Parallel algorithm for constructing decision tree – partitioned approach

Problem description

The Decision Tree is one of the most popular classification algorithms in current use in Data Mining and Machine Learning. Decision tree is the form of classification tree. Each interior node corresponds to a variable; an arc to a child represents a possible value of that variable. A leaf represents a possible value of target variable given the values of the variables represented by the path from the root.

Construction of decision trees is usually done with recursive algorithm, which splits data-set into subsets. This process is repeated on each derived subset in a recursive manner. The recursion is completed when splitting is either non-feasible, or a singular classification can be applied to each element of the derived subset.

Parallel algorithms are invented to speed up decision tree constructing. Since training data-sets should be as large as possible, sequential algorithms are two slow to cope with very large data-sets, hence highly parallel algorithms are desired.

Background

Problem of construction of decision tree is well studied. There exists a lot of different algorithms for constructing decision trees. The same goes to parallel versions. Some gives almost linear speed up with larger number of processors. There are two parallel formulations of classification decision tree learning algorithm based on induction [1]. They are *Synchronous Tree Construction Approach* and *Partitioned Tree Construction Approach*. We are going to implement the latter.

Description of the own solution (new algorithms, methods, techniques or modifications of some algorithms)

As we've mentioned, the *Partitioned Tree Construction Approach* will be described and implemented. We've chosen Erlang programming language. Erlang is a general-purpose concurrent, parallel, functional, distributed programming language (and run-time system), invented specifically for soft real-time, distributed, fault-tolerant applications. It was invented by Ericsson and then opensourced. Erlang employees actor model for concurrency. Each actor is called "process". Processes uses asynchronous message passing for communication. Each erlang run-time system is called "node". Distribution mechanisms are builtin into Erlang run-time system, so that communication of process at node A with process at node B is fully transparent.

We've used slightly modified version of ID3 algorithm, implemented in partitioned parallel way. Each node (processor) has local access to the data-set. Decision tree construction is handled by tree construction superviser, which is supervising the process. Work processes are spawned on each node, which are connected to the cluster, using builtin rpc mechanism. After spawning these processes information about their location is not used (since inter-process communication across nodes is transparent).

Consider the case in which a group of processors *Pn* cooperate to expand node *n*. The algorithm consists of following steps:

1. Processors in *Pn* cooperate to expand node *n* using the following method
   - Each processor stores disjoint set of ids of data-items and, when asked for, gathers extensive distribution information on subset of data-set defined by these ids.
   - Each processor sends this information to tree_construction superviser, which sums this information and, when all information is gathered, decides what test is to be used at the tree node

- Once the node *n* is expanded in to successor nodes, *n1* , *n2* , ... , *nk* , then the processor group *Pn* is also partitioned, and the successor nodes are assigned to processors as follows
  1. If the number of successor nodes is greater than |Pn|,
     1. Partition the successor nodes into |Pn| groups. Assign each processor to one node group.
     2. Shuffle the training data such that each processor has data items (in fact their ids) that belong to the nodes it is responsible for.
     3. Now the expansion of the sub-trees rooted at a node group proceeds completely independently at each processor as in the serial algorithm.
  4. Otherwise (if the number of successor nodes is less than |Pn|),
     1. Assign a subset of processors to each node
     2. Shuffle the training cases such that each subset of processors has training cases that belong to the nodes it is responsible for.
     3. Processor subsets assigned to different nodes develop sub-trees independently. Processor subsets that contain only one processor use the sequential algorithm to expand the part of the classification tree rooted at the node assigned to them. Processor subsets that contain more than one processor proceed by following the above steps recursively.

At the beginning, all processors work together to expand the root node of the classification tree. At the end, the whole classification tree is constructed by combining sub-trees of each processor.

User’s Manual

To run this application you have to install Erlang R12B2 [2], to have "erl" in the PATH variable (or you can point it explicitly) and to have GNU "make" installed on computer.

