

A survey of parallel algorithms for classification

Venu Satuluri

March 15, 2007

Abstract

Classification of objects based on their features into pre-defined categories is a widely studied problem with applications in fraud detection, artificial intelligence and many other fields. Parallel formulations of classification methods are desirable to improve training times for problems with large training sets as well as to exploit existing computing hardware. I survey the literature on parallel formulations of building decision trees from large training sets. Most approaches exploit data parallelism by distributing the work of evaluating splitting points at each node of the tree and building the tree in a breadth-first, synchronous manner. The alternative is to exploit task parallelism by assigning a processor or a group of processors the task of building a specific subtree of the global decision tree.

1 Introduction

Classification is the problem of automatically assigning an object to one of several pre-defined categories based on the attributes of the object. It has been recognised as one of the core tasks in data mining, the field that is concerned with the extraction of knowledge or patterns from databases through the building of predictive or descriptive models [4]. For example, insurance companies might want to classify a customer as being either high-risk or low-risk using various attributes about the customer - such as credit history, annual income, age etc. More examples of the problem include hand-written digit recognition, text classification, intrusion detection, credit card fraud detection and so on.

The problem is usually formulated as follows: we are given a *training set* of objects (also called *instances* or *records*) and their attributes (also called *features*) as well as the categories or classes that these objects belong to. The attributes of each record can either be *categorical* or *continuous*. We then try to build a classifier that makes use of the training set to build a model to predict the class of a new record, given the attributes of the new record. Because we are given a training dataset, the classification problem is also known as supervised induction or supervised learning.

Several methods for classification have been introduced over the years - e.g. decision trees [1] [11], artificial neural networks, nearest neighbour classifiers [9], support vector machines and so on. Neural nets and support vector machines are now-a-days considered to be the state-of-the-art, but decision trees also have decent accuracy and moreover are easier to interpret, which is a crucial advantage when it comes to data

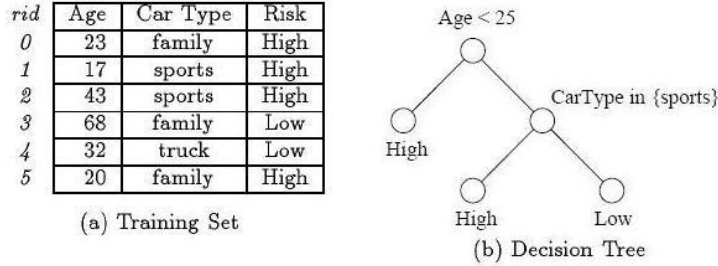


Figure 1: A toy decision tree built for classifying car insurance customers as high/low risk. [12]

mining. I also suspect that once an algorithm gains acceptance, it takes time before scalable and parallelised versions of that algorithm appear. For these reasons, the majority of the work on scaling up classification algorithms has been in formulating parallel versions of decision trees. Hence, this paper surveys only parallel algorithms for decision tree induction. Other surveys on this topic include [14] and [10].

2 Decision Trees

A decision tree model consists of internal nodes and leaves. An internal node represents a test on one attribute of the data - this test splits the data into two portions (or, more generally, k portions) and the portions are respectively assigned to the children of that node. This means that each node is also associated with all those records which satisfy tests at all the intermediate nodes along the path from this node to the root of the decision tree. Each leaf is associated with a class - all the records which satisfy all the tests along the path from this leaf to the root will be assigned the class associated with this leaf. Consult Figure 1 for an example.

Decision tree induction takes place in two steps - a growth or construction phase where the tree is built from the training set, and a pruning phase where the tree is reduced in size to avoid overfitting with the training set. The growth phase is computationally much more time-consuming [12], hence all algorithms concentrate on parallelizing the tree construction phase. The growth phase consists of a recursive algorithm that finds a split point, partitions the training set and recursively builds the trees for the partitions (consult Figure 2). The tree is thus built in a depth-first manner.

