Saving time and memory in computational intelligence system with machine unification and task spooling

Krzysztof Grąbczewski, Norbert Jankowski*

Department of Informatics
Nicolaus Copernicus University
ul. Grudziądzka 5
87-100 Toruń, Poland

Abstract

There are many knowledge-based data mining frameworks and it is common to think that new ones can not come up with anything new. This article refutes such claims. We propose a sophisticated unification mechanism and two-tier machine cache system aimed at saving time and memory. No machine is run twice. Instead, machines are reused wherever they are repeatedly requested (regardless of request context). We also present an exceptional task spooler. Its unique design facilitates efficient automated management of large numbers of tasks with natural adjustment to available computational resources. Dedicated task scheduler cooperates with machine unification mechanism to save time and space. The solutions are possible thanks to very general and universal design of machine, configuration, machine context, unique machine life cycle, machine information exchange, configuration templates and other necessary concepts. Results gained by machines are stored in a uniform way, facilitating easy results exploration and collection by means of a special query system and versatile analysis with series transformations. No knowledge about internals of particular machines is necessary to extensively explore the results. The ideas presented here, have been implemented and verified inside Intemi framework for data mining and meta-learning tasks. They are general engine-level mechanisms that may be fruitful in all aspects of data analysis, all applications of knowledge-based data mining, computational intelligence, machine learning or neural networks methods.

Key words: Knowledge-based systems, Data mining, Data mining tools, Computational intelligence, Meta-learning, Machine learning.

1. Introduction

Automation of advanced data analysis exploiting knowledge-based systems or Computational Intelligence (CI) has recently become a very important challenge. The community has formulated many algorithms for data transformation and for solving classification, approximation and other optimization problems (for a compact review see [1] or see some handbooks [2–11]). The algorithms may be combined in many ways, so that the tasks of finding optimal solutions are very hard and require sophisticated tools. Nontriviality of model selection is evident when browsing the results of NIPS 2003 Challenge in Feature Selection [12, 13], WCCI Performance Prediction Challenge [14] in 2006 or other similar contests. The competitions show that in real applications, optimal solutions are often complex models and require atypical ways of learning. Problem complexity is even clearer when solving more difficult problems in text mining or bioinformatics, where only good cooperation between different machines may provide a competitive solution. This means that before application of a final decision learner (for example a classifier) we have to prepare some transformations (and/or their ensembles) which facilitate success in further decision making.

To perform successful learning from data in an automated manner, we need some meta-knowledge

*Corresponding author. Tel.: +48 56 6113307, fax: +48 56 6221543

Email addresses: kg@is.umk.pl (Krzysztof Grąbczewski), norbert@is.umk.pl (Norbert Jankowski)

i.e. knowledge about how to build efficient learning machines providing accurate solutions to the problem being solved. Our interest is to provide tools for automated meta-level analysis, to support finding the most appropriate (usually complex) models for particular problems. The meta-task is independent from particular object-level tasks within data mining and computational intelligence. We present a general approach, applicable to any kind of learning problems.

The term *meta-learning* encompasses the whole spectrum of techniques aiming at gathering meta-knowledge and exploiting it in learning processes. Although many different particular goals of meta-learning have been defined, the superior goal is to use meta-knowledge to create more accurate models and/or to find them sooner. To reach such goal, a robust CI framework that efficiently manages time and memory must be used. Here, we show some aspects of our approach to time and memory efficiency as one of the pillars of successful meta-learning.

Some meta-learning approaches [15–18] base mainly on data characterization techniques (characteristics of data like number of features/vectors/-classes, features variances, information measures on features, also from decision trees etc.) or on *landmarking* (machines are ranked on the basis of simple machines performances before starting the more power consuming ones). Although the projects are really interesting, they still suffer from significant limitations. The whole space of possible and interesting models is not browsed so thoroughly, thereby some types of solutions can not be found with this kind of approaches.

We do not believe that on the basis of some simple and inexpensive description of data, it is possible to predict the structure and configuration of the most successful learner. Thus, in our approach the term *meta-learning* encompasses the whole complex process of model construction including adjustment of training parameters for different parts of the model hierarchy, construction of hierarchies, combining miscellaneous data transformation methods and other adaptive processes, performing model validation and complexity analysis, etc. So in fact, our approach to meta-learning is a search process, driven by heuristics (created and adjusted according to proper meta-knowledge) protecting from spending time on learning processes of poor promise and from the danger of combinatorial explosion. The problem of driving the search resembles the problems of context-aware design agents, where context is understood in a very broad sense including experience [19].

Meta-learning can be regarded as successful only if it efficiently uses the time it is given. It must be realized within as efficient CI environment as possible. Therefore, we have designed and implemented a general environment for complex machines learning and analysis. This article describes some of the crucial elements of the system with special emphasis on efficiency of time and memory usage. We introduce some completely new concepts in the realm of knowledge based systems including unprecedented machine unification system and dedicated approach to task spooling and running. Section 2 presents more detailed justification of the need for a new architecture and, at the same time, presents the main features of our system. Section 3 describes some of the substantial aspects of kernel design. Sections 4 and 5 present two aspects of the system that predispose it for meta-learning: task management and machine unification. Deep analysis of learning and testing results is possible thanks to the query subsystem presented in section 7. In section 8 we present our meta parameter search machine and its basic applications that illustrate the mechanisms described in preceding sections. All the solutions and applications have been realized in practice—they are not parts of a future project but of an existing and already working system. We do not include exhaustive tables of results obtained with the system, because the aim of this article is not to discuss particular results but to present some interesting mechanisms and illustrate how they work. The final section 9 summarizes and discusses future perspectives of the environment.

2. Why yet another data mining system was indispensable

In order to conduct robust meta-learning, we need a universal, versatile, but also efficient and easy to use framework. It must facilitate unhampered manipulation of complex machine configurations and learning results. Because in meta-learning we must perform huge amounts of tests and compare them reliably, we need a framework capable of avoiding multiple calculations of the same tests and facilitating robust comparisons between old and new calculations, so learning must be performed for the same data samples etc. Therefore,

we need a system providing all of the following features:

1. Engine-level architecture:
   (a) A unified encapsulation of most aspects of handling CI models like learning machines creation, running and removal, defining inputs and outputs of adaptive methods and their connections, adaptive processes execution, etc.
   (b) The same way of handling and operation of simple learning machines and complex, heterogeneous structures. Easy definition, configuration and running of machine hierarchies (submachines creation and management).
   (c) Easy and uniform access to learners' parameters.
   (d) Easy and uniform mechanisms for representation of machine inputs and outputs and for universal information interchange.
2. Task management:
   (a) Efficient and transparent multitasking environment for processes queuing/spooling and running on local and remote CPUs.
   (b) Versatile time and memory management for optimal usage of the computational resources.
3. Results acquisition and analysis:
   (a) Easy and uniform access to exhaustive browsing and analysis of the machine learning results.
   (b) Simple and efficient methods of validation of the learning processes, conducive to fair validation i.e. not prone to testing embezzlements.
   (c) Variety of tools for estimation of model *relevance*, analysis of reliability, complexity and statistical significance of differences [20].
4. Machine management efficiency:
   (a) Mechanisms of machine unification and a machine cache system preventing repeated calculations (significant in so large scale calculations like meta-learning).
   (b) Multilevel random seed management to facilitate providing the same learning environment and data for different learning machines.
5. Others:
   (a) Templates for creation and manipulation of complex–structure machines, equipped with exchangeable parts, instantiated during meta-learning.
   (b) Rich library of fundamental methods (especially learning machines) providing high functionality, versatility and diversity.
   (c) Simple and highly versatile Software Development Kit (SDK) for programming system extensions.

There are plenty of data mining or knowledge-based systems available today [21–24]. Software packages or systems like Weka, RapidMiner, Knime, SPSS Modeler (former Clementine), GhostMiner etc. are designed to prepare and validate different computational intelligence models, but we have found none, that would have implemented all the features mentioned above or would facilitate an extension to support the features without significant rearrangements within the architecture of the kernel. Moreover, some of the features, we propose, have not been provided by any of the systems mentioned above. These features are not just attractive—they are necessary to provide advanced and efficient data analysis using advanced meta-learning techniques.

Commercial products (like SPSS Modeler), are addressed rather to business users than to academic researchers, so they are designed as tools for data analysis on the basis of obtained models and as such they often do not reveal their internals in a satisfactory way for efficient meta-analysis. The fundamental purposes are different in the case of open source packages like Weka, RapidMiner or Knime, but they do not support the features listed above in a satisfactory way either. Some systems are limited to one research field, e.g. SNNS [25] is limited to some algorithms around neural networks. In most such systems, developing all the features listed above is practically impossible.

Description of all the advantages and drawbacks of a number of popular systems would require a lot of space and is not the purpose of this article. Here, we just point some major drawbacks of the most popular frameworks to better show the needs of advanced meta-learning approach. To avoid arbitrary judgments with insufficient explanation, we describe the most important aspects instead of presenting a raw (questionable) feature table. Although our decision to create a new system from

scratch was undertaken a couple of years ago, below we relate our prerequisites to the current state of other systems, to make the analysis easier to verify. In fact, most of our arguments are still valid today, so comparing with the current state is definitely more reasonable.

Ad. 1: All data mining frameworks try to unify miscellaneous processes, to handle them in a uniform way, so they usually address the aspects 1a–1d in some way, but the approaches are quite diverse. The fundamentals of different engines look similar: in our approach, projects consist of interconnected *machines* (see section 3), while RapidMiner components are called *operators* and Knime ones are *nodes* and also have some interconnected I/O ports, but the scopes of entities included by the terms makes the systems significantly different from each other. RapidMiner and Knime went so far in their unified views that their operators and nodes include both advanced learning machines and simple functions related to results visualization and statistical tests like t-test. Knime even contains a node responsible for assigning colors to different categories for the purpose of further visualization. In our approach, we have also enclosed many different processes into machines (see sec. 3), but created separate tools for results manipulation and visualization. Separating them into different levels supports easier creation and analysis of complex processes.

Similarly, it is very advantageous to notice (and reflect in the system architecture) the differences between configuration time and run time of complex processes, especially those including repeated subprocesses. No other system has introduced similar distinction so far. The lack of such solutions makes projects less intuitive. For example, both RapidMiner and Knime projects containing cross-validation tests are not natural to handle, because single subprocess hierarchy is adequate to the configuration time but not to run time form of the process. As a result, at run time, only the last instance of the subprocess is available, small configuration changes may require recomputation of the whole process or a large part of the project, including items which are not affected by the change etc.

The differences at low-level system organization also imply significant differences in user interaction with the system. Introducing such nodes like Knime's "Color Manager" may result in the necessity of running the project in parts, with some user interaction between them, because it may be impossible to configure such nodes properly before other nodes have provided their results. Necessity of user interaction in the middle of the project is equally inconvenient as no possibility of user interaction during the life of the project (the case of RapidMiner). The user must be allowed to interact with the project but must also be able to design complex data analysis projects which can be run with no human supervision.

The aspect 1c is nowadays mostly a matter of programming language selection as some languages provide tools for meta-level analysis of their code (meta-level of the programming language). Currently, C# is the most advanced programming language regarding the meta-level analysis possibilities, hence we have chosen it for our system. C++ provided just a little bit (its RunTime Type Identification system). Java can also be satisfactory from this point of view, so it was chosen as the language of RapidMiner and Knime source codes.

Ad. 2: Meta-learning, like most advanced data mining tasks, requires bulk computing, so it is very desirable that the system can run subprocesses on many different CPUs. The authors of Knime kept in mind such requirements from the start of the system, but RapidMiner, being a continuation of Weka and Yale efforts, does not provide satisfactory tools for parallelization.

Engine-level solutions have important influence on many other aspects of higher level system behavior. RapidMiner introduces tools for manual ordering of the operators creation, as its architecture does not allow for fully automatic determination of the order in all cases.

In earlier versions of RapidMiner, parallelization was left completely to the authors of operators. Operators like `ParallelXValidation` or `ICA` were solving multithreading on their own. Although with the advent of version 5, the approach is different, the engine-level solutions still do not solve the problems with determination of the order of running operators.

