Increasing Efficiency of Data Mining Systems by Machine Unification and Double Machine Cache

Norbert Jankowski and Krzysztof Grąbczewski

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

Abstract. In advanced meta-learning algorithms and in general data mining systems, we need to search through huge spaces of machine learning algorithms. Meta-learning and other complex data mining approaches need to train and test thousands of learning machines while searching for the best solution (model), which often is quite complex. To facilitate working with projects of any scale, we propose intelligent mechanism of machine unification and cooperating mechanism of machine cache. Data mining system equipped with the mechanisms can deal with projects many times bigger than systems devoid of machine unification and cache. Presented solutions also reduce computational time needed for learning and save memory.

1 Introduction

All data mining systems are always limited by the amount of memory of the computer system used to solve given task and by the time assigned to solve the task. The limits will always exist. We can never spend unlimited time or use unlimited memory resources. Today, none of commonly known data mining systems like Weka [1], Rapid Miner [2], Knime [3], SPSS Clementine [4], GhostMiner [5] (see [6] for more) care so much about that limits. As a result, when given learning machine is needed again, it is constructed again. It is not a rare case that given machine is used several times. For example consider testing several configurations of classifier committees which share some member machines. Comparably frequent example is that a data transformation precedes several classifiers. In consequence, time is being lost for repeated processes, while machines sharing configurations should be built just once.

Another problem is observed when *learning from data* of large size. Memory quickly gets full, when checking a number of learning machine configurations (manually or by meta-learning). Of course, there are methods of dealing with huge data, but we want to point this problem from a little different point of view. It is possible to use advanced machine learning techniques on huge data using complex configurations of machines, if only the data mining system being used, is supported by specialized mechanism of intelligent machine caching.

L. Rutkowski et al. (Eds.): ICAISC 2010, Part I, LNAI 6113, pp. 380–387, 2010.

This is why we investigate the ideas of machine unification and machine cache in the following sections. All described elements have been implemented in Intemi—a general data mining system [7].

2 Machine Unification and Machine Cache

In general, the idea of unification lies in determination whether given machine (with given configuration and specification of input sources) has already been constructed (as a result of earlier request) and is available for reuse. It means that unification can reduce CPU consumption by not running any machine process twice. Up to now, no data mining system has proposed such feature, although it may significantly save computational power. Additionally, when it is not necessary to compute machine again, we do not use additional memory resources either, which would take place in the case of relearning of given machine.

To reuse already created machines, the machine cache is constructed. In fact, we propose two cooperating caches. One works in memory and cooperates with the other (a disk cache), which is used to extend the capacity of memory. But in contrary to swap mechanism nested in operating systems, it is not oriented in page swapping but in machine swapping, which is much more effective.

Machine, Configuration and Learning Task. Before we move into deeper detail about possibility of defining machine unification and construction of efficient machine cache, we introduce a general, formal view of learning machine.

By *learning machine* we mean any (implemented) instance of learning algorithm. The term *model* means the result of learning. For example, in consequence of running a classifier machine we obtain a ready to use classification model. It is more advantageous to extend the term of learning machine to more general term of *machine*. Machine need not be a *learning* machine. In this way, the same entity may embrace, for example, algorithms to standardize data, to provide cross-validation tests or even to load or import data. The common feature of all machines is that they have *inputs*, their inner configuration (states of free parameters) and that they provide *outputs*. Inputs and outputs reflect machine roles, the goal of machine: a classifier output plays the role of a classifier, data standardization routine play the role of data transformer etc.

Each *machine configuration* may be defined by

$$C := \langle p, io, \{C_i : i = 1, \ldots, n\}\rangle \tag{1}$$

where p represents machine process parameters, io specifies inputs and outputs (counts, names and types), and $\{C_i : i = 1, \ldots, n\}$ is a set of optional subconfigurations (enables construction of complex machines—each machine may have submachines).

The output of given machine may be *direct* (provided directly by given machine) or *indirect* (provided as a submachine output—it is defined by submachine path and output name). This is very important that outputs may be defined in

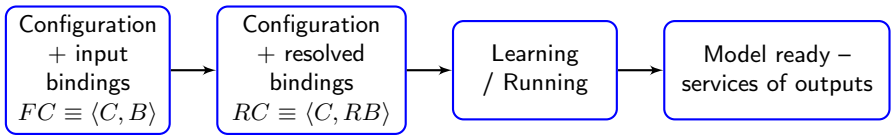

Fig. 1. Machine construction time line and learning task

so flexible way. Thanks to this, future complex machines do not need to reimplement each output but the output of a submachine may be simply exhibited.

