Task Management in Advanced Computational Intelligence System

Krzysztof Grąbczewski and Norbert Jankowski

Department of Informatics, Nicolaus Copernicus University, Toruń, Poland
{kg,norbert}@is.umk.pl
http://www.is.umk.pl

Abstract. Computational intelligence (CI) comes up with more and more sophisticated, hierarchical learning machines. Running advanced techniques, including meta-learning, requires general data mining systems, capable of efficient management of very complex machines. Requirements for running complex learning tasks, within such systems, are significantly different than those of running processes by operating systems. We address major requirements that should be met by CI systems and present corresponding solutions tested and implemented in our system. The main focus are the aspects of task spooling and multitasking.

1 Introduction

Thanks to continuous growth of computing power, that could be observed during recent decades, we can build more and more sophisticated software for solving more and more serious problems. Even home computers are now equipped with multi-core processors, so it is not too difficult to gather a number of quite powerful CPUs for scientific explorations. Naturally, it is still (and will always be) very important not to waste computing resources, so we need data mining systems capable of taking full advantage of available power. Because hardware development goes toward more parallel computations, computational intelligence (CI) systems must follow this direction and efficiently solve multitasking problems at kernel level. Advanced CI applications can easily be parallelized, because solving contemporary problems always requires testing many complex machines. How complex can the models be and how many computational experiments must be performed to obtain satisfactory solutions, can be seen by examples of recent data mining contests, e.g. the NIPS Feature Selection Challenge [6,5], different meta-learning enterprises [1,2,8,4] and miscellaneous ensemble machines approaches. Therefore, we need versatile tools for easy manipulation of learning machines. Intemi—the system, we have recently developed [3,7] is a general data mining tool, designed with emphasis on efficiency and eligibility for meta-learning.

In this paper, we present some kernel-level solutions of Intemi, which facilitate building variety of machines in simple and efficient way. After short introduction to basic architecture of the system (section 2), we present Intemi task management system. Major prerequisites can be found in section 3, then we describe Intemi task life cycle (section 4) and finally, the task spooling module of the system (section 5).

L. Rutkowski et al. (Eds.): ICAISC 2010, Part I, LNAI 6113, pp. 331–338, 2010.
© Springer-Verlag Berlin Heidelberg 2010

2 Intemi Basics

Intemi handles CI algorithms in a unified way based on the concept of *machines*. A general view of machine is presented in figure 1. The abstract concept of machine encircles not only learning machines but also all other algorithms related to data management (data loaders, data transformations, tests etc.). Machines exchange information by means of their inputs and outputs. Before a machine may be created, its process parameters and input bindings must be specified. After creation, machine process must be started, to prepare the machine for further services provided by outputs and possibly deposit adequate results in *results repository*. The process may be designed for arbitrary goals, from simple data loading to advanced data mining. Each machine is allowed to create other machines called its *submachines* or *child machines*. Machine trees, representing parent-child relations, and graphs of input–output interconnections are managed within *projects*.

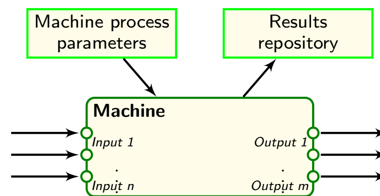

Fig. 1. The abstract view of machine

An example of complex machine is presented in figure 2. It is an instance of Repeater—a machine that, a specified number of times, repeats a test for subsequent collections of outputs provided by a distributor machine. The information about the number of repetitions, the test and the distributor is provided within Repeater machine configuration. The *schemes* included in the figure are machines responsible just for creation of a number of child machines in appropriate configuration. In the figures, child machines are depicted as boxes drawn within the area of their parents.

Advanced Intemi projects may produce large numbers of different machines. To handle them efficiently while preserving ease of use, the project must provide flexible tools. One of them is the task spooling system, presented in this article.

When dealing with large number of machines, it is quite probable, that the same machine is requested many times. To prevent from running the same machine repeatedly, Intemi is equipped with machine unification mechanism and a cache system. The problem has been solved at system level, because it is definitely more versatile than any machine-level solution and makes machine development easier. Task management module must also be aware of unification possibilities, to ensure that two identical machines will never be run—in such cases, the same machine must be assigned to all equivalent requests.