Our approach to the output-input bindings facilitates fully automated management of machine creation and parallelization for arbitrarily complex machine hierarchies. The most important mechanisms of our task management system are sketched in section 4.

Ad. 3: Efficient results analysis is another very important feature of a tool to be used for meta-learning. The results system is closely bound up with other engine-level solutions. The problems with management of multilevel hierarchies of oper-

ators and nodes make the results systems of RapidMiner and Knime not flexible enough. We usually have to foresee all the results that will be needed further and all the interesting results must be deposited at run time, since after the whole process is finished, only the last instance of each operator/node is available. GhostMiner keeps all machines (when desired), so it makes all results available, but provides no general framework for results access and analysis. From meta-learning perspective it is important to access different kinds of results in a common manner so that no specific knowledge about particular machines is required to collect and analyze their results. No available system provides satisfactory tools for that, so we have designed a uniform results repository and a query system for this purpose. They are briefly presented in section 7.

Data mining frameworks should also support fair validation and comparison of results obtained with different methods. They should not suggest unfair solutions (though preventing all embezzlements is not possible). Therefore validation mechanisms should not separate data preprocessing from proper model construction, because fair validation must treat any sequence of supervised methods as a whole and validate it as a whole. For example, Weka used to exhibit distinction between data preprocessing and proper analysis which may seem to justify some incorrect ways of data mining project construction. Even some books on data mining suggest almost total separation between preprocessing and classification or prediction. For more about incorrect methodology in data analysis see section 7.6.

Ad. 4: Up to our knowledge, there is no other data mining system providing automated mechanisms for machine unification. Even in a single project, when two identical models are scheduled they are built twice and two identical copies reside in memory. Of course, the project designer may prevent from repetitions where possible by proper project construction, however there are always circumstances, where some repetitions can not be foreseen (e.g. in feature selection tasks). With efficient machine unification, we may restart the project (also with some changes within it) and all repeated parts are reused saving time and memory. How we have reached the goal of run time machine unification is described in section 5.

The functionality of machine unification would be seriously restricted without a seed control mechanism for random processes designed at system-level. To make comparisons of complex multi-level tests most reliable and reasonable, the seeds of random processes must be set in the same way for separate runs (repetitions/folds) of the tests. For example, when repeating cross-validation tests several times for better error estimation, subsequent data splits must be different for each CV repetition, but the whole sequence of seeds should be the same for different test runs in order to use paired t-test or other statistical tests for paired samples. In RapidMiner one can use global or local seed in particular operators, but no automated intelligent seed policy can be performed for complex multi-level structures of operators. Knime does not care for such random processes control, and the task is left to the authors of particular operators. The more so, mechanisms of automated management of multi-level node structures are lacking. One may use macros in RapidMiner or variables in Knime to implement the desired behavior for a particular project, but this way, each project requires a lot of additional effort to obtain the goal.

Our design facilitates automated and versatile seed control for random processes of any complexity. It is addressed in section 5.3.

Ad. 5: Out of many little solutions supporting meta-learning, machine templates (feature 5a) are worth a special mention, because they help configure and manage complex machine structures. We introduce them in section 3.5 and preliminarily use in the meta parameter search machine (section 8) which can be seen as one of the first steps toward proper meta-learning. The features 5b and 5c concern system libraries and are out of the scope of this article, which puts stress on fundamental mechanisms facilitating efficiency of advanced learning processes within computational intelligence.

To conveniently and efficiently perform meta-learning tasks, all the listed features should be incorporated into a uniform and robust system. Theoretically, each system available today could be developed to fulfill our requirements, but according to our estimation, the extent of effort to do so would be too large, even in the case of GhostMiner which we know from cover to cover because we have designed and implemented it. Instead of wrestling with shortages and bottlenecks of existing systems, we have designed a new system from scratch, keeping in mind all the meta-learning requirements (the substantial ones listed above and

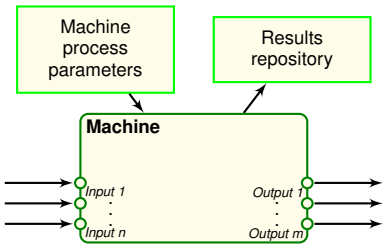

Figure 1: The abstract view of a machine.

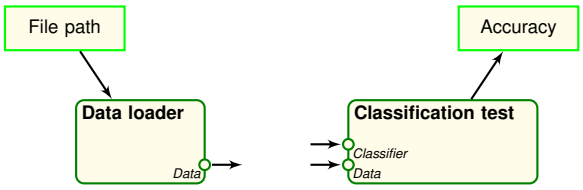

Figure 2: Machine examples: a Data loader and a Classification test.

many others, maybe less spectacular).

Some of our solutions are presented in more detail in the following sections.

3. System architecture and information exchange

Designing an architecture fulfilling so many expectations as those listed in the preceding section is not a trivial task, especially, when one of the strong requirements for the system is simplicity of use. Fortunately, proper system kernel foundations reconciled the two requirements, despite they seem contradictory.

The main key to the new possibilities is the unified view of *machine* and *model*. First of all, it is advisable to distinguish the two terms to avoid ambiguities. A machine is any process that can be configured and run to bring some results. A model is the result of such a process. For example an MLP network algorithm (the MLP machine) can be configured by the network structure, initial weights, learning parameters etc. It can be run on some training data, and as a result we get a trained network—the MLP model created by the learning process of the MLP machine.

We deliberately avoid the term "learning machine", since in our approach a machine can perform any process which we would, not necessarily, call a learning process such as loading the data from a disk file, data standardization or testing a classifier on external data.

A general view of machine is presented in figure 1. The *machine configuration* consists of:

- specification of inputs and outputs (how many, names and types),
- machine process parameters,
- submachines configuration (it is not depicted in figure 1 to keep the figure clear; in further figures, starting with figure 3, the submachines are visible as boxes placed within the parent machine).

More formally, each machine configuration is defined as

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

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

Given *parameters* of a machine process together with submachines configurations and *inputs* to be precessed, the machine runs to create its model. The results of the process may be exhibited as *outputs* and/or deposited in a collection called *results repository*.

The inputs and outputs serve as sockets for information exchange between machines. The difference between machine inputs and parameters is that inputs come from other machines while the parameters are specific to the process and are provided directly by the user. Similarly, outputs exhibit parts of the model to other machines (to be passed as their inputs, when proper input–output connection is established). Results repository serves as kind of report from machine run and contains an excerpt from the model. It is up to the machine author whether the machine receives any inputs, whether it has some adjustable parameters and whether it has outputs and/or puts results in the repository.

Two examples of simple machines are presented in figure 2. The Data loader machine receives no inputs and declares a single parameter which is a string containing the file name from which the data

is to be loaded. The machine process exposes the data series as the output and deposits no entries into the results repository.

The machine of Classification test, presented in figure 2 on the right, takes inputs but no parameters. The inputs introduce the classifier (more exactly an interface with the classification routine) to be tested and the test data series. As a result we get information about accuracy of the classifier. The machine provides no outputs, because the information it gains, is not expected as input of any other machine.

In the following examples, we will not expose machine parameters or results deposited in the repository, as they are not so important from the point of view of the presentation. More important is the information flow, so inputs, outputs and their interconnections will be thoroughly presented.

The unified concept of machine does not introduce any kind of machine type, so we do not split machines to classifiers, data loaders, data transformers etc. Instead the inputs and outputs of a machine define possible contexts of its application i.e. any machine providing an output of type *Classifier* may be called a classifier and used in the context of a classifier, for example may be tested by the Classification test machine when its *Classifier* input is bound to the *Classifier* output of the former machine. Thus, a single machine may be useful in a number of ways. For example a decision tree may expose a *Classifier* output and also a *Feature ranking* output (generated on the basis of features occurring in the conditions defining the tree nodes). It will let the decision tree machine occur both in the context of a classifier and a feature ranking.

3.1. Scheme machine

A very important machine, from the organizational point of view, is the Scheme machine (see an example in figure 3). It is a machine with a function of a machine container or machine group. The process of a scheme machine simply runs all the submachine processes. The submachine relation is depicted as placement of the submachine within the area of the parent machine. The input-output interconnections of figure 3 are a part the scheme machine parameters, since they are just instructions about how to bind inputs when creating submachines.

The scenario configured within the scheme in figure 3, assumes two data collections coming through the inputs (one for training and one for test). Two

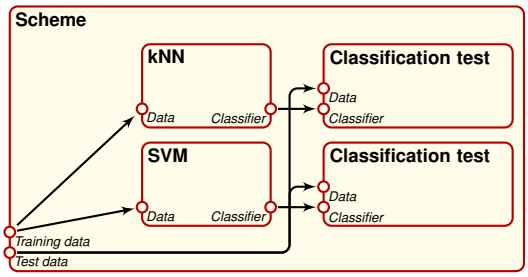

Figure 3: A scheme machine example.

learning machines: k Nearest Neighbors (kNN) and Support Vector Machine (SVM) are to be trained on the training data and their classifier outputs tested by separate test machines on the test data. Such a scenario is very useful for detailed comparison of the results of the two learning algorithms, because they are trained on exactly the same data series and tested on the same data, so each particular decision can be reliably compared (see section 7).

3.2. Transform and classify machine

The schemes may be used by other machines to facilitate user configuration of scenarios to be performed to obtain required results. For example, a machine performing data transformation and classification called Transform and classify (T&C) presented in figure 4, defines two scheme submachines to be filled by the user at configuration time. Such definition makes the T&C a general machine capable of performing any data transformation and classification scenarios. For example, one can put a standardization machine inside the Transformation scheme and an SVM machine into the Classifier scheme to perform SVM classification after data standardization (the lower part of figure 4). The interconnections within the two schemes (submachines of T&C), define the behavior within the schemes. There are no interconnections between the inputs of the parent machine and submachines or between the outputs of submachines and the outputs of the parent machine, because they are not configurable—the T&C machine will take care of the appropriate connections at run time. The schemes inside the T&C machine can be filled with an arbitrarily complex scenario, not just a single machine. The only requirement is that all the outputs of the schemes are bound to some outputs of contained machines.

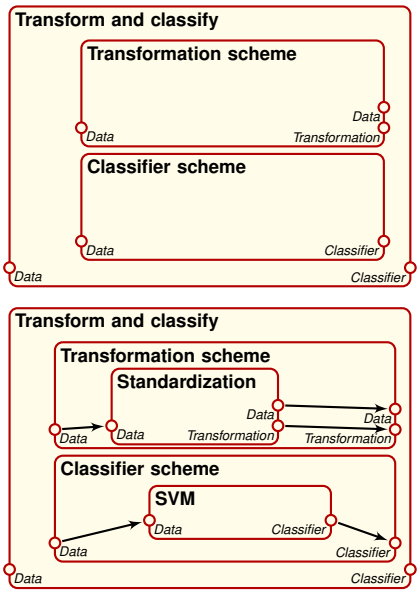

Figure 4: A configuration of the Transform and classify machine (raw and filled).

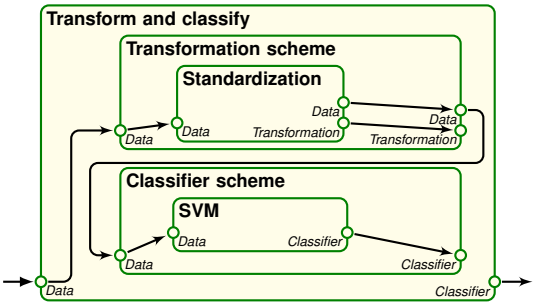

Figure 5: T&C machine at run time.

