises a collection of representative approaches for scaling up machine learning and data mining methods on parallel and distributed computing platforms. Demand for parallelizing learning algorithms is highly task-specific: in some settings it is driven by the enormous dataset sizes, in others by model complexity or by real-time performance requirements. Making task-appropriate algorithm and platform choices for large-scale machine learning requires understanding the benefits, trade-offs, and constraints of the available options.

Solutions presented in the book cover a range of parallelization platforms from FPGAs and GPUs to multi-core systems and commodity clusters; concurrent programming frameworks that include CUDA, MPI, MapReduce, and DryadLINQ; and various learning settings: supervised, unsupervised, semi-supervised, and online learning. Extensive coverage of parallelization of boosted trees, support vector machines, spectral clustering, belief propagation, and other popular learning algorithms accompanied by deep dives into several applications make the book equally useful for researchers, students, and practitioners.

Dr. Ron Bekkerman is a computer engineer and scientist whose experience spans across disciplines from video processing to business intelligence. Currently a senior research scientist at LinkedIn, he previously worked for a number of major companies including Hewlett-Packard and Motorola. Ron's research interests lie primarily in the area of large-scale unsupervised learning. He is the corresponding author of several publications in top-tier venues, such as ICML, KDD, SIGIR, WWW, IJCAI, CVPR, EMNLP, and JMLR.

Dr. Mikhail Bilenko is a researcher in the Machine Learning Group at Microsoft Research. His research interests center on machine learning and data mining tasks that arise in the context of large behavioral and textual datasets. Mikhail's recent work has focused on learning algorithms that leverage user behavior to improve online advertising. His papers have been published in KDD, ICML, SIGIR, and WWW among other venues, and I have received best paper awards from SIGIR and KDD.

Dr. John Langford is a computer scientist working as a senior researcher at Yahoo! Research. Previously, he was affiliated with the Toyota Technological Institute and IBM T. J. Watson Research Center. John's work has been published in conferences and journals including ICML, COLT, NIPS, UAI, KDD, JMLR, and MLJ. He received the Pat Goldberg Memorial Best Paper Award, as well as best paper awards from ACM EC and WSDM. He is also the author of the popular machine learning weblog, hunch.net.

Scaling Up Machine Learning

Parallel and Distributed Approaches

Edited by
Ron Bekkerman
Mikhail Bilenko
John Langford

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town,
Singapore, São Paulo, Delhi, Tokyo, Mexico City

Cambridge University Press
32 Avenue of the Americas, New York, NY 10013-2473, USA

www.cambridge.org
Information on this title: www.cambridge.org/9780521192248

First published 2012

Printed in the United States of America

A catalog record for this publication is available from the British Library.

Library of Congress Cataloging in Publication data

Scaling up machine learning : parallel and distributed approaches / [edited by] Ron Bekkerman, Mikhail Bilenko, John Langford.
p. cm.
Includes index.
ISBN 978-0-521-19224-8 (hardback)
1. Machine learning. 2. Data mining. 3. Parallel algorithms. 4. Parallel programs (Computer programs) I. Bekkerman, Ron. II. Bilenko, Mikhail. III. Langford, John.
Q325.5.S28 2011
006.3′1–dc23 2011016323

ISBN 978-0-521-19224-8 Hardback

Contents

Contributors

Polina Akselrod
Yale University, New Haven, CT, USA

Arthur Asuncion
University of California, Irvine, CA, USA

Hongjie Bai
Google Research, Beijing, China

Sugato Basu
Google Research, Mountain View, CA, USA

Roberto J. Bayardo
Google Research, Mountain View, CA, USA

Ron Bekkerman
LinkedIn Corporation, Mountain View, CA, USA

Mikhail Bilenko
Microsoft Research, Redmond, WA, USA

Jeff Bilmes
University of Washington, Seattle, WA, USA

Mihai Budiu
Microsoft Research, Mountain View, CA, USA

Christopher J. C. Burges
Microsoft Research, Redmond, WA, USA

Srihari Cadambi
NEC Labs America, Princeton, NJ, USA

Srimat Chakradhar
NEC Labs America, Princeton, NJ, USA

Edward Y. Chang
Google Research, Beijing, China

Wen-Yen Chen
University of California, Santa Barbara, CA, USA

Jike Chong
Parasians LLC, Sunnyvale, CA, USA

Adam Coates
Stanford University. Stanford, CA, USA

Eric Cosatto
NEC Labs America, Princeton, NJ, USA

Eugenio Culurciello
Yale University, New Haven, CT, USA

Igor Durdanovic
NEC Labs America, Princeton, NJ, USA

Clément Farabet
New York University, New York, NY, USA

Dennis Fetterly
Microsoft Research, Mountain View, CA, USA

Amol Ghoting
IBM Research, Yorktown Heights, NY, USA

Ekaterina Gonina
University of California, Berkeley, CA, USA

Joseph Gonzalez
Carnegie Mellon University, Pittsburgh, PA, USA

Hans Peter Graf
NEC Labs America, Princeton, NJ, USA

Carlos Guestrin
Carnegie Mellon University, Pittsburgh, PA, USA

Joshua S. Herbach
Google Inc., Mountain View, CA, USA

Daniel Hsu
Rutgers University, Piscataway, NJ, USA and University of Pennsylvania, Philadelphia, PA, USA

Meichun Hsu
HP Labs, Palo Alto, CA, USA

Michael Isard
Microsoft Research, Mountain View, CA, USA

Venkata Jakkula
NEC Labs America, Princeton, NJ, USA

Nikos Karampatziakis
Cornell University, Ithaca, NY, USA

Koray Kavukcuoglu
NEC Labs America, Princeton, NJ, USA

Kurt Keutzer
University of California, Berkeley, CA, USA

Jeremy Kubica
Google Inc., Pittsburgh, PA, USA

John Langford
Yahoo! Research, New York, NY, USA

Yann LeCun
New York University, New York, NY, USA

Jian Li
Google Research, Beijing, China

Chih-Jen Lin
National Taiwan University, Taipei, Taiwan

Nathan Liu
Hong Kong University of Science and Technology, Kowloon, Hong Kong

Yucheng Low
Carnegie Mellon University, Pittsburgh, PA, USA

Abhinandan Majumdar
NEC Labs America, Princeton, NJ, USA

Berin Martini
Yale University, New Haven, CT, USA

Frank McSherry
Microsoft Research, Mountain View, CA, USA

Ramesh Natarajan
IBM Research, Yorktown Heights, NY, USA

David Newman
University of California, Irvine, CA, USA

Andrew Y. Ng
Stanford University. Stanford, CA, USA

Biswanath Panda
Google Inc., Mountain View, CA, USA

Srinivasan Parthasarathy
Ohio State University, Columbus, OH, USA

Edwin Pednault
IBM Research, Yorktown Heights, NY, USA

Ian Porteous
Google Inc., Kirkland, WA, USA

Zhihuan Qiu
Google Research, Beijing, China

Rajat Raina
Facebook Inc., Palo Alto, CA, USA

Martin Scholz
HP Labs, Palo Alto, CA, USA

Sameer Singh
University of Massachusetts, Amherst, MA, USA

Alex J. Smola
Yahoo! Research, Santa Clara, NY, USA

Padhraic Smyth
University of California, Irvine, CA, USA

Yangqiu Song
Tsinghua University, Beijing, China

Daria Sorokina
Yandex Labs, Palo Alto, CA, USA

Amarnag Subramanya
Google Research, Mountain View, CA, USA

Krysta M. Svore
Microsoft Research, Redmond, WA, USA

Selcuk Talay
Yale University, New Haven, CT, USA

Shirish Tatikonda
IBM Research, San Jose, CA, USA

Scott Triglia
University of California, Irvine, CA, USA

Hao Wang
Google Research, Beijing, China

Max Welling
University of California, Irvine, CA, USA

Ren Wu
HP Labs, Palo Alto, CA, USA

Evan Xiang
Hong Kong University of Science and Technology, Kowloon, Hong Kong

Qiang Yang
Hong Kong University of Science and Technology, Kowloon, Hong Kong

Elad Yom-Tov
Yahoo! Research, New York, NY, USA

Kisun You
Seoul National University, Seoul, Korea

Yuan Yu
Microsoft Research, Mountain View, CA, USA

Bin Zhang
HP Labs, Palo Alto, CA, USA

Kaihua Zhu
Google Research, Beijing, China

Preface

This book attempts to aggregate state-of-the-art research in parallel and distributed machine learning. We believe that parallelization provides a key pathway for scaling up machine learning to large datasets and complex methods. Although large-scale machine learning has been increasingly popular in both industrial and academic research communities, there has been no singular resource covering the variety of approaches recently proposed. We did our best to assemble the most representative contemporary studies in one volume. While each contributed chapter concentrates on a distinct approach and problem, together with their references they provide a comprehensive view of the field.

We believe that the book will be useful to the broad audience of researchers, practitioners, and anyone who wants to grasp the future of machine learning. To smooth the ramp-up for beginners, the first five chapters provide introductory material on machine learning algorithms and parallel computing platforms. Although the book gets deeply technical in some parts, the reader is assumed to have only basic prior knowledge of machine learning and parallel/distributed computing, along with college-level mathematical maturity. We hope that an engineering undergraduate who is familiar with the notion of a classifier and had some exposure to threads, MPI, or MapReduce will be able to understand the majority of the book's content. We also hope that a seasoned expert will find this book full of new, interesting ideas to inspire future research in the area.

We are deeply thankful to all chapter authors for significant investments of their time, talent, and creativity in preparing their contributions to this volume. We appreciate the efforts of our editors at Cambridge University Press: Heather Bergman, who initiated this project, and Lauren Cowles, who worked with us throughout the process, guiding the book to completion. We thank chapter reviewers who provided detailed, thoughtful feedback to chapter authors that was invaluable in shaping the book: David Andrzejewski, Yoav Artzi, Arthur Asuncion, Hongjie Bai, Sugato Basu, Andrew Bender, Mark Chapman, Wen-Yen Chen, Sulabh Choudhury, Adam Coates, Kamalika Das, Kevin Duh, Igor Durdanovic, Clément Farabet, Dennis Fetterly, Eric Garcia, Joseph Gonzalez, Isaac Greenbaum, Caden Howell, Ferris Jumah, Andrey Kolobov, Jeremy

Kubica, Bo Li, Luke McDowell, W. P. McNeill, Frank McSherry, Chris Meek, Xu Miao, Steena Monteiro, Miguel Osorio, Sindhu Vijaya Raghavan, Paul Rodrigues, Martin Scholz, Suhail Shergill, Sameer Singh, Tom Sommerville, Amarnag Subramanya, Narayanan Sundaram, Krysta Svore, Shirish Tatikonda, Amund Tveit, Jean Wu, Evan Xiang, Elad Yom-Tov, and Bin Zhang.

