Templates for scalable data analysis

2 Synchronous Templates

Amr Ahmed, Alexander J Smola, Markus Weimer
Yahoo! Research & UC Berkeley & ANU
Running Example

Inbox

Spam
Running Example

- Inbox
- Spam

Spam Filter
Running Example
Running Example

Spam Filter

Inbox

Spam
Running Example

Spam Filter

Inbox

Spam
Running Example
Running Example

Spam Filter

Inbox

Spam
Running Example

Spam Filter

Inbox

Spam

Webmail UI
Running Example

Inbox

Spam

Filter

Webmail UI
Running Example

- Inbox
- Spam
- Spam Filter
- Webmail UI

Logged event
Running Example
ML Workflow

Log Files
Images
Text
Geodata
Sound
Videos
Strokes

Raw Data
Example
Model
ML Workflow

- Log Files
- Images
- Text
- Strokes
- Geodata
- Sound
- Videos

Example Formation

Raw Data • Example • Model
ML Workflow

Raw Data  Example  Model
ML Workflow

- Raw Data
- Example
- Model

Steps:
- Example Formation
- Training
- Prediction

Data Types:
- Log Files
- Videos
- Images
- Text
- Strokes
- Geodata
- Sound
ML Workflow

Raw Data  Example  Model
ML Workflow

Raw Data  Example  Model
• Example Formation in Pig
• Modeling today Hadoop, Spark, Pregel
• Future Declarative Systems
Example Formation
Example Formation

EMail

Log Files
Example Formation

EMail → ID → Bag of Words

Log Files
Example Formation

EMail -> ID -> Bag of Words

Log Files -> ID -> Label
Example Formation

- EMail
  - ID
  - Bag of Words

- Log Files
  - ID
  - Label

- Label
  - Bag of Words
Example Formation
Example Formation

Feature Extraction

Log Files → ID → Label

Example

Bag of Words

Label

ID
Example Formation

Feature Extraction

Label Extraction

Example
Requirements

EMail

Log Files
Requirements

Data Parallel Functions

EMail → ID → Bag of Words

Log Files → ID → Label
Requirements

- EMail
  - ID
  - Bag of Words

- Log Files
  - ID
  - Label

- Label
- Large Scale Join
- Bag of Words
Apache Pig

• Relational Query Language
• Similar to SQL
• Performs runtime optimizations
• Executes Queries on Apache Hadoop
• Developed and heavily used by Yahoo!
• Open Source (Apache)

http://pig.apache.org
Pig: Example Formation

- Example formation
- User Defined Function
- Feature and Label Extraction
- JOIN between the outputs of the above
- Applied via FOREACH ... GENERATE
Machine Learning in MapReduce
MapReduce

- Parallel, Distributed programming framework

User defines two functions:
  - `map(x)` **emits** *(key, value)* pairs
  - `reduce(k, x[])` **gets all values for a key,** produces output
MapReduce

Map
MapReduce

Map

GroupBy (Shuffle)

Reduce
MapReduce

Map

GroupBy (Shuffle)

Reduce
MapReduce

Map

GroupBy (Shuffle)

Reduce
MapReduce

Map

GroupBy (Shuffle)

Reduce
MapReduce

Map

GroupBy (Shuffle)

Reduce
MapReduce

Map

GroupBy (Shuffle)

Reduce
MapReduce

Map

GroupBy (Shuffle)

Reduce
• Open Source MapReduce Implementation:
  • **HDFS**: Distributed File System
  • **YARN**: Resource Management
  • **MapReduce**: Programming Framework

http://hadoop.apache.org
• Open Source MapReduce Implementation
  - **HDFS**: Distributed File System
  - **YARN**: Resource Management
  - **MapReduce**: Programming Framework

New in Hadoop 0.23

http://hadoop.apache.org
Efficient Noise-Tolerant Learning from Statistical Queries

MICHAEL KEARNS
AT&T Laboratories—Research, Florham Park, New Jersey

Abstract. In this paper, we study the problem of learning in the presence of classification noise in the probabilistic learning model of Valiant and its variants. In order to identify the class of "robust" learning algorithms in the most general way, we formalize a new but related model of learning from statistical queries. Intuitively, in this model, a learning algorithm is forbidden to examine examples of the unknown target function, but is given access to an oracle providing probabilities over the sample space of random examples.

One of our main results shows that any class of functions learnable from statistical queries is in fact learnable with classification noise in Valiant's model, with a noise rate approaching the information-theoretic barrier of 1/2. We then demonstrate the generality of the statistical query model, showing that practically every class learnable in Valiant's model and its variants can also be learned in the new model (and thus can be learned in the presence of noise). A notable exception to this statement is the class of parity functions, which we prove is not learnable from statistical queries, and for which no noise-tolerant algorithm is known.

Categories and Subject Descriptors: F. [Theory of Computation]; G.3 [Probability and Statistics]; I.2. [Artificial Intelligence]; I.5 [Pattern Recognition]

General Terms: Computational learning theory, Machine learning

Additional Key Words and Phases: Computational learning theory, machine learning

1. Introduction

In this paper, we study the extension of Valiant's learning model [Val 1984] in which the positive or negative classification label provided with each example may be corrupted by random noise. This extension was first examined in the learning theory literature by Angluin and Laird [1988], who formalized the simplest type of white label noise and then sought algorithms tolerant of the highest possible rate of noise. In addition to being the subject of a number of theoretical studies [Angluin and Laird 1988; Laird 1988; Sloan 1988; Ke...]

MapReduce for ML

- Learning algorithm can access the learning problem only through a statistical query oracle
- The statistical query oracle returns an estimate of the expectation of a function $f(x,y)$ (averaged over the data distribution).
MapReduce for ML

- **Rephrase oracle in summation form.**

- **Map:** Calculate function estimates over sub-groups of data.

- **Reduce:** Aggregate the function estimates from various sub-groups.