At run time, the machine gets a form depicted in figure 5. Beside the connections defined at configuration time, the internal schemes' *input bindings* can be seen—they were decided by the parent machine according to the aim of the submachines: the transformation is run on the incoming training data and the classifier is trained on the transformed training data. When the classifier output of the main machine is questioned to classify a series of data objects, it first transforms the data using the *Transformation* output of the transformation scheme (in fact of the standardization machine) and then classifies the transformed data with the *Classifier* output of the classifier scheme (in fact SVM output). This procedure guarantees that the test data

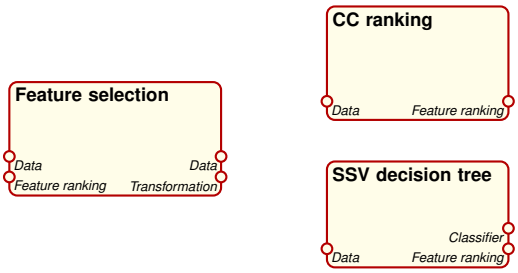

Figure 6: Configuration of the Feature selection machine and two feature ranking machines.

series is standardized (with respect to the statistics of the training data) and then classified. The T&C machine can be used wherever a classifier is expected (in the same contexts as, for example an SVM), because of its *Classifier* output.

Additional advantage of the Transform and classify machine is that such form of cooperation between machines is also suitable for further optimization and test procedures. For example, classifiers may be easily combined with feature selection methods (as it can be seen below) and the parameters like the number of features can be optimized in a simple and universal way, using machines like MPS (see section 8).

3.3. Feature selection and rankings

The family of feature selection methods based on ranking measures is another interesting illustration of the information flow between machines. The object oriented encapsulation idea leads to a split of the feature selection process into two stages: generation of feature ranking and selection of the adequate number of the top-ranked features. It results in just one feature selection machine for all the feature selection purposes and a number of feature ranking machines implementing particular algorithms.

As shown in figure 6 the feature selection machine declares two inputs (the data and feature ranking) and outputs two interfaces (providing data series obtained by filtering the input data to keep just the selected features and the transformation of feature selection for external data transformation). How many features are to be selected is defined within the feature selection machine parameters: arbitrary number of top-level features or the features with the ranking value exceeding given threshold. Naturally, there is no single rule for optimal setting

of these parameters for all data (all problems), so they must be determined separately for each data. It can be obtained with meta-learning: sometimes so simple methods as meta parameter search presented in section 8 are satisfactory and sometimes more advanced approaches combining feature selection with other data transformation methods are inevitable [26, 27].

3.4. Repeater machine

In data mining, we very often need to perform similar tasks many times. A typical example is cross-validation technique used for validation of model parameters or to test generalization capabilities of an algorithm. For such purposes, we have created a general machine named Repeater. Initial view of the repeater configuration is presented in figure 7 on the left. It declares two subschemes: one for inputs generation (Distributor scheme) and one for a scenario to be repeated (Test scheme). A single cycle of the repeater job is to generate the distributor according to the first subconfiguration, and pass its proper outputs to a number of subsequent instances of the test scheme. The distributor's outputs are so called multi-outputs i.e. collections of output objects. The collections size (must be the same for all distributor's outputs) defines the number of instances of the test scheme that will be created by the repeater. Moreover, the repeater has a parameter defining how many times the whole scenario will be repeated.

The example presented in figure 7 on the right is a repeater configured to perform a comparative CV test of kNN and SVM machines. The CV distributor receives the data as input and splits it into a number of training data sets and the same number of test data sets, according to the rule of n-fold cross-validation. During configuration of the repeater, adding the CV distributor to the distributor scheme results in propagation of the outputs from the distributor to the scheme and also to test scheme inputs, so that the test scheme can also be properly configured. Requesting a machine of this configuration produces machine hierarchy similar to the one presented in figure 8. Performing twice (independently) 2-fold CV we generate two distributors (one for each CV cycle) and four test schemes (two per CV cycle). The CV distributor outputs are two training sets and two test sets—the first elements go to the inputs of the first test scheme and the second elements to the second scheme.

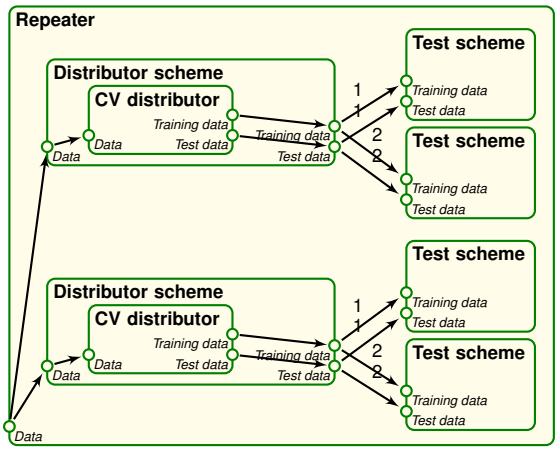

Figure 8: Run time view of Repeater machine configured to perform twice 2-fold CV. Test schemes are simplified for clearer view—in fact each one contains four submachines as in figure 7.

3.5. Configuration templates

Schemes are also very useful from the point of view of meta-learning. They may play the role of placeholders for machine configurations (of single machines or complex machine scenarios). Such incomplete configurations are called *machine configuration templates*. With simple substitutions, templates can be easily converted into fully specified (feasible) machine configurations. It is especially important during meta-search to easily generate and test different machine configurations. An example of a template is the raw configuration of Transform and classify machine presented in figure 4. It contains two empty schemes, which can be seen as "placeholders" for subconfigurations performing functions defined by the inputs and outputs of the scheme. As a consequence, the raw T&C configuration may be regarded as a template of configurations.

Similar empty scheme may be used to parameterize configuration of boosting algorithm, where a placeholder for a classifier may be filled in proper time. It means that the raw configuration of boosting is also a template.

Another substantial template is the one for performing feature selection (see figure 9). The dashed box represents a placeholder for a ranking. In practice, it can be a scheme with adequate definitions of the input and the output, so that all other elements of the configuration (the interconnections) may be realized. After replacing the Ranking scheme by a

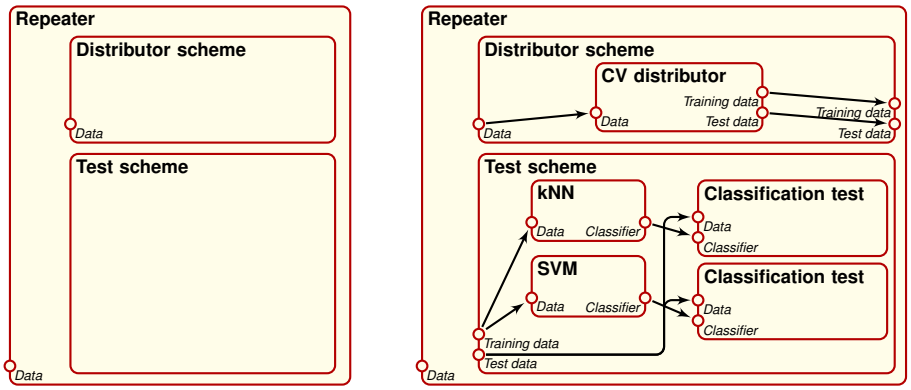

Figure 7: Repeater machine configuration (raw and filled).

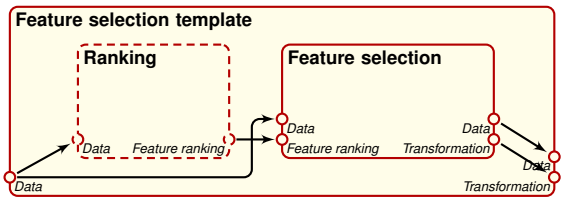

Figure 9: Feature selection template.

feasible scenario the whole construct can be created and run or put into another configuration (for example into the Transformation scheme of the T&C machine configuration.

Just the two templates described above, augmented with information about available simple machines for feature ranking, some other data transformations (like standardization) and classification learning machines, comprise quite a robust tool for machine configuration generation and as a consequence for robust meta-learning.

4. Machine life cycle

Thanks to the unified view of machine, we can simply state, that the fundamental task of a data mining system is to create and run machines in given contexts. Therefore, in our approach, each *request for machine* precisely defines *machine configuration* (as defined and discussed in section 3) and *machine context* which is the information about:

- the parent machine (handled automatically by the system, when a machine orders creation of another machine) and the child index,
- input bindings i.e. the specification of other machines outputs that are to be passed as inputs to the requested machine.

The system serves the request so as to assign adequate machine to it and, if necessary, run the machine process to make the machine ready for further analysis of its outputs and the information deposited in the results repository. Thanks to the concept of machine context, the system may assign the same machine to many contexts. Then, such machine may provide its services and be analyzed in different contexts within arbitrarily complex machine structures. As shown further, sharing machines in different contexts yields significant savings in CPU time and memory.

The possible paths, machine requests go through, are presented in figure 10. The flowchart presents different states of the requests while the dashed lines encircle the areas corresponding to particular system modules.

4.1. Input bindings

Each machine request is first analyzed to determine the machine contexts providing inputs to the requested machine.

This is why at the configuration stage each input has to be bound to an output of another machine (or a number of outputs). The input bindings may be defined as:

$$B = \{\langle input\ name, \{binding\}\rangle\}, \tag{2}$$

where $\{binding\}$ denotes a set of bindings.

In consequence, *full configuration* of machine is defined by pair of configuration and their bindings:

$$FC = \langle C, B\rangle. \tag{3}$$

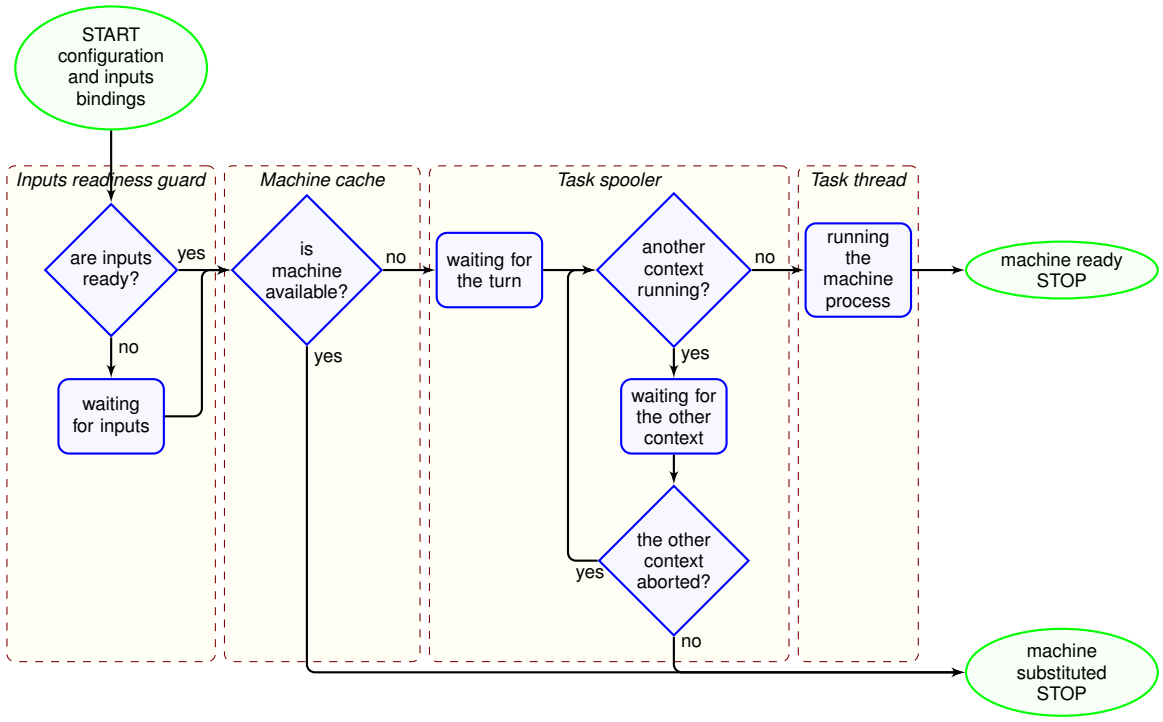

Figure 10: Machine request life cycle.