For categorical attributes, the test takes the form of $a_i \in A'_i$ where $A'_i \subset A_i$ and A_i is the domain of the i^{th} attribute. For continuous attributes, the test takes the form of $a_i < x$ or an equivalent inequality. The x s are usually midpoints between all the sorted values of that attribute in the training set, hence choosing a split point for continuous attribute requires the attribute values to be sorted. For each test or split, we have to evaluate the goodness of the split. One measure is the *gini* index [1] - for a data set S containing examples from n classes, $gini(S)$ is defined as $gini(S) = 1 - \sum p_j^2$ where p_j is the relative frequency of class j in S . The index of a split that divides S into

```

Partition(Data  $S$ )
  if (all points in  $S$  are of the same class) then
    return;
  for each attribute  $A$  do
    evaluate splits on attribute  $A$ ;
  Use best split found to partition  $S$  into  $S_1$  and  $S_2$ ;
  Partition( $S_1$ );
  Partition( $S_2$ );

Initial call: Partition(TrainingData)

```

Figure 2: The serial algorithm for the growth phase of learning a decision tree. [12]

two subsets S_1 and S_2 is given by $gini_{split}(S) = \frac{n_1}{n}gini(S_1) + \frac{n_2}{n}gini(S_2)$. Since the number of decision trees that can be constructed on a given training set is huge, one typically employs a greedy search over the decision tree space by selecting those splits with the least *gini* value as the next node in the tree.

3 Survey of Approaches

3.1 SLIQ

SLIQ [8] was one of the first scalable algorithms for decision tree induction which also laid down the groundwork that much subsequent work ([12] [6] [5] [13]) would reuse. SLIQ uses a vertical data format, meaning all values of an attribute were stored as a list, which was sorted at the start of the algorithm. This meant that the attributes need not be sorted repeatedly at each node as was the case in existing algorithms. SLIQ also evaluated the split points for each node at the same level in one scan through the training data, thus growing the tree in a breadth-first manner. The calculation of gini index for each possible split point can be done efficiently by storing class distributions in histograms, one per class per node. However SLIQ uses a memory-resident data structure called *class list* which stores the class labels of each record. This data structure limits the size of the datasets SLIQ can handle.

3.2 SPRINT

Shafer et. al. [12] present a more memory-efficient version of SLIQ, called SPRINT and also give parallel versions of SPRINT. This algorithm associates the class label along with the record id for each value in the attribute lists. It splits each of the attribute lists evenly among the processors ensuring that the continuous attributes maintain their sorted order on values at each nodes. (In other words, the data is partitioned in a row-block fashion.) Each processor determines the best split points for all the records it has, and there happens a round of all-to-all broadcast to determine the best global split point. Once this decision has been made, the attribute lists of the splitting attribute can be partitioned easily. In order to split other lists, a hash table is formed to maintain a mapping of record ids to nodes (of the decision tree in the next level), which needs to be communicated all the other processors from the processor that produced the winning split point. The other processors then use the hash table to split their own

attribute lists. The size of this hash table is proportional to the number of records at the current node, which means that for the root of the decision tree, the size of the hash table is proportional to the total number of records in the training set.

3.3 ScalParC

ScalParC [6] improves on SPRINT by using a distributed hash table that need not be locally constructed and thus eliminates the bottleneck that the hash table presented for SPRINT. Each processor stores a $\frac{1}{p}$ part of the hash table, and each global record id j is hashed using the function $h(j) = (j \div \frac{N}{p}, j \% \frac{N}{p})$, where N is the size of the database, p is the number of processors, and the first member of the ordered pair gives us the processor that is associated with the record id j and the second member of the ordered pair gives us the location in the buffer of that processor where this record is stored. Once each processor gets to know the winning split decision, it makes a buffer corresponding to each processor, where it puts the records that need to be communicated to that processor. Once all processors make such buffers, an all-to-all personalised communication is performed which ensures that each processor has the records it needs to handle. (This step is similar to the data exchange step in the parallel sample sorting algorithm.)