Before given machine may be created, it must be completely defined by pair of machine configuration and its *input bindings*:

$$FC := \langle C, B \rangle. \tag{2}$$

The input bindings define symbolic input connections (input connections point symbolically appropriate source of input i.e. an output of another machine). Because machine may have several inputs, the input bindings B have a form of a set of pairs:

$$B := \{\langle input\ name, binding \rangle\}, \tag{3}$$

which assigns a binding to each input.

The inputs compose acyclic directed graph (machines are vertices and input–output connections are edges). On the other hand we may see complex machines as machine trees, because each machine may have submachines (the tree subnodes).

Input binding may have one of three types:

$$\begin{aligned} binding := {} & \langle parent\ input\ name \rangle \mid \\ & \langle id\ of\ sibling\ machine, sibling\ output\ name \rangle \mid \\ & \langle submachine\ path, output\ name \rangle \end{aligned} \tag{4}$$

It may be a connection to a parent input, to a sibling machine output or to an output of a machine defined by a submachine path (which may point a child, a grand-child, etc.). Such three types of bindings facilitate sufficient and flexible definition of input–output connections (even for extremely complex machines).

Machine construction timeline is sketched in Figure 1. When a fully defined machine (defined by configuration and input bindings) is requested, the request waits until all inputs become ready to use (the learning processes of machines which provide those inputs are finished). When all the inputs are ready, the input bindings are transformed to *resolved inputs*:

$$RB := \langle input\ name, rbinding \rangle \tag{5}$$

i.e. a collection of resolved inputs of the form:

$$rbinding := \langle machine\ stamp, output\ name, output\ stamp \rangle, \tag{6}$$

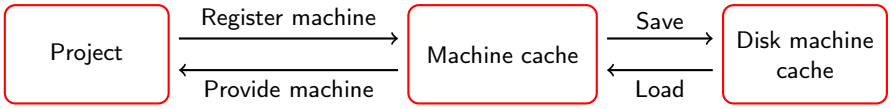

Fig. 2. Cooperation between project, machine cache and disk cache

Resolved input provides machine stamp[1] (identifying the machine that really provides the required output) and the name of output of that machine. Note that some outputs are indirect and resolved binding provides link to the (final) machine which really provides the output.

The input resolving process, converts the pair FC to:

$$RC := \langle C, RB \rangle. \tag{7}$$

The pair RC provides all information about configuration of the requested machine, necessary to run the machine process, so the request is either submitted to the task spooler (when it is the first request for the machine) or directly filled with proper machine reference (when such machine has already been constructed and may be reused). Below we discuss the latter case in more detail.

Machine Caching and Unification. The goal of machine cache is to save computational time and reuse previously constructed and trained machines. When a machine is requested and the cache can not return it instantly, because the machine has not been constructed and trained yet, the request becomes a task to be accomplished by proper task running server. Each machine, just after learning, is *registered* in the cache and each next request for the same machine will be instantly served by the machine cache. Please compare Figure 2.

The *provide machine* functionality is not trivial, because each machine type may be configured in different ways and get different inputs. Moreover, machines are often complex, so their configurations include subconfigurations (of different machines). Another important aspect of machine unification is that it can not be based just on machine configuration and input bindings FC as in Eq. 2, because such FC is not yet definitively determined. This means that because of dynamical symbolic definition of bindings and especially because of opportunity of defining indirect outputs for machines, the definite and unambiguous is the pair RC from Eq. 7. The resolved input bindings point (completely and directly) the machines that provide required outputs, and name the outputs. The FC may be converted to RC when all connected machines are ready to use. In consequence, machine requests have to wait until all inputs can be resolved (the inputs are ready to use). After conversion from FC to RC the project may *ask* machine cache to provide machine basing on the pair RC.

Another two important problems occur, when the machine cache is expected to keep thousands of machines (not just tens):

[1] The necessity of machine stamps will be clarified later.

– Memory for saving machines is limited. In real cases the number of machines which can be kept in memory is relatively small. This is solved by disk cache cooperating with the memory cache.
– Searching for machines in machine cache must be highly effective. This is achieved by efficient machine unification based on the *RC* pairs.