An input may be bound in one of three ways including binding to a parent's input (as it was often the case in the figures presented so far), to a sibling's output (an output of a machine that does not exist at the time of configuration) or to the capsule (which can be seen as an path in the machines subtree, from a machine called the root of the path to its ancestor following the line of direct parent-child dependencies). More formally:

$$
\begin{aligned}
&\mathit{binding} = && (4)\\
&\quad \langle \mathit{parent\ input\ name} \rangle \mid && (5)\\
&\quad \langle \mathit{id\ of\ sibling\ machine}, \mathit{sibling\ output\ name} \rangle \mid\\
&\quad \langle \mathit{submachine\ path}, \mathit{output\ name} \rangle
\end{aligned}
$$

Therefore, the abstract information must be transformed into outputs specification containing machine context references. After that, the request is passed to the *input readiness control* module, where, if necessary, it waits for other machines (the machines providing inputs) processes to finish. When all the inputs are ready, the request is examined by *machine cache*, whether the machine is already available because of earlier calculations. If the unification is not possible the machine request goes 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 the parent machine or by the system user. This may influence the flows of other requests for the same machine. Below we discuss the crucial stages of machine request management.

4.2. Inputs readiness guard

To facilitate optimal parallelization of running machine processes, the requests must be deposited in the task spooler as soon as all the inputs for particular machine are ready. Therefore, the input readiness guard controls, in real time, all the changes in machine readiness and submits runnable requests to the spooler. The most important functionality of the module is performed by two methods: Control and MachineFinished. They are launched by the events of new machine requests and machine processes finalization, respectively. The sketch of their behavior is presented in the following pseudocode:

```
1 void Control(MachineContext machineContext)
2 {
```

```
3   Set_of_contexts dependenceContexts,
4     notReadyContexts;
5   dependenceContexts =
6     machineContext.DependenceContexts();
7   notReadyContexts =
8     FilterNotReady(dependenceContexts);
9   if (notReadyContexts.Count == 0)
10    ProvideMachine(machineContext);
11  else
12    AddWaiting(machineContext,
13      notReadyContexts);
14 }
15 void MachineFinished(Machine machine)
16 {
17   foreach (context bound with machine)
18     foreach (waitingContext
19       awaiting machine outputs)
20     {
21       Bind2ResolvedBind(waitingContext);
22       if (waitingContext has all inputs ready)
23         ProvideMachine(waitingContext);
24       RemoveWaiting(waitingContext, machine);
25     }
26   RemoveWaiting(machine);
27 }
```

When a machine context is defined and passed to the inputs readiness guard, all the machine contexts, being input providers for the machine, are determined (line 6). If all of them are ready, the method ProvideMachine is called (line 10) to pass control over the context to another module (here, to the machine cache, for analysis of unification possibilities). Otherwise, the information about the context and its dependencies is added to proper internal structures for further control.

When a machine process run is finished, a signal is sent to the inputs readiness guard and it's MachineFinished method is called. The method tests all the contexts waiting for the machine, and triggers ProvideMachine method (line 23) if the machines outputs are the only outputs awaited by given context. Because machine outputs may be awaited through its different contexts, all the contexts (and their output readers) are examined. The information about waiting machines is maintained up-to-date with calls of RemoveWaiting methods of the pseudocode.

4.3. Resolved input bindings

The goal of function Bind2ResolvedBind in line 21 is to transform the binding from the symbolic form to the *resolved* form—the final form which points directly the appropriate output. At start, the machine contexts providing inputs were found, but not always the machines contained within these contexts are the real output providers of interest. For example, an output of a scheme is just a "transit" from another machine output. The same occurs when a machine exhibits output which indeed is an output of a submachine. *Resolved input bindings* contain precise information about the machines providing inputs to the requested machine. This information is especially important from the point of view of optimization of machine information exchange in distributed processing.

In that process each *binding* is transformed to its resolved form:

$$rbinding = \langle machine\ stamp, output\ name \mid output\ stamp \rangle, \quad (6)$$

When all the inputs of a requested machine are ready (the if-condition in line 22 becomes true), they are *resolved*, i.e. the machines, that actually provide the inputs are determined:

$$RB = \langle input\ name, \{rbinding\} \rangle \quad (7)$$

The necessity of that process will be visible also in section 5.

4.4. Unification within the machine cache

From the point of view of a machine request life, the unification stage is very simple: the machine cache is checked, whether exactly the same machine has been requested (and successfully finished) before, which means that the machine of requested configuration and assigned resolved inputs already exists in cache and may be reused. The advantages of the machine cache and the unification process, with special emphasis on meta-learning applications, are discussed in more detail in section 5.

4.5. Task spooler

A machine context, that can not be unified with any context analyzed earlier, is equipped with proper task information and the task is pushed to the *task spooler*, where it waits for its turn and for a free processing thread. The process of task spooling is quite nontrivial because different requests may be subject to unification also within the spooler, canceling a task must be properly managed (to not cancel other requests for the same machine) etc.

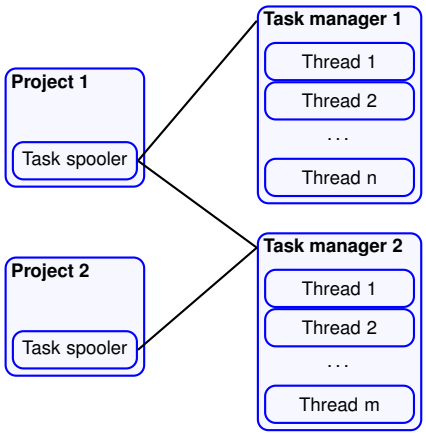

Figure 11: Two projects and two task managers.

Also the structure of the spooler is very important from the point of view of efficiency of calculations. More detailed discussion of these ideas is presented in section 6.

4.6. Task running

Machine requests are popped from the task spooler by task managers (computation servers) to be run within a thread. The idea is depicted by figure 11. Each project can subscribe to services of any number of *task managers* executed either on local or remote computers. Moreover subscribing and unsubscribing to task managers may be performed at project run time, so the CPU power can be assigned dynamically. Each task manager serves the computational power to any number of projects. Task managers run a number of threads in parallel to make all the CPU power available to the projects. Each project and each task manager presented in figure 11 may be executed on different computer.

A task thread runs machine processes one by one. When one task is finished, the thread queries for another task to run. If a task goes into waiting mode (a machine requests some submachines and waits for them) the task manager is informed about it and starts another task thread, to keep the number of truly running processes constant.

Machine tasks may need information from other machines of the project (for example input providers or submachines). In the case of remote task managers a *project proxy* is created to supply the necessary project machines to the remote computer. Only the necessary data is marshaled, to optimize the information flow.

Naturally, all the operations are conducted automatically by the system. The only duty of a project author is to subscribe to and unsubscribe from task manager services—each is just a single method call.

5. Machine unification and machine caches

In advanced data mining project it is inevitable that a machine with the same configuration and inputs is requested twice or even more times. It would not be right, if an intelligent data analysis system were running the same adaptive process more than once and keep two equivalent models in memory. Therefore, we have introduced machine contexts as objects separate from proper machines. Different contexts may request for the same machine and may share the machine. This general goal is realized by two cooperating mechanism: the machine unification system and the two-tier machine caches.

Different requests are guaranteed to result in the same machine if:

- configurations of the machines (parameters of the processes) are equal and
- the inputs are bound to the same or equivalent outputs.

Such requests may be *unified* which means that they may share the machine—the request served later gets the machine from the cache and next instance of the machine is not run.

5.1. Machine unification

Unification possibility may be judged after the machine inputs are ready. It means that we can not determine the equality of pairs $FC = \langle C, B\rangle$ (Eq. 3), but we need to check equivalence of the resolved configurations. It is obtained in the input resolving process, in which the pair FC is converted to *resolved configuration*:

$$RC = \langle C, RB\rangle. \tag{8}$$

where RB describes the resolved inputs (defined in Eq. 7).

The equivalence of machine configurations (the C part of FC) does not change, but only provided the resolved bindings (which point the final outputs) we can determine the equivalence. The control performed at the appropriate time guarantees that no unification possibility is overlooked.

Proper definition of machine configuration includes a method to determine equality of two parameter structures of the same type, which is used by the unification system. The term "machine parameters" includes information about pseudo-random processes performed by the machine (if any) and subconfigurations (if present). Therefore, we have formalized the way machines should deal with randomness. It is described in subsection 5.3.

Comparisons of sets of inputs are performed in two stages to reflect "same" and "equivalent" inputs respectively: first it is checked whether they refer to exactly the same output objects and then, if the input bindings seem different after the first check and the output types implement proper interface, they are compared with adequate method of the interface.

When the two conditions of machine unification are satisfied, new machine is not created and the new context just refers to already existent machine.

Machine unification must be controlled at different stages of machine life (depicted in figure 10). Appropriate check takes place before machine is sent to the spooler. If another request for the same machine has been fully served before (i.e. the machine is already in the cache), than its machine can be instantly assigned to the new context, and the new request is not spooled. When two (or more) requests are unified before any of them is finalized, then two tasks (of the same machine creation) are pushed to the task spooler (with different priorities). Spooler services control the requests to prevent running two requests for the same machine in parallel and producing two identical machines.

More details about the spooler, also in the context of machine unification, are presented in section 6.

For efficient realization of unification, another two important problems had to be solved. The problems would come out, when the machine cache had to keep thousands of machines (not just tens):

- Memory for saving machines is limited. In real cases, the numbers of machines which can be kept in memory are relatively small. This is solved by a disk cache cooperating with the memory cache (see section 5.2).

- Searching for machines in machine cache must be highly effective. This is achieved by efficient machine unification mechanisms described just below.

Firstly: 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 the searched RC pair with each RC pair in the cache. Secondly: when keeping machines in plain structures the search time would depend linearly on the number of machines in machine cache, but when the RC pairs are complex, the complexity is proportional to the sum over all parameters of all RC's in the cache:

$$\sum_{rc \in cache} |rc|, \tag{9}$$

where $|rc|$ is the length of rc. Such solution is not acceptable.

To make machine search very much quicker, we have built a specialized machine cache using three hash dictionaries to realize three types of mappings:

- unificator dictionary, mapping RC pairs to unique machine stamps. It means that the machine cache may provide appropriate machine only if the unificator dictionary contains appropriate RC *key*.

- unificatorRev dictionary, providing the mapping inverse to unificator (from machine stamps to RC pairs).

- cache dictionary, 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 instance 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 to machine in approximated complexity $O(|rc|)$ and independence from the number of machines in the cache (very important for scalability of data mining systems). This means that each machine configuration has to implement two methods: equality of configurations and the hash function of the configuration. Provided high quality of the methods, the maintenance of the unification system costs very little.

To better see the advantages of machine unification, imagine a project to test and compare suitability of different feature selection methods for a classification problem. In the case of our system

we will compare feature ranking methods rather than feature selection methods. To make the test credible, we should, for instance, perform a cross-validation (CV) of complex machines consisting of feature ranking, feature selection and classification machines. The complex machine could have the form of Transform and classify machine (see figure 4) with transformation scheme replaced by the feature selection template of figure 6 filled with proper feature ranking. When performing the CV test with different ranking machines, it may turn out that the selection of top-most features of different rankings result in the same set of features, so it makes no sense to train and test the classifier twice for the same data.

Table 1 shows feature rankings obtained for Wisconsin breast cancer data from the UCI repository with eight different methods: three based on indices estimating feature's eligibility for target prediction (F-score, correlation coefficient and entropy based mutual information index), one based on internals of trained SVM model and four based on decision trees using different split criteria (Gini index, information gain, information gain ratio and SSV [28]). To test a classifier on all sets of top-ranked features for each of the eight rankings, we would need to perform 72 tests, if we did not control subsets identity. An analysis of the 72 subsets brings a conclusion that there are only 37 different sets of top-ranked features, so we can avoid 35 repeated calculations. Very similar savings occur, when the rankings are determined inside the CV test (for each training sample, not for the whole data set as in the case of rankings presented in the table).

Table 1: Feature rankings for UCI Wisconsin breast cancer data.