---

**Abstract**

We are at the beginning of the multicore era. Computers will have increasingly many cores (processors), but there is still no good programming framework for these architectures, and thus no simple and unified way for machine learning to take advantage of the potential speed up. In this paper, we develop a broadly applicable parallel programming method, one that is easily applied to many different learning algorithms. Our work is in distinct contrast to the tradition in machine learning of designing (often ingenious) ways to speed up a single algorithm at a time. Specifically, we show that algorithms that fit the Statistical Query model [15] can be written in a certain “summation form,” which allows them to be easily parallelized on multicore computers. We adapt Google’s map-reduce [7] paradigm to demonstrate this parallel speed up technique on a variety of learning algorithms including locally weighted linear regression (LWLR), k-means, logistic regression (LR), naive Bayes (NB), SVM, ICA, PCA, gaussian discriminant analysis (GDA), EM, and backpropagation (NN). Our experimental results show basically linear speedup with an increasing number of processors.

**1 Introduction**

Frequency scaling on silicon—the ability to drive chips at ever higher clock rates—is beginning to hit a power limit as device geometries shrink due to leakage, and simply because CMOS consumes power every time it changes state [9, 10]. Yet Moore’s law [20], the density of circuits doubling every generation, is projected to last between 10 and 20 more years for silicon based circuits [10].
Example: Gradient

\[(x, y)_1\]
\[(x, y)_2\]
\[(x, y)_3\]
\[(x, y)_4\]
Example: Gradient

ComputeGradient

\[(x, y)_1 \rightarrow g_1\]
\[(x, y)_2 \rightarrow g_2\]
\[(x, y)_3 \rightarrow g_3\]
\[(x, y)_4 \rightarrow g_4\]
Example: Gradient

\[(x, y)\]

\[g_1\]

\[g_2\]

\[g_3\]

\[g_4\]

\[\sum\]

\[g\]
Example: Gradient

ComputeGradient

\[ g = \sum g \]
Example: Gradient

Map:

\((x, y)_1\) \(\rightarrow\) \(g_1\)
\((x, y)_2\) \(\rightarrow\) \(g_2\)
\((x, y)_3\) \(\rightarrow\) \(g_3\)
\((x, y)_4\) \(\rightarrow\) \(g_4\)

Reduce:

\(\sum\)

\(g\)
• Machine Learning Library
• Implementations of many algorithms, both on Hadoop MapReduce and stand-alone
• Open Source (Apache)
• Welcoming, helpful community

http://mahout.apache.org
• **Recommender Systems**, e.g.
  • User and Item based recommenders
  • Collaborative Filtering
• Clustering (K-Means, Mean Shift, …)
• Topic Models (LDA)
• Supervised ML
  • (Logistic) Regression
  • Linear SVMs
• Decision Trees and Forests
Efficient Noise-Tolerant Learning from Statistical Queries

MICHAEL KEARNS

Abstract. In this paper, we study the problem of learning in the presence of classification noise in the probabilistic learning model of Valiant and its variants. In order to identify the class of “robust” learning algorithms in the most general way, we formalize a new but related model of learning from statistical queries. Intuitively, in this model, a learning algorithm is forbidden to examine individual examples of the unknown target function, but is given access to an oracle providing estimates of probabilities over the sample space of random examples.

One of our main results shows that any class of functions learnable from statistical queries is in fact learnable with classification noise in Valiant’s model, with a noise rate approaching the information-theoretic barrier of 1/2. We then demonstrate the generality of the statistical query model: showing that practically every class learnable in Valiant’s model and its variants can also be learned in the new model (and thus can be learned in the presence of noise). A notable exception to this statement is the class of parity functions, which we prove is not learnable from statistical queries, and for which no noise-tolerant algorithm is known.

Categories and Subject Descriptors: F [Theory of Computing]: G.3 [Probability and Statistics]; I.2 [Artificial Intelligence]; I.5 [Pattern Recognition]

General Terms: Computational learning theory, Machine learning

Additional Key Words and Phases: Computational learning theory, machine learning

1. Introduction

In this paper, we study the extension of Valiant’s model of learnability (Valiant 1984) in which the positive or negative classification of an example may be corrupted by random noise. This extension was first examined in Laird (1988), who formalized the noise-tolerant algorithms to analog the being the subject of a number of theoretical studies (Angluin and Laird 1988; Sloan 1989; Kearns and Li 1993), the classification noise model has become a common paradigm for many research in machine learning.
Efficient Noise-Tolerant Learning from Statistical Queries

MICHAEL KEARNS
AT&T Laboratories—Research, Florham Park, New Jersey

Abstract. In this paper, we study the problem of learning in the presence of classification noise in the probabilistic learning model of Valiant and its variants. In order to identify the class of “robust” learning algorithms in the most general way, we formalize a new but related model of learning from statistical queries. Intuitively, in this model, a learning algorithm is forbidden to examine individual examples of the unknown target function, but is given access to an oracle providing estimates of probabilities over the sample space of random examples.

One of our main results shows that any class of functions learnable from statistical queries is in fact learnable with classification noise in Valiant’s model, with a noise rate approaching the information-theoretic barrier of 1/2. We then demonstrate the generality of the statistical query model, showing that practically every class learnable in Valiant’s model and its variants can also be learned in the new model (and thus can be learned in the presence of noise). A notable exception to this statement is the class of parity functions, which we prove is not learnable from statistical queries, and for which no noise-tolerant algorithm is known.

