Parallelizing ListNet Training using Spark

Shilpa Shukla
School of Information
University of Texas at Austin
shilpa@ischool.utexas.edu

Matthew Lease
School of Information
University of Texas at Austin
ml@ischool.utexas.edu

Ambuj Tewari
Dept. of Computer Science
University of Texas at Austin
ambuj@cs.utexas.edu

ABSTRACT

As ever-larger training sets for *learning to rank* are created, scalability of learning has become increasingly important to achieving continuing improvements in ranking accuracy [2]. Exploiting independence of "summation form" computations [3], we show how each iteration in ListNet [1] gradient descent can benefit from parallel execution. We seek to draw the attention of the IR community to use *Spark* [7], a newly introduced distributed cluster computing system, for reducing training time of iterative learning to rank algorithms. Unlike MapReduce [4], Spark is especially suited for iterative and interactive algorithms. Our results show near linear reduction in ListNet training time using Spark on Amazon EC2 clusters.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]

Keywords

Learning to Rank, Distributed Computing

1. INTRODUCTION

A potentially easy way to improve learning to rank accuracy is to simply train on more data. The challenge, of course, is scalability [2]. Prior work showed that many machine learning algorithms, such as those based on a "summation form", can be easily parallelized [3]. While summation terms can be independently computed via MapReduce, MapReduce is not well-suited to computing iterative learning algorithms and relatively inefficient for such usage.

Instead, we adopt the recently developed *Spark*[1] cluster computing system [6]. Spark is not only well-suited to such iterative (and interactive) algorithms such as gradient descent, but it runs on existing Hadoop[2] clusters. Spark supports reuse of a working set of data across multiple parallel operations via a distributed memory abstraction called Resilient Distributed Datasets (RDDs), which support parallel, in-memory computations on large clusters while retaining similar fault tolerance as MapReduce to reconstruct a lost partition whenever faults occur. As with Hadoop, the same Spark program can be run *stand-alone* on a single node or distributed on a compute cluster.

ListNet [1] uses gradient descent to minimize the loss between gold relevance scores and predicted scores. Scores can then be sorted to produce document ranking. Suppose there are m queries in the training set. Denoting gold relevance scores and predicted scores on query i by $y^{(i)}$ and $z^{(i)}$ respectively, the total loss is given in "summation form" by:

$$\text{TotalLoss}(w) = \sum_{i=1}^{m} L(y^{(i)}, z^{(i)}) \tag{1}$$

We sum loss over queries in the dataset. ListNet attempts to minimize this loss using gradient descent:

$$w \leftarrow w - \alpha \nabla_w \text{TotalLoss}(w) \tag{2}$$

where α is a *step-size parameter* and $\nabla_w \text{TotalLoss}(w)$ is the gradient of the loss (1) evaluated at w. Since we have:

$$\nabla_w \text{TotalLoss}(w) = \sum_{i=1}^{m} \nabla_w L(y^{(i)}, z^{(i)}) \ . \tag{3}$$

the gradient is in "summation form" and its computation can be parallelized. Optimization of w iteratively alternates between (2) and (3). We are not aware of any existing work that parallelizes ListNet training on commodity clusters. There is some recent work [5] on parallelizing learning to rank for information retrieval but it proposed a new algorithm based on evolutionary computation.

Figure 1 shows our implementation of "vanilla" gradient descent for ListNet using Spark. Using `SparkContext`, we create an RDD by reading in the training set from a file in Amazon S3. This RDD is parsed to populate a datastructure using the `map` transformation, creating a list such that each element in the list consists of a query, documents to be ranked, gold relevance labels, and feature vectors. We then call a `cache` function to advise Spark to keep the RDD in memory of the worker nodes to improve performance. Further, we use a shared variable `gradient` which is a Spark *accumulator*, similar to Hadoop counters. The `gradient` accumulator is used to sum up the contributions to the gradient calculated by all the worker nodes for computing the weight vector in the driver code.

EVALUATION. We use the Microsoft Learning to Rank datasets[3] MSLR-WEB10K and MSLR-WEB30K. There is a trade-off involved in choosing the step size α. If it is too small, the convergence will be slow. If it is too large, we may not have convergence. We tried α values in the

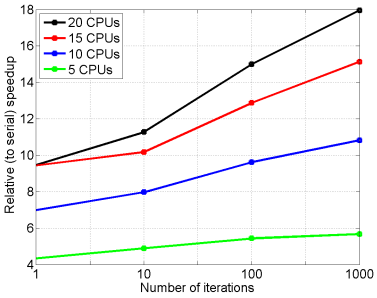
[1] `http://www.spark-project.org`
[2] `http://hadoop.apache.org`