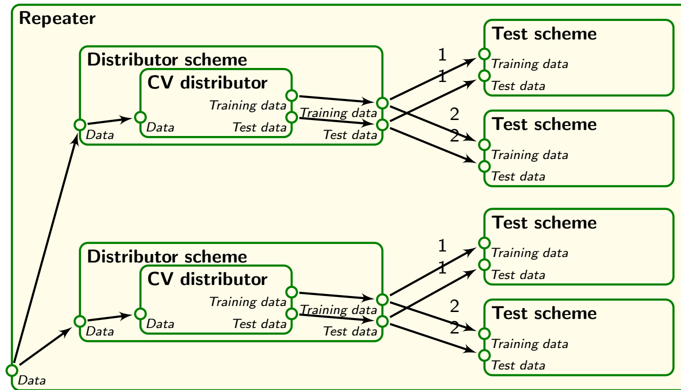

Fig. 2. Run time view of Repeater machine configured to perform twice 2-fold cross-validation (CV). Test schemes are simplified for clearer view—in fact each one contains four submachines as in figure 3.

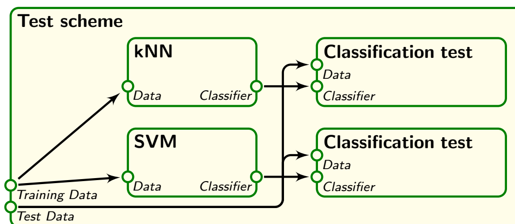

Fig. 3. A test scheme example

3 CI Task Management Prerequisites

Large scale CI computations require parallelization. It can be easily done, because in most projects, many machines may run independently. On the other side, we can not run all the requested machines instantly, because it could quickly overfill available memory. Therefore, we need to control the number of subprocesses running in parallel.

Task management in a CI system has its own specificity, quite different from task management inside operating systems (OS). Although in both problems, priorities are assigned to tasks, the differences are fundamental. For example, in operating systems we need a fair scheduler, i.e. a system of equal CPU allocation to all process of the same priority, while in advanced CI, memory saving is more important, because huge amounts of machines running at the same time would paralyze the system. Therefore, CI systems should rather care for finishing tasks as soon as possible and start new ones only if no further parallelization of currently running tasks is possible.

Another significant difference between CI and OS task management is that in CI we would like the system to reflect task dependencies in an automated way. Here, by *dependencies* we mean the input-output relation. We can create a task with inputs bound to outputs of a machine that is not started yet or is currently running, so that the new task can not be started until some other tasks are finished. Task synchronization in OS is a completely different problem.

The most important requirements for a CI task management system are:

- efficient exploitation of CPU resources i.e. taking advantage of as much CPU power as possible,
- protection against system memory exhaustion,
- automated resolving of dependencies between tasks,
- support for unification system in preventing from running equivalent tasks more than once.

The system, we have implemented within Intemi, satisfies all these conditions.

4 Task Life Cycle

Intemi machine request life follows one of many possible paths, according to the flowchart presented in figure 4. The flowchart blocks represent different states of the request, while dashed lines encircle the areas corresponding to particular system modules.

Each machine request is first analyzed to determine the machine contexts providing inputs to the requested machine. As it can be seen in figures 2 and 3, an input may be bound in a number of ways including binding to parent's input, to sibling's output (thus an output of a machine that does not exist at the time of configuration), etc. Therefore, the abstract information provided at machine configuration time, must be transformed into outputs specification containing references to machine instances. After that, the request is passed to the *input readiness control* module, where, if necessary, it waits for processes of other machines (the machines providing inputs) to finish. The inputs readiness guard keeps the collection of awaiting machine requests up to date, thanks to receiving (in real time) the information about each new machine request and about all the changes in machine (outputs) readiness. When all the inputs of a particular machine request are ready, it is examined by *machine cache*, whether the machine is already available because of earlier calculations. If the unification is not possible, the machine request is pushed to the *task spooler* and finally it can be fulfilled by a *task running* thread. The life of a request may be aborted at any time by parent machine or by system user. This may influence the flows of other requests for the same machine.