Ron Bekkerman would like to thank Martin Scholz for his personal involvement in this project since its initial stage. Ron is deeply grateful to his mother Faina, wife Anna, and daughter Naomi, for their endless love and support throughout all his ventures.

CHAPTER 1

Scaling Up Machine Learning: Introduction

Ron Bekkerman, Mikhail Bilenko, and John Langford

Distributed and parallel processing of very large datasets has been employed for decades in specialized, high-budget settings, such as financial and petroleum industry applications. Recent years have brought dramatic progress in usability, cost effectiveness, and diversity of parallel computing platforms, with their popularity growing for a broad set of data analysis and machine learning tasks.

The current rise in interest in scaling up machine learning applications can be partially attributed to the evolution of hardware architectures and programming frameworks that make it easy to exploit the types of parallelism realizable in many learning algorithms. A number of platforms make it convenient to implement concurrent processing of data instances or their features. This allows fairly straightforward parallelization of many learning algorithms that view input as an unordered batch of examples and aggregate isolated computations over each of them.

Increased attention to large-scale machine learning is also due to the spread of very large datasets across many modern applications. Such datasets are often accumulated on distributed storage platforms, motivating the development of learning algorithms that can be distributed appropriately. Finally, the proliferation of sensing devices that perform real-time inference based on high-dimensional, complex feature representations drives additional demand for utilizing parallelism in learning-centric applications. Examples of this trend include speech recognition and visual object detection becoming commonplace in autonomous robots and mobile devices.

The abundance of distributed platform choices provides a number of options for implementing machine learning algorithms to obtain efficiency gains or the capability to process very large datasets. These options include customizable integrated circuits (e.g., Field-Programmable Gate Arrays – FPGAs), custom processing units (e.g., general-purpose Graphics Processing Units – GPUs), multiprocessor and multicore parallelism, High-Performance Computing (HPC) clusters connected by fast local networks, and datacenter-scale virtual clusters that can be rented from commercial cloud computing providers. Aside from the multiple platform options, there exists a variety of programming frameworks in which algorithms can be implemented. Framework choices tend

to be particularly diverse for distributed architectures, such as clusters of commodity PCs.

The wide range of platforms and frameworks for parallel and distributed computing presents both opportunities and challenges for machine learning scientists and engineers. Fully exploiting the available hardware resources requires adapting some algorithms and redesigning others to enable their concurrent execution. For any prediction model and learning algorithm, their structure, dataflow, and underlying task decomposition must be taken into account to determine the suitability of a particular infrastructure choice.

Chapters making up this volume form a representative set of state-of-the-art solutions that span the space of modern parallel computing platforms and frameworks for a variety of machine learning algorithms, tasks, and applications. Although it is infeasible to cover every existing approach for every platform, we believe that the presented set of techniques covers most commonly used methods, including the popular "top performers" (e.g., boosted decision trees and support vector machines) and common "baselines" (e.g., k-means clustering).

Because most chapters focus on a single choice of platform and/or framework, the rest of this introduction provides the reader with unifying context: a brief overview of machine learning basics and fundamental concepts in parallel and distributed computing, a summary of typical task and application scenarios that require scaling up learning, and thoughts on evaluating algorithm performance and platform trade-offs. Following these are an overview of the chapters and bibliography notes.

1.1 Machine Learning Basics

Machine learning focuses on constructing algorithms for making predictions from data. A machine learning task aims to identify (to *learn*) a function $f\colon \mathcal{X} \to \mathcal{Y}$ that maps input domain $\mathcal{X}$ (of data) onto output domain $\mathcal{Y}$ (of possible predictions). The function f is selected from a certain function class, which is different for each family of learning algorithms. Elements of $\mathcal{X}$ and $\mathcal{Y}$ are application-specific representations of data objects and predictions, respectively.

Two canonical machine learning settings are *supervised learning* and *unsupervised learning*. Supervised learning algorithms utilize *training data* to construct a prediction function f, which is subsequently applied to *test* instances. Typically, training data is provided in the form of *labeled examples* $(x, y) \in \mathcal{X} \times \mathcal{Y}$, where x is a data instance and y is the corresponding *ground truth* prediction for x.

The ultimate goal of supervised learning is to identify a function f that produces accurate predictions on test data. More formally, the goal is to minimize the prediction error (*loss*) function $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, which quantifies the difference between any $f(x)$ and y – the predicted output of x and its ground truth label. However, the loss cannot be minimized directly on test instances and their labels because they are typically unavailable at training time. Instead, supervised learning algorithms aim to construct predictive functions that *generalize* well to previously unseen data, as opposed to performing optimally just on the given training set, that is, *overfitting* the training data.

The most common supervised learning setting is *induction*, where it is assumed that each training and test example (x, y) is sampled from some unknown joint probability

distribution P over $\mathcal{X} \times \mathcal{Y}$. The objective is to find f that minimizes *expected loss* $\mathbb{E}_{(x,y)\sim P}\, l(f(x), y)$. Because the joint distribution P is unknown, expected loss cannot be minimized in closed form; hence, learning algorithms approximate it based on training examples. Additional supervised learning settings include *semi-supervised learning* (where the input data consists of both labeled and unlabeled instances), *transfer learning*, and *online learning* (see Section 1.6.3).

Two classic supervised learning tasks are *classification* and *regression*. In classification, the output domain is a finite discrete set of categories (*classes*), $\mathcal{Y} = \{c_1, ..., c_k\}$, whereas in regression the output domain is the set of real numbers, $\mathcal{Y} = \mathbb{R}$. More complex output domains are explored within advanced learning frameworks, such as *structured learning* (Bakir et al., 2007).

The simplest classification scenario is *binary*, in which there are two classes. Let us consider a small example. Assume that the task is to learn a function that predicts whether an incoming email message is spam or not. A common way to represent textual messages is as large, sparse vectors, in which every entry corresponds to a vocabulary word, and non-zero entries represent words that are present in the message. The label can be represented as 1 for spam and -1 for nonspam. With this representation, it is common to learn a vector of weights w optimizing $f(x) = \text{sign}\left(\sum_i w_i x_i\right)$ so as to predict the label.

The most prominent example of unsupervised learning is *data clustering*. In clustering, the goal is to construct a function f that partitions an *unlabeled* dataset into $k = |\mathcal{Y}|$ clusters, with $\mathcal{Y}$ being the set of cluster indices. Data instances assigned to the same cluster should presumably be more similar to each other than to data instances assigned to any other cluster. There are many ways to define similarity between data instances; for example, for vector data, (inverted) Euclidean distance and cosine similarity are commonly used. Clustering quality is often measured against a dataset with existing class labels that are withheld during clustering: a quality measure penalizes f if it assigns instances of the same class to different clusters and instances of different classes to the same cluster.

We note that both supervised and unsupervised learning settings distinguish between *learning* and *inference* tasks, where learning refers to the process of identifying the prediction function f, while inference refers to computing $f(x)$ on a data instance x. For many learning algorithms, inference is a component of the learning process, as predictions of some interim candidate f' on the training data are used in the search for the optimal f. Depending on the application domain, scaling up may be required for either the learning or the inference algorithm, and chapters in this book present numerous examples of speeding up both.

1.2 Reasons for Scaling Up Machine Learning

There are a number of settings where a practitioner could find the scale of a machine learning task daunting for single-machine processing and consider employing parallelization. Such settings are characterized by:

1. **Large number of data instances:** In many domains, the number of potential training examples is extremely large, making single-machine processing infeasible.

2. **High input dimensionality:** In some applications, data instances are represented by a very large number of features. Machine learning algorithms may partition computation across the set of features, which allows scaling up to lengthy data representations.
3. **Model and algorithm complexity:** A number of high-accuracy learning algorithms either rely on complex, nonlinear models, or employ computationally expensive subroutines. In both cases, distributing the computation across multiple processing units can be the key enabler for learning on large datasets.
4. **Inference time constraints:** Applications that involve sensing, such as robot navigation or speech recognition, require predictions to be made in real time. Tight constraints on inference speed in such settings invite parallelization of inference algorithms.
5. **Prediction cascades:** Applications that require sequential, interdependent predictions have highly complex joint output spaces, and parallelization can significantly speed up inference in such settings.
6. **Model selection and parameter sweeps:** Tuning hyper-parameters of learning algorithms and statistical significance evaluation require multiple executions of learning and inference. Fortunately, these procedures belong to the category of so-called *embarrassingly parallelizable* applications, naturally suited for concurrent execution.

The following sections discuss each of these scenarios in more detail.

1.2.1 Large Number of Data Instances

Datasets that aggregate billions of events per day have become common in a number of domains, such as internet and finance, with each event being a potential input to a learning algorithm. Also, more and more devices include sensors continuously logging observations that can serve as training data. Each data instance may have, for example, thousands of non-zero features on average, resulting in datasets of 10^{12} instance–feature pairs per day. Even if each feature takes only 1 byte to store, datasets collected over time can easily reach hundreds of terabytes.

The preferred way to effectively process such datasets is to combine the distributed storage and bandwidth of a cluster of machines. Several computational frameworks have recently emerged to ease the use of large quantities of data, such as MapReduce and DryadLINQ, used in several chapters in this book. Such frameworks combine the ability to use high-capacity storage and execution platforms with programming via simple, naturally parallelizable language primitives.

1.2.2 High Input Dimensionality

Machine learning and data mining tasks involving natural language, images, or video can easily have input dimensionality of 10^6 or higher, far exceeding the comfortable scale of $10 - 1{,}000$ features considered common until recently. Although data in some of these domains is sparse, that is not always the case; sparsity is also lost in the parameter space of many algorithms. Parallelizing the computation across features can thus be an attractive pathway for scaling up computation to richer representations, or just for speeding up algorithms that naturally iterate over features, such as decision trees.

1.2.3 Model and Algorithm Complexity

Data in some domains has inherently nonlinear structure with respect to the basic features (e.g., pixels or words). Models that employ highly nonlinear representations, such as decision tree ensembles or multi-layer (deep) networks, can significantly outperform simpler algorithms in such applications. Although feature engineering can yield high accuracies with computationally cheap linear models in these domains, there is a growing interest in learning as automatically as possible from the base representation. A common characteristic of algorithms that attempt this is their substantial computational complexity. Although the training data may easily fit on one machine, the learning process may simply be too slow for a reasonable development cycle. This is also the case for some learning algorithms, the computational complexity of which is superlinear in the number of training examples.

