Universal meta-learning architecture and algorithms

Norbert Jankowski norbert@is.umk.pl
Krzysztof Grąbczewski kg@is.umk.pl
Department of Informatics
Nicolaus Copernicus University
Toruń, Poland

Editor:

Abstract

There are hundreds of algorithms within data mining. Some of them are used to transform data, some to build classifiers, others for prediction, etc. Nobody knows well all these algorithms and nobody can know all the arcana of their behavior in all possible applications. How to find the best combination of transformation and final machine which solves given problem?

The solution is to use configurable and efficient meta-learning to solve data mining problems. Below, a general and flexible meta-learning system is presented. It can be used to solve different problems with computational intelligence, basing on learning from data.

The main ideas of our meta-learning algorithms lie in complexity controlled loop, searching for most adequate models and in using special functional specification of search spaces (the meta-learning spaces) combined with flexible way of defining the goal of meta-searching.

Keywords: Meta-Learning, Data Mining, Learning Machines, Computational Intelligence, Data Mining System Architecture, Computational Intelligence System Architecture

1. Introduction

Recent decades have brought large amount of data, eligible for automated analysis that could result in valuable descriptions, classifiers, approximators, visualizations or other forms of models. The Computational Intelligence (CI) community has formulated many algorithms for data transformation and for solving classification, approximation and other optimization problems (Jankowski and Grąbczewski, 2006). 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 (Guyon, 2003; Guyon et al., 2006), WCCI Performance Prediction Challenge (Guyon, 2006) in 2006 or other similar contests.

Most real life learning problems can be reasonably solved only by complex models, revealing good cooperation between different kinds of learning machines. To perform successful learning from data in an automated manner, we need to exploit meta-knowledge i.e. the knowledge about how to build an efficient learning machine providing an accurate solution to the problem being solved.

One of the approaches to meta-learning develops methods of decision committees construction, different stacking strategies, also performing nontrivial analysis of member mod-

els to draw committee conclusions (Chan and Stolfo, 1996; Prodromidis and Chan, 2000; Todorovski and Dzeroski, 2003; Duch and Itert, 2003; Jankowski and Grąbczewski, 2005). Another group of meta-learning enterprises (Pfahringer et al., 2000; Brazdil et al., 2003; Bensusan et al., 2000; Peng et al., 2002) base 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) and try to learn the relation between such data descriptions and accuracy of different learning methods. Although the projects are really interesting, they still suffer from many limitations and may be extended in a number of ways. 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.

In gating neural networks (Kadlec and Gabrys, 2008) authors use neural networks to predict performance of proposed *local experts* (machines proceeded by transformations) and decide about final decision (the best combination learned by regression) of the whole system. Another application of meta-learning for optimization problems by building relation between elements which characterize problem and performance of algorithms can be found in (Smith-Miles, 2008).

We do not believe that on the basis of some, not very sophisticated or expensive, description of the 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, however not a naive search throughout the whole space of possible models, but a search driven by heuristics protecting from spending time on learning processes of poor promise and from the danger of combinatorial explosion.

This article presents many aspects of our meta-learning approach. In Section 2 we present some basic assumptions and general ideas of our efforts. Section 3 presents the main ideas of the computational framework we have developed to make deeper meta-level analysis possible. Next, Section 4 describes the Meta Parameter Search Machine, which supports simple searches within the space of possible models. Section 5 is the most important part which describe main parts of meta-learning algorithm (definition of configuration of meta-learning, elements of scheme of main algorithm presented in Section 2, complexity control engine). Section 7 presents example application of proposed meta-learning algorithm for variety of benchmark data streams.

2. General Meta-learning Framework

First the difference between learning and meta-learning should be pointed out. Both the learning and meta-learning are considering in the context of learning from data, which is common around computational intelligence problems. Learning process of a *learning machine* $\mathcal{L}$ is a function $\mathcal{A}(\mathcal{L})$:

$$\mathcal{A}(\mathcal{L}) : \mathcal{K}_{\mathcal{L}} \times \mathcal{D} \to \mathcal{M}, \tag{1}$$

where $\mathcal{K}_{\mathcal{L}}$ represents the space of configuration parameters of given learning machine $\mathcal{L}$, $\mathcal{D}$ defines the space of data streams (typically a single data table, sometimes composed by few independent data inputs), which provide the learning material, and $\mathcal{M}$ defines the space of goal models. Models should play a role (assumed by $\mathcal{L}$) like classifier, feature selector, feature extractor, approximator, prototype selector, etc.