To compile You have to go to root of package and run

```
make
```

To run program in one node mode type (ignore warnings if they appears)

```
make run
```

To run program in master node mode type

```
make run_master
```

To run program in child node mode type (then you will have to choose name of the node - name should be unique and name "master" is reserved)

```
make run_child
```

In one node or master node you'll see program's main menu:

```
Nodes: [nonode@nohost]

1. Create tree
2. List nodes
0. Quit
>
```

To correctly configure multi-machine cluster consult [3]

First line says what nodes are currently connected to cluster. You may create tree by typing "1" and pressing Enter. To see list of connected nodes type "2". To quite type "0" and press Enter. When you chose to create tree you'll be asked about path to the csv file:

```
Current directory: /home/gleber/workspace/erlang/dec_tree
Enter filename: priv/mices.csv
```

Enter relative or absolute path to the csv file (for format - see below). When tree is created you'll see the following:

```
1. Print tree
2. Print unpruned tree
3. Test tree on original data
0. Back
>
```

Commands are self-describing.

File format

Program handles only comma-separated files of special form it should consist of at least 3 lines. The first contains columns names. The second contains column types ("int" for continuous data or "str" for discreet data). The third contains if column contains dependent variables ("dep" - only one dependent column is allowed), independent variables ("ind") or is to be ignored ("ign"). After these header columns data columns are to be present. Here is an example of file:

```
"Id","Level","Sex","Time","Death"
"int","str","str","int","str"
"ign","ind","ind","ind","dep"
1,"Control","Female",70,1
2,"Control","Female",77,0
3,"Control","Female",83,1
4,"Control","Female",87,0
5,"Control","Female",92,0
6,"Control","Female",92,0
7,"Control","Female",93,1
```

Technical documentation (comments in source, description of main data structures and procedures)

Modules

Program consists of few modules:

- gui - program UI is implemented here
- dec_tree - part of decision tree logic is implemented here (functions for calculating entropies, information gains and similar)
- data_server - global data_server, tree_constructor and data_worker are implemented here
- dataset_erl - functions for handling data-set stored in ETS [4] is here
- csv - helper module for CSV reading

Program is separated in few parts:

- data_server - global process which handles managing (spawning and handling fails) of workers (data_worker), data-set distribution and serving meta data on data-set (such as columns, their types, its dependency/independency
- data_worker - process which is spawned at every node (Erlang run-time system) by the data_server, which stores whole data-set, information about subsets it's responsible for (for every tree node it is responsible for), statistic gathering and distribution of this data among other workers (when asked by tree_constructor)
- tree_constructor - this process is supervising tree construction process. It fetches statistics from workers, analyse it and chooses how the node is to be expanded. Next it decides how subsets of data is splitted between workers.

Use case

Creation of one decision tree triggers the following operations:

1. Erlang run-time system is ran on different machines, where exactly one is ran as the master
2. At master machine UI functions from gui.erl handles user input (eg. file name)
3. Functions inside gui.erl reads data and metadata from the csv file
4. data_server is started
    1. data_server spawns worker processes at every machine connected to the cluster
2. Metadata is sent to data_server
3. UI sends data to data_server, which redistributes it among workers
    1. Data is stored only at workers (data_server does not use data itself)
2. data_server:distributed_create_tree is called and tree_constructor is created
3. tree_constructor claims all workers from data_server
4. tree_constructor tells workers to split data (using ids) among them evenly (in fact whole data-set is stored at every data_worker, but every data_worker works only with subset defined by list of ids)
5. tree_constructor tells workers to start gathering statistics and waits for it
6. When statistics from all workers for given node is gathered
    1. It is checked if given tree node should be leaf, if true, then this node is transformed into leaf corresponding to specific class (and it is removed from list of nodes to be expanded)
    2. Entropy is calculated for every possible test for given subset
    3. Test with smallest entropy is chosen as the way to split the node into sub-nodes
    4. Child nodes are created
    5. tree_constructor tells data_workers responsible for the node to redistribute ids among them according to selected test
    6. tree_constructor tells data_workers to start gathering statistics
    7. The node is removed from list of nodes to be expanded, child nodes are added to list of nodes to be expanded
    8. Point 11 (1 - 6) is repeated for for every, while list of nodes to be expanded is not empty
9. tree_constructor reconstructs tree from internal representation into representation usable by UI and sends it to the process, which called data_server:distributed_create_tree
10. data_server:distributed_create_tree returns created tree to UI
11. UI simplifies tree by pruning
12. UI gives a user tools to test tree against data-set and print it on the screen

Data structures

Every process stores it's internal state variable as argument of corresponding tail-recursive function.