For problems of this nature, parallel multinode or multicore implementations appear viable and have been employed successfully, allowing the use of complex algorithms and models for larger datasets. In addition, coprocessors such as GPUs have also been employed successfully for fast transformation of the original input space.

1.2.4 Inference Time Constraints

The primary means for reducing the testing time is via embarrassingly parallel replication. This approach works well for settings where *throughput* is the primary concern – the number of evaluations to be done is very large. Consider, for example, evaluating 10^{10} emails per day in a spam filter, which is not expected to output results in real time, yet must not become backlogged.

Inference latency is generally a more stringent concern compared to throughput. Latency issues arise in any situation where systems are waiting for a prediction, and the overall application performance degrades rapidly with latency. For instance, this occurs for a car-driving robot making path planning decisions based on several sensors, or an online news provider that aims to improve user experience by selecting suggested stories using on-the-fly personalization.

Constraints on throughput and latency are not entirely compatible – for example, data pipelining trades throughput for latency. However, for both of them, utilizing highly parallelized hardware architectures such as GPUs or FPGAs has been found effective.

1.2.5 Prediction Cascades

Many real-world problems such as object tracking, speech recognition, and machine translation require performing a sequence of interdependent predictions, forming *prediction cascades*. If a cascade is viewed as a single inference task, it has a large joint output space, typically resulting in very high computational costs due to increased computational complexity. Interdependencies between the prediction tasks are typically tackled by stagewise parallelization of individual tasks, along with adaptive task management, as illustrated by the approach of Chapter 21 to speech recognition.

1.2.6 Model Selection and Parameter Sweeps

The practice of developing, tuning, and evaluating learning algorithms relies on workflow that is embarrassingly parallel: it requires no intercommunication between the tasks with independent executions on the same dataset. Two particular processes of this nature are parameter sweeps and statistical significance testing. In parameter sweeps, the learning algorithm is run multiple times on the same dataset with different settings, followed by evaluation on a validation set. During statistical significance testing procedures such as cross-validation or bootstrapping, training and testing is performed repeatedly on different dataset subsets, with results aggregated for subsequent measurement of statistical significance. Usefulness of parallel platforms is obvious for these tasks, as they can be easily performed concurrently without the need to parallelize actual learning and inference algorithms.

1.3 Key Concepts in Parallel and Distributed Computing

Performance gains attainable in machine learning applications by employing parallel and distributed systems are driven by concurrent execution of tasks that are otherwise performed serially. There are two major directions in which this concurrency is realized: *data parallelism* and *task parallelism*. Data parallelism refers to simultaneous processing of multiple inputs, whereas task parallelism is achieved when algorithm execution can be partitioned into segments, some of which are independent and hence can be executed concurrently.

1.3.1 Data Parallelism

Data parallelism refers to executing the same computation on multiple inputs concurrently. It is a natural fit for many machine learning applications and algorithms that accept input data as a batch of independent samples from an underlying distribution. Representation of these samples via an instance-by-feature matrix naturally suggests two orthogonal directions for achieving data parallelism. One is partitioning the matrix rowwise into subsets of instances that are then processed independently (e.g., when computing the update to the weights for logistic regression). The other is splitting it columnwise for algorithms that can decouple the computation across features (e.g., for identifying the split feature in decision tree construction).

The most basic example of data parallelism is encountered in embarrassingly parallel algorithms, where the computation is split into concurrent subtasks requiring no intercommunication, which run independently on separate data subsets. A related simple implementation of data parallelism occurs within the *master–slave communication model*: a master process distributes the data across slave processes that execute the same computation (see, e.g., Chapters 8 and 16).

Less obvious cases of data parallelism arise in algorithms where instances or features are not independent, but there exists a well-defined relational structure between them that can be represented as a graph. Data parallelism can then be achieved if the computation can be partitioned across instances based on this structure. Then, concurrent execution on different partitions is interlaced with exchange of information across them; approaches presented in Chapters 10 and 15 rely on this algorithmic pattern.

The foregoing examples illustrate coarse-grained data parallelism over subsets of instances or features that can be achieved via algorithm design. Fine-grained data parallelism, in contrast, refers to exploiting the capability of modern processor architectures that allow parallelizing vector and matrix computations in hardware. Standard libraries such as BLAS and LAPACK[1] provide routines that abstract out the execution of basic vector and matrix operations. Learning algorithms that can be represented as cascades of such operations can then leverage hardware-supported parallelism by making the corresponding API calls, dramatically simplifying the algorithms' implementation.

1.3.2 Task Parallelism

Unlike data parallelism defined by performing the same computation on multiple inputs simultaneously, task parallelism refers to segmenting the overall algorithm into parts, some of which can be executed concurrently. Fine-grained task parallelism for numerical computations can be performed automatically by many modern architectures (e.g., via pipelining) but can also be implemented semimanually on certain platforms, such as GPUs, potentially resulting in very significant efficiency gains, but requiring in-depth platform expertise. Coarse-grained task parallelism requires explicit encapsulation of each task in the algorithm's implementation as well as a scheduling service, which is typically provided by a programming framework.

The partitioning of an algorithm into tasks can be represented by a directed acyclic graph, with nodes corresponding to individual tasks, and edges representing inter-task dependencies. Dataflow between tasks occurs naturally along the graph edges. A prominent example of such a platform is MapReduce, a programming model for distributed computation introduced by Dean and Ghemawat (2004), on which several chapters in this book rely; see Chapter 2 for more details. Additional cross-task communication can be supported by platforms via point-to-point and broadcast messaging. The Message Passing Interface (MPI) introduced by Gropp et al. (1994) is an example of such messaging protocol that is widely supported across many platforms and programming languages. Several chapters in this book rely on it; see Section 4.4 of Chapter 4 for more details. Besides wide availability, MPI's popularity is due to its flexibility: it supports both point-to-point and collective communication, with synchronous and asynchronous mechanisms.

For many algorithms, scaling up can be most efficiently achieved by a mixture of data and task parallelism. Capability for hybrid parallelism is realized by most modern platforms: for example, it is exhibited both by the highly distributed DryadLINQ framework described in Chapter 3 and by computer vision algorithms implemented on GPUs and customized hardware as described in Chapters 18 and 19.

1.4 Platform Choices and Trade-Offs

Let us briefly summarize the key dimensions along which parallel and distributed platforms can be characterized. The classic taxonomy of parallel architectures proposed

[1] http://www.netlib.org/blas/ and http://www.netlib.org/lapack/.

by Flynn (1972) differentiates them by concurrency of algorithm execution (single vs. multiple instruction) and input processing (single vs. multiple data streams). Further distinctions can be made based on the configuration of shared memory and the organization of processing units. Modern parallel architectures are typically based on hybrid topologies where processing units are organized hierarchically, with multiple layers of shared memory. For example, GPUs typically have dozens of multiprocessors, each of which has multiple stream processors organized in "blocks". Individual blocks have access to relatively small locally shared memory and a much larger globally shared memory (with higher latency).

Unlike parallel architectures, distributed computing platforms typically have larger (physical) distances between processing units, resulting in higher latencies and lower bandwidth. Furthermore, individual processing units may be heterogeneous, and direct communication between them may be limited or nonexistent either via shared memory or via message passing, with the extreme case being one where all dataflow is limited to task boundaries, as is the case for MapReduce.

The overall variety of parallel and distributed platforms and frameworks that are now available for machine learning applications may seem overwhelming. However, the following observations capture the key differentiating aspects between the platforms:

- **Parallelism granularity:** Employing hardware-specific solutions – GPUs and FPGAs – allows very fine-grained data and task parallelism, where elementary numerical tasks (operations on vectors, matrices, and tensors) can be spread across multiple processing units with very high throughput achieved by pipelining. However, using this capability requires redefining the entire algorithm as a dataflow of such elementary tasks and eliminating bottlenecks. Moving up to parallelism across cores and processors in generic CPUs, the constraints on defining the algorithm as a sequence of finely tuned stages are relaxed, and parallelism is no longer limited to elementary numeric operations. With cluster- and datacenter-scale solutions, defining higher-granularity tasks becomes imperative because of increasing communication costs.
- **Degree of algorithm customization:** Depending on platform choice, the complexity of algorithm redesign required for enabling concurrency may vary from simply using a third-party solution for automatic parallelization of an existing imperative or declarative-style implementation, to having to completely re-create the algorithm, or even implement it directly in hardware. Generally, implementing learning algorithms on hardware-specific platforms (e.g., GPUs) requires significant expertise, hardware-aware task configuration, and avoiding certain commonplace software patterns such as branching. In contrast, higher-level parallel and distributed systems allow using multiple, commonplace programming languages extended by APIs that enable parallelism.
- **Ability to mix programming paradigms:** Declarative programming languages are becoming increasingly popular for large-scale data manipulation, borrowing from a variety of predecessors – from functional programming to SQL – to make parallel programming easier by expressing algorithms primarily as a mixture of logic and dataflow. Such languages are often hybridized with the classic imperative programming to provide maximum expressiveness. Examples of this trend include Microsoft's DryadLINQ,

Google's Sawzall and Pregel, and Apache Pig and Hive. Even in applications where such declarative-style languages are insufficient for expressing the learning algorithms, they are often used for computing the basic first- and second-order statistics that produce highly predictive features for many learning tasks.

- **Dataset scale-out:** Applications that process datasets too large to fit in memory commonly rely on distributed filesystems or shared-memory clusters. Parallel computing frameworks that are tightly coupled with distributed dataset storage allow optimizing task allocation during scheduling to maximize local dataflows. In contrast, scheduling in hardware-specific parallelism is decoupled from storage solutions used for very large datasets and hence requires crafting manual solutions to maximize throughput.
- **Offline vs online execution:** Distributed platforms typically assume that their user has higher tolerance for failures and latency compared to hardware-specific solutions. For example, an algorithm implemented via MapReduce and submitted to a virtual cluster typically has no guarantees on completion time. In contrast, GPU-based algorithms can assume dedicated use of the platform, which may be preferable for real-time applications.

Finally, we should note that there is a growing trend for hybridization of the multiple parallelization levels: for example, it is now possible to rent clusters comprising multicore nodes with attached GPUs from commercial cloud computing providers. Given a particular application at hand, the choice of the platform and programming framework should be guided by the criteria just given to identify an appropriate solution.