3.4 pCLOUDS

The approaches presented so far are *data parallel* i.e. they exploit the parallelism available among the different attributes. This approach is efficient at building the top parts of the tree where split points can be evaluated in parallel. One could also formulate *task parallel* algorithms that exploit the parallelism in building different subtrees of the decision tree, which approach is more efficient during construction of the lower parts of the decision tree. pCLOUDS [13] combines both these approaches, where a data parallel approach is used when the tree growth is at a stage where the (tree) nodes are large, and a transition to a task parallel approach is made once the tree nodes become small enough. The data of each task is redistributed to its destination processors after which building the subtree is performed locally within the processor. The assigning and processing of small nodes are delayed until all the large nodes have been processed to reduce the number of message startups. pCLOUDS also departs from earlier SLIQ-derived algorithms by not pre-sorting the attribute lists and deriving splitting points by *sampling* the splitting points, in which they evaluate only a subset of the splitting points along each numeric attribute. They claim this does not significantly affect the accuracy of the decision tree so constructed.

4 Performance Evaluation

Except for SLIQ (which suffers from the bottleneck of a memory-resident data structure), the other three algorithms presented above show good speedups with increase in the number of processors. However, objective judgement of the different algorithms is difficult as none of the papers I have summarised tries to compare the running times

of their algorithms with the times of the algorithms they are trying to improve upon, by using benchmark datasets.

5 Discussion and Conclusion

There exist more parallel versions of decision tree induction, such as RainForest [5], SUBTREE [15], Parallel Decision Tree [7] and so on. I have chosen not to report on these mainly because I felt the papers I have summarised above contain some of the main ideas and issues involved in the parallelisation of decision tree induction and also because an exhaustive summary is outside the scope of this paper. Other surveys on this topic include [14] and [10].

One is also lead to wonder why there are no parallel versions of other classification methods. Ensemble methods for classification, such as arcing classifiers [2] and random forests [3], use multiple *base classifiers* and perform some kind of averaging or polling on the outputs of the decisions of said base classifiers to output the final classification. These methods have been recognized in recent times to be highly effective and are ideally suited for parallelization.

References

- [1] Breiman, Friedman, Olshen, and Stone. *Classification and Regression Trees*. Wadsworth, 1984.
- [2] L. Breiman. Arcing classifiers. *Annals of Statistics*, 26(3):801–849, 1998.
- [3] L. Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.
- [4] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery: An overview. *Advances in Knowledge Discovery and Data Mining*, 1996.
- [5] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest - a framework for fast decision tree construction of large datasets. *Data Mining and Knowledge Discovery*, 4(2/3):127–162, 2000.
- [6] Joshi, Karypis, and Kumar. Scalparc: A new scalable and efficient parallel classification algorithm for mining large datasets. In *IPPS: 11th International Parallel Processing Symposium*. IEEE Computer Society Press, 1998.
- [7] R. Kufirin. Decision trees on parallel processors. In *IJCAI-95 Workshop on Parallel Processing for Artificial Intelligence*, 1995.
- [8] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Extending Database Technology*, pages 18–32, 1996.
- [9] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.
- [10] F. J. Provost and V. Kolluri. A survey of methods for scaling up inductive algorithms. *Data Mining and Knowledge Discovery*, 3(2):131–169, 1999.

- [11] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1993.
- [12] J. C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. 22nd Int. Conf. Very Large Databases, VLDB*, pages 544–555, 1996.
- [13] M. K. Sreenivas, K. Alsabti, and S. Ranka. Parallel out-of-core divide-and-conquer techniques with application to classification trees. In *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 555–562, Washington, DC, USA, 1999. IEEE Computer Society.
- [14] M. J. Zaki and C.-T. Ho, editors. *Large-Scale Parallel Data Mining*. Springer-Verlag, 2000.
- [15] M. J. Zaki, C.-T. Ho, and R. Agrawal. Parallel classification for data mining on shared-memory multiprocessors. In *ICDE: IEEE International Conference on Data Engineering*, pages 198–205, 1999.