Categories and Subject Descriptors: F. [Theory of Computation]; G.3 [Probability and Statistics]; I.2. [Artificial Intelligence]; I.5 [Pattern Recognition]

General Terms: Computational learning theory, Machine learning

Additional Key Words and Phrases: Computational learning theory, machine learning

1. Introduction

In this paper, we study the extension of Valiant’s model [Valiant 1984] in which the positive or negative classification of an example may be corrupted by random noise. This is an area of study in the learning theory literature by Angluin and Laird [1988] in which the target function’s ground label is the object of highest prior probability, and then the target function can be learned by an algorithm for the highest probability rate of noise. In addition to this, there are other theoretical studies [Angluin and Laird 1988; Laird 1988; Sloan 1989; Kearns and Li 1993], the classification noise model has become a common paradigm for investigating the learning problems of designing (often ingenious) ways to speed up a learning algorithm at a single core, but there is still no good programming framework for handling many cores (processors), but there is still no good programming framework for handling many cores (processors), but there is still no good programming framework for handling many cores (processors), but there is still no good programming framework for handling many cores (processors).
Further Reading

Map-Reduce for Machine Learning on Multicore

Cheng-Tao Chu*, Sang Kyun Kim*, Yi-An Lin*, Yuan Yuan Yu*, Gary Bradski†, Andrew Y. Ng*
Kunle Olukotun*

Abstract

We are at the beginning of the multicore era. Computers will have increasingly many cores (processors), but there is still no good programming framework for these architectures, and thus no simple and unified way for machine learning to take advantage of the potential speed up. In this paper, we develop a broadly applicable parallel programming method, one that is distinct from the traditional learning algorithms. Our work is in distinct contrast to the tradition in machine learning of designing (often ingenious) ways to parallelize on multicore computers. We adapt Google’s map-reduce [7] parallelization technique to a variety of learning algorithms, including locally weighted linear regression (LWR), k-means, logistic regression (LR), naive Bayes (NB), SVM, ICA, PCA, gaussian discriminant analysis (GDA), EM, and backpropagation (NN). Our experimental results show basically linear speedup with an increasing number of processors.
Further Reading

Map-Reduce for Machine Learning on Multicore

Cheng-Tao Chu *  sangkyunkim * yi-an.lin
chengtao@stanford.edu  skkim38@stanford.edu  ian1@stanford.edu

Yuan Yuan Yu *  garybradski@gmail  andrew.y.ng
yuyan@stanford.edu  ang@cs.stanford.edu

Kunle Olukotun *
kunle@cs.stanford.edu

† CS. Department, Stanford University 353 Serra Mall, Stanford University, Stanford CA 94305-9025.
† RexEx Inc.

Abstract

We are at the beginning of the multicore era. Computers will have increasingly many cores (processors), but there is still no good programming framework for these architectures, and thus no simple and unified way for machine learning to take advantage of the potential speedup. In this paper, we develop a broadly applicable parallel programming method, one that can be applied to many different learning algorithms. Our work is in distinct contrast to the traditional learning of designing (often ingenious) ways to parallelize on multicore computers. We adapt Google’s Map-Reduce [7] technique to a variety of learning algorithms, including linear regression, k-means, logistic regression (LR), naive Bayes (NB), SVM, ICA, PCA, Gaussian discriminant analysis (GDA), EM, and backpropagation (NN). Our experimental results show basically linear speedup with an increasing number of processors.

Efficient Noise-Tolerant Learning from Statistical Queries

Introduction

1. Introduction

In this paper, we study the extension of Valiant’s learning model [18] to the simplest type of white label noise and then sought algorithms tolerating the example may be corrupted by random noise. This extension was first examined in [15] and extended in various directions in [12]. In this paper, we consider the efficient noise-tolerant learning problem, and show the results of our theoretical studies [Angluin and Laird 1988; Laird 1988; Sloan 1988; Kearns and Valiant 1989]. We demonstrate this parallel speed up technique on a variety of learning algorithms. Our work is in distinct contrast to the traditional learning of designing (often ingenious) ways to parallelize on multicore computers. We adapt Google’s map-reduce [7] technique to a variety of learning algorithms, including linear regression, k-means, logistic regression (LR), naive Bayes (NB), SVM, ICA, PCA, Gaussian discriminant analysis (GDA), EM, and backpropagation (NN). Our experimental results show basically linear speedup with an increasing number of processors.
Further Reading

Session 2: Modeling with Hadoop
Algorithms in MapReduce
Vijay K Narayanan
Principal Scientist, Yahoo! Labs, Yahoo!

Tutorial @ KDD 2011
http://www.slideshare.net/hadoop
Further Reading

Session 2: Modeling with Hadoop
Algorithms in MapReduce

Vijay K Narayanan
Principal Scientist, Yahoo! Labs, Yahoo!

Tutorial @ KDD 2011
http://www.slideshare.net/hadoop
1 Introduction

Map-Reduce for Machine Learning on Multicore
Cheng-Tao Chu
chuchengtao@stanford.edu
Sang Kyun Kim
skkim38@stanford.edu
Yi-An Lin
ianl@stanford.edu

Efficient Noise-Tolerant Learning from Statistical Queries
MICHAEL KEARNS
AT&T Laboratories—Research, Florham Park, New Jersey

In this paper, we study the extension of Valiant's learning model [Valiant 1984] to the presence of classification noise. One of our main results shows that any class of functions learnable from statistical queries is in fact noise-tolerant. Intuitively, in this model, an learning algorithm is not allowed to examine individual examples, but is given access to an oracle providing estimates of the unknown target function.

Further Reading
http://www.slideshare.net/hadoop
Further Reading
Further Reading
Trouble

- ML is iterative
- Each iteration is a Job
- Overhead per job (>45s)
  - Scheduling
  - Program Distribution
  - Data Loading and Parsing
  - State Transfer

Map-Reduce for Machine Learning on Multicore

Cheng-Tao Chu *  
chengtao@stanford.edu

Sang Kyun Kim *  
skkim38@stanford.edu

Yi-An Lin *  
ianl@stanford.edu

YuanYuan Yu *  
yuanyuan@stanford.edu

Gary Bradski †  
garybradski@gmail

Andrew Y. Ng *  
ang@cs.stanford.edu

Kunle Olukotun *  
kunle@cs.stanford.edu

* CS. Department, Stanford University 353 Serra Mall, Stanford University, Stanford CA 94305-9025.
† Rexee Inc.