Indeed, learning should be seen as the process in which some *free parameters* of the machine M are adjusted or determined according to a *strategy* (algorithm) of the learning machine $\mathcal{L}$. After the learning process of $\mathcal{L}$, the model M should be ready to use as a classifier, data transformer, etc. depending on the goal of $\mathcal{L}$.

From such point of view meta-learning is another or rather specific learning machine. In the case of meta-learning the learning phase learn how to learn, to learn as well as possible. This means that the target model of a meta-learning machine (as the output of meta-learning) is a configuration of a learning machine extracted by meta-learning algorithm. The configuration produced by meta-learning should play the goal-role (like, already mentioned, classifier, approximator, data transformer, etc.) of meta-learning task. It is important to see that meta-learning is obligated to chose machine type (it may be even very complex one) and their strict configuration. This is because different configuration may provide incomparable behavior of given learning machine. Of course such definition does not indicate, how the meta-learning should search for the best type of learning machine and their best configuration.

Almost always meta-learning algorithms learn by observation and testing of nested learning machines. Meta-learning differ in the strategy of selection, which learning machines to observe and what to observe in chosen machines to find possibly best or at least satisfactory conclusions. From the theoretical point of view meta-learning, in general, is not limited in any way except the limitation of memory and time.

We propose a unified scheme of meta-learning algorithms (MLAs) which base on learning from observations. It is depicted in Figure 1.

The initialization step is a *link* between given configuration of meta-learning (which is very important question—see Section 5.2) and the further steps.

The meta-learning algorithm, after some initialization, starts the main loop, which up to the given *stop condition*, *runs* different learning processes, *monitors* them and *concludes* from their gains. In each repetition, it defines a number of tasks which test the behavior of appropriate learning configurations (i.e. configurations of single or complex learning machines)—step *start some tasks*. In other words, at this step it is decided, which (when and whether) machines are tested and how it is done (the strategy of given MLA). In the next step (*wait for any task*) the MLA waits until any test task is finished, so that the main loop may be continued. A test task may finish in a natural way (at the assumed end of the task) or due to some exception (different types of errors, broken by meta-learning because of exceeded time limit and so on). After a task is finished, its results are analyzed and *evaluated*. In this step some results may be accumulated (for example saving information about best machines) and new knowledge items created (e.g. about different machines cooperations). Such knowledge may have crucial influence on further parts of the meta-learning (tasks formulation and the control of the search through the space of learning machines). Precious conclusions may be drawn, even if a task is finished in a non-natural way.

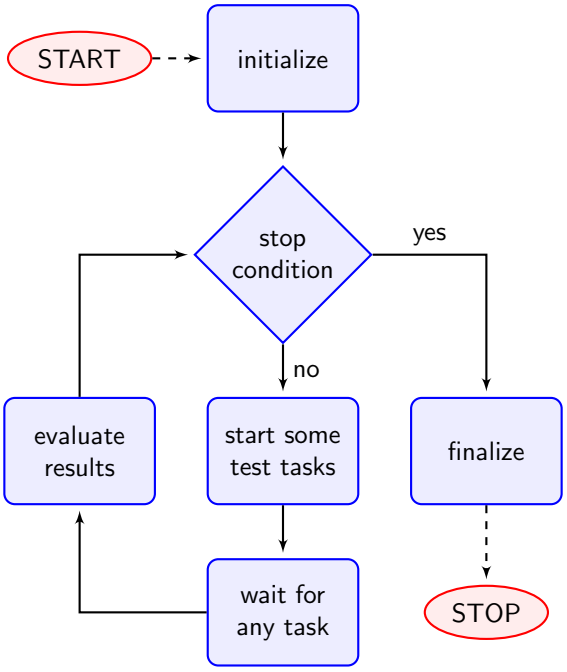


Figure 1: General meta-learning algorithm.

When the *stop condition* becomes satisfied, the MLA prepares and returns the final result: the configuration of chosen learning machine or, in more general way, even a ranking of learning machines (ordered by a degree of goal satisfaction), comments on chosen learning machines and their interactions, etc.

Each of the key steps of this general meta-learning algorithm may be realized in different ways yielding different meta-learning algorithms.