Ranking method	Feature ranking								
F-score	6	3	2	7	1	8	4	5	9
Correlation coefficient	3	2	6	7	1	8	4	5	9
Information theory	2	3	6	7	5	8	1	4	9
SVM	6	1	3	7	9	4	8	5	2
Decision tree (DT), Gini	2	6	8	1	5	4	7	3	9
DT, information gain	2	6	1	7	3	4	8	5	9
DT, information gain ratio	2	6	1	5	7	4	3	8	9
DT, SSV	2	6	1	8	7	4	5	3	9

Naturally, being aware of saving possibilities, one can design the project in such a way, that different rankings are analyzed first to check such redundancies and avoid them, but it requires programming a special machine to perform the test in proper way. System feature of avoiding repeated calculations, eliminates the overload without any special effort from the user and is a general and very efficient solution which may help in many other circumstances.

In advanced meta-learning, it would be highly nontrivial to predict all the possibilities of repeated machines and prevent them in advance. For a more complex example of machine unification advantages see section 8. To maximize the gains, we have solved the unification problem at the kernel level of our system. Each machine created and run within the project is registered in *machine cache*. Each request for a new machine is checked against the machine cache, whether the machine is already available and if it is, the machine request is substituted by the ready machine and the request does not go to the task spooler. Using hash codes in comparing different machine contexts makes the cost of machine cache management very small, so the overall balance is definitely positive.

5.2. Two-tier cache system

Machine cache can be successful if it has access to as many machines as possible. Thus it is advantageous to keep all the machines run within the project, but it may result in too much memory consumption. On the other hand, sometimes we must deal with so large data and models, that keeping too many of them in memory at the same time would thwart even basic tests. Therefore, as mentioned earlier, our cache system is a *two-tier* solution. If possible, machines are kept in memory, but to make more memory available they are swapped to specialized *disc cache*.

Each machine is saved in the disc cache after its adaptive process is finished. It is kept in memory as long as it is needed by any other machine. The information interchange through inputs and outputs is a subject to open–close management, so that the system receives all the information about machines and their outputs being in use. The requirement to open and close machine inputs facilitates also optimization of machine exchange between the project and computational servers running the project tasks (possibly remote servers).

Additionally, the unique, specialized structure of the disk cache and its management guarantee that the access to machines (loading/saving) does not depend on the number of machines in the disc cache but just on the length of binary representation of

the machine. It keeps the whole unification and cache system as efficient as possible.

A question might be asked here: why to introduce a disc cache instead of relying on the operating system virtual memory mechanisms? The answer is that our internally managed disk cache can be much more effective, because it can take advantage of the information about project internal structure and act without any delay that could disable performing the tasks scheduled in the project. Moreover, the operating system cache has no information about which machines are necessary at the moment or will be necessary in further steps in contrary to our system which has full knowledge about that, because each machine usage is registered. The knowledge lets the system decide whether given machine should be kept in the memory cache or just in the disc cache or should be completely discarded from the cache while system's virtual memory system has no such information.

5.3. Control over random processes

Another detail which gets quite important when more advanced data analysis is to be performed, and which has not been completely solved by any data mining package we know of, is the way of random processes management. It is common to include a seed value (controlling the randomness) inside configuration structures of CI algorithms. Then, the seed value can be set arbitrarily or chosen randomly (usually on the basis of current time to provide better randomness). However in the case of complex machines, it is not satisfactory, especially when we like the mechanisms of machine unification to be maximally functional. Our design of seed control aimed to facilitate:

- unification of complex, multi-level machines, possibly with pseudo-random behavior of chosen machines at different levels,
- robust comparisons of different CI methods based on testing them on exactly the same data, even when the tests are made within distinct projects.

These two features are not guaranteed, when we may configure machines to use either fixed or random seed. For example: when we want to perform 10 repetitions of 10-fold cross-validation, we configure just one machine providing cross-validation data, which is to be repeated. If we set the seed property to "fixed", each repetition will result in the same sets of training and test data, and exactly the same 10-fold CV will be performed 10 times. If we set the seed to "random" (dependent on the time), it will not be possible to repeat the same splits in further tests. What we need is a system that allows the subsequent CVs to run with different seeds and makes it possible to repeat the whole procedure with the same sequence of seeds. The same need occurs, for example, when we perform a CV of a neural network initialized by random weights.

For full functionality, we assumed that the seed configuration should facilitate three different ways of seed control:

1. The seed is fixed to a given integer value.
2. The seed value is determined by a pseudo-random number generator initialized with the time of machine preparation.
3. The seed is managed automatically by the system to reflect the seed settings of parent machines.

The third option of automatic seed management means, that the machines can get different seeds in a machine branch, but the seeds will depend on the seed of the root machine of the branch and the child machine index. So, when we need a 10-fold CV to be run 10 times independently, but in such a way that can be repeated at any time, we may set a fixed seed for the repeater machine, and assign the auto mode to the seed of the machine generating data for CV. In such case, each repeated CV will get different seed, but dependent on the seed of the repeater, providing both diversity of different CV repetitions and possibility to repeat the whole procedure. The same 10 times repeated 10-fold CV can be obtained later with the same configuration of the seed at the repeater machine level and automatic setting for the CV data generator. Moreover, when we repeat the whole scenario for exactly the same configuration of a classifier, the whole repeater machine may be unified with the former one, and no calculations are necessary because the whole structure is already available.

The possibility of performing tests with exactly the same training and test data, also in completely different projects opens the gates to most adequate comparisons of different machines results including paired t-test, Wilcoxon test or even McNemar test.

6. Task spooler

The task spooler of our system is not a simple standard queue. To optimize the efficiency of task running we have introduced a system of hierarchical priorities. Each parent machine can assign priorities to its children, so that they can be run in proper order. It prevents starting many unrelated tasks in parallel i.e. from too large consumption of memory and computation time. Because of the priorities and machine unification mechanisms, which may result in spooling the same task twice with different priorities, the task spooling system gets quite different than operating systems task schedulers. The number of tasks may be quite large and we can not use the policy of equal CPU time gratification, because it would very often lead to memory exhaustion. As a result the main part of the spooler has the form of tree containing nodes with priorities. Apart from the tree, there is a container for machine requests waiting for machines being run for the sake of other contexts—when a machine is being run, other requests for the same machine may not be run, because it would not obey the rule of not creating the same machine twice.

When the task gets its turn, it is popped out from the spooler and, if the machine ordered within the task, is not currently running, it is created and run. When a task related to another context of the same machine is currently running, the new task for the same machine must wait until the other context is fully serviced. If the other context finishes successfully, the machine is assigned to all its contexts, otherwise (i.e. the parent machine of the other context aborted its run) the waiting task is started the same way as when it is not unified with any other task. The functionality described above is expressed algorithmically by the following meta-code:

```
1  Task GetNextTask()
2  {
3    foreach (Task t
4        in waiting_for_another_context)
5    {
6      if (the other context aborted)
7        return t;
8      if (the other context is ready) {
9        t.Status = Substituted;
10       continue;
11     }
12   }
13   while (true) {
14     if (spooler.IsEmpty)
15       return null;
16     Task t = spooler.Pop();
17     if (machine of t is
18         finished by another context) {
19       t.Status = Substituted; continue;
20     }
21     if (machine of t is
22         running within another context) {
23       waiting_for_another_context.Add(t);
24       continue;
25     }
26     return t;
27   }
28 }
```

The collection waiting_for_another_context contains the tasks that received their turn to run while other tasks requesting the same machines were running (see line 23). When such running task is canceled, the tasks from waiting_for_another_context are processed before the tasks that have not got their turn yet (see lines 3–12). The main loop, processing the spooler, is contained within lines 13–27. The method GetNextTask is called whenever a task thread calls for a new task to perform.

After the machine process is finished, the machine gets the *ready* status. It is not the end of the life of the machine, but from the point of view of its different states it is the final state, in which the machine results and outputs can be exploited, but the machine does not change anymore.

We need to stress 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 of the created model(s).

To observe the advantages of our spooling system 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 figure 8, 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 12. The repeater machine creates 10 distributor schemes (ds_1-ds_{10}) and 100 test schemes (ts_1^1-$ts_{10}^1 \ldots ts_1^{10}$-$ts_{10}^{10}$). Each distributor scheme cre-

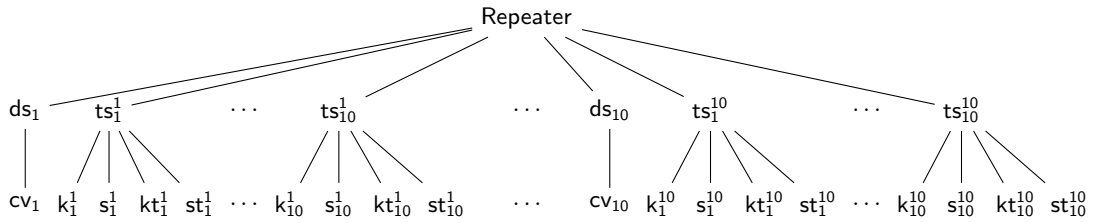

Figure 12: Repeater submachine tree performing 10 × 10-fold CV of tests defined in figure 8. Notation: ds – distributor scheme, ts – test scheme, cv – CV distributor, k – kNN, kt – classification test for kNN, s – SVM, st – classification test for SVM.

ates one cross-validation distributor (cv_i) and each test scheme requests 4 child machines: kNN ($\mathsf{k}_\mathsf{j}^\mathsf{i}$), classification test of kNN ($\mathsf{kt}_\mathsf{j}^\mathsf{i}$), SVM ($\mathsf{s}_\mathsf{j}^\mathsf{i}$) and classification test of SVM ($\mathsf{st}_\mathsf{j}^\mathsf{i}$).

To avoid randomness of the process due to parallel calculation, we assume that all the tasks are calculated by one task manager with one running thread. The request for the repeater machine pushes the root node of the tree (of figure 12) 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,...,ts_{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:

$$\mathsf{Repeater},\ \mathsf{ds}_1,\ \mathsf{cv}_1,\ \mathsf{ts}_1^1,\ \mathsf{k}_1^1,\ \mathsf{s}_1^1,\ \mathsf{kt}_1^1,\ \mathsf{st}_1^1,\ \ldots,\ \mathsf{ts}_{10}^1,\ \mathsf{k}_{10}^1,\ \mathsf{s}_{10}^1,\ \mathsf{kt}_{10}^1,\ \mathsf{st}_{10}^1,\ \ldots,\ \mathsf{ds}_{10},\ \mathsf{cv}_{10},\ \mathsf{ts}_1^{10},\ \mathsf{k}_1^{10},\ \mathsf{s}_1^{10},\ \mathsf{kt}_1^{10},\ \mathsf{st}_1^{10},\ \ldots,\ \mathsf{ts}_{10}^{10},\ \mathsf{k}_{10}^{10},\ \mathsf{s}_{10}^{10},\ \mathsf{kt}_{10}^{10},\ \mathsf{st}_{10}^{10},$$

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

$$\mathsf{Repeater},\ \mathsf{ds}_1,\ \mathsf{ts}_1^1,\ \ldots,\ \mathsf{ts}_{10}^1,\ \ldots,\ \mathsf{ds}_{10},\ \mathsf{ts}_1^{10},\ \ldots,\ \mathsf{ts}_{10}^{10},\ \mathsf{cv}_1,\ \mathsf{k}_1^1,\ \mathsf{s}_1^1,\ \mathsf{kt}_1^1,\ \mathsf{st}_1^1,\ \ldots,\ \mathsf{k}_{10}^1,\ \mathsf{s}_{10}^1,\ \mathsf{kt}_{10}^1,\ \mathsf{st}_{10}^1,\ \ldots,\ \mathsf{cv}_{10},\ \mathsf{k}_1^{10},\ \mathsf{s}_1^{10},\ \mathsf{kt}_1^{10},\ \mathsf{st}_1^{10},\ \ldots,\ \mathsf{k}_{10}^{10},\ \mathsf{s}_{10}^{10},\ \mathsf{kt}_{10}^{10},\ \mathsf{st}_{10}^{10}.$$

Since, thanks to the spooling system, Intemi keeps just three running machines at a time, both memory and CPU time are saved significantly. When running this example on the UCI Wisconsin breast cancer data, peak memory usage was about **30 MB**, while with standard queue it was over **160 MB**. Also the overall time used by the project was significantly reduced (around **15%**) although the calculations were exactly the same—handling the process with smaller number of threads and less memory consumption turned out to be so less time consuming for the operating system.