Abstract

We are at the beginning of the multicore era. Computers will have increasingly many cores (processors), but there is still no good programming framework for these architectures, and thus no simple and unified way for machine learning to take advantage of the potential speed up. In this paper, we develop a broadly applicable parallel programming method, one that is easily applied to many different learning algorithms. Our work is in distinct contrast to the tradition in machine learning of designing (often ingenious) ways to speed up a single algorithm at a time. Specifically, we show that algorithms that fit the Statistical Query model [15] can be written in a certain “summation form,” which allows them to be easily parallelized on multicore computers. We adapt Google’s map-reduce [7] paradigm to demonstrate this parallel speed up technique on a variety of learning algorithms including locally weighted linear regression (LWLR), k-means, logistic regression (LR), naive Bayes (NB), SVM, ICA, PCA, gaussian discriminant analysis (GDA), EM, and backpropagation (NN). Our experimental results show basically linear speedup with an increasing number of processors.

1 Introduction

Frequency scaling on silicon—the ability to drive chips at ever higher clock rates—is beginning to hit a power limit as device geometries shrink due to leakage, and simply because CMOS consumes power every time it changes state [9, 10]. Yet Moore’s law [20], the density of circuits doubling every generation, is predicted to last between 10 and 20 more years for silicon based circuits [10].
Abstract

We are at the beginning of the multicore era. Computers will have increasingly many cores (processors), but there is still no good programming framework for these architectures, and thus no simple and unified way for machine learning to take advantage of the potential speed up. In this paper, we develop a broadly applicable parallel programming method, one that is easily applied to many different learning algorithms. Our work is in distinct contrast to the tradition in machine learning of designing (often ingenious) ways to speed up a single algorithm at a time. Specifically, we show that algorithms that fit the Statistical Query model [15] can be written in a certain “summation form,” which allows them to be easily parallelized on multicore computers. We adapt Google’s map-reduce [7] paradigm to demonstrate this parallel speed up technique on a variety of learning algorithms including locally weighted linear regression (LWLR), k-means, logistic regression (LR), naive Bayes (NB), SVM, ICA, PCA, gaussian discriminant analysis (GDA), EM, and backpropagation (NN). Our experimental results show basically linear speedup with an increasing number of processors.

1 Introduction

Frequency scaling on silicon—the ability to drive chips at ever higher clock rates—is beginning to hit a power limit as device geometries shrink due to leakage, and simply because CMOS consumes power every time it changes state [9, 10]. Yet Moore’s law [20], the density of circuits doubling every generation, is projected to last between 10 and 20 more years for silicon based circuits [10].
Beyond MapReduce today
Solutions

- Local (subsampling)
- MPI
- Spark
- Pregel
Subsampling

• Form examples on the cluster
• Subsample the data on the cluster
• Train a model on a single machine
Model Averaging

Per-Partition Training

Averaging
Model Averaging

Per-Partition Training → Averaging
Model Averaging

Per-Partition Training

Averaging
Model Averaging

Per-Partition Training

Averaging
Model Averaging

Per-Partition Training

Map

Reduce

Averaging
Message Passing Interface

- Mature HPC standard
- Supported on many clusters (e.g. OpenMPI)
- Available in C, Fortran and Scripting Languages
- Key operation here: AllReduce
AllReduce
AllReduce

... AllReduce()
AllReduce

... AllReduce()

... AllReduce()

... AllReduce()
AllReduce

... AllReduce()

... AllReduce()

... AllReduce()
AllReduce

... AllReduce()

... AllReduce()

... AllReduce()

... AllReduce()
AllReduce

... AllReduce()

... AllReduce()

... AllReduce()

... AllReduce()
AllReduce

Synchronization Barrier

... AllReduce()

... AllReduce()
AllReduce

... AllReduce()

... AllReduce()

... AllReduce()

... AllReduce()
State Persists Across Iterations
Use Hadoop for
- Data local scheduling
- Good machine identification

Use MPI for
- AllReduce

30x Speedup over Hadoop MapReduce

ABSTRACT
We present a system and a set of techniques for learning linear predictors with convex losses on terascale datasets, with trillions of features, billions of training examples, and millions of parameters in an hour using a cluster of 1000 machines. Individually none of the component techniques is new, but the careful synthesis required to obtain an efficient implementation is a novel contribution. The result is, up to our knowledge, the most scalable and efficient linear learning system reported in the literature. We describe and thoroughly evaluate the components of the system, showing the importance of the various design choices.

1. INTRODUCTION
Distributed machine learning is a research area that has seen a growing body of literature in recent years. Much work focuses on problems of the form

$$\min_{w \in \mathbb{R}^d} \sum_{i=1}^{n} f(x_i; y_i) + \lambda R(w),$$

where $x_i$ is the feature vector of the $i$-th example, $y_i$ is the label, $w$ is the linear predictor, $f$ is a loss function and $R$ a regularizer. Much of this work exploits the natural decomposability over examples in (1), partitioning the examples over different nodes in a distributed environment such as a cluster.

Perhaps the simplest learning strategy when the number of samples $n$ is very large is to subsample a smaller set of examples that can be tractably learned with. However, this strategy only works if the problem is simple enough or the number of parameters is very small. The setting of interest here is when a large number of samples is really needed to learn a good model, and distributed algorithms are a natural choice for such scenarios.

Some prior works (McDonald et al., 2010; Zinkevich et al., 2010) consider online learning with averaging and F et al. (2010a) propose gossip-style mesage passing algorithms extending the existing literature on distributed convex optimization (Bertsekas and Tsitsiklis, 1989). Lan et al. (2009) analyze a delayed version of distributed online learning. Dekel et al. (2010) consider mini-batch variants of online algorithms which are extended to delay-based up in Agarwal and Duchi (2011). A recent article of Boyd et al. (2011) describes an application of the ADMM technique to distributed learning problems. GraphLab (Low et al., 2010) is a parallel computation framework on graphs. More closely related to our work is that of Teo et al. (2007) who use a gossip-style randomized algorithm to parallelize a bundle method for optimization.

However, all of the aforementioned approaches leave something to be desired empirically when deployed on large clusters. In particular their throughput—measured as the input size divided by the wall clock running time—is extremely small. This is a factor of about a factor of the interface of a single machine.

An additional benefit of our system is its compatibility with MapReduce clusters such as Hadoop (unlike MPI-1 systems) and minimal additional programming effort to parallelize existing learning algorithms (unlike MapReduce approaches).