It is important to see that such a general scheme is not limited to a single strategy of MLA or searching by observing task by task (MLA autonomously decides about current group of started test tasks). This scheme does not apply any limits to the learning machine search space which in general can be a non-fixed space and may evaluate in the progress of meta search, for example to produce complex substitutions of machines. This opens the gates even to directing the functional space of learning machines according to collected meta-knowledge. Also the stop condition may be defined according to the considered problem and their limits.

This meta-scheme may be used to solve different types of problems. It is not limited only to classification or approximation problems.

First, note that finding an optimal model for given data mining problem $\mathcal{P}$ is almost always NP hard. Because of that, meta-learning algorithms should focus on finding approximation to the optimal solution independently of the problem type. Second, it would be very useful if the meta-learning could find solutions which at least are not worst than the ones that can be found by human experts in data mining in given limited amount of time.

„At least" because usually meta-learning should find more attractive solutions, sometimes even of surprising structure. In general meta-learning is more open to make deeper observation of intermediate test tasks and the search procedure may be more exhaustive and consistent. Experts usually restrict their tests to a part of algorithms and to some schemes of using them. Sophisticated meta-learning may quite easily overcome such disadvantages simultaneously keeping high level of flexibility.

This is why our general goal of the meta-learning is *to maximize the probability of finding, as optimal as possible, solution of given problem $\mathcal{P}$ in as short time as possible for determined searching space.*

As a consequence of such definition of the goal, the construction of meta-learning algorithm should carefully advise the order of testing tasks during the progress of the search and build meta-knowledge based on the experience from passed tests. Meta-knowledge may cover experience of so different kinds, among others: the correlations between subparts of machines in the context of performance, experience connected to the complexities of machines etc.

In our meta-learning approach, the algorithms search not only among *base* learning machines, but also produce and test different, sometimes quite complex machines like compositions of (several) transformations and classifiers (or other final e.i. decision making machines), committees of different types of machines, including complex ones (like composition of a transformer and a classifier as a single classifier inside the committee). Also, the transformations may be nested or compose chains. The compositions of complex machines may vary in their behavior and goal.

In the past, we have come up with the idea that meta-learning algorithms should favorite simple solutions and start the machines providing them before more complex ones. It means that MLAs should start with maximally plain learning machines, then they should test some plain compositions of machines (plain transformations with plain classifiers), after that more and more complex structures of learning machines (complex committees, multi-transformations etc.). But the problem is that the order of such generated tasks does not reflect real complexity of the tasks in the context of problem $\mathcal{P}$ described by data D. Let's consider two testing tasks T_1 and T_2 of computational complexities $O(mf^2)$ and $O(m^2f)$ respectively. Assume the data D is given in the form of data table and m is the number of instances and f is the number of features. In such case, it is not possible to compare time consumption of T_1 and T_2 until the final values m and f are known. What's more, sometimes a composition of a transformation and a classifier may be indeed of smaller complexity than the classifier without transformation. It is true because when using a transformation, the data passed to the learning process of the classifier may be of smaller complexity and, as a consequence, classifier's learning is simpler and the difference between the classifier learning complexities, with and without transformation may be bigger than the cost of the transformation. This proves that real complexity is not reflected directly by structure of learning machine.

To obtain the right order in the searching queue of learning machines, a complexity measure should be used. Although the Kolmogorov complexity (Kolmogorov, 1965; Li and Vitányi, 1993)

$$C_K(P) = \min_p\{l(p) \ : \text{program } p \text{ prints } P\} \tag{2}$$

is very well defined from theoretical point of view, it is unrealistic from practical side—the program p may work for a *very long* time. Levin's definition (Li and Vitányi, 1993) introduced a term responsible for time consumption:

$$C_L(P) = \min_p \{c_L(p) \ : \text{program } p \text{ prints } P \text{ in time } t^p\} \tag{3}$$

where

$$c_L(p) = l(p) + \log(t^p). \tag{4}$$

This definition is much more realistic in practical realization because of the time limit (Jankowski, 1995; Li and Vitányi, 1993). Such definition of complexity (or similar, as it will be seen further in this paper) helps prepare the order according to the real complexity of test tasks.