7. Results and query system

Apart from efficient machine creation and running, a successful data mining system must provide tools for handling machine results. From the point of view of meta-learning applications it is extremely important to manage the results in a unified way, facilitating analysis of results of machines completely unknown to the meta-learners.

As described before, machine outputs are handled in a standardized way, independently of particular machine peculiarities. Also the results that are not expected to have the form of outputs, can be deposited in a standard manner. We have designed three standard ways of exposing such information:

- depositing to the machine's results repository by the machine itself,
- commenting machines by their parent machines,
- commenting machines by commentators.

Putting the results into these repositories is advantageous also from the perspective of memory usage. Machines can be discarded from memory when no other machine needs their outputs, while the results and comments repositories (which should be filled with moderation) stay in memory and are available for further analysis.

The information can be accessed directly (it can be called a low level access) or by running a query (definitely recommended) to collect the necessary information from a machine subtree.

7.1. Results repositories

The system creates a *results repository* for each machine and the machine can put there the information describing the model, the learning procedure etc. Results repositories are dictionaries mapping string labels to the results objects. Machines can add results to their repositories by calling the AddToResultsRepository method of the derived class Machine (each machine class is obliged to derive from it). For example the Classification test machine adds calculated accuracy to the repository in the following way:

```
double accuracy;
... //calculating the accuracy
AddToResultsRepository("Accuracy", accuracy);
```

There is no limit for the number of elements that can be put into the repository or their size, however it is advisable to put there only the most important information, in order to save memory.

7.2. Parent's comments

Each machine can comment its submachines to augment further analysis of the submachines structure. For example, the repeater of figure 8 comments each of its submachines with labels denoting which repetition and which CV fold the subscheme belongs to. Thanks to this, we can run queries filtering appropriate results, for example we can select all the accuracies of the first CV cycle or all the accuracies of the second folds of each CV cycle etc.

A machine can comment a child with a call to the AddChildComments method of the Machine class. The child index, comment label and comment value must be passed to the method. In the case of the example mentioned above, the repeater comments each subscheme with the code:

```
AddChildComments(givenMachineId,
  "Repetition", group + 1);
AddChildComments(givenMachineId,
  "Fold", machineId + 1);
```

7.3. Query

To simplify the management of submachines' results, which constitute a tree hierarchy, we have provided tools for quick and easy results collection and analysis. A *series* of results selected from a machine tree can be obtained by running a *query*. A query is defined by:

- the root machine of the query search (root of the tree),
- a *qualifier* i.e. a filtering object—the one that decides whether an item corresponding to a machine in the tree, is added to the result series or not,
- a *labeler* i.e. the object collecting the *results objects* that describe a machine qualified to the result series.

Running a query means performing a search through the tree structure of submachines of the root machine and collecting a dictionary of label-value mappings (the task of the labeler) for each tree node qualified by the qualifier.

For example, consider a repeater machine producing a run time hierarchy of submachines as in figure 8. After the repeater is finished, its parent wants to collect all the accuracies of SVM machines, so it runs the following code:

```
Query.Series results = Query(repeaterCapsule,
  new Query.Qualifier.RootSubconfig(1, 3),
  new Query.Labeler.FixedLabelList("Accuracy"));
```

The method Query takes three parameters: the first repeaterCapsule is the result of the CreateChild method which had to be called to create the repeater, the second defines the qualifier and the third—the labeler. The qualifier RootSubconfig selects the submachines, which were generated from the subconfiguration of repeater corresponding to path "1, 3". The two-element path means that the source configuration is the subconfiguration 3 of subconfiguration 1 of the repeater. A look at the repeater configuration in figure 7 clarifies, that subconfiguration 1 of the repeater is the configuration of the test scheme (0-based indices are used) and its subconfiguration 3 is the SVM Classification test. So the qualifier will accept all the machines generated on the basis of the configuration Classification test taking *Classifier* input from SVM machine. These are classification tests, so they put Accuracy to the results repository. The labeler FixedLabelList of the example, simply describes each selected machine by the object put into the results repository with label Accuracy.

As a result we obtain a series of four descriptions containing mappings of the label Accuracy to the floating point value of the accuracy.

We have proposed a number of qualifiers and labelers, most likely to be needed by researchers. Instead of the RootSubconfig(1, 3) qualifier, one could use ConfigType(typeof(ClassTestConfig)). This would collect the results for all the machine tree nodes for which ClassTestConfig is the configuration class i.e. from all the classification test machines. The result series would contain not four but eight elements since both Classification test machines (taking *Classifier* input from kNN and SVM machines) would be qualified by the query.

The labelers are given access to the whole path of machine results: from the query root submachines to the leaves of the machine tree. Thus, the labelers can use labels from any level. For instance, the labeler AllLabels collects all the labels commenting the whole path. It lets us easily collect, for example, the accuracies calculated by the classification test machines and fold identifiers given by the repeater machine to its submachines (distribution and test schemes). Partial results of a $n\times5$ CV are presented in table 2. The Repetition and Fold entries are the comments made by the repeater on its submachines.

7.4. Series and series transformations

The result of a query is contained within an object of the Series class. The series is a collection of label-value pairs describing subsequent items. In the case of the series presented in table 2, each item is described by three values assigned to the labels Accuracy, Repetition and Fold respectively.

The series are often just intermediate results that undergo more or less sophisticated analysis. Each series can be transformed by specialized objects called series transformations. The transformations get a number of series objects and return another series object. One of the basic transformations is the BasicStatistics which transforms a series into a single item series containing the information about minimum, mean, maximum values and standard deviation.

Quite advanced manipulation of series of results can be performed with groups related transformations. As mentioned before, when a query uses a ConfigType qualifier pointing to the configuration type of classification test, it collects all the classification test results (regardless which classifier it tests—kNN or SVM in the example shown before). So, a series obtained in this way, contains descriptions of kNN classification tests and SVM classification tests, by turns. We can easily create two groups, separating kNN results from SVN results with a call to the GroupModulo transformation:

```
Series inGroups = allresults.Transform(
  new GroupModulo(2));
```

It is also easy to group series items containing common values for particular label. For example, when we repeat 7 times a 10-fold CV and collect all available labels from classification test machines, we can group the resulting series by the "Repetition":

```
Series repets = allresults.Transform(
  new Group("Repetition"));
```

and obtain a series of 7 series containing separated results of each repetition of the whole CV cycle.

A grouped series can be traversed to transform each of its subseries. The MAP transformation performs a given transformation on all the subseries. For example, running the code:

Table 2: Partial results obtained with labeler AllLabels for a $n \times 5$ CV.

Item	1	2	3	4	5	6	7	8	9	10	11	...
Accuracy	0.93	1.0	0.97	0.93	0.97	0.9	0.97	1	1.0	1.0	0.97	...
Repetition	1	1	1	1	1	2	2	2	2	2	3	...
Fold	1	2	3	4	5	1	2	3	4	5	1	...

```
Series stats = repets.Transform(
  new MAP(new BasicStatistics));
```

we can calculate basic statistics for all the series within repets. The result remains a series of series (as repets was, so it must be ungrouped with the Ungroup transformation, to obtain a plain series of basic CV statistics for each repetition.

A shorthand notation is available to speed up writing code for such manipulations:

```
Series flatStats = allresults.Group("Repetition")
  .MAP(new BasicStatistics()).Ungroup();
```

One of the labels contained within the series can be set as the *main label*. It is necessary for some transformations to know the default (main) label to operate on. For example the BasicStatistics transformation calculates the statistics for the values assigned to the main label.

There are also basic arithmetic operators available and they also act on the main label values. They can be used in a natural manner as infix operators. The code:

```
Series diff = kNNResults - SVNResults;
```

calculates the differences of accuracies (assuming, that the Accuracy is the main label in the series kNNResults and SVNResults). An example of such operation on results of 10-fold CV (performed once) is shown in table 3.

One of the most important aspects of such results manipulation is easy testing of statistical hypotheses about the results differences. Thanks to universality of proposed ideas, we can easily run statistical tests like t-test, paired t-test, Wicoxon, McNemar etc. They are implemented as series transformers, so it is possible to call just:

```
Series tTest = new TTest().Transform(
  kNNResults, SVNResults);
Series tTestP = new TTestPaired().Transform(
  kNNResults, SVNResults);
```

The results are new series, containing the information about the test results. TTest and TTestPaired transformations return series with single items labeled with t value and p value presenting the value of calculated statistic (t) and the estimated probability of the null hypothesis (about equality of the means) being true. The results calculated for the 10-fold CV accuracies from table 3 are presented in table 4. In this case the t-test allows us to reject

Table 4: Statistical significance tests results.

Test name	t-test	paired t-test	McNemar
statistic	2.370	3.772	12.25
p-value	0.0292	0.0044	0.0005

the null hypothesis with 95% confidence ($\alpha = 0.05$), but not with 99% confidence ($\alpha = 0.01$). The paired t-test, provided with the information about differences in subsequent passes, confirms that the results are statistically significantly different with more than 99% confidence.

The mechanisms of query and series facilitate such analyses with very simple means. There are many more predefined series transformations and new transformations can also be easily implemented.

7.5. Commentators

Queries collect and series transformers analyze data gathered in results repositories. By default, machines put in the repositories only the most important information. Such strategy saves both memory and computation time. When additional information is necessary it can be deposited in the system as a comment on a machine.

Each machine can order a comment on its successors at the moment of requesting for creation of a submachine. To continue the example of kNN and SVM test, if we need to perform a McNemar test of statistical significance of difference between performance of kNN and SVM, we request comments on both classification tests. The CorrectnessCommentator comments a classification test machine with a binary vector of the length

Table 3: 5NN and SVN accuracies (in %) for 10-fold CV for UCI image segmentation data.

Fold	1	2	3	4	5	6	7	8	9	10	Mean±St.dev.
5NN	95.2	81.0	90.5	81.0	95.2	81.0	90.5	85.7	90.5	90.5	88.1±5.6
SVM	81.0	76.2	90.5	71.4	81.0	76.2	90.5	76.2	90.5	81.0	81.4±6.9
5NN-SVM	14.2	4.8	0.0	9.6	14.2	4.8	0.0	9.5	0.0	9.5	6.7±5.6

equal to the number of objects in the test data and containing a 1 for each correct answer and 0 for incorrect answer. It would be wasteful to generate and keep such vectors for each classification test machine, so it can be done on demand by means of commentators.

This is an example of C# code that compares classification test results (for the example scenario testing kNN and SVM) with McNemar test:

```
Series s = project.Query(repeaterCaps,
  new ConfigType(typeof(ClassTestConfig)),
  new FixedLabelList("Correctness"));
s = s.GroupModulo(2).MAP(new Unpack());
Series r = new McNemar().Transform(s[0], s[1]);
```

First three lines run proper query. The fourth line groups the results of kNN and SVM and maps the two groups with Unpack transformation, which unfolds all the collections of binary results added by the CorrectnessCommentator into a single, long series. The fifth line runs McNemar test on the two long series of correctness binary flags and returns the χ^2 statistic value and the p value as two values in a single item series. Last column of table 4 contains the results of the McNemar test run for the example being discussed.

7.6. How to avoid testing frauds?

Using tools without the mechanisms described in this article, like uniform representation and management of simple and complex machines, clear distinction between the configuration and runtime lives of machines, results repositories, queries etc., often makes CI research prone to miscellaneous embezzlements. Apparently, some of them are disguised enough to be accepted in numerous scientific articles. The unified view of machines, we have proposed, makes the embezzlements easier to see and avoid. Of course, it is not possible to completely prevent the frauds with system level protection, because that would mean significant restrictions of the system.