It needs to be stressed that the whole machine life cycle is managed completely automatically. From the user point of view, only the start and the end of the path, machine goes through, must be taken care of, i.e. the user orders a machine providing its configuration and inputs bindings, and then just waits for the machine (or for a collection of submachines) to be ready for further analysis.

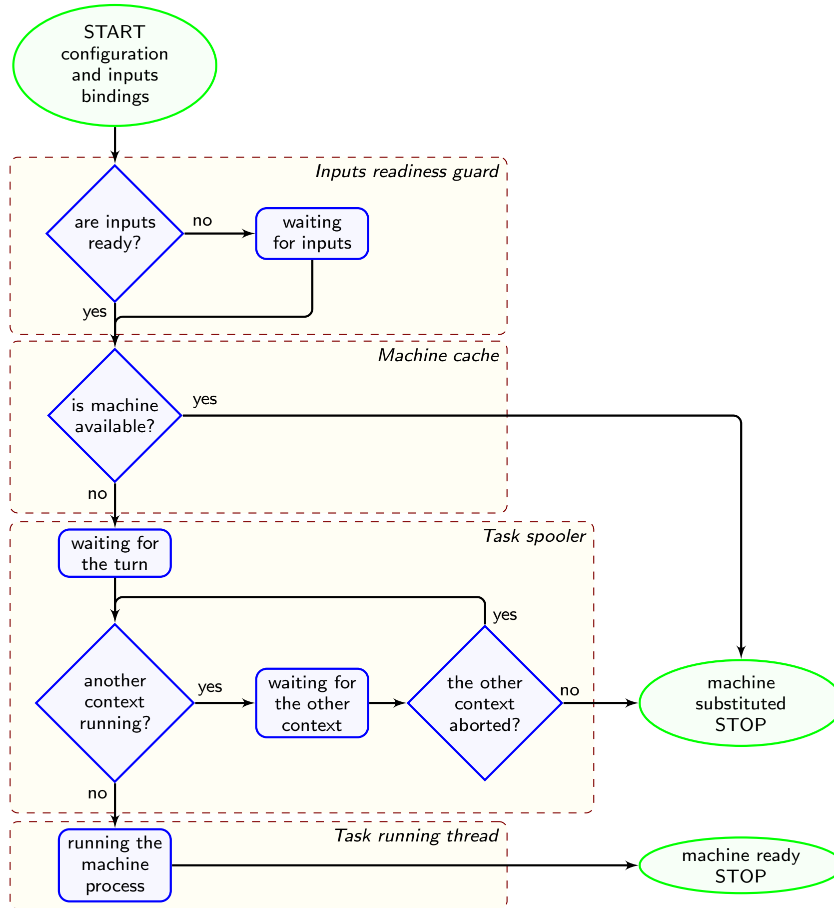

Fig. 4. Machine request life cycle

5 Task Spooling

In most computer systems dealing with tasks, structures of queues are quite successful in task ordering and preventing from running too large number of tasks in parallel. As signaled in section 3, in the case of CI projects, fundamental requirements are different, which makes solutions based on first-in-first-out (FIFO) methodology not appropriate.

In a CI project, it is not desirable to run all the subtasks in parallel and grant each of them with the same amount of CPU time. Instead, we prefer finishing the subtasks already running, over starting completely new ones. To address this issue, we have based Intemi task spooling system on a *tree structure with ordered nodes*. The tree nodes correspond to machines (running or scheduled for running) and the subnode relation is the submachine relation. The order of child machines reflects priorities assigned to the submachines at request time.