1.5 Thinking about Performance

The term "performance" is deeply ambiguous for parallel learning algorithms, as it includes both predictive accuracy and computational speed, each of which can be measured by a number of metrics. The variety of learning problems addressed in the chapters of this book makes the presented approaches generally incomparable in terms of predictive performance: the algorithms are designed to optimize different objectives in different settings. Even in those cases where the same problem is addressed, such as binary classification or clustering, differences in application domains and evaluation methodology typically lead to incomparability in accuracy results. As a consequence of this, it is not possible to provide a meaningful quantitative summary of relative accuracy across the chapters in the book, although it should be understood in every case that the authors strove to create effective algorithms.

Classical analysis of algorithms' complexity is based on O-notation (or its brethren) to bound and quantify computational costs. This approach meets difficulties with many machine learning algorithms, as they often include optimization-based termination conditions for which no formal analysis exists. For example, a typical early stopping algorithm may terminate when predictive error measured on a holdout test set begins to rise – something that is difficult to analyze because the core algorithm does not have access to this test set by design.

Nevertheless, individual subroutines within learning algorithms do often have clear computational complexities. When examining algorithms and considering their application to a given domain, we suggest asking the following questions:

1. What is the computational complexity of the algorithm or of its subroutine? Is it linear (i.e., $O(\text{input size})$)? Or superlinear? In general, there is a qualitative difference between algorithms scaling as $O(\text{input size})$ and others scaling as $O(\text{input size}^{\alpha})$ for $\alpha \geq 2$. For all practical purposes, algorithms with cubic and higher complexities are not applicable to real-world tasks of the modern scale.
2. What is the bandwidth requirement for the algorithm? This is particularly important for any algorithm distributed over a cluster of computers, but is also relevant for parallel algorithms that use shared memory or disk resources. This question comes in two flavors: What is the aggregate bandwidth used? And what is the maximum bandwidth of any node? Answers of the form $O(\text{input size})$, $O(\text{instances})$, and $O(\text{parameters})$ can all arise naturally depending on how the data is organized and the algorithm proceeds. These answers can have a very substantial impact on running time, as the input dataset may be, say, 10^{14} bytes in size, yet have only 10^{10} examples and 10^{8} parameters.

Key metrics used for analyzing computational performance of parallel algorithms are speedup, efficiency, and scalability:

- *Speedup* is the ratio of solution time for the sequential algorithms versus its parallel counterpart.
- *Efficiency* measures the ratio of speedup to the number of processors.
- *Scalability* tracks efficiency as a function of an increasing number of processors.

For reasons explained earlier, these measures can be nontrivial to evaluate analytically for machine learning algorithms, and generally should be considered in conjunction with accuracy comparisons. However, these measures are highly informative in empirical studies. From a practical standpoint, given the differences in hardware employed for parallel and sequential implementations, viewing these metrics as functions of costs (hardware and implementation) is important for fair comparisons.

Empirical evaluation of computational costs for different algorithms should be ideally performed by comparing them on the same datasets. As with predictive performance, this may not be done for the work presented in subsequent chapters, given the dramatic differences in tasks, application domains, underlying frameworks, and implementations for the different methods. However, it is possible to consider the general *feature throughput* of the methods presented in different chapters, defined as $\frac{\text{running time}}{\text{input size}}$. Based on the results reported across chapters, well-designed parallelized methods are capable of obtaining high efficiency across the different platforms and tasks.

1.6 Organization of the Book

Chapters in this book span a range of computing platforms, learning algorithms, prediction problems, and application domains, describing a variety of parallelization techniques to scale up machine learning. The book is organized in four parts. The

Table 1.1. *Chapter summary.*

Chapter	Platform	Parallelization Framework	Learning Setting	Algorithms/ Applications
2	Cluster	MapReduce	Clustering, classification	*k*-Means, decision tree ensembles
3	Cluster	DryadLINQ	Multiple	*k*-Means, decision trees, SVD
4	Cluster	MPI	Multiple	Kernel *k*-means, decision trees, frequent pattern mining
5	GPU	CUDA	Clustering, regression	*k*-Means, regression *k*-means
6	Cluster	MPI	Classification	SVM (IPM)
7	Cluster, multicore, FPGA	TCP, UDP, threads, HDL	Classification, regression	SVM (SMO)
8	Cluster	MPI	Ranking	LambdaMART, web search
9	Cluster	MPI	Regression, classification	Transform regression
10	Cluster	MPI	Inference	Loopy belief propagation
11	Cluster	MPI	Inference	MCMC
12	Cluster	MapReduce, MPI	Clustering	Spectral clustering
13	Cluster	MPI	Clustering	Information-theoretic clustering
14	Cluster	TCP, threads	Classification, regression	Online learning
15	Cluster, multicore	TCP, threads	Semi-supervised learning (SSL)	Graph-based SSL
16	Cluster	MPI	Transfer learning	Collaborative filtering
17	Cluster	MapReduce	Classification	Feature selection
18	GPU	CUDA	Classification	Object detection, feature extraction
19	FPGA	HDL	Classification	Object detection, feature extraction
20	Multicore	Threads, task queue	Pattern mining	Frequent subtree mining
21	Multicore, GPU	CUDA, task queue	Inference	Speech recognition

first part focuses on four distinct programming frameworks, on top of which a variety of learning algorithms have been successfully implemented. The second part focuses on individual learning algorithms, describing parallelized versions of several high-performing supervised and unsupervised methods. The third part is dedicated to task settings that differ from the classic supervised versus unsupervised dichotomy, such as online learning, semi-supervised learning, transfer learning, and feature selection. The final, fourth part describes several application settings where scaling up learning has been highly successful: computer vision, speech recognition, and frequent pattern mining. Table 1.1 contains a summary view of the chapters, prediction tasks considered, and specific algorithms and applications for each chapter.

1.6.1 Part I: Frameworks for Scaling Up Machine Learning

The first four chapters of the book describe programming frameworks that are well suited for parallelizing learning algorithms, as illustrated by in-depth examples of specific algorithms provided in each chapter. In particular, the implementation of *k*-means clustering in each chapter is a shared example that is illustrative of the similarities, differences, and capabilities of the frameworks.

Chapter 2, the first contributed chapter in the book, provides a brief introduction to MapReduce, an increasingly popular distributed computing framework, and discusses the pros and cons of scaling up learning algorithms using it. The chapter focuses on employing MapReduce to parallelize the training of decision tree ensembles, a class of algorithms that includes such popular methods as boosting and bagging. The presented approach, PLANET, distributes the tree construction process by concurrently expanding multiple nodes in each tree, leveraging the data partitioning naturally induced by the tree, and modulating between parallel and local execution when appropriate. PLANET achieves a two-orders-of-magnitude speedup on a 200 node MapReduce cluster on datasets that are not feasible to process on a single machine.

Chapter 3 introduces DryadLINQ, a declarative data-parallel programming language that compiles programs down to reliable distributed computations, executed by the Dryad cluster runtime. DryadLINQ presents the programmer with a high-level abstraction of the data, as a typed collection in .NET, and enables numerous useful software engineering tools such as type-safety, integration with the development environment, and interoperability with standard libraries, all of which help programmers to write their program correctly before they execute it. At the same time, the language is well matched to the underlying Dryad execution engine, capable of reliably and scalably executing the computation across large clusters of machines. Several examples demonstrate that relatively simple programs in DryadLINQ can result in very efficient distributed computations; for example, a version of *k*-means is implemented in only a dozen lines. Several other machine learning examples call attention to the ease of programming and demonstrate strong performance across multi-gigabyte datasets.

Chapter 4 describes the IBM Parallel Machine Learning Toolbox (PML) that provides a general MPI-based parallelization foundation well suited for machine learning algorithms. Given an algorithm at hand, PML represents it as a sequence of operators that obey the algebraic rules of commutativity and associativity. Intuitively, such operators correspond to algorithm steps during which training instances are exchangeable and can be partitioned in any way, making their processing easy to parallelize. Functionality provided by PML is particularly beneficial for algorithms that require multiple passes over data – as most machine learning algorithms do. The chapter describes how a number of popular learning algorithms can be represented as associative-commutative cascades and gets into the details of their implementations in PML. Chapter 9 from the second part of the book discusses *transform regression* as implemented in PML.

Chapter 5 provides a gentle introduction to Compute Unified Device Architecture (CUDA) programming on GPUs and illustrates its use in machine learning applications by describing implementations of *k*-means and regression *k*-means. The chapter offers important insights into redesigning learning algorithms to fit the CPU/GPU

computation model, with a detailed discussion of *uniformly fine-grained* data/task parallelism in GPUs: parallel execution over vectors and matrices, with inputs pipelined to further increase efficiency. Experiments demonstrate two-orders-of-magnitude speedups over highly optimized, multi-threaded implementations of k-means and regression k-means on CPUs.

1.6.2 Part II: Supervised and Unsupervised Learning Algorithms

The second part of the book is dedicated to parallelization of popular supervised and unsupervised machine learning algorithms that cover key approaches in modern machine learning. The first two chapters describe different approaches to parallelizing the training of Support Vector Machines (SVMs): one showing how the Interior Point Method (IPM) can be effectively distributed using message passing, and another focusing on customized hardware design for the Sequential Minimal Optimization (SMO) algorithm that results in a dramatic speedup. Variants of boosted decision trees are covered by the next two chapters: first, an MPI-based parallelization of boosting for ranking, and second, transform regression that provides several enhancements to traditional boosting that significantly reduce the number of iterations. The subsequent two chapters are dedicated to graphical models: one describing parallelizing Belief Propagation (BP) in factor graphs, a workhorse of numerous graphical model algorithms, and another on distributed Markov Chain Monte Carlo (MCMC) inference in unsupervised topic models, an area of significant interest in recent years. This part of the book concludes with two chapters on clustering, describing fast implementations of two very different approaches: spectral clustering and information-theoretic co-clustering.

Chapter 6 is the first of the two parallel SVM chapters, presenting a two-stage approach, in which the first stage computes a kernel matrix approximation via parallelized Incomplete Cholesky Factorization (ICF). In the second stage, the Interior Point Method (IPM) is applied to the factorized matrix in parallel via a nontrivial rearrangement of the underlying computation. The method's scalability is achieved by partitioning the input data over the cluster nodes, with the factorization built up one row at a time. The approach achieves a two-orders-of-magnitude speedup on a 500-node cluster over a state-of-the-art baseline, LibSVM, and its MPI-based implementation has been released open-source.

