Parallel Learning to Rank for Information Retrieval

Shuaiqiang Wang*
Shandong University of Finance, Jinan, China
wangsq@sdfi.edu.cn

Byron J. Gao
Texas State University, San Marcos, TX, USA
bgao@txstate.edu

Ke Wang
Simon Fraser University, Burnaby, BC, Canada
wangk@cs.sfu.ca

Hady W. Lauw
Institute for Infocomm Research, Singapore
hwlauw@i2r.a-star.edu.sg

ABSTRACT

Learning to rank represents a category of effective ranking methods for information retrieval. While the primary concern of existing research has been accuracy, learning efficiency is becoming an important issue due to the unprecedented availability of large-scale training data and the need for continuous update of ranking functions. In this paper, we investigate parallel learning to rank, targeting simultaneous improvement in accuracy and efficiency.

Categories and Subject Descriptors: I.2.6 [Artificial Intelligence]: Learning; H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval

General Terms: Algorithms, Performance.

Keywords: Learning to rank, Parallel algorithms, Cooperative coevolution, MapReduce, Information retrieval.

1. INTRODUCTION

Learning to rank represents a category of effective ranking methods for information retrieval (IR) systems. Given training data, in the form of a set of queries each associated with a list of search results labeled by relevance degree, learning to rank returns a ranking function that can be used to order search results for future queries [7].

While learning accuracy has thus far been the primary concern, learning efficiency is increasingly a crucial issue [1]. Due to the diversity of queries and documents, learning to rank involves increasingly larger training data with many features. For example, the CSearch dataset used in ListNet [1] contains ∼25 million query-document pairs with 600 features. In addition, due to the rapid growth of the Web, ranking functions need to be re-learned repeatedly. Thus, it is important for learning to rank to achieve high efficiency through parallelization while maintaining accuracy.

Many learning to rank approaches have been proposed, e.g., ListNet [1], RankBoost [4] and RankSVM [6]. However, these studies were mainly concerned with accuracy and did not seek for improvement in learning efficiency through parallelization. Many parallel machine learning frameworks have been introduced, e.g., IBM Parallel Machine Learning Toolbox (www.alphaworks.ibm.com/tech/pml/) and cooperative coevolution (CC) [9]. However, none of these parallel machine learning methods have been applied to learning to rank.

In this paper, we investigate parallel learning to rank for information retrieval. In particular, we propose CCRank, an CC-based parallel learning to rank framework targeting simultaneous improvement in accuracy and efficiency. We also discuss other ways of achieving parallelization for learning to rank, such as MapReduce [3].

*This work was done while the first author was a postdoctoral fellow at Texas State University.

SIGIR'11, July 24–28, 2011, Beijing, China.
ACM 978-1-4503-0757-4/11/07..

2. PARALLEL LEARNING TO RANK

2.1 The CCRank Framework

Overview. Evolutionary algorithms (EAs) are stochastic search methods mimicking the metaphor of natural biological evolution. They operate on a population of potential solutions, called *individuals*, applying the principle of survival of the fittest to produce better and better approximations to the optimal solution.

Cooperative coevolution (CC) is a framework advantageous in solving problems with exceptionally large search space and complex structures [9]. In CC, a collection of EAs co-evolve simultaneously, where the EAs interact with one another in a cooperative manner. The fitness of an individual is based on how well it cooperates with other interacting individuals. CC follows a divide-and-conquer strategy, decomposing a problem into sub-problems and combining sub-solutions in the end to form a complete solution. The nature of CC allows easy parallelization.

CCRank adapts parallel CC to learning to rank. It starts with problem *decomposition*, followed by a parallel iterative *coevolution* process. At the end of each generation, parallel execution is suspended and a complete candidate solution is produced by *combination*. CCRank returns the best solution selected from all the candidates.

Decomposition and combination. Each complete solution is decomposed into a collection of individuals. Initially L solutions are generated randomly from the full feature space. Then, each solution is decomposed into N sub-individuals, resulting in N populations, each having L individuals. Each population will be assigned an EA to evolve. Combination is a reverse process of decomposition, which assemblies individuals into a complete solution.

Coevolution. Coevolution proceeds iteratively, where N populations co-evolve *in parallel* from generation to generation. Each population $\mathcal{P}_j$ maintains a collection of individ-

Algorithm 1: CCRank Framework

Input : Training set $\mathcal{T}$, number of generations G, number of populations N
Output: Ranking function f

1 $\mathcal{P}^{(0)}_{1,\ldots,N} \leftarrow$ `Initialize` () `// decomposition`
2 **for** $g \leftarrow 1$ **to** G **do**
3 $\quad \mathcal{P}^{(g)}_{1,\ldots,N} \leftarrow$ `Evolve` $(\mathcal{P}^{(g-1)}_{1,\ldots,N})$ `// coevolution`
4 $\quad f^{(g)} \leftarrow$ `Combine` $(w_{1,\ldots,N})$ `// combination`
5 $\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{f^{(g)}\}$

uals and a *winner* w_j, the best individual with the highest fitness score. Under the CC framework, fitness of individuals is based on how well they cooperate with other populations. First, individual i_j from $\mathcal{P}_j$ and winners of *other* populations selected in the *previous* generation are combined into a solution. Then, the evaluation measure, e.g., MAP, for the combined solution is calculated using the training data, and the resulting score is assigned to i_j as its fitness score.