The disk cache as a general concept is not described in this paper, because of space limit. For the purpose of this article it is enough to know that disk cache provides functionality of machine saving and machine loading as it was depicted in Figure 2. The functions are used on demand via machine cache functions as can be seen below—see functions RegisterMachine and ProvideMachine.

The problem of machine unification can not be realized by means of plain comparison of two *RC* configurations. It would be too slow to compare searched *RC* pair with each *RC* pair in the cache. The search complexity depends linearly on the number of machines in machine cache, but when the *RC* pairs are complex, complexity is equal to the sum over all parameters of all *RC*'s in cache:

$$\sum_{rc \in cache} |rc|, \tag{8}$$

where $|rc|$ is the length of rc.

To make machine search much quicker than the naive solution, we have built specialized machine cache using three hash dictionaries for three types of mappings:

– unificator dictionary, mapping from *RC* pair to unique machine stamp. It means that the machine cache may provide appropriate machine only if the unificator dictionary contains appropriate *RC key*.
– unificatorRev dictionary, providing mapping inverse to unificator (from machine stamps to *RC* pairs).
– cache, mapping machine stamps to machines. It cooperates with the disk cache: before a machine is released from memory, it is first saved in the disk cache. Thanks to this, a single machine may be shared in many places (for example in several complex machines).

The three hash dictionaries obviously need fast calculation of hash codes, but as a result, they guarantee access in approximated complexity $O(|rc|)$ and independence from the number of machines in machine cache (very important for scalability of data mining systems).

Machine Cache Functionality. As mentioned above, each machine, just after it is ready, is registered in the machine cache. The registration is sketched in Figure 3.

In the first line, outputs of machines are registered, to simplify further unification and to accelerate the process of transformation of input bindings into resolved bindings. Basing on such construction, translation to resolved binding may be done just once for each output, even if the output is used many times.

```
1  function RegisterMachine(machine);
2      RegisterOutputs(machine.Outputs);
3      s = GetNextUniqStamp();
4      RC_pair = machine.GetRC();
5      unificator[RC_pair] = s;
6      unificatorRev[s] = RC_pair;
7      cache[s] = machine;
8      diskCache.PushSaveRequest(s, machine);
9  end
10
11 function ProvideMachine(RC_pair);
12     s = unificator[RC_pair];
13     if (s != 0) {
14         machine = cache[s];
15         if (machine == 0)
16             machine = cache[s] =
17                 diskCache.Load(s);
18     } else {
19         machine =
20             taskSpooler(RC_pair);
21         RegisterMachine(machine);
22     }
23     return machine;
24 end
```

Fig. 3. Listings of RegisterMachine and ProvideMachine

In line 3, new unique stamp is assigned to the newly prepared machine. Next, the resolved configuration RC and the stamp are used to define two mappings: from RC to stamp (in unificator dictionary) and vice versa (unificatorRev dictionary). After that, the final mapping from the machine stamp to machine is added to the cache dictionary. The last line of RegisterMachine pushes the request for asynchronous save of the machine to disk cache. Note that complexity of this procedure is $O(|rc|)$ except the subcall of procedure RegisterOutputs which complexity depends linearly on the number of machine outputs (usually a very small number).

The machine cache search for requested machine, basing on requested RC pair of configuration and resolved input bindings, is realized by ProvideMachine function—see Figure 3. First, the unificator dictionary is checked whether it already contains requested RC pair. If it does, then the machine is contained within the machine cache or at least within disk cache (if it has been released from memory). In such case the machine is extracted from cache (code line 14) or loaded back to memory cache (code line 17).

If machine has not been deposited in machine cache yet, then the task spooler is used to construct and learn machine basing on RC pair (see code line 20). After that, in the next line, the machine is registered in machine cache.

It is very important that each machine is provided in the way presented above, so that each two equivalent machines may be unified, even if one (or both) of them is a submachine. From the cache point of view each machine (regardless of whether they are submachines at any level) are handled in the same way. They are identified just by their stamps.

Another advantage of machine cache is that even if a machine is removed from the project or it is no longer a submachine, it is not necessary to remove it from machine cache. Next requests for this machine will find it ready for reuse.