[3] `http://research.microsoft.com/en-us/projects/mslr/`

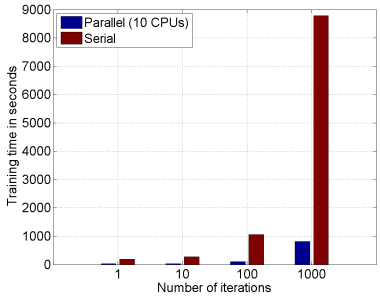
SIGIR'12, August 12–16, 2012, Portland, Oregon, USA.
ACM 978-1-4503-1472-5/12/08.

```
for (i <- 1 to ITERATIONS) {
            val gradient = sparkContext.accumulator(spark.examples.Vector.zeros(dim))
            val loss = sparkContext.accumulator(0.0)
            for (q <- queries) {
                val expRelScores = q.relScores.map(y => math.exp(beta*y.toDouble))
                val ourScores = q.docFeatures.map(x => w dot x); val expOurScores = ourScores.map(z => math.exp(z))
                val sumExpRelScores = expRelScores.reduce(_ + _); val sumExpOurScores = expOurScores.reduce(_ + _)

                val P_y = expRelScores.map(y => y/sumExpRelScores); val P_z = expOurScores.map(z => z/sumExpOurScores)

                var lossForAQuery = 0.0; var gradientForAQuery = spark.examples.Vector.zeros(dim)
                for (j <- 0 to q.relScores.length-1) {
                    gradientForAQuery += (q.docFeatures(j) * (P_z(j) - P_y(j)))
                    lossForAQuery += -P_y(j) * math.log(P_z(j))
                }
                gradient += gradientForAQuery; loss += lossForAQuery
            }
            w -= gradient.value * stepSize
        }
```

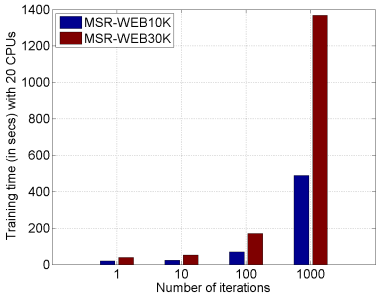
Figure 1: Parallel implementation of ListNet algorithm in Scala (`www.scala-lang.org`) using Spark. The code `for(q <- queries){body}` is equivalent to `queries.foreach(q => {body})`. As `foreach` is a parallel operation in Spark, all queries will be processed in parallel. The variables `gradient` and `loss` are *accumulators* that support only "add" operations and are used to sum values obtained by various worker nodes.



(a) Relative speedups on the 10K training set as more CPUs are used.



(b) Training times for MSLR-WEB10K for serial (1 CPU) vs. parallel (10 CPUs).



(c) Run times on 10K and 30K training sets with 20 CPUs.

Figure 2: Performance improvement in ListNet training time enabled by parallelization.

range 10^{-4} to 10^{-2} and chose the largest one that did not lead to divergence. Standard MSLR dataset partitioning of queries facilitates computation of NDCG accuracy using five-fold cross-validation. Average NDCG@10 ranking accuracy achieved over all five test folds of MSLR-WEB10K is 0.252; greater NDCG accuracy might be achieved by using more sophisticated line search procedures in the gradient descent algorithm shown in Figure 1.

Figure 2(a) shows that as the number of iterations increases, the overhead of using Spark reduces, achieving linear speedup in training time. Figure 2(b) shows that with a parallelism of 10 we achieve roughly 11x speedup on the MSLR-WEB10K dataset as compared to serial implementation. When the degree of parallelism is 1 and the cache is insufficient, some recomputations might become inevitable leading to slightly super-linear speedup. Figure 2(c) shows that training times for MSLR-WEB30K, which has 3x as many queries as MSLR-WEB10K, is actually less than 3x the corresponding times for MSLR-WEB10K, showing greater benefit from parallelization as the training size increases.

2. REFERENCES

[1] Cao, Z., Qin, T., Liu T.-Y., Tsai M.-F., and Li, H.: Learning to Rank: From Pairwise Approach to Listwise Approach. In: ICML '07, (2007).

[2] Chapelle, O., Chang, Y., and Liu, T.-Y.: Future Directions in Learning to Rank. In: JMLR Workshop and Conference Proceedings 14, pp. 91-100, (2011).

[3] Chu, C.T., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G., Ng, A., and Olukotun, K.: Map-reduce for machine learning on multicore. In: NIPS '06, (2007).

[4] S. Ghemawat and J. Dean.: MapReduce: Simplified data processing on large clusters. In: OSDI '04, pages 137–149 (2004).

[5] Wang, S., Gao, B. J. and Wang, K. and Lauw, H. W.: Parallel learning to rank for information retrieval. In: SIGIR '11, (2011).

[6] Zaharia M., Chowdhury, M., Das, T., Dave, A., Ma J., McCauley, M., Franklin, M., Shenker S. and Stoica, I.: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In: NSDI '12 (2012).

[7] Zaharia M., Chowdhury, M., Franklin, M. J., Shenker S. and Stoica, I.: Spark: Cluster Computing with Working Sets. In: HotCloud '10, (2010).