There are many different techniques of data preparation for final adaptive model construction. Many researchers clearly split the whole data mining process into stages including separate data *preprocessing* and final learning or testing. It is dangerous, because it suggests that machine generalization testing may be performed on preprocessed data without unjust consequences. As a result there are many publications describing such unfair approaches to data mining.

The most common mistake is **using supervised data transformation methods before testing generalization abilities of learning algorithms**, instead of including the supervised transformations in a complex model to be tested. The set of most popular fraud techniques of supervised data **preprocessing** include:

- supervised data discretization,
- supervised missing values substitutions,
- supervised feature extraction.

An extreme example of such unfair calculations would be adding the targets to the data. In such a case everybody would have no doubt that it is a cheat. Unfortunately, less radical examples quite frequently go successfully through the sieve of peer review processes.

In the list of fraud techniques, we deliberately emphasize the word **supervised**. Naturally using unsupervised methods (for example data standardization) as data preprocessing, before testing generalization is not a cheat, although would also be a bit strange in practical applications. For instance, in medicine, usually data is gathered for some time period to enable training of adaptive machines. When models are created, they are used to classify new data collected for new patients. Standardization of the whole data, which is the union of training and test data sets, would require waiting until all the data, we want to classify, is collected. After that the machines would be trained and the decisions for all the new patients determined. So if we want to simulate practical application of machine learning tools, we should treat all the data manipulation

before training as a part of the training process and in the case of cross-validation, repeat the whole procedure from the very beginning for each CV fold. However, assuming the training data is representative for the whole population, unsupervised data preprocessing methods like different forms of normalizations, give very similar results on the whole data and the training part, and the differences are very unlikely to affect the further validation results.

An example of a fraud technique is filling missing values in a supervised manner within the data preprocessing stage. If we allow the method to be supervised, we could create a method to fill the missing values with the class label of the particular data object. It would introduce very precious information and significantly decrease the estimated errors. In an extreme case, we could delete all the values of a feature and then fill the missing values (obtained this way) by the class label. Then, simple identity function would be perfect classifier, but what would we need to do with new data to be classified? We should do the same we did for the training data i.e. put the class label in proper place, so we would need to now the class label in order to predict it. It is important to notice, that filling missing values with the mean values observed within the adequate class is practically equivalent to filling with the class labels, since, almost always, the means are unique for particular classes, so they are equally informative as class labels.

Another example of influence of supervised data preprocessing on validation result may be adding new, more informative features, on the basis of an analysis of a model built for the whole data set. For example: SSV decision tree algorithm [28] produces a very simple and relatively very accurate description of the appendicitis data from UCI repository:

$$\text{if HNEA} < 7520.5 \wedge \text{MBAP} < 12 \text{ then class 2 else class 1.}$$

The rule accuracy calculated for the whole set is 91.5%. Extending the original data, by adding a binary feature corresponding to the value of the rule premises, introduces the knowledge gained by SSV (by an analysis of the whole data set) to the data. Therefore, most classifiers can find the information and reach the CV test accuracy of up to 91.5%—of course the SSV decision tree does, while when trained in a fair way, its average CV accuracy is about 86.1%. So the alleged CV test result is in fact the reclassification accuracy. It is a less spectacular and less obvious example than adding the target as one of the features, but reveals the same vulnerability.

Even unsupervised transformation, when used as *preprocessing*, may be **very** dangerous. For example, a typical cure for small number of instances is multiplication of instances (sometimes with addition of a bit of noise) to enable learning of some machines. When this transformation is used before separation of testing data part, it causes that we can find the same instances in training and testing data[1]. As a result, if the multiplication is meticulous, the miracle is nearly guaranteed, especially with machines like 1NN, which will find *original* clone-instances as nearest neighbors.

Another abuse in data mining is using single **external test files** for numerous machine tests. Although in the case of different contests it seems the best way of estimation of models' generalization abilities, it is not good to select the test data in data sets distributed with test data targets. An extreme example, in this case, is a "learner" that does not learn, but simply guesses the target model. It is just the matter of sufficiently high number of trials, to find a model performing perfectly on the test set. But it would no longer be a fair result, because we have selected a model on the basis of its behavior on the test data i.e. we have used the test data for a peculiar type of training.

In the case of the hypothyroid data from the UCI repository, SSV decision trees (with some manipulation of parameters) reach 99.73% accuracy in reclassification of the training data and $99.09-99.36\%$ when classifying the "test data". The success of the most accurate configuration parameters can not be regarded as fair, since we would never guess that it is so accurate model, if we had not checked it on the test data. Similarly with a kNN machine equipped with mechanisms of feature weighting, one can obtain almost as good results as with SSV decision tree, but running a CV on the union of training and test sets results in definitely lower results, showing that the model with weighting was so accurate, as a result of a coincidence. Training and testing machines many times, just makes finding an accurate model more and more probable.

For more about trustful and successful methods of testing and parameter or transformation combin-

[1] In the cases of adding noise, very similar instances can be found.

ing please read [29, 30]. These articles show how to successfully deal with real problems by examples of challenges in data mining.

8. Meta parameter search

One of the first steps toward advanced meta-learning is a machine capable of efficient searching in the space of model parameters. Different data mining systems introduce such tools, but the implementations are either very limited or look like external patches that do not fit the overall inner architecture. Shortages of the engine level design do not allow for advanced meta-learning in a natural way. In our approach the meta-learning requirements decided about many solutions at the engine level of the system (e.g. those presented in preceding sections), so meta-learning machines are built in the same manner as all other machines and are nothing special also from the point of view of system operation. Simple parameter search machine is the first step toward more sophisticated meta-learning and constitutes a good illustration of cooperation of different mechanisms described above.

The aim of the meta parameter search (MPS) machine is to repeatedly create a submachine and test different values of parameters included in the submachine configuration. Our algorithm of the MPS machine allows for arbitrarily complex submachine and can search for optimal values of parameters of any part of the submachine configuration hierarchy (any depth etc.).

The configuration of the parameter search machine includes:

- **test configuration** defining arbitrarily complex machine structure,
- **scenario** of parameter changes,
- **query specification** which determines the way of estimation of the test results.

The process realized by the parameter search machine creates a sequence of submachines for the test configuration with some parameters changed each time, according to the configured scenario. In each pass, it:

- gets a candidate configuration from the scenario object,
- creates a submachine to test the configuration,
- runs a query on the submachine branch to collect proper results,
- transforms the series of values resulting from the query by given series transformer to obtain the final measure of the configuration parameters performance,
- passes back the performance result to the parameter changes scenario object, so that it can adjust the process of producing subsequent machine configurations.

Normally, the test submachine configuration defines the whole test process, e.g. multiple training and test of a classifier, so it is convenient to derive it from a test template scheme, introduced in section 3.5, by filling placeholders with proper machine configurations e.g. putting classifier configuration within a cross-validation test template.

The parameter search scenario may just iterate over a set of possible values of a parameter or perform a sophisticated process exploiting specialized meta-knowledge and the feedback from subsequent tests. Although it is not possible to define a single scenario that would guarantee satisfactory results in a short time, machine authors may define default scenarios for machine parameters i.e. the default way of searching for optimal parameter values (suggestions of scenarios that should perform well on average).

The following source code presents an application of the MPS machine:

```
DataLoaderConfig dCfg =
  new DataLoaderConfig();
dCfg.InputFileName = "breast-cancer.dat";
Capsule dCaps =
  project.CreateMachine(dCfg, null, null);

RepeaterConfig rCfg = new RepeaterConfig();
rCfg.SetCVDistributor();
rCfg.TestScheme.Add(0, new StdConfig(),
  new Bindings().Bind("Data", "Training data"));
rCfg.TestScheme.Add(1,
  new SVMClassifierConfig(),
  new Bindings().Bind("Data", 0, "Data"));
rCfg.TestScheme.Add(2,
  new ExtTransformationConfig(), new Bindings()
  .Bind("Data", "Test data").Bind("Transformer",
  0, "Transformer"));
rCfg.TestScheme.Add(3, new ClassTestConfig(),
  new Bindings().Bind("Data", 2, "Data")
```

```
20      .Bind("Classifier", 1, "Classifier"));
21  rCfg.CVParams(5, 2);
22
23  ParamSearchConfig psCfg =
24    new ParamSearchConfig();
25  psCfg.TestingConfiguration = rCfg;
26
27  int[] subcfgPath = new int[] { 1, 1 };
28  StepScenario_D SigmaScenario =
29    new StepScenario_D(subcfgPath,
30    new string[] { "Kernel", "Sigma" },
31      StepScenario_D.StepTypes.Power2, -12, 2, 8);
32  StepScenario_D CScenario =
33    new StepScenario_D(subcfgPath,
34      new string[] { "C" },
35      StepScenario_D.StepTypes.Power2, -1, 2, 7);
36  psCfg.Scenario = new StackedScenario(
37    StackedScenario.StackTypes.Grid,
38    new IScenario[] { SigmaScenario, CScenario });
39
40  psCfg.QueryDefinition = new QueryDefinition(
41    new Intemi.Query.Qualifier.RootSubconfig(1, 1),
42    new Intemi.Query.Labeler.FixedLabelList(
43      "Accuracy"),
44    new Intemi.Query.BasicStatistics());
45
46  Capsule psCaps = project.CreateMachine(psCfg,
47    new Bindings().Bind("Data", dCaps, "Data"),
48    null);
49  project.WaitAll();
```

Lines 1-5 define configuration of data loading machine and order the machine. Lines 7-21 prepare configuration of a repeater performing 5 times 2-fold cross-validation of an SVM machine trained on standardized training part of CV data and tested on the test part after standardization performed according to the statistics calculated for the training data. Thus the machines built for StdConfig and ExtTransformationConfig are put inside the repeater test scheme. The repeater configuration is then (in line 25) passed to the configuration of the MPS machine as the configuration of test procedure.

Lines 27-38 define the scenario of the parameters search process. The search has a form of a grid (see line 37), i.e. each declared value of the Gaussian kernel function σ parameter is tried against each declared value of the C parameter. Both parameters belong to the SVM machine, which is identified by the path `1, 1` (see lines 27, 30 and 33) of subconfigurations of the MPS testing configuration (the testing configuration is the repeater defined in lines 7-21 and indices are 0-based, so the repeater's subconfiguration 1 is its test scheme containing SVM configuration as subconfiguration 1—see line 11). Both parameters are of exponential nature, so we explore the sets $\{2^i : i = -12, -10, \ldots, 4\}$ and $\{2^i : i = -1, 1, 3, \ldots, 11\}$. The sets specification is contained within lines 31 and 35, containing the declaration of the exponential type, start exponent, step (the increment of the exponent), and how many values are to be tried. We have provided also many other predefined parameter search scenarios and completely new scenarios can also be easily added.

Lines 40-44 specify the query to collect each test results and the transformation of the resulting series to another series obtaining the final estimate of the configuration performance. The estimate implemented here is simply average accuracy obtained within the repeated CV test (BasicStatistics transformation calculates mean value, minimum, maximum, standard deviation etc. setting the mean as the main label of the resulting series). The results obtained with the code, are presented in table 5. They are available as an output of the parameter search machine after the whole process is finished.

Finally, lines 46-49 order the MPS machine from the project and wait for all the machines to finalize their processes.

The project takes care for all the machines that must be built to provide the requested machine. It creates subsequent machine tasks, when it can be determined that such a machine must be created, and wherever possible, reuses machines created earlier with the unification mechanism. In this project the repeater is performed many times with changed configuration of the SVM machine but without changes to the configuration of the CV processes. Thus many submachines in this project, e.g. responsible for data splits, are reused to save time and memory.

The project configuration is visualized in figure 13. At run time, a single Data loader and a single Meta parameter search are created. Within the MPS process, a Repeater machine is created for each of the $56 = 8 \times 7$ pairs of parameters. Each repeater performs 5×2 CV, so it creates 5 distributor schemes and 10 test schemes. Thus, the overall number of needed machines is quite large. Thanks to the unification framework (described in section 5), the number of machines that are really created and run is significantly smaller. The numbers are presented in table 6.