One of the key components in our system is a commodity infrastructure that efficiently accumulates and broadcasts values across all nodes of a computation. It is functionally similar to MPI AllReduce (hence we use the name). A recent paper of McDonald et al. (2010) shows a connection with Hadoop

http://hunch.net/~vw
MPI: Conclusion

- **The Good**
  - Computational Performance
  - Well established software available

- **The Bad**
  - No fault tolerance

- **The Ugly**
  - Ignorance of shared clusters
  - Systems-Level decisions at the algorithm layer
Spark: Intro

• Open Source cluster computation framework
• Developed at UC Berkeley by the AMP Lab
• Aimed at interactive and iterative use cases
• 30x faster than Hadoop for those
• User interface: Embedded Domain Specific Language in Scala

http://spark-project.org/
Spark: Example

val points = spark.textFile(...).
    map(parsePoint).
    partitionBy(HashPartitioner(NODES)).
    cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(computeGradient(_,w)).reduce(_ + _)
  w -= gradient
}
val points = spark.textFile(...).
    map(parsePoint).
    partitionBy(HashPartitioner(NODES)).
    cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(computeGradient(_,w)).reduce(_ + _)

  w -= gradient
}
val points = spark.textFile(...).
               map(parsePoint).
               partitionBy(HashPartitioner(NODES)).
               cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(computeGradient(_,w)).reduce(_ + _)

  w -= gradient
}
val points = spark.textFile(...).
    map(parsePoint).
    partitionBy(HashPartitioner(NODES)).
    cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
    val gradient = points.map(computeGradient(_,w)).reduce(_ + _)
    w -= gradient
}
val points = spark.textFile(...).
            map(parsePoint).
            partitionBy(HashPartitioner(NODES)).
            cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(computeGradient(_,w)).reduce(_ + _)

  w -= gradient
}

Spark: Example
Spark: Example

```scala
val points = spark.textFile(...).
           map(parsePoint).
           partitionBy(HashPartitioner(NODES)).
           cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(computeGradient(_, w)).reduce(_ + _)

  w -= gradient
}
```

- Computes a gradient per data point
- Spark: Example
val points = spark.textFile(...).
    map(parsePoint).
    partitionBy(HashPartitioner(NODES)).
    cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(computeGradient(_,w)).reduce(_ + _)
  w -= gradient
}

Spark: Example

Computes a gradient per data point

Sums them up
val points = spark.textFile(...).
    map(parsePoint).
    partitionBy(HashPartitioner(NODES)).
    cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
    val gradient = points.map(computeGradient(_,w)).reduce(_ + _)

    w -= gradient
}
val points = spark.textFile(...).
             map(parsePoint).
             partitionBy(HashPartitioner(NODES)).
             cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(computeGradient(_,w)).reduce(_ + _)

  w -= gradient
}
val points = spark.textFile(...).
               map(parsePoint).
               partitionBy(HashPartitioner(NODES)).
               cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(computeGradient(_,w)).reduce(_ + _)

  w -= gradient
}

Trouble! (physical layer shows through)
Spark: Conclusion

• **The Good**
  • Speed (ca. MPI speed)
  • Fault Tolerance
  • Ease of Programming

• **The Bad**
  • Main Memory Assumption

• **The Ugly**
  • Systems aspects creep up
Graph Computation framework

Developed by Google

Per vertex function update() processes incoming messages and sends new ones

Computation is Bulk Synchronous Parallel

Synchronous Parallel Computation is Bulk sends new ones incoming messages and update().

Graph Computation

Per vertex function update() processes incoming messages and sends new ones

Computation is Bulk Synchronous Parallel

Pregel: A System for Large-Scale Graph Processing

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski

Google, Inc.

{malewicz,austern,ajcbik,dehnert,ilan,naty,gczaj}@google.com

Abstract

Practical computing problems concern large graphs. Standard examples include the Web graph and various social networks. The scale of these graphs—in some cases billions of vertices, trillions of edges—poses challenges to their efficient processing. In this paper we present a computational model suitable for this task. Programs are expressed as a sequence of iterations, in each of which a vertex can process messages sent in the previous iteration, send messages to other vertices, and modify its own state and that of its outgoing edges or mutate graph topology. This vertex-centric approach is flexible enough to express a broad set of algorithms and easy to program. It is a framework for processing large graphs that is expressive and fault tolerant implementation on clusters of thousands of commodity computers and its implied synchronous model is designed for efficient, scalable fault-tolerant implementation on clusters of thousands of commodity computers. The model has been designed for efficient, scalable fault-tolerant implementation on clusters of thousands of commodity computers, and its implied synchronous model is designed for efficient, scalable fault-tolerant implementation on clusters of thousands of commodity computers. The Internet made the Web graph a popular object of scientific work—have been processed for decades. Pregel, an existing parallel graph system, has similar flavor of clustering, and variations on the page rank themes. There are many other graph computing systems, e.g., minimum cut and connected components.

Efficient processing of large graphs is challenging. Algorithms often exhibit poor locality of memory access, little work per vertex, and a changing degree of parallelism over the course of execution. Distribution over many machines exacerbates the locality issue, and increases the probability that a machine will fail during computation. Despite the ubiquity of large graphs and their commercial importance, we know of no scalable general-purpose systems for implementing arbitrary graph algorithms over arbitrary graph representations in a large-scale distributed environment.

Implementing an algorithm to process a large graph means choosing among the following options:

1. Crafting a custom distributed infrastructure, requiring a substantial implementation effort that must be repeated for each new algorithm or graph representation.
2. Relying on an existing distributed computing platform, for example, MapReduce, which is well suited for a wide array of scale computing problems. It is sometimes used for small large graphs, but this can lead to optimal performance and usability issues. The models for processing data have been extended to facilitate aggregation and SQL-like queries but these extensions are usually not ideal for graph algorithms that often better fit a message passing model.
3. Using a single-computer graph algorithm library such as BGL, LEDA, NetworkX, JDSI, Stanford GraphBase, or FGL, limited scale of problems that can be addressed.
4. Using an existing parallel graph system. The Pi-Bot, Pregel, and PGF/Boost libraries, among others, extend the Pregel model.