Chapter 7 also considers parallelizing SVMs, focusing on the popular SMO algorithm as the underlying optimization method. This chapter is unique in the sense that it offers a hybrid high-level/low-level parallelization. At the high level, the instances are distributed across the nodes and SMO is executed on each node. To ensure that the optimization is going toward the global optimum, all locally optimal working sets are merged into the globally optimal working set in each SMO iteration. At the low level, specialized hardware (FPGA) is used to speed up the core kernel computation. The cluster implementation uses a custom-written TCP/UDP multicast-based communication interface and achieves a two-orders-of-magnitude speedup on a cluster of 48 dual-core nodes. The superlinear speedup is notable, illustrating that linearly increasing memory with efficient communication can significantly lighten the computational bottlenecks. The implementation of the method has been released open-source.

Chapter 8 covers LambdaMART, a boosted decision tree algorithm for learning to rank, an industry-defining task central to many information retrieval applications. The authors develop several distributed LambdaMART variants, one of which partitions features (rather than instances) across nodes and uses a master–slave structure to execute the algorithm. This approach achieves an order-of-magnitude speedup with an MPI-based implementation using 32 nodes and produces a learned model exactly equivalent to a sequential implementation. The chapter also describes experiments with instance-distributed approaches that approximate the sequential implementation.

Chapter 9 describes Transform Regression, a powerful classification and regression algorithm motivated by gradient boosting, but departing from it in several aspects that lead to dramatic speedups. Notably, transform regression uses prior trees' predictions as features in subsequent iterations, and employs linear regression in tree leaves. The algorithm is efficiently parallelized using the PML framework described in Chapter 4 and is shown to obtain high-accuracy models in fewer than 10 iterations, thus reducing the number of trees in the ensemble by two orders of magnitude, a gain that directly translates into corresponding speedups at inference time.

Chapter 10 focuses on approximate inference in probabilistic graphical models using loopy Belief Propagation (BP), a widely applied message-passing technique. The chapter provides a comparative analysis of several BP parallelization techniques and explores the role of message scheduling in efficient parallel inference. The culmination of the chapter is the Splash algorithm that sequentially propagates messages along spanning trees, yielding a provably optimal parallel BP algorithm. It is shown that the combination of dynamic scheduling and over-partitioning are essential for high-performance parallel inference. Experimental results in shared and distributed memory settings demonstrate that the Splash algorithm strongly outperforms alternative approaches, for example, achieving a 23-fold speedup on a 40-node distributed memory cluster, versus 14-fold for the next-best method.

Chapter 11 is dedicated to parallelizing learning in statistical latent variable models, such as topic models, which have been increasingly popular for identifying underlying structure in large data collections. The chapter focuses on distributing collapsed Gibbs sampling, a Markov Chain Monte Carlo (MCMC) technique, in the context of Latent Dirichlet Allocation (LDA) and Hierarchical Dirichlet Processes (HDP), two popular topic models, as well as for Bayesian networks in general, using Hidden Markov Models (HMMs) as an example. Scaling up to large datasets is achieved by distributing data instances and exchanging statistics across nodes, with synchronous and asynchronous variants considered. An MPI-based implementation over 1,024 processors is shown to achieve almost three-orders-of-magnitude speedups, with no loss in accuracy compared to baseline implementations, demonstrating that the approach successfully scales up topic models to multi-gigabyte text collections. The core algorithm is open source.

Chapter 12 is the first of two chapters dedicated to parallelization of clustering methods. It presents a parallel *spectral clustering* technique composed of three stages: sparsification of the affinity matrix, subsequent eigendecomposition, and obtaining final clusters via k-means using projected instances. It is shown that sparsification is vital for enabling the subsequent modules to run on large-scale datasets, and although it is the most expensive step, it can be distributed using MapReduce. The following

steps, eigendecomposition and k-means, are parallelized using MPI. The chapter presents detailed complexity analysis and extensive experimental results on text and image datasets, showing near-linear overall speedups on clusters up to 256 machines. Interestingly, results indicate that matrix sparsification has the benefit of improving clustering accuracy.

Chapter 13 proposes a parallelization scheme for *co-clustering*, the task of simultaneously constructing a clustering of data instances and a clustering of their features. The proposed algorithm optimizes an information-theoretic objective and uses an elemental *sequential* subroutine that "shuffles" the data of two clusters. The shuffling is done in parallel over the set of clusters that is split into pairs. Two results are of interest here: a two-orders-of-magnitude speedup on a 400-core MPI cluster, and evidence that *sequential* co-clustering is substantially better at revealing underlying structure of the data than an easily parallelizable k-means-like co-clustering algorithm that optimizes the same objective.

1.6.3 Part III: Alternative Learning Settings

This part of the book looks beyond the traditional supervised and unsupervised learning formulations, with the first three chapters focusing on parallelizing online, semi-supervised, and transfer learning. The fourth chapter presents a MapReduce-based method for scaling up feature selection, an integral part of machine learning practice that is well known to improve both computational efficiency and predictive accuracy.

Chapter 14 focuses on the *online learning* setting, where training instances arrive in a stream, one after another, with learning performed on one example at a time. Theoretical results show that delayed updates can cause additional error, so the algorithms focus on minimizing delay in a distributed environment to achieve high-quality solutions. To achieve this, features are partitioned ("sharded") across cores and nodes, and various delay-tolerant learning algorithms are tested. Empirical results show that a multicore and multinode parallelized version yields a speedup of a factor of 6 on a cluster of nine machines while sometimes even improving predictive performance. The core algorithm is open source.

Chapter 15 considers *semi-supervised learning*, where training sets include large amounts of unlabeled data alongside the labeled examples. In particular, the authors focus on graph-based semi-supervised classification, where the data instances are represented by graph nodes, with edges connecting those that are similar. The chapter describes *measure propagation*, a top-performing semi-supervised classification algorithm, and develops a number of effective heuristics for speeding up its parallelization. The heuristics reorder graph nodes to maximize the locality of message passing and hence are applicable to the broad family of message-passing algorithms. The chapter addresses both multicore and distributed settings, obtaining 85% efficiency on a 1,000-core distributed computer for a dataset containing 120 million graph-node instances on a key task in the speech recognition pipeline.

Chapter 16 deals with *transfer learning*: a setting where two or more learning tasks are solved consequently or concurrently, taking advantage of learning across the tasks. It is typically assumed that inputs to the tasks have different distributions that share supports. The chapter introduces DisCo, a distributed transfer learning framework,

where each task is learned on its own node concurrently with others, with knowledge transfer conducted over data instances that are shared across tasks. The chapter shows that the described parallelization method results in an order-of-magnitude speedup over a centralized implementation in the domains of recommender systems and text classification, with knowledge transfer improving accuracy of tasks over that obtained in isolation.

Chapter 17 is dedicated to distributed feature selection. The task of feature selection is motivated by the observation that predictive accuracy of many learning algorithms can be improved by extracting a subset of all features that provides an informative representation of data and excludes noise. Reducing the number of features also naturally decreases computational costs of learning and inference. The chapter focuses on Forward Feature Selection via Single Feature Optimization (SFO) specialized for logistic regression. Starting with an empty set of features, the method proceeds by iteratively selecting features that improve predictive performance, until no gains are obtained, with the remaining features discarded. A MapReduce implementation is described based on data instances partitioned over the nodes. In experiments, the algorithm achieves a speedup of approximately 16 on a 20-node cluster.

1.6.4 Part IV: Applications

The final part of the book presents several learning applications in distinct domains where scaling up is crucial to both computational efficiency and improving accuracy. The first two chapters focus on hardware-based approaches for speeding up inference in classic computer vision applications, object detection and recognition. In domains such as robotics and surveillance systems, model training is performed offline and can rely on extensive computing resources, whereas efficient inference is key to enabling real-time performance. The next chapter focuses on frequent subtree pattern mining, an unsupervised learning task that is important in many applications where data is naturally represented by trees. The final chapter in the book describes an exemplary case of deep-dive bottleneck analysis and pattern-driven design that lead to crucial inference speedups of a highly optimized speech recognition pipeline.

Chapter 18 describes two approaches to improving performance in vision tasks based on employing GPUs for efficient feature processing and induction. The first half of the chapter demonstrates that a combination of high-level features optimized for GPUs, synthetic expansion of training sets, and training using boosting distributed over a cluster yields significant accuracy gains on an object detection task. GPU-based detectors also enjoy a 100-fold speedup over their CPU implementation. In the second half, the chapter describes how Deep Belief Networks (DBNs) can be efficiently trained on GPUs to learn high-quality feature representations, avoiding the need for extensive human engineering traditionally required for inducing informative features in computer vision.

Chapter 19 shows how large parallel filter banks, commonly used for feature selection in vision tasks, can be effectively deployed via customized hardware implemented on FPGAs or ASICs (application-specific integrated circuits). Convolutional neural networks are tested, with their implementation using a data flow model enabling efficient parallelism. Comparisons with CPU and GPU implementations on standard

computer vision benchmarks demonstrate that customized hardware leads to 100-fold gains in overall efficiency measured with respect to power consumption.

Chapter 20 considers the problem of mining frequent subtrees, an important task in a number of domains ranging from bioinformatics to mining logs of browsing behavior. Detecting frequently occurring subtrees is computationally challenging in cases where tree instances can be arbitrarily complex in structure and large in size. The chapter demonstrates how frequent subtree mining can be efficiently parallelized on multicore systems, providing insights into various design aspects including memory utilization and load balancing. The chapter's approach is based on adaptive parallelization: employing multiple levels of task granularity to maximize concurrency. Multi-resolution task parallelism leads to high utilization of system resources, as demonstrated by near-linear speedups on standard web log, XML, and Natural Language Processing (NLP) benchmark datasets.

Chapter 21 focuses on parallelizing the inference process for Automatic Speech Recognition (ASR). In ASR, obtaining inference efficiency is challenging because highly optimized modern ASR models involve irregular graph structures that lead to load balancing issues in highly parallel implementations. The chapter describes how careful bottleneck analysis helps exploit the richest sources of concurrency for efficient ASR implementation on both GPUs and multicore systems. The overall application architecture presented here effectively utilizes single-instruction multiple-data (SIMD) operations for execution efficiency and hardware-supported atomic instructions for synchronization efficiency. Compared to an optimized single-thread implementation, these techniques provide an order-of-magnitude speedup, achieving recognition speed more than trice faster than real time, empowering development of novel ASR-based applications that can be deployed in an increasing variety of usage scenarios.

1.7 Bibliographic Notes

The goal of this book is presenting a practical set of modern platforms and algorithms that are effective in learning applications deployed in large-scale settings. This collection is by no means an exhaustive anthology: compiling one would be impossible given the breadth of ongoing research in the area. However, the references in each chapter provide a comprehensive overview of related literature for the described method as well as alternative approaches. The remainder of this section surveys a broader set of background references, along with pointers to software packages and additional recent work.