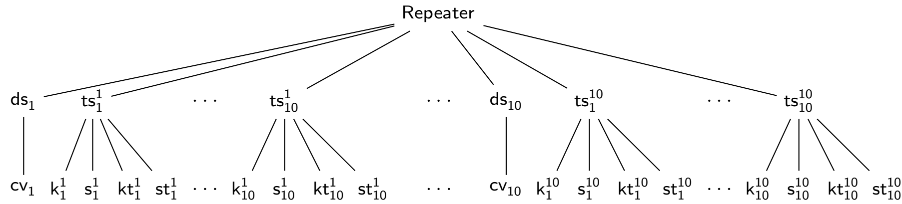

Fig. 5. Repeater submachine tree performing 10 × 10-fold CV of tests defined in figure 3. Notation: ds – distributor scheme, ts – test scheme, cv – CV distributor, k – kNN, kt – classification test for kNN, s – SVM, st – classification test for SVM.

In the case of equal priorities, the time of request determines the order. When a machine is popped from the spooling tree, it is searched with hill climbing technique (i.e. depth first search respecting the order of child nodes).

To observe the advantages of Intemi spooling system structure in comparison to standard queue, let's analyze the progress of calculating 10 repetitions of 10-fold CV to compare classification accuracy of kNN and SVM algorithms. Such configuration results in a repeater machine as presented in figures 2 and 3, but with 10 distributor schemes instead of 2 and 10 test schemes per distributor scheme, in place of 2. The resulting machine hierarchy is sketched in figure 5. The repeater machine creates 10 distributor schemes and 100 test schemes. Each distributor scheme creates one cross-validation distributor and each test scheme requests 4 child machines: kNN, classification test of kNN, SVM and classification test of SVM.

To avoid randomness of the process due to parallel calculation, we assume that all the tasks are calculated by a task manager with one running thread. The request for the repeater machine pushes the root node of the tree (of figure 5) into the spooler. When the request is popped out, the repeater process is run and puts all the repeater children to the queue: the first distribution scheme (ds_1), 10 test schemes (ts_1^1,...,ds_{10}^1) bound to ds_1 outputs, the second distribution scheme (ds_2) and so on. Thus, 110 machine requests go to the queue. After that, the repeater starts waiting for its children and the task manager calls for next task to run (ds_1 is poped). The distributor scheme requests the CV distributor machine (cv_1) and starts waiting until cv_1 is ready.

When a standard queue is used as the spooler, there are 109 requests in the queue before cv_1, so it will be run after all the 109 preceding requests are popped, run and start waiting after pushing all their requests for children to the spooler. It means that when cv_1 gets its turn to run, 111 threads are in waiting mode (the repeater machine and all 110 of its children) and all the 410 machines of the third level are in the queue. So, the task manager controls 112 task threads. It costs a lot: the operating system must deal with many waiting threads and all the started machines occupy memory.

With Intemi spooling system based on tree with ordered nodes, the history of machine requests and pops is quite different. Only the begin is similar, because

the repeater machine is popped, run and it requests its 110 children. Then, ds_1 is popped out and run. It pushes cv_1 to the spooler and starts waiting. Next pop from the spooler returns not ts_1^1 as in the case of standard queue, but cv_1, because the ds_1 branch is favored over all the other children of the repeater. When cv_1 is finished, ds_1 can be finished too, and ts_1^1 is run. It requests its 4 children, which are finished before ts_2^1 is started, thanks to the ordered tree based spooling system. As a result, only two waiting machine processes and one running may be observed at the same time, so the task manager controls only 3 threads. This is because the machines are popped from the spooler in the following order:

$\mathsf{Repeater}$, ds_1, cv_1, ts_1^1, k_1^1, s_1^1, kt_1^1, st_1^1, …, ts_{10}^1, k_{10}^1, s_{10}^1, kt_{10}^1, st_{10}^1, …, ds_{10}, cv_{10}, ts_1^{10}, k_1^{10}, s_1^{10}, kt_1^{10}, st_1^{10}, …, ts_{10}^{10}, k_{10}^{10}, s_{10}^{10}, kt_{10}^{10}, st_{10}^{10},

while in the case of a standard FIFO the order is:

$\mathsf{Repeater}$, ds_1, ts_1^1, …, ts_{10}^1, …, ds_{10}, ts_1^{10}, …, ts_{10}^{10}, cv_1, k_1^1, s_1^1, kt_1^1, st_1^1, …, k_{10}^1, s_{10}^1, kt_{10}^1, st_{10}^1, …, cv_{10}, k_1^{10}, s_1^{10}, kt_1^{10}, st_1^{10}, …, k_{10}^{10}, s_{10}^{10}, kt_{10}^{10}, st_{10}^{10}.