Categories and Subject Descriptors

D.2.13 [Software Engineering]: Reusable Software—Reusable libraries

General Terms

Programming Techniques: Concurrent Programming—Distributed programming. Reusable Software—Reusable libraries

Keywords

Distributed computing, graph algorithms

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to use Digital Library distribution system for reproduction and archiving purposes, and in combination with other materials, is granted.

Copyright 2010 ACM 978-1-4503-0032-2/10/06...

ACM Digital Library, last accessed [date] Copyright 2010 ACM 978-1-4503-0032-2/10/06...

SIGMOD’10, June 6–11, 2010, Indianapolis, Indiana, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to use Digital Library distribution system for reproduction and archiving purposes, and in combination with other materials, is granted.

Copyright 2010 ACM 978-1-4503-0032-2/10/06...

SIGMOD’10, June 6–11, 2010, Indianapolis, Indiana, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to use Digital Library distribution system for reproduction and archiving purposes, and in combination with other materials, is granted.

Copyright 2010 ACM 978-1-4503-0032-2/10/06...

SIGMOD’10, June 6–11, 2010, Indianapolis, Indiana, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to use Digital Library distribution system for reproduction and archiving purposes, and in combination with other materials, is granted.

Copyright 2010 ACM 978-1-4503-0032-2/10/06...

SIGMOD’10, June 6–11, 2010, Indianapolis, Indiana, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to use Digital Library distribution system for reproduction and archiving purposes, and in combination with other materials, is granted.

Copyright 2010 ACM 978-1-4503-0032-2/10/06...

SIGMOD’10, June 6–11, 2010, Indianapolis, Indiana, USA.
Giraph

- Apache Open Source implementation of Pregel
- Runs on Hadoop, (ab)uses mappers to do so
- Used at LinkedIn and Facebook

http://incubator.apache.org/giraph/
Pregel Visually

t=0
Pregel Visually

Messages Arrive and Are Processed

t=0
Pregel Visually

Messages Arrive and Are Processed

t=0
Pregel Visually

Barrier

t=0
t=1
Pregel Visually

t=0   t=1
Pregel Visually

Messages are being sent
Pregel Visually

t=0  t=1
Pregel Visually

t=0  t=1  t=2
Pregel Visually

Termination: No more messages
Pregel Visually
• `update()` receives the PageRank of all neighbors
• Updates its local PageRank
• Sends new PageRank around if it changed enough
Pregel: Conclusion

• **The Good**
  • Excellent Map for Graph problems
  • Fast

• **The Bad**
  • Memory Model
  • Main Memory Assumption

• **The Ugly**
  • Wrong computational model (stay for the afternoon)
Open Problems

- No complete isolation of user / systems code
  - Unlike MapReduce

- No one system for example formation and modeling
  - Learning Effort
  - Orchestration
  - Wasted resources in distributed clusters
A Declarative Approach
Joint Work With

Yingyi Bu, Vinayak Borkar, Michael J. Carey
University of California, Irvine

Joshua Rosen, Neoklis Polyzotis
University of California, Santa Cruz
Goals

- Unify Example Formation and Modeling
  - Relational Algebra Operators
  - Iteration Support
  - A unified runtime

- Increase Productivity via high-level language
  - Insulate the user from the systems aspects
  - Debugging and IDE support
Approach
Approach

ScalOps

High Level Language
Relational Algebra and Loops
Approach

ScalOps

High Level Language
Relational Algebra and Loops

Datalog

Declarative Language
Captures the Recursive Dataflow
**Approach**

ScalOps

- High Level Language
  - Relational Algebra and Loops

Datalog

- Declarative Language
  - Captures the Recursive Dataflow

Recursive Dataflow

- Suite of data-parallel operators
  - Selected by an Optimizer / Compiler
**Approach**

- **ScalOps**
  - High Level Language
  - Relational Algebra and Loops

- **Datalog**
  - Declarative Language
  - Captures the Recursive Dataflow

- **Recursive Dataflow**
  - Suite of data-parallel operators
  - Selected by an Optimizer / Compiler

- **Hyracks Dataflow**
  - Unified Runtime
  - Scalability + High performance
Approach

ScalOps

High Level Language
Relational Algebra and Loops

Datalog

Declarative Language
Captures the Recursive Dataflow

Recursive Dataflow

Suite of data-parallel operators
Selected by an Optimizer / Compiler

Hyracks Dataflow

Unified Runtime
Scalability + High performance
Approach

ScalOps

High Level Language
Relational Algebra and Loops

Datalog

Declarative Language
Captures the Recursive Dataflow

Recursive Dataflow

Suite of data-parallel operators
Selected by an Optimizer / Compiler

Hyracks Dataflow

Unified Runtime
Scalability + High performance
ScalOps

- Internal Domain Specific Language (DSL)
  - Written in Scala
- Relational Algebra (Filter, Join, GroupBy, ...)
- Iteration support
- Rich UDF support
  - Inline Scala function calls / literals
  - Byte-code compatible with Java
- Support in major IDEs
def train(xy: Table[Example],
    compute_grad:(Example, Vector) => Vector,
    compute_loss:(Example, Vector) => Double) = {

  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment

  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) { env => {
    val gradient = xy.map(x=>compute_grad(x,env.w)).reduce(_+_)
    val loss = xy.map(x=>compute_loss(x,env.w)).reduce(_+_)
    env.w = env.w - gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}
}
def train(xy: Table[Example],
          compute_grad:(Example, Vector) => Vector,
          compute_loss:(Example, Vector) => Double) = {

  class Env(w:VectorType, lastError:DoubleType, delta:DoubleType) extends Environment

  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x=>compute_grad(x,env.w)).reduce(_+_
    val loss = xy.map(x=>compute_loss(x,env.w)).reduce(_+_
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}
def train(xy: Table[Example],
            compute_grad:(Example, Vector) => Vector,
            compute_loss:(Example, Vector) => Double) = {
  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment
  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)
  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
    val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}