Concluding this section, the meta-learning algorithm presented below as a special realization of the general meta-learning scheme described above, can be shortly summarized by the following items:

- The search is performed in a functional space of different learning algorithms and of different kinds of algorithms. Learning machines for the tests will be generated using specialized machines generators.
- The mail loop is controlled by checking the complexity of the test tasks. Complexity control is also useful to handle with halting problems of subsequent tasks, started by meta-learning.
- Meta-learning collects meta-knowledge basing on the intermediate test tasks. Using this knowledge the algorithm provides some correction of complexities, and changes the behavior of advanced machine generators, what has crucial role in defining the meta-space of learning machines. The knowledge may be accumulated per given problem but also may survive like for example in the case of the knowledge about complexities. Meta-learning algorithms may use meta-knowledge collected from other learning tasks.

What can be the role of meta-learning in the context of *no free lunch* theorem? Let's start from another side, from the point of view of a learning machine which has satisfactory level of validated(!) performance on the training data and smallest complexity among other machines of similar performance, such simplest solution has the highest probability of success on a test set, and it was shown in literature from several perspectives like bias-variance, minimum description length, regularization, etc (Bishop, 1995; Duda et al., 2001; Hastie et al., 2001; Rissanen, 1978; Mitchell, 1997). From this side, in the case of classification problems, the process of meta-learning gives closer solution to the optimal Bayesian classifier than single (accidental?) learning machine.

The problem is that *no free lunch theorem* does not assume any relation of the distribution $P(X, Y)$ of the learning data D with the distribution $P(X', F(X'))$ $(X \subset X')$ of unknown target function $F(.)$ except being not contradictory at points of the data D.

Within the context of given learning data D, not all targets have similarly high (or highest) probability of evidence.

The *perfect learning machine* should discover not the origin-target, but the most probable target. In other words: the goal of generalization is not to predict an unpredictable model.

3. General System Architecture

Advanced data mining, including meta-learning, is not possible without a general and versatile framework for easy and efficient management of different models, performing tests etc. One of the main keys to such a system is a unified view of *machines* and *models*. We define a machine as any process that can be configured and run to bring some results. The results of the processes constitute models. For example an MLP network algorithm (Werbose, 1974) as the MLP machine can be configured by the network structure, initial weights, learning parameters etc. It can be run on some training data, and the result is a trained network—the MLP model created by the learning process of the MLP machine.

We deliberately avoid using 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 data from a disk file, data standardization or testing a classifier on external data.

A general view of a machine is presented in Figure 2. Before a machine may be created it

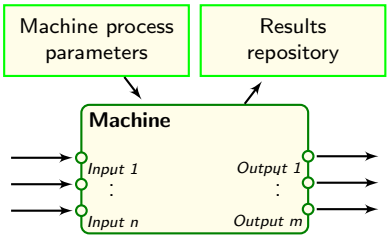


Figure 2: The abstract view of a machine.

must be completely configured and its context must be defined. Full 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 2 to keep the figure clear; in further figures, starting with Figure 3, the submachines are visible as boxes placed within the parent machine).

Machine context 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 machine to be created.

Some parts of machine configuration are fixed and do not require verbatim specification each time, a machine is created (e.g. the collection of inputs and outputs, for most machines, are always the same). Other configuration items usually have some (most common or most sensible) default values, so the machine user needs to specify only the items, that are different from the defaults.

A properly configured machine can be run (more precisely, the machine process is run to create the model). After the machine process is finished, the results may be deposited in the *results repository* and/or exhibited as *outputs*.

The inputs and outputs serve as sockets for information exchange between machines. The difference between machine inputs and configuration is that inputs come from other machines and the configuration contains the parameters of the process provided by the user. It is up to the machine author whether the machine receives any inputs and whether it has some adjustable parameters.

Similarly, machine outputs provide information about the model to other machines, and results repositories contain additional information about machine processes—the information that is not expected by other machines in the form of inputs.

The interconnections between outputs and inputs of different machines define the information flow within the project. Therefore, it is very important to properly encapsulate the CI functionality into machines. For more flexibility, each machine can create and manage a collection of submachines, performing separate, well defined parts of more complex algorithms. This facilitates creating multi-level complex machines while keeping each particular machine simple and easy to manipulate. An example of a machine with submachines is the repeater machine presented in Figure 3 The submachines are placed within the area of their