Releasing machines. Machine cache periodically observes (with TryRelease function) the usage of each machine and depending on the usage statistics, it decides whether given machine can be released or not. Each time a machine is

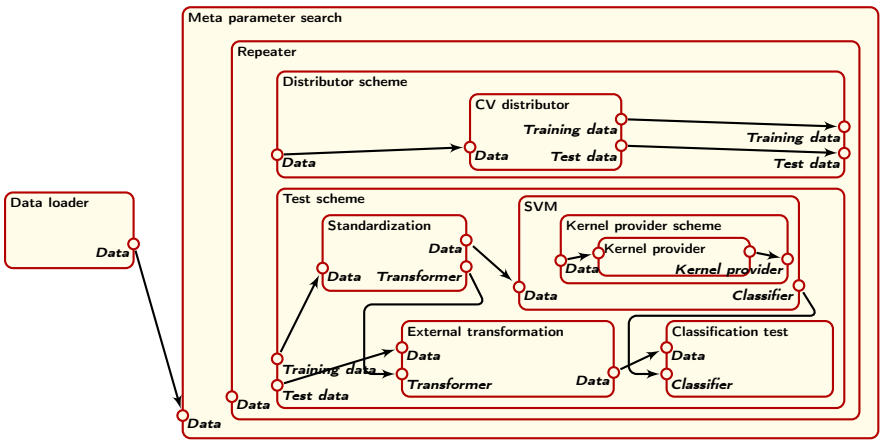

Fig. 4. A meta parameter search project configuration

constructed or becomes not used, its lastUseTime is updated and it is added to special list analyzed within TryRelease (see the code below). Of course, machine can not be released before it is saved in the disk cache.

```
25 function TryRelease(machineList)
26     foreach m in machineList
27         if (now − m.lastUseTime > timelimit && m.isSavedInCache) {
28             cache[m.stamp] = null;
29             machineList.Remove(m);
30         }
31 end
```

3 Unification Example

As mentioned in section 2, machine unification is especially advantageous in meta-learning. Even one of the simplest meta-learning approaches, a simple meta parameter search (MPS), is a good example. Imagine a project configuration depicted in Figure 4, where the MPS machine is designed to repeat 5 times 2-fold CV of an attractiveness test for different values of SVM C and kernel σ parameters. MPS machines are hierarchical and use different submachines to examine specified test tasks. To test the C parameter within the set $\{2^{-12}, 2^{-10}, \ldots, 2^{2}\}$ and σ within $\{2^{-1}, 2^{1}, \ldots, 2^{11}\}$, we need to perform the whole 5×2 CV process 8×7 times.

As enumerated in Table 1, such a project contains (logically) 4538 machines. Thanks to the unification system, only 1928 different machines are created, saving both time and memory. The savings are possible, because we perform exactly the same CV many times, so the data sets can be shared and also the SVM machine is built many times with different C parameters and the same kernel σ, which facilitates sharing the kernel tables by quite large number of SVM machines.

Table 1. Numbers of machines that exist in the project logically and physically

Machine	logical count	physical count
Data loader	1	1
Meta parameter search	1	1
Repeater	56	56
Distributor scheme	280	5
CV distributor	280	5
Test scheme	560	560
Standardization	560	10
External transformation	560	10
SVM	560	560
Kernel provider scheme	560	80
Kernel provider	560	80
Classification test	560	560
Sum	4538	1928

4 Summary

The concepts of machine unification and machine cache, presented above, are efficient and sufficiently general to be usable for many purposes in the scope of data mining. Discussed advantages clearly show that a machine realizing given configuration should be run once and next should be reused wherever requested, instead of running it many times, as it has been done in the past. Very attractive complexity of presented machine unification and disk cache management, saves memory and CPU time and facilitates much more sophisticated data analysis including advanced meta-learning.

References

1. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann, San Francisco (2005)
2. Rapid miner: Rapid miner 4.4 (2009), `http://rapid-i.com`
3. Knime: Knime konstanz information miner (2009), `http://www.knime.org/`
4. SPSS: Clementine—pasw modeler (2009)
5. Jankowski, N., Grąbczewski, K., Duch, W.: GhostMiner 3.0. FQS Poland, Fujitsu, Kraków, Poland (2004)
6. KDnuggets: Software suites for Data Mining and Knowledge Discovery (2009), `http://www.kdnuggets.com/software/suites.html`
7. 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)