Table is our Dataset type
class Example(x:Vector, y:Double)

def train(xy:Table[Example],
    compute_grad:(Example, Vector) => Vector,
    compute_loss:(Example, Vector) => Double) = {
    class Env(w:VectorType, lastError:DoubleType, delta:DoubleType) extends Environment
    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    loop(initialValue, (env: Env) => env.delta < eps) { env =>
        val gradient = xy.map(x=>compute_grad(x,env.w)).reduce(_+_)
        val loss = xy.map(x=>compute_loss(x,env.w)).reduce(_+_)
        env.w -= gradient
        env.delta = env.lastLoss - loss
        env.lastLoss = loss
        env
    }
}
def train(xy: Table[Example],
    compute_grad:(Example, Vector) => Vector,
    compute_loss:(Example, Vector) => Double) = {

    class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment

    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    loop(initialValue, (env: Env) => env.delta < eps) { env => {
        val gradient = xy.map(x=>compute_grad(x, env.w)).reduce(_+_)
        val loss = xy.map(x=>compute_loss(x, env.w)).reduce(_+_)
        env.w = env.w - gradient
        env.delta = env.lastLoss - loss
        env.lastLoss = loss
        env
    }
}
}
def train(xy: Table[Example],
        compute_grad: (Example, Vector) => Vector,
        compute_loss: (Example, Vector) => Double) = {

class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment

val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
    val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
}
}
def train(xy: Table[Example],
        compute_grad:(Example, Vector) => Vector,
        compute_loss:(Example, Vector) => Double) = {

  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment

  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
    val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}

def train(xy: Table[Example],
    compute_grad: (Example, Vector) => Vector,
    compute_loss: (Example, Vector) => Double) = {

  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment {

    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    loop(initialValue, (env: Env) => env.delta < eps) { env =>
      val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
      val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
      env.w -= gradient
      env.delta = env.lastLoss - loss
      env.lastLoss = loss
      env
    }
  }
}
def train(xy: Table[Example],
compute_grad: (Example, Vector) => Vector,
compute_loss: (Example, Vector) => Double) = {
  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment
  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)
  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
    val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}
def train(xy: Table[Example],
compute_grad:(Example, Vector) => Vector,
compute_loss:(Example, Vector) => Double) = {
  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment {
    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    loop(initialValue, (env: Env) => env.delta < eps) { env => {
      val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
      val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
      env.w -= gradient
      env.delta = env.lastLoss - loss
      env.lastLoss = loss
      env
    }
  }
}
def train(xy: Table[Example],
compute_grad: (Example, Vector) => Vector,
compute_loss: (Example, Vector) => Double) = {
  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment {
    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)
    loop(initialValue, (env: Env) => env.delta < eps) { env =>
      val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
      val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
      env.w -= gradient
      env.delta = env.lastLoss - loss
      env.lastLoss = loss
      env
    }
  }
}
def train(xy: Table[Example],
    compute_grad:(Example, Vector) => Vector,
    compute_loss:(Example, Vector) => Double) = {

  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment

  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
    val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}
def train(xy: Table[Example],
    compute_grad: (Example, Vector) => Vector,
    compute_loss: (Example, Vector) => Double) = {

    class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment
    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    loop(initialValue, (env: Env) => env.delta < eps) { env =>
        val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
        val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
        env.w -= gradient
        env.delta = env.lastLoss - loss
        env.lastLoss = loss
        env
    }
}
def train(xy: Table[Example],
    compute_grad: (Example, Vector) => Vector,
    compute_loss: (Example, Vector) => Double) = {
  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment {

  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
    val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}
}
def train(xy: Table[Example],
    compute_grad: (Example, Vector) => Vector,
    compute_loss: (Example, Vector) => Double) = {

  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment

  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
    val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}
def train(xy: Table[Example],
            compute_grad: (Example, Vector) => Vector,
            compute_loss: (Example, Vector) => Double) = {

    class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment{
        val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

        loop(initialValue, (env: Env) => env.delta < eps) { env =>
            val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_+_)
            val loss = xy.map(x => compute_loss(x, env.w)).reduce(_+_)

            env.w -= gradient
            env.delta = env.lastLoss - loss
            env.lastLoss = loss

            env
        }
    }
}
def train(xy: Table[Example],
compute_grad:(Example, Vector) => Vector,
compute_loss:(Example, Vector) => Double) = {

  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment

  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
    val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}

def train(xy: Table[Example],
    compute_grad: (Example, Vector) => Vector,
    compute_loss: (Example, Vector) => Double) = {

    class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment

    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    loop(initialValue, (env: Env) => env.delta < eps) { env =>
        val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_+_)
        val loss = xy.map(x => compute_loss(x, env.w)).reduce(_+_)
        env.w -= gradient
        env.delta = env.lastLoss - loss
        env.lastLoss = loss
        env
    }
}
def train(xy: Table[Example],
compute_grad:(Example, Vector) => Vector,
compute_loss:(Example, Vector) => Double) = {

  class Env (w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment {
    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    def loop(initialValue, (env: Env) => env.delta < eps) {
      env => {
        val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
        val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
        env.w -= gradient
        env.delta = env.lastLoss - loss
        env.lastLoss = loss
        env
      }
    }
  }

  loop(initialValue, (env: Env) => env.delta < eps) {
    env => {
      val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
      val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
      env.w -= gradient
      env.delta = env.lastLoss - loss
      env.lastLoss = loss
      env
    }
  }
}
def train(xy: Table[Example],
          compute_grad: (Example, Vector) => Vector,
          compute_loss: (Example, Vector) => Double) = {
  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment
  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_+_)
    val loss = xy.map(x => compute_loss(x, env.w)).reduce(_+_)
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}
```scala
def train(xy: Table[Example],
compute_grad: (Example, Vector) => Vector,
compute_loss: (Example, Vector) => Double) = {
  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment {
    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)
    loop(initialValue, (env: Env) => env.delta < eps) {
      val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
      val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
      env.w -= gradient
      env.delta = env.lastLoss - loss
      env.lastLoss = loss
      env
    }
  }
}
```
Approach