Many modern machine learning techniques rely on formulating the training objective as an optimization problem, allowing the use of the large arsenal of previously developed mathematical programming algorithms. Distributed and parallel optimization algorithms have been a fruitful research area for decades, yielding a number of theoretical and practical advances. Censor and Zenios (1997) is a canonical reference in this area that covers the parallelization of several algorithm classes for linear and quadratic programming, which are centerpieces of many modern machine learning techniques.

Parallelization of algorithms to enable scaling up to large datasets has been an active research direction in the data mining community since early nineties. The monograph of Freitas and Lavington (1998) describes early work on parallel data mining from a database-centric perspective. A survey by Provost and Kolluri (1999) provides a structured overview of approaches for scaling up inductive learning algorithms, categorizing them into several groups that include parallelization and data partitioning. Two subsequent edited collections (Zaki and Ho, 2000; Kargupta and Chan, 2000) are representative of early research on parallel mining algorithms and include chapters that describe several prototype frameworks for concurrent mining of partitioned data collections.

In the statistical machine learning community, scaling up kernel-based methods (of which Support Vector Machines are the most prominent example) has been a topic of significant research interest due to the super-linear computational complexity of most training methods. The volume edited by Bottou et al. (2007) presents a comprehensive set of modern solutions in this area, which primarily focus on algorithmic aspects, but also include two parallel approaches, one of which is extended in Chapter 7 of the present book.

One parallelization framework that has been a subject of study in the distributed data mining community is Peer-To-Peer (P2P) networks, which are decentralized systems composed of nodes that are highly non-stationary (nodes often go offline), where communication is typically asynchronous and has high latency. These issues are counterbalanced by the potential for very high scalability of storage and computational resources. Designing machine learning methods for P2P settings is a subject of ongoing work (Datta et al., 2009; Bhaduri et al., 2008; Luo et al., 2007).

Two recently published textbooks (Lin and Dyer, 2010; Rajaraman and Ullman, 2010) may be useful companion references for readers of the present book who are primarily interested in algorithms implemented via MapReduce. Lin and Dyer (2010) offer a gentle introduction to MapReduce, with plentiful examples focused on text processing applications, whereas Rajaraman and Ullman (2010) describe a broad array of mining tasks on large datasets, covering MapReduce and parallel clustering in depth.

MapReduce and DryadLINQ presented in Chapters 1 and 3 are representative samples of an increasingly popular family of distributed platforms that combine three layers: a parallelization-friendly programming language, a task execution engine, and a distributed filesystem. *Hadoop*[2] is a prominent, widely used open-source member of this family, programmable via APIs for popular imperative languages such as Java or Python, as well as via specialized languages with a strong functional and declarative flavor, such as Apache Pig and Hive.[3] Another, related set of tools such as Aster Data[4] or Greenplum[5] provide a MapReduce API for distributed databases. Finally, MADlib[6] provides a library of learning tools on top of distributed databases, while Apache Mahout[7] is a nascent library of machine learning algorithms being developed

[2] http://hadoop.apache.org/.
[3] http://pig.apache.org/ and http://hive.apache.org/.
[4] http://www.asterdata.com/resources/mapreduce.php.
[5] http://www.greenplum.com.
[6] http://madlib.net.
[7] http://mahout.apache.org.

for Hadoop. In this book, PML (presented in Chapter 4) is an example of an off-the-shelf machine learning toolbox based on a general library of parallelization primitives especially suited for learning algorithms.

Since starting this project, a few other parallel learning algorithms of potential interest have been published. Readers of Chapter 11 may be interested in a new cluster parallel Latent Dirichlet Allocation algorithm (Smola and Narayanamurthy, 2010). Readers of Chapter 8 may be interested in a similar algorithm made to interoperate with the Hadoop file system (Ye et al., 2009).

References

Bakir, G., Hofmann, T., Schölkopf, B., Smola, A., Taskar, B., and Vishwanathan, S. V. N. (eds). 2007. *Predicting Structured Data*. Cambridge, MA: MIT Press.

Bhaduri, K., Wolff, R., Giannella, C., and Kargupta, H. 2008. Distributed Decision-Tree Induction in Peer-to-Peer Systems. *Statistical Analysis and Data Mining*, **1**, 85–103.

Bottou, L., Chapelle, O., DeCoste, D., and Weston, J. (eds). 2007. *Large-Scale Kernel Machines*. MIT Press.

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

Datta, S., Giannella, C. R., and Kargupta, H. 2009. Approximate Distributed K-Means Clustering over a Peer-to-Peer Network. *IEEE Transactions on Knowledge and Data Engineering*, **21**, 1372–1388.

Dean, Jeffrey, and Ghemawat, Sanjay. 2004. MapReduce: Simplified Data Processing on Large Clusters. In: *Sixth Symposium on Operating System Design and Implementation (OSDI-2004)*.

Flynn, M. J. 1972. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, **21**(9), 948–960.

Freitas, A. A., and Lavington, S. H. 1998. *Mining Very Large Databases with Parallel Processing*. Kluwer.

Gropp, W., Lusk, E., and Skjellum, A. 1994. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press.

Kargupta, H., and Chan, P. (eds). 2000. *Advances in Distributed and Parallel Knowledge Discovery*. Cambridge, MA: AAAI/MIT Press.

Lin, J., and Dyer, C. 2010. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool.

Luo, P., Xiong, H., Lu, K., and Shi, Z. 2007. Distributed classification in peer-to-peer networks. Pages 968–976 of: *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Provost, F., and Kolluri, V. 1999. A survey of methods for scaling up inductive algorithms. *Data Mining and Knowledge Discovery*, **3**(2), 131–169.

Rajaraman, A., and Ullman, J. D. 2010. *Mining of Massive Datasets*. http://infolab.stanford.edu/~ullman/mmds.html.

Smola, A. J., and Narayanamurthy, S. 2010. An Architecture for Parallel Topic Models. *Proceedings of the VLDB Endowment*, **3**(1), 703–710.

Ye, J., Chow, J.-H., Chen, J., and Zheng, Z. 2009. Stochastic Gradient Boosted Distributed Decision Trees. In: *CIKM '09 Proceeding of the 18th ACM Conference on Information and Knowledge Management*.

Zaki, M. J., and Ho, C.-T. (eds). 2000. *Large-scale Parallel Data Mining*. New York: Springer.

PART ONE

Frameworks for Scaling Up Machine Learning

CHAPTER 2

MapReduce and Its Application to Massively Parallel Learning of Decision Tree Ensembles

Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo

In this chapter we look at leveraging the MapReduce distributed computing framework (Dean and Ghemawat, 2004) for parallelizing machine learning methods of wide interest, with a specific focus on learning ensembles of classification or regression trees. Building a production-ready implementation of a distributed learning algorithm can be a complex task. With the wide and growing availability of MapReduce-capable computing infrastructures, it is natural to ask whether such infrastructures may be of use in parallelizing common data mining tasks such as tree learning. For many data mining applications, MapReduce may offer scalability as well as ease of deployment in a production setting (for reasons explained later).

We initially give an overview of MapReduce and outline its application in a classic clustering algorithm, *k*-means. Subsequently, we focus on PLANET: a scalable distributed framework for learning tree models over large datasets. PLANET defines tree learning as a series of distributed computations and implements each one using the *MapReduce* model. We show how this framework supports scalable construction of classification and regression trees, as well as ensembles of such models. We discuss the benefits and challenges of using a MapReduce compute cluster for tree learning and demonstrate the scalability of this approach by applying it to a real-world learning task from the domain of computational advertising.

MapReduce is a simple model for distributed computing that abstracts away many of the difficulties in parallelizing data management operations across a cluster of commodity machines. By using MapReduce, one can alleviate, if not eliminate, many complexities such as data partitioning, scheduling tasks across many machines, handling machine failures, and performing inter-machine communication. These properties have motivated many companies to run MapReduce frameworks on their compute clusters for data analysis and other data management tasks. MapReduce has become in some sense an industry standard. For example, there are open-source implementations such as Hadoop that can be run either in-house or on cloud computing services such as Amazon EC2.[1]

[1] http://aws.amazon.com/ec2/.

Startups such as Cloudera[2] offer software and services to simplify Hadoop deployment, and companies including Google, IBM, and Yahoo! have granted several universities access to MapReduce clusters to advance parallel computing research.[3]

Despite the growing popularity of MapReduce, its application to standard data mining and machine learning tasks needs to be better studied. In this chapter we focus on one such task: tree learning. We believe that a tree learner capable of exploiting a MapReduce cluster can effectively address many scalability issues that arise in building tree models on massive datasets. Our choice of focusing on tree models is motivated primarily by their popularity. Tree models are used in many applications because they are interpretable, can model complex interactions, and can easily handle both numerical and categorical features. Recent studies have shown that tree models, when combined with ensemble techniques, provide excellent predictive performance across a wide variety of domains (Caruana et al., 2008; Caruana and Niculescu-Mizil, 2006). The effectiveness of boosted trees has also been separately validated by other researchers; for example, Gao et al. (2009) present an algorithm for model interpolation and ensembles using boosted trees that performs well on *web search ranking*, even when the test data is quite different from the training data.

This chapter describes our experiences with developing and deploying a MapReduce-based tree learner called PLANET, which stands for Parallel Learner for Assembling Numerous Ensemble Trees. The development of PLANET was motivated by a real application in sponsored search advertising, in which massive clickstreams are processed to develop a model that can predict the quality of user experience following the click on a sponsored search ad (Sculley et al., 2009). We show how PLANET effectively scales to large datasets, describe experiments that highlight the performance characteristics of PLANET, and demonstrate the benefits of various optimizations that we implemented within the system. We show that although MapReduce is not a panacea, it still provides a powerful basis on which scalable tree learning can be implemented.

2.1 Preliminaries

Let us first define some notation and terminology that we will use in the rest of the chapter. Let $\mathcal{X} = \{X_1, X_2, \ldots X_N\}$ be a set of features with domains $\mathbb{D}_{X_1}, \mathbb{D}_{X_2}, \ldots \mathbb{D}_{X_N}$ respectively. Let Y be the class label with domain $\mathbb{D}_Y$. Consider a dataset $D = \{(\mathbf{x}_i, y_i) | \mathbf{x}_i \in \mathbb{D}_{X_1} \times \mathbb{D}_{X_2} \times \ldots \mathbb{D}_{X_N}, y_i \in \mathbb{D}_Y\}$ sampled from an unknown distribution, where the ith data vector $\mathbf{x}_i$ has a class label y_i associated with it. Given the dataset D, the goal of supervised learning is to learn a function (or *model*) $F : \mathbb{D}_{X_1} \times \mathbb{D}_{X_2} \times \ldots \mathbb{D}_{X_N} \rightarrow \mathbb{D}_Y$ that minimizes the difference between the predicted and the true values of Y, on unseen data drawn from the same distribution as D. If $\mathbb{D}_Y$ is continuous, the learning problem is a regression problem; if $\mathbb{D}_Y$ is categorical, it is a classification problem. In contrast, in unsupervised learning (e.g., clustering), the goal is to learn a function

[2] www.cloudera.com/.