Pseudocode. Algorithm 1 summarizes the CCRank framework. Line 1 performs initialization. Lines 2–6 show the entire evolution process from generation to generation.

2.2 Discussion

Parallel learning to rank. It is possible, but non-trivial, to adapt parallel machine learning algorithms to learning to rank. Some of such algorithms are included in IBM Parallel Machine Learning Toolbox. In addition, Proximal SVM [2] and Cascade SVM [5] are recent parallel SVMs that have demonstrated promising performance. On the other hand, MapReduce [3] provides a simple and unified parallel framework that has been widely applied in various domains. In the following we discuss how to adapt it to learning to rank.

Adapting MapReduce to learning to rank. MapReduce [3] is based on two fundamental functions, `Map` and `Reduce`. The `Map` function takes an input key/value pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the `Reduce` function. The `Reduce` function accepts an intermediate key and a list of intermediate values, and merges these values to form a value for the same key.

To adapt MapReduce to learning to rank, it is desirable to incorporate many existing learning to rank algorithms. However, since MapReduce achieves parallelism by dividing processes of algorithms, it is infeasible to have a unifying framework incorporating all learning to rank algorithms that have different processes. We propose an framework that is able to incorporate an important category of learning to rank algorithms by identifying and parallelizing some common and time-consuming operations.

This category of algorithms perform direct optimization of evaluation measures. They use ranking measures such as MAP and $NDCG$, or some measure-based formulae, as their loss functions directly. A common, time-consuming, and repeating key operation for these algorithms is evaluation of loss functions, which requires ranking of all associated documents for each query. We propose to parallelize this operation. In particular, each process of `Map` evaluates the ranking measure m_q of each query q for candidate ranking function f, and emits the key/value pair (f, m_q) to `Reduce`. Then, each process of `Reduce` calculates the value m of the loss function for f with measures $m_{q_1}, \ldots, m_{q_{|Q|}}$ of each query from different `Map` processes, and emits m.

3. EXPERIMENTS

We implemented CCRank based on RankIP [8]. $N = 8$ EAs are maintained, each containing $L = 70$ individuals that co-evolve up to $G = 30$ generations. The depth of the complete solution is $d = 8$.

For datasets, we used MQ2007 and MQ2008, a collection of benchmarks released in 2009 by Microsoft Research Asia (research.microsoft.com/en-us/um/beijing/projects/letor/). For comparison partners, we used state-of-the-art algorithms AdaRank [10], RankBoost [4], RankSVM [6] and ListNet [1].

We used the Map measure for accuracy comparison. For MQ2007, CCRank and RankBoost share the best performance. For MQ2008, CCRank outperformed all other algorithms, gaining 1.13%, 0.901%, 2.60% and 0.901% respectively. Note that the best and worst performances from those comparison partners differ by merely 1.67%.

To demonstrate the gain in efficiency by parallel evolution, we extracted 25%, 50%, and 100% portions of MQ2008 and generated 3 datasets, which have 3,803, 7,606, and 15,211 instances respectively. Then we ran CCRank on these datasets varying the number of processors (1, 2, 4, 8, and 16).

From the execution time analysis, we can see that parallel evolution leads to significant speed-up in CCRank. Comparing to the case of 1 processor, the averaged relative speed-ups are 173%, 299%, 486%, and 736% respectively, for the cases of 2, 4, 8, and 16 processors.

4. REFERENCES

[1] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li. Learning to rank: from pairwise approach to listwise approach. In *ICML*, 2007.

[2] R. Collobert, Y. Bengio, and S. Bengio. A parallel mixture of svms for very large scale problems. In *NIPS*, 2004.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[4] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer. An efficient boosting algorithm for combining preferences. *J. Mach. Learning Res.*, 4(1):933–969, 2003.

[5] H. P. Graf, E. Cosatto, L. Bottou, I. Durdanovic, and V. Vapnik. Parallel support vector machines: The cascade SVM. In *NIPS*, 2004.

[6] T. Joachims. Optimizing search engines using clickthrough data. In *KDD*, 2002.

[7] T.-Y. Liu. Learning to rank for information retrieval. In *WWW*, 2009.

[8] S. Wang, J. Ma, and J. Liu. Learning to rank using evolutionary computation: Immune programming or genetic programming? In *CIKM*, 2009.

[9] R. P. Wiegand. *An Analysis of Cooperative Coevolutionary Algorithms*. PhD thesis, George Mason University, 2004.

[10] J. Xu and H. Li. AdaRank: a boosting algorithm for information retrieval. In *SIGIR*, 2007.