Data-set is stored in ETS (inside state variable of each data_worker) as a set of tuples identified by unique id number, these tuples represents one data item with additional unique id number (hence size of tuple is (number of columns + 1)).

Description of tests

Sample data-sets are located in "priv/" directory. Program is capable of handling any of those.

Data-set dm2.csv

It was artificially generated with [5]. Thereexists very clear dependency between column "C" and dependent column:

```
Enter filename: priv/dm2.csv
1. Print tree
2. Print unpruned tree
3. Test tree on original data
0. Back
> 1
  test equals on "C" (100)
    "an*":
      decision: "c1" in 1 of 1 entries
    "d*":
      decision: "c2" in 2 of 2 entries
    "o*":
      decision: "c1" in 1 of 1 entries
    "ad*":
      decision: "c2" in 1 of 1 entries
    "ag*":
      decision: "c1" in 1 of 1 entries
    "e*":
      decision: "c1" in 1 of 1 entries
    "x*":
      decision: "c2" in 1 of 1 entries
    "?":
      decision: "c2" in 1 of 1 entries
    "at*":
      decision: "c2" in 1 of 1 entries
    "m*":
      decision: "c2" in 1 of 1 entries
    "bc*":
      decision: "c2" in 1 of 1 entries
    "d":
      decision: "c2" in 35 of 35 entries
    "ag":
      decision: "c1" in 53 of 53 entries
1. Print tree
2. Print unpruned tree
3. Test tree on original data
0. Back
> 2
100 of 100 correct classified
```

It is clear, that program was able to reveal this dependency.

Data-set mices.csv

Data-set from [6] containing "Types of Death of Sacrificed Mice Receiving Red Dye No. 40".

```
Enter filename: priv/mices.csv
1. Print tree
2. Print unpruned tree
3. Test tree on original data
0. Back
> 1
  test less on "Time" (122)
    103:
      decision: "0" in 84 of 117 entries
    other:
     test equals on "Sex" (5)
       "Female":
         decision: "1" in 2 of 2 entries
       "Male":
         decision: "0" in 3 of 3 entries
1. Print tree
2. Print unpruned tree
3. Test tree on original data
0. Back
> 3
89 of 122 correct classified
```

Classification rate of 72% is quite good for such simple algorithm. Probably it may possible to increase this rate by replacing information gain with the gain ratio in function which decides what test should be chosen at given tree node.

Conclusions, comments

No benchmarking was done. Except for few tests which shown, that given implementation of parallel algorithms is flawed, since there was only 5% to 10% speed up when second machine is added to the cluster. Most probable cause of this behaviour is the fact that there is a lot of communication between workers. Functions which splits data-set at given tree node into should take into consideration locality. Probably this implementation is also flawed by well known problem of partitioned approach to parallel construction of decision trees - at some moment frontier of tree is too big and there is a lot of communication between processors - in many cases it would be simpler to process data locally, because communication cost will be too big.

List of references to literature, web pages.

1. http://www-users.cs.umn.edu/~kumar/papers/classpar-icpp.ps
2. http://erlang.org/

3. http://www.erlang.org/doc/reference_manual/distributed.html
4. http://www.erlang.org/doc/man/ets.html
5. http://www.datasetgenerator.com/
6. http://lib.stat.cmu.edu/datasets/Andrews/T42.1
7. http://www.mini.pw.edu.pl/~brys/www/
   ?Strona_g%B3%F3wna:Przedmioty_z_semestru_zimowego:Data_Mining