ScalOps

High Level Language
Relational Algebra and Loops

Datalog

Declarative Language
Captures the Recursive Dataflow

Recursive Dataflow

Suite of data-parallel operators
Selected by an Optimizer / Compiler

Hyracks Dataflow

Unified Runtime
Scalability + High performance
Approach

High Level Language
Relational Algebra and Loops

Declarative Language
Captures the Recursive Dataflow

Suite of data-parallel operators
Selected by an Optimizer / Compiler

Unified Runtime
Scalability + High performance

ScalOps

Datalog

Recursive Dataflow

Hyracks Dataflow
def train(xy: Table[Example],
    compute_grad:(Example, Vector) => Vector,
    compute_loss:(Example, Vector) => Double) = {

  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment

  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) { env => {
    val gradient = xy.map(x=>compute_grad(x,env.w)).reduce(_.+)
    val loss = xy.map(x=>compute_loss(x,env.w)).reduce(_.+)
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}
}
def train(xy: Table[Example],
          compute_grad: (Example, Vector) => Vector,
          compute_loss: (Example, Vector) => Double) = {

  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment

  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
    val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}
def train(xy: Table[Example],
          compute_grad: (Example, Vector) => Vector,
          compute_loss: (Example, Vector) => Double) = {
  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment
  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, { env =>
    val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
    val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)

    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  })
}
def train(xy: Table[Example],
    compute_grad: (Example, Vector) => Vector,
    compute_loss: (Example, Vector) => Double) = {

    class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment

    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    loop(initialValue, (env: Env) => env.delta < eps) { env =>
        val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
        val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
        env.w -= gradient
        env.delta = env.lastLoss - loss
        env.lastLoss = loss
        env
    }
}
def train(xy: Table[Example],
    compute_grad: (Example, Vector) => Vector,
    compute_loss: (Example, Vector) => Double) = {

  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment {
  
  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) { env => {

    val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
    val loss     = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)

    env.w       -= gradient
    env.delta   = env.lastLoss - loss
    env.lastLoss = loss

    env
  }
}
}
Logical Plan

MapReduce

Map()  Reduce()

Model

Continue()

Loop

(Model, Performance)

Update()

Sequential

Aggregate Statistics

Training Data
**Approach**

- **ScalOps**
  - High Level Language
  - Relational Algebra and Loops

- **Datalog**
  - Declarative Language
  - Captures the Recursive Dataflow

- **Recursive Dataflow**
  - Suite of data-parallel operators
  - Selected by an Optimizer / Compiler

- **Hyracks Dataflow**
  - Unified Runtime
  - Scalability + High performance
**Approach**

- **ScalOps**
  - High Level Language
    - Relational Algebra and Loops
  - Declarative Language
    - Captures the Recursive Dataflow
- **Datalog**
  - Suite of data-parallel operators
  - Selected by an Optimizer / Compiler
- **Recursive Dataflow**
- **Hyracks Dataflow**
  - Unified Runtime
    - Scalability + High performance
Some Optimizations

• Caching, Rocking

• Scheduling: Data-Local, Iteration-Aware

• Avoid (de-)serialization

• Minimize #network connections

• Pipelining
Physical Plan

Data Loading

- HDFS

Cached Records

Iterative Computation

- HDFS

model

(map)

CR

Iteration Barrier

(model)

Aggregation tree (reduce)

Sequential (update)

HDFS

Driver (loop)
Physical Plan

Data Loading

HDFS → CR

Cached Records

Iterative Computation

HDFS

(model)

CR

(map)

Aggregation tree
(reduce)

Sequential (update)

HDFS

Driver (loop)

How Many?
Physical Plan

Data Loading

HDFS → Cached Records

Iterative Computation

HDFS → model

(map) → Aggregation tree

(reduce) → Sequential (update)

Driver (loop) → HDFS

How Many?

Structure?
Fan-In
Fan-In
Fan-In: Blocking
Fan-In: Blocking

\[ h = \log_f(N) = \frac{\ln(N)}{\ln(f)} \]
Fan-In: Time per Level

\[ h = \log_f(N) = \frac{\ln(N)}{\ln(f)} \]
Fan-In: Time per Level

\[ t = fA \]

\[ h = \log_f(N) = \frac{\ln(N)}{\ln(f)} \]
$$t = fA$$

$$h = \log_f(N) = \frac{\ln(N)}{\ln(f)}$$
Fan-In: Total Time

\[ t = f A \]

\[ h = \log_f(N) = \frac{\ln(N)}{\ln(f)} \]

\[ t = h \times t \]

\[ = \frac{f}{\ln(f)} \ln(N) \times A \]
Fan-In: Total Time

\[ t = f A \]

\[ h \cdot t = \frac{f}{\ln(f)} \ln(N) \cdot A \]

Minimized for \( \hat{f} = e \)
Partitioning

- Aggregation time increases logarithmically with number of machines
- Map time decreases linearly with the number of machines
- Closed form solutions available (but omitted here)
Evaluation

• As fast as
  • Vowpal Wabbit
  • Spark

• Faster than Hadoop (doh!)

• Much, much less code
Evaluation

- Iteration time (seconds)
  - 20 CPUs: 50 seconds
  - 24 CPUs: 100 seconds
  - 40 CPUs: 150 seconds
  - 60 CPUs: 200 seconds
  - 80 CPUs: 300 seconds
  - 120 CPUs: 500 seconds

- Iteration Cost (CPU-seconds)
  - 20 CPUs: 3,000
  - 24 CPUs: 4,000
  - 40 CPUs: 5,000
  - 60 CPUs: 6,000
Evaluation

Optimizer: Cheapest
Evaluation

Optimizer: Cheapest

Optimizer: Fastest

Iteration Cost (CPU-seconds)

Iteration time (seconds)

CPUs

20 24 40 60 80 120

3,000 4,000 5,000

50 100
Summary

• **Example Formation**
  • Use Pig

• **Modeling**
  • Hadoop (maybe not)
  • Subsampling (now)
  • Spark / Pregel (now)
  • ScalOps (as soon as we are done)