Since, thanks to the spooling system, Intemi keeps just three running machines at a time, both memory and CPU time are saved significantly. In one of the example projects, peak memory usage was about **30 MB**, while with standard queue it was over **160 MB**. Also the overall time used by the projects was significantly reduced (around **15%**).

The idea of machine unification is that a machine must not be constructed twice. To make it possible, the task spooler must also be involved in unification, because the same machine request may occur in the spooler twice (two requests for the same machines but in different contexts, with different priorities). The spooling system may not allow for popping the same request twice, so when a request is popped, all its other instances must be blocked until the request is handled. When the task is finished successfully, all other requests are provided with the model, otherwise, the remaining requests stay in the spooler with proper priorities, to be started again in adequate time. The functionality of popping next task to run, is expressed algorithmically by the following meta-code:

```
1  Task GetNextTask() {
2    foreach (Task t in waiting_for_another_context) {
3      if (the other context aborted) return t;
4      if (the other context is ready)
5        { t.Status = Substituted; continue; }
6    }
7    while (true) {
8      if (spooler.IsEmpty) return null;
9      Task t = spooler.Pop();
10     if (machine of t is finished by another context)
11       { t.Status = Substituted; continue; }
12     if (machine of t is running within another context)
13       { waiting_for_another_context.Add(t); continue; }
14     return t;
15   }
16 }
```

The tasks, that receive their turn while other tasks requesting the same machine are running, are moved to the waiting_for_another_context collection, which is served in a privileged manner. It is processed in lines 2–6, before the main loop, processing the spooler (the tasks that have not got their turn yet), coded in lines 7–15.

6 Summary

Although task management module stays in background, it constitutes a very important part of any CI system. As we have presented, proper organization of task life cycle and task spooling may significantly improve performance and possibilities of the system. Naturally, task management must seamlessly cooperate with other modules of the system, e.g. with machine unification. System design respecting all aspects of machine learning processes, results in high efficiency and reliability of the system. Intemi task management approach has yielded many advantages including reduction of the number of simultaneously started machine processes. It facilitates computing more complex machines and reduces the time of computations. The scale of improvements has exceeded our expectations.

References

1. Brazdil, P., Soares, C., Da Costa, J.P.: Ranking learning algorithms: Using IBL and meta-learning on accuracy and time results. Machine Learning 50(3), 251–277 (2003)
2. Fürnkranz, J., Petrak, J.: An evaluation of landmarking variants. In: Giraud-Carrier, C., Lavra, N., Moyle, S., Kavsek, B. (eds.) Proceedings of the ECML/PKDD Workshop on Integrating Aspects of Data Mining, Decision Support and Meta-Learning (2001)
3. Grąbczewski, K., Jankowski, N.: Meta-learning architecture for knowledge representation and management in computational intelligence. International Journal of Information Technology and Intelligent Computing 2(2), 27 (2007)
4. Grąbczewski, K., Jankowski, N.: Meta-learning with machine generators and complexity controlled exploration. In: Rutkowski, L., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) ICAISC 2008. LNCS (LNAI), vol. 5097, pp. 545–555. Springer, Heidelberg (2008)
5. Guyon, I., Gunn, S., Nikravesh, M., Zadeh, L.: Feature extraction, foundations and applications. Springer, Heidelberg (2006)
6. Guyon, I.: Nips 2003 workshop on feature extraction (December 2003), http://www.clopinet.com/isabelle/Projects/NIPS2003
7. Jankowski, N., Grąbczewski, K.: Learning machines information distribution system with example applications. In: Computer Recognition Systems 2. Advances in Soft Computing, pp. 205–215. Springer, Heidelberg (2007)
8. Kalousis, A., Hilario, M.: Model selection via meta-learning: a comparative study, pp. 406–413 (2000)