Table 5: Meta parameters search results for SVM tested by 5×2-fold CV on Wisconsin breast cancer data.

C	Gaussian kernel σ							
	0.0010	0.0039	0.02	0.06	0.25	1	4	16
0.5	0.6635	0.9393	0.9642	**0.9671**	**0.9674**	0.9588	0.9419	0.8798
2	0.9396	0.9645	**0.9677**	**0.9665**	**0.9659**	0.9605	0.9508	0.9159
8	0.9645	**0.9677**	**0.9668**	0.9645	0.9591	0.9479	0.9508	0.9159
32	**0.9677**	**0.9668**	0.9634	0.9474	0.9428	0.9451	0.9508	0.9159
128	**0.9674**	0.9617	0.9456	0.9413	0.9322	0.9422	0.9508	0.9159
512	0.9617	0.9474	0.9428	0.9365	0.9293	0.9416	0.9508	0.9159
2048	0.9476	0.9416	0.9391	0.9339	0.9273	0.9416	0.9508	0.9159

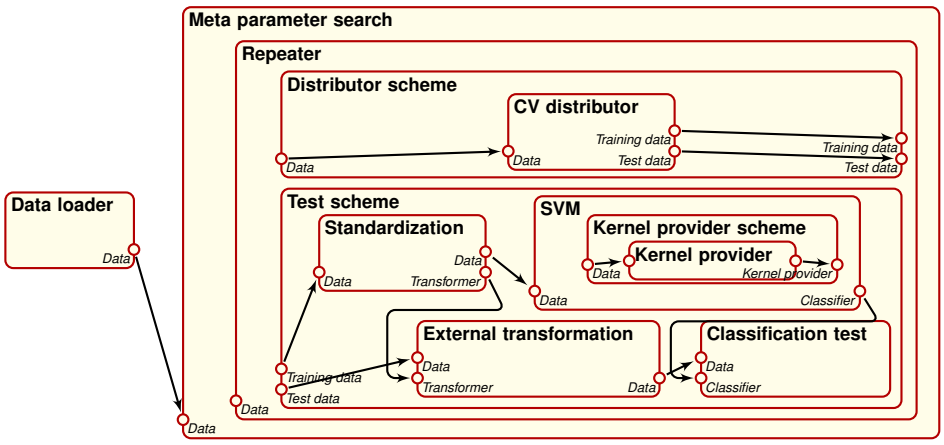

Figure 13: Meta parameter search project configuration, corresponding to the source code.

We consciously did not extend table 6 with columns describing other data mining systems, because no other system is able to reuse machines, and even if one were 10% faster in the first learning of given machine, it would not be faster after next computation of the machine, since our system is able to reuse machines so serving repeated requests costs almost nothing. Learning processes are sometimes really CPU consuming and this problem can not be trivially neglected.

It can be easily seen in table 6, which machines are reused many times: distributor scheme, CV distributor, standardization, external transformation, kernel provider scheme and kernel provider. Indeed, there is no point in repeating CV data distribution for different repeater instances, since each time the data split is the same. So only 5 different CV data pairs are needed—the repeater performs 5 independent splits. Similarly the training and test data coming as test scheme inputs are cyclically repeated, so only 10 different standardization machines and 10 different external transformation machines are necessary.

A nice surprise, can be the unification of the SVM kernel calculations. Our realization of the SVM machine, extracted kernel calculations to a separate machine (submachine of SVM) to enable multiple use of the same kernel table. More precisely, as visible in figure 13 SVM defines the Kernel provider scheme which can be filled with any machine structure providing kernel calculation interface. In our test, the scheme contains a single Kernel provider machine which outputs Gaussian kernel calculation routine. We run SVMs with different parameters of C and Gaussian kernel σ. Two SVM machines trained on the same data with different C parameter and the same σ may share the kernel table (i.e. use the same Kernel provider scheme and Kernel provider). Thus, only 80 different kernel tables are needed in the project (8 different values of σ and 10 different training data sets).

As a result of all the savings, out of **4538** ma-

Table 6: 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

chines comprising the project there are only **1928** different machines. Naturally it means completely different peak memory usage: more than **250 MB** and less than **60 MB** respectively. Moreover, less machines to be created and run, means also time savings: the analysis described above, for Wisconsin breast cancer data, with machine unification takes just **10.5 s** while without unification it takes **16.5 s** of a 2 GHz CPU. So different ratios describing savings in the number of machines, memory occupation and CPU time consumption result from the fact, that the machines that were unified in this project need much memory, but little time (the distributors that split data and data standardization machines; even SVM kernel providers did not affect the result significantly though their complexity is $O(n^2)$ in both space and time).

The unification possibilities are detected automatically and no algorithm is run twice with the same parameters, saving time and memory during project run and time of machine implementers, since they do not need to predict when exactly machines can be reused.

Currently, we have also several more advanced meta-learning tools, but we do not describe them in detail in this article, since here we concentrate on advantages of proper engine level solutions, from the point of view of memory and time savings.

9. Summary

Our plans of advanced meta-learning bore the need for very efficient data mining framework capable of handling very large projects. Although some kinds of meta parameter search are available in some systems available today, it would be very difficult to go forward to more complex meta-learning applications. Therefore, we have designed and implemented Intemi as a new versatile data analysis architecture, devoid of numerous drawbacks of other systems. The unified view of computational machines made meta-level analysis as simple as object-level processes. It opens gates to more advanced, trustworthy and successful research including very complex meta-learning and miscellaneous applications.

Here, we have described some of the crucial mechanisms of the new framework, including unprecedented solutions in machine representation, machine life cycle, task management (spooling and running), machine unification, results management and query system. The unification system guarantees that no machine process is run twice and no identical models are kept in memory. This yields very significant savings in time and memory consumption. Task spooling and running is also designed with special care about efficient usage of CPU time and available memory. Uniform results repository with specialized query language and rich set of series transformations provide robust tools for meta-level analysis of performed processes. All these features, augmented with two-tier cache system and other interesting solutions make the system unique in the realm of data mining frameworks.

The example of meta parameter search machine depicts how some of the new features of our system work in practice. Currently developed applications take even more advantage of the novel architecture, since the meta parameter search tool is just a starting point of the advanced data mining, feasible with Intemi.

We are currently working on more advanced meta-learning procedures performing

- general analysis of broad collections of diverse object-level learning algorithms,
- specialized explorations in some subgroups of machines like decision tree based classifiers,

to learn more on meta-level behavior of the algorithms and to improve object-level learning gains.

We also plan to prepare the system for distribution to a broad range of researchers. To make it possible, we need

- a fully-functional graphical user interface, to make all the possibilities available to the users not keen on programming, i.e. without the necessity to write any code,
- a thoroughly documented Software Development Kit, including wizards to support most common implementations of machines, queries etc.

It may also become sensible to create an external, large machine cache, also available on-line, to avoid repeated comparisons between different projects. These shall significantly reduce the time necessary for versatile tests of new meta-learning approaches.

References

[1] N. Jankowski, K. Grąbczewski, Learning machines, in: I. Guyon, S. Gunn, M. Nikravesh, L. Zadeh (Eds.), Feature extraction, foundations and applications, Springer, 2006, pp. 29–64.

[2] R. O. Duda, P. E. Hart, D. G. Stork, Pattern Classification, 2nd Edition, Wiley, 2001.

[3] T. Mitchell, Machine learning, McGraw Hill, 1997.

[4] C. M. Bishop, Neural Networks for Pattern Recognition, Oxford University Press, 1995.

[5] S. Haykin, Neural Networks - A Comprehensive Foundation, Maxwell MacMillian Int., New York, 1994.

[6] V. Cherkassky, F. Mulier, Learning from data, Adaptive and learning systems for signal processing, communications and control, John Wiley & Sons, Inc., New York, 1998.

[7] R. Schalkoff, Pattern Recognition: statistical, structural and neural approaches, Wiley, 1992.

[8] J. P. M. de Sá, Pattern Recognition. Concepts, Methods and Applications, Springer Verlag, 2001.

[9] T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Springer Series in Statistics, Springer, 2001.

[10] B. D. Ripley, Pattern Recognition and Neural Networks, Cambridge University Press, Cambridge, 1996.

[11] B. Schölkopf, A. Smola, Learning with Kernels, MIT Press, Cambridge, MA, 2002.

[12] I. Guyon, Nips 2003 workshop on feature extraction, http://www.clopinet.com/isabelle/Projects/NIPS2003 (Dec. 2003).

[13] I. Guyon, S. Gunn, M. Nikravesh, L. Zadeh, Feature extraction, foundations and applications, Springer, 2006.

[14] I. Guyon, Performance prediction challenge, http://www.modelselect.inf.ethz.ch (Jul. 2006).

[15] B. Pfahringer, H. Bensusan, C. Giraud-Carrier, Meta-learning by landmarking various learning algorithms, in: Proceedings of the Seventeenth International Conference on Machine Learning, Morgan Kaufmann, 2000, pp. 743–750.

[16] P. Brazdil, C. Soares, J. P. da Costa, Ranking learning algorithms: Using IBL and meta-learning on accuracy and time results, Machine Learning 50 (3) (2003) 251–277.

[17] H. Bensusan, C. Giraud-Carrier, C. J. Kennedy, A higher-order approach to meta-learning, in: J. Cussens, A. Frisch (Eds.), Proceedings of the Work-in-Progress Track at the 10th International Conference on Inductive Logic Programming, 2000, pp. 33–42.

[18] Y. Peng, P. Falch, C. Soares, P. Brazdil, Improved dataset characterisation for meta-learning, in: The 5th International Conference on Discovery Science, Springer-Verlag, Luebeck, Germany, 2002, pp. 141–152.

[19] J. S. Gero, G. J. Smith, Context, situations, and design agents, Knowledge-Based Systems 22 (8) (2009) 600–609.

[20] R. Lowry, Concepts and applications of inferential statistics, http://faculty.vassar.edu/lowry/webtext.html (2005).

[21] W. Duch, Software and datasets, http://www.is.umk.pl/~duch/software.html (2006).

[22] KDnuggets, Software suites for Data Mining and Knowledge Discovery, http://www.kdnuggets.com/software/suites.html (2009).

[23] I. I. Bittencourt, E. Costa, M. Silva, E. Soares, A computational model for developing semantic web-based educational systems, Knowledge-Based Systems 22 (4) (2009) 302–315.

[24] M. Gaeta, F. Orciuoli, P. Ritrovato, Advanced ontology management system for personalized e-learning, Knowledge-Based Systems 22 (4) (2009) 292–301.

[25] Institute of Parallel and Distributed High-Performance Systems (IPVR), Stuttgart Neural Networks Simulator (SNNS), http://www.informatik.uni-stuttgart.de/ipvr/bv/projekte/snns/snns.html.

[26] N. Jankowski, K. Grąbczewski, Building meta-learning algorithms basing on search controlled by machine's complexity and machines generators, in: IEEE World Congress on Computational Intelligence, IEEE Press, 2008, pp. 3600–3607.

[27] K. Grąbczewski, N. Jankowski, Meta-learning with machine generators and complexity controlled exploration, in: Artificial Intelligence and Soft Computing, Lecture notes in computer science, Springer, 2008, pp. 545–555.

[28] K. Grąbczewski, W. Duch, The Separability of Split Value criterion, in: Proceedings of the 5th Conference on Neural Networks and Their Applications, Zakopane, Poland, 2000, pp. 201–208.

[29] K. Grąbczewski, N. Jankowski, Mining for complex models comprising feature selection and classification, in: I. Guyon, S. Gunn, M. Nikravesh, L. Zadeh (Eds.), Feature extraction, foundations and Applications, Studies in fuzziness and soft computing, Springer, 2006, pp. 473–489.

[30] N. Jankowski, K. Grąbczewski, Handwritten digit recognition — road to contest victory, in: IEEE Symposium Series on Computational Intelligence, IEEE Press, USA, 2007, pp. 491–498.

Contents