[3] For example, see www.youtube.com/watch?v=UBrDPRlplyo and www.nsf.gov/news/news_summ.jsp?cntn_id=111470.

$F : \mathbb{D}_{X_1} \times \mathbb{D}_{X_2} \times \ldots \mathbb{D}_{X_N} \times \mathbb{D}_Y$ that best approximates the joint distribution of X and Y in D. For notational simplicity, we will use Y both to denote a class label in supervised methods and to denote a cluster label in clustering.

Let $\mathcal{L}$ be a function that quantifies the disagreement between the value of the function $F(\mathbf{x}_i)$ (predicted label) and the actual class label y_i, for example, the squared difference between the actual label and the predicted label, known as the *squared loss*. A model that minimizes the net loss $\sum_{(\mathbf{x}_i, y_i) \in D} \mathcal{L}(F(\mathbf{x}_i), y_i)$ on the *training set* D may not generalize well when applied to unseen data (Vapnik, 1995). Generalization is attained through controlling model complexity by various methods, e.g., pruning and ensemble learning for tree models (Breiman, 2001). The learned model can be evaluated by measuring its net loss when applied to a holdout dataset.

2.1.1 MapReduce

MapReduce (Dean and Ghemawat, 2004) provides a framework for performing a two-phase distributed computation on large datasets, which in our case is a training dataset D. In the *Map* phase, the system partitions D into a set of disjoint units that are assigned to worker processes, known as *mappers*. Each mapper (in parallel with the others) scans through its assigned data and applies a user-specified map function to each record. The output of the map function is a set of key–value pairs that are collected by the *Shuffle* phase, which groups them by key. The master process redistributes the output of shuffle to a series of worker processes called *reducers*, which perform the *Reduce* phase. Each reducer applies a user-specified reduce function to all the values for a key and outputs the value of the reduce function. The collection of final values from all of the reducers is the final output of MapReduce.

MapReduce Example: Word Histogram

Let us demonstrate how MapReduce works through the following simple example. Given a collection of text documents, we would like to compute the word histogram in this collection, that is, the number of times each word occurs in all the documents. In the Map phase, the total set of documents is partitioned into subsets, each of which is given to an individual mapper. Each mapper goes through the subset of documents assigned to it and outputs a series of $\langle word_i, count_i \rangle$ values as the key–value pair, where $count_i$ is the number of times $word_i$ occurs among the documents seen by the mapper. Each reducer takes the values associated with the a particular key (in this case, a word) and aggregates the values (in this case, word counts) for each key. The output of the reducer phase gives us the counts per word across the entire document collection, which is the desired word histogram.

MapReduce Example: k-means Clustering

MapReduce can be used to efficiently solve supervised and unsupervised learning problems at scale. In the rest of the chapter, we focus on using MapReduce to learn ensembles of decision trees for classification and regression. In this section, we briefly

describe how MapReduce can be used for k-means clustering, to show its efficiency in unsupervised learning.

The k-means clustering algorithm (MacQueen, 1967) is a widely used clustering method that applies iterative relocation of points to find a locally optimal partitioning of a dataset. In k-means, the total distance between each data point and a representative point (centroid) of the cluster to which it is assigned is minimized. Each iteration of k-means has two steps. In the cluster assignment step, k-means assigns each point to a cluster such that, of all the current cluster centroids, the point is closest to the centroid of that cluster. In the cluster re-estimation step, k-means re-estimates the new cluster centroids based on the reassignments of points to clusters in the previous step. The cluster re-assignment and centroid re-estimation steps proceed in iterations until a specified convergence criterion is reached, such as when the total distance between points and centroids does not change substantially from one iteration to another.

In the MapReduce implementation of k-means, each mapper in the Map phase is assigned a subset of points. For these points, the mapper does the cluster assignment step – it computes y_i, the index of the closest centroid for each point $\mathbf{x}_i$, and also computes the relevant cluster aggregation statistics: S_j, the sum of all points seen by the mapper assigned to the jth cluster; and n_j, the number of points seen by the mapper assigned to the jth cluster. At the end of the Map phase, the cluster index and the corresponding cluster aggregation statistics (sum and counts) are output. The Map algorithm is shown in Algorithm 1.

Algorithm 1: k-means::Map

Input: Training data $\mathbf{x} \in D$, number of clusters k, distance measure d

If first Map iteration **then**
  Initialize the k cluster centroids $\mathbf{C}$ randomly
Else
  Get the k cluster centroids $\mathbf{C}$ from the previous Reduce step
Set $S_j = 0$ and $n_j = 0$ for $j = \{1, \cdots, k\}$
For each $\mathbf{x}_i \in \mathbf{D}$ **do**
  $y_i = \arg\min_j d(\mathbf{x}_i, \mathbf{c}_j)$
  $S_{y_i} = S_{y_i} + \mathbf{x}_i$
  $n_{y_i} = n_{y_i} + 1$
For each $j \in \{1, \cdots, k\}$ **do**
  Output(j, $< S_j, n_j >$)

The reducer does the centroid re-estimation step – it combines the values for a given clusterid key by merging the cluster statistics. For each cluster j, the reducer gets a list of cluster statistics $[< S_j^l, n_j^l >]$, where l is an index over the list – the lth partial sum S_j^l in this list represents the sum of some points in cluster j seen by any particular mapper, whereas the lth number n_j^l in the list is the count of the number of points in that set. The reducer calculates the average of S_j^l to get the updated centroid $\mathbf{c}_j$ for cluster j. The Reduce algorithm is shown in Algorithm 2.

Algorithm 2: k-means::Reduce

Input: List of centroid statistics – partial sums and counts $[< S_j^l, n_j^l >]$ – for each centroid $j \in \{1, \cdots, k\}$

1: **For each** $j \in \{1, \cdots, k\}$ **do**
2: Let λ be the length of the list of centroid statistics
3: $n_j = 0, S_j = 0$
4: **For each** $l \in \{1, \cdots, \lambda\}$ **do**
5: $n_j = n_j + n_j^l$
6: $S_j = S_j + S_j^l$
7: $\mathbf{c}_j = \frac{S_j}{n_j}$
8: Output(j, $\mathbf{c}_j$)

The whole clustering is run by a Master, which is responsible for running the Map (cluster assignment) and Reduce (centroid re-estimation) steps iteratively until k-means converges.

2.1.2 Tree Models

Classification and regression trees are one of the oldest and most popular data mining models (Duda et al., 2001). Tree models represent F by recursively partitioning the data space $\mathbb{D}_{X_1} \times \mathbb{D}_{X_2} \times \ldots \mathbb{D}_{X_N}$ into non-overlapping regions, with a simple model in each region.

Figure 2.1 shows an example tree model. Non-leaf nodes in the tree define region boundaries in the data space. Each region boundary is represented as a predicate

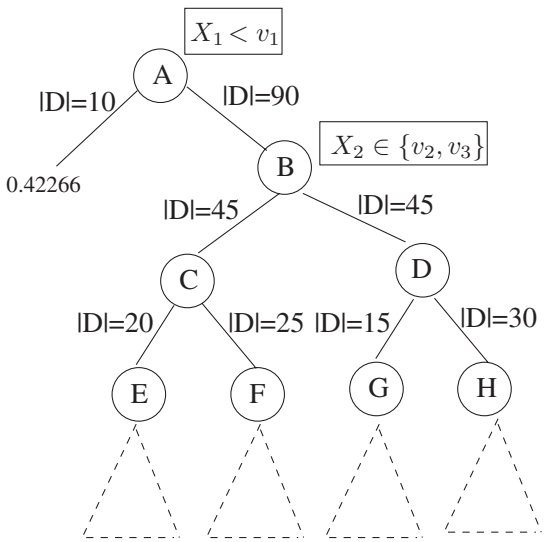

Figure 2.1 Example Tree. Note that the labels on the nodes (in boxes) are the split predicates, whereas the labels on the edges are the sizes of the dataset in each branch (|D| denotes the dataset size in that branch).

on a feature in $\mathcal{X}$. If the feature is numerical, the predicate is of the form $X < v$, $v \in \mathbb{D}_X$ (e.g., Node A in Figure 2.1). Categorical features have predicates of the form $X \in \{v_1, v_2, \ldots v_k\}$, $v_1 \in \mathbb{D}_X, v_2 \in \mathbb{D}_X, \ldots v_k \in \mathbb{D}_X$, (e.g., Node B in Figure 2.1). The path from the root to a leaf node in the tree defines a region. Leaf nodes (e.g., the left child of A in Figure 2.1) contain a region prediction that in most cases is a constant value or some simple function. To make predictions on an unknown $\mathbf{x}$, the tree is traversed to find the region containing $\mathbf{x}$. The region containing $\mathbf{x}$ is the path from the root to a leaf in the tree along which all non-leaf predicates are true when evaluated on $\mathbf{x}$. The prediction given by this leaf is used as the value for $F(\mathbf{x})$.

In our example tree model, predicate evaluations at non-leaf nodes have only two outcomes, leading to binary splits. Although tree models can have non-binary splits, for the sake of simplicity we focus only on binary splits for the remainder of this chapter. All our techniques also apply to tree algorithms with non-binary splits with straightforward modifications.

Tree models are popular because they are interpretable, capable of modeling complex classification and regression tasks, and handle both numerical and categorical domains. Caruana and Niculescu-Mizil (2006) show that tree models, when combined with ensemble learning methods such as bagging (Breiman, 1996), boosting (Freund and Schapire, 1996), and forests (Breiman, 2001), outperform many other popular learning methods in terms of prediction accuracy. A thorough discussion of tree models and different ensemble methods is beyond the scope of this chapter – see Rokach and Maimon (2008) for a good review.

2.1.3 Learning Tree Models

Previous work on learning tree models is extensive. For a given training dataset D, finding the optimal tree is known to be NP-Hard; thus most algorithms use a greedy top-down approach to construct the tree (Algorithm 3) (Duda et al., 2001). At the root of the tree, the entire training dataset D is examined to find the *best* split predicate for the root. The dataset is then partitioned along the split predicate and the process is repeated recursively on the partitions to build the child nodes.

Algorithm 3: InMemoryBuildNode

Input: Node n, Data D

```
(n →split,D_L,D_R)=FindBestSplit(D)
If StoppingCriteria(D_L) then
    n →left_prediction=FindPrediction(D_L)
Else
    InMemoryBuildNode(n →left,D_L)
If StoppingCriteria(D_R) then
    n →right_prediction=FindPrediction(D_R)
Else
    InMemoryBuildNode(n →right,D_R)
```

Finding the best split predicate for a node (Line 3 of Algorithm 3) is the most important step in the greedy learning algorithm and has been the subject of much of the research in tree learning. Numerous techniques have been proposed for finding the right split at a node, depending on the particular learning problem. The main idea is to reduce the *impurity* (I) in a node. Loosely defined, the impurity at a node is a measure of the dissimilarity in the Y values of the training records D that are input to the node. The general strategy is to pick a predicate that maximizes $I(D) - (I(D_L) + I(D_R))$, where D_L and D_R are the datasets obtained after partitioning D on the chosen predicate. At each step, the algorithm greedily partitions the data space to progressively reduce region impurity. The process continues until all Y values in the input dataset D to a node are the same, at which point the algorithm has isolated a pure region (Lines 3 and 7). Some algorithms do not continue splitting until regions are completely pure and instead stop once the number of records in D falls below a predefined threshold.

Popular impurity measures are derived from measures such as entropy, Gini index, and variance (Rokach and Maimon, 2008), to name only a few. PLANET uses an impurity measure based on variance (Var) to evaluate the quality of a split. The higher the variance in the Y values of a node, the greater the node's impurity. Further details on the split criteria are discussed in Section 2.1.4. Although we focus concretely on variance as our split criterion for the remainder of this chapter, as long as a split metric can be computed on subsets of the training data and later aggregated, PLANET can be easily extended to support it.

Scalability Challenge

The greedy tree induction algorithm we have described is simple and works well in practice. However, it does not scale well to large training datasets. FindBestSplit requires a full scan of the node's input data, which can be large at higher levels of the tree. Large inputs that do not fit in main memory become a bottleneck because of the cost of scanning data from secondary storage. Even at lower levels of the tree where a node's input dataset D is typically much smaller than D, loading D into memory still requires reading and writing partitions of D to secondary storage multiple times.

Previous work has looked at the problem of building tree models from datasets that are too large to fit completely in main memory. Some of the known algorithms are disk-based approaches that use clever techniques to optimize the number of reads and writes to secondary storage during tree construction (e.g., Mehta, Agrawal, and Rissanen, 1996). Other algorithms scan the training data in parallel using specialized parallel architectures (e.g., Bradford, Fortes, and Bradford, 1999). We defer a detailed discussion of these approaches and how they compare to PLANET to Section 2.7. As we show in Section 2.7, some of the ideas used in PLANET have been proposed in the past; however, we are not aware of any efforts to build massively parallel tree models on commodity hardware using the MapReduce framework.

Post-pruning learned trees to prevent overfitting is also a well studied problem. However, with ensemble models (Section 2.4), post-pruning is not always needed. Because PLANET is primarily used for building ensemble models, we do not discuss post-pruning in this chapter.

2.1.4 Regression Trees

Regression trees are a special case of tree models where the output feature Y is continuous (Breiman, 2001). We focus primarily on regression trees within this chapter because most of our use cases require predictions on continuous outputs. Note that any regression tree learner also supports binary (0-1) classification tasks by modeling them as instances of logistic regression. The core operations of regression tree learning in Algorithm 3 are implemented as follows.

FindBestSplit(D): In a regression tree, D is split using the predicate that results in the largest reduction in variance. Let $Var(D)$ be the variance of the class label Y measured over all records in D. At each step the tree learning algorithm picks a split that maximizes

$$|D| \times Var(D) - (|D_L| \times Var(D_L) + |D_R| \times Var(D_R)), \tag{2.1}$$

where $D_L \subset D$ and $D_R \subset D$ are the training records in the left and right subtree after splitting D by a predicate.

Regression trees use the following policy to determine the set of predicates whose split quality will be evaluated:

- For numerical domains, split predicates are of the form $X_i < v$, for some $v \in \mathbb{D}_{X_i}$. To find the best split, D is sorted along X_i, and a split point is considered between each adjacent pair of values for X_i in the sorted list.
- For categorical domains, split predicates are of the form $X_i \in \{v_1, v_2, \ldots v_k\}$, where $\{v_1, v_2, \ldots v_k\} \in \mathcal{P}(\mathbb{D}_{X_i})$, the power set of $\mathbb{D}_{X_i}$. Breiman et al. (1984) present an algorithm for finding the best split predicate for a categorical feature without evaluating all possible subsets of $\mathbb{D}_{X_i}$. The algorithm is based on the observation that the optimal split predicate is a subsequence in the list of values for X_i sorted by the average Y value.

StoppingCriteria(D): A node in the tree is not expanded if the number of records in D falls below a threshold. Alternatively, the user can also specify the maximum depth to which a tree should be built.

FindPrediction(D): The prediction at a leaf is simply the average of all the Y values in D.

2.2 Example of PLANET

The PLANET framework breaks up the process of constructing a tree model into a set of MapReduce tasks. Dependencies exist between the different tasks, and PLANET uses clever scheduling methods to efficiently execute and manage them. Before delving into the technical details of the framework, we begin with a detailed example of how tree induction proceeds in PLANET.

The example introduces the different components in PLANET, describes their roles, and provides a high-level overview of the entire system. To keep the example simple,

we discuss only the construction of a single tree. The method extends naturally to ensembles of trees, as we discuss in Section 2.4.

Example Setup: Let us assume that we have a training dataset D^* with 100 records. Further assume that tree induction stops once the number of training records at a node falls below 10. Let the tree in Figure 2.1 be the model that will be learned if we run Algorithm 3 on a machine with sufficient memory. Our goal in this example is to demonstrate how PLANET constructs the tree in Figure 2.1 when there is a memory constraint limiting Algorithm 3 to operating on inputs of size 25 records or less.

2.2.1 Components

At the heart of PLANET is the *Controller*, a single machine that initiates, schedules, and controls the entire tree induction process. The Controller has access to a compute cluster on which it schedules MapReduce jobs. In order to control and coordinate tree construction, the Controller maintains the following:

- *ModelFile* (M): The Controller constructs a tree using a set of MapReduce jobs, each of which builds different parts of the tree. At any point, the model file contains the entire tree constructed so far.

Given the ModelFile (M), the Controller determines the nodes at which split predicates can be computed. In the example of Figure 2.1, if M has nodes A and B, then the Controller can compute splits for C and D. This information is stored in two queues.

- *MapReduceQueue* (MRQ): This queue contains nodes for which D is too large to fit in memory (i.e., > 25 in our example).
- *InMemoryQueue* (IMQ): This queue contains nodes for which D fits in memory (i.e., ≤ 25 in our example).

As tree induction proceeds, the Controller dequeues nodes off MRQ and IMQ and schedules MapReduce jobs to find split predicates at the nodes. Once a MapReduce job completes, the Controller updates M with the nodes and their split predicates and then updates MRQ and IMQ with new nodes at which split predicates can be computed. Each MapReduce job takes as input a set of nodes (N), the training data set (D^*), and the current state of the model (M). The Controller schedules two types of MapReduce jobs:

- Nodes in MRQ are processed using MR_ExpandNodes, which for a given set of nodes N computes a candidate set of good split predicates for each node in N.
- Nodes in IMQ are processed using MR_InMemory. Recall that nodes in IMQ have input datasets D that are small enough to fit in memory. Therefore, given a set of nodes N, MR_InMemory completes tree induction at nodes in N using Algorithm 3.

We defer details of the MapReduce jobs to Section 2.3. In the remainder of this section, we tie the foregoing components together and walk through the example.

2.2.2 Walkthrough

When tree induction begins, M, MRQ, and IMQ are all empty. The only node the Controller can expand is the root (A). Finding the split for A requires a scan of the entire training dataset of 100 ($\geq$ 25) records. Because this set is too large to fit in memory, A is pushed onto MRQ and IMQ stays empty.

After initialization, the Controller dequeues A from MRQ and schedules a job MR_ExpandNodes({A}, M, D^*). This job computes a set of good splits for node A along with some additional information about each split. Specifically, for each split we compute (1) the quality of the split (i.e., the reduction in impurity), (2) the predictions in the left and right branches, and (3) the number of training records in the left and right branches.

The split information computed by MR_ExpandNodes gets sent back to the Controller, which selects the best split for node A. In this example, the best split has 10 records in the left branch and 90 records in the right. The selected split information for node A is then added into the ModelFile. The Controller next updates the queues with new nodes at which split predicates can be computed. The left branch of A has 10 records. This matches the stopping criteria, and hence no new nodes are added for this branch. For the right branch with 90 records ($\geq$ 25), node B can be expanded and is pushed onto MRQ.

Tree induction continues by dequeuing node B and scheduling MR_ExpandNodes({B}, M, D^*). Note that for expanding node B, we need only the records that went down the right subtree of A, but to minimize bookkeeping, PLANET passes the entire training dataset to the MapReduce. As we describe in Section 2.3.3, MR_ExpandNodes uses the current state of the ModelFile to determine the subset of D^* that will be input to B.

Once the Controller has received the results for the MapReduce on node B and updated M with the split for B, it can now expand both C and D. Both of these nodes get 45 records as input and are therefore pushed on to MRQ. The Controller can now schedule a single MR_ExpandNodes({C, D}, M, D^*) job to find the best splits for both nodes C and D. Note that by expanding C and D in a single step, PLANET expands trees breadth first as opposed to the depth first process used by the in-memory Algorithm 3.

Once the Controller has the obtained the splits for C and D, it can schedule jobs to expand nodes E, F, G, and H. Of these, H uses 30 records, which still cannot fit in memory and hence get added to MRQ. The input sets to E, F, G are small enough to fit into memory, and hence tree induction at these nodes can be completed in-memory. The Controller pushes these nodes into the IMQ.

The Controller next schedules two MapReduce jobs simultaneously. MR_InMemory({E,F,G}, M, D^*) completes tree induction at nodes E, F, and G because the input datasets to these nodes are small. MR_ExpandNodes({H}, M, D^*) computes good splits for H. Once the InMemory job returns, tree induction for the subtrees rooted at E, F, and G is complete. The Controller updates MRQ and IMQ with the children of node H and continues tree induction. PLANET aggressively tries to maximize the number of nodes at which split predicates can be computed in parallel and schedules multiple MapReduce jobs simultaneously.