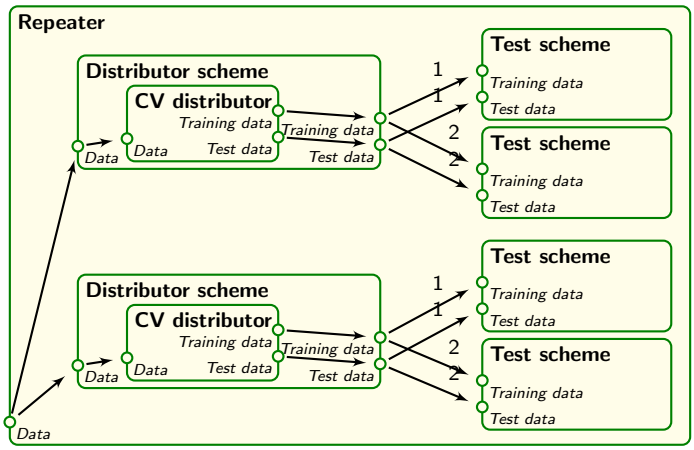


Figure 3: Run time view of Repeater machine configured to perform twice 2-fold CV. Test schemes are simplified for clearer view—in fact each one contains a scenario to be repeated within the CV, for example the one of Figure 4.

parent machines.

The repeater in the example, performed two independent runs of 2-fold cross-validation (CV). It has generated 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. In this example the repeater machine has 6 submachines, each having further submachines.

3.1 Schemes and Machine Configuration Templates

When configuring complex machines like the repeater in Figure 3, it is important to be provided with simple tools for machine hierarchy construction. To be properly defined, the repeater needs definitions of two schemes: one defining the distributor (i.e. the machine providing training and test data for each fold) and one to specify the test performed in each fold (Figure 4 shows an example test scenario, where kNN (k-nearest neighbors (Cover

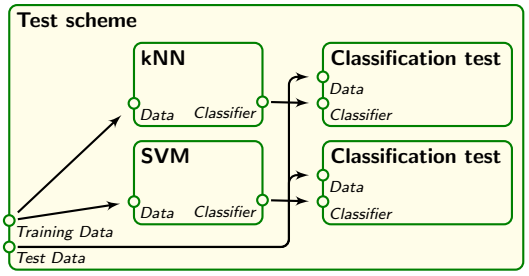


Figure 4: A test scheme example.

and Hart, 1967)) and SVM (support vector machines (Boser et al., 1992; Vapnik, 1998)) classifiers are, in parallel, created, trained on the training data and tested on the test data). At run time the repeater creates and runs the distributor scheme, and then creates and runs a number of test schemes with inputs bound with subsequent data series exhibited as distributor scheme outputs.

The repeater's children are *schemes*, i.e. machines especially designated for constructing machine hierarchies. Schemes do not perform any advanced processes for themselves, but just run graphs of their children (request creation of the children and wait till all the children are ready to eventually exhibit their outputs to other machines).

A machine configuration with empty scheme (or empty schemes) as child machine configuration (scheme which contain information only about types and names of inputs and outputs) is called a *machine template* (or more precisely a *machine configuration template*). Machine templates are very advantageous in meta-learning, since they facilitate definition of general learning strategies that can be filled with different elements to test a number of similar components in similar circumstances. Empty scheme may be filled by one or more configurations. The types of empty scheme inputs and outputs defines general type of role of scheme. For example, the feature selection template, presented in Figure 5, may be very useful for testing different feature ranking methods from the point of view of their eligibility for feature selection tasks. The dashed box represents a *placeholder* (empty scheme with

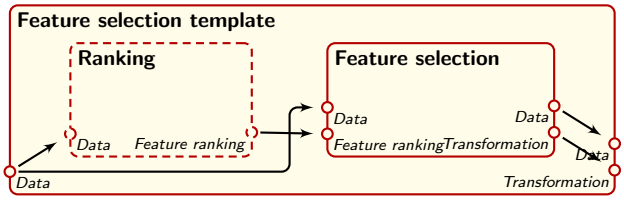


Figure 5: Feature selection template.

defined types and names of inputs and outputs) for a ranking. In case of ranking the scheme has single input with data and single output with information about ranking of features. After replacing the Ranking placeholder by a compatible configuration the whole construct can be created and run or put into another configuration for more complex experiments.

3.2 Query System

Standardization of machine results management makes the technical aspects of results analysis completely independent of the specifics of particular machines. Therefore, we have designed the *results repository*, where the information may be exposed in one of three standard ways:

- the machine itself may deposit its results to the repository (e.g. classification test machines put the value of accuracy into the repository),
- parent machines may comment their submachines (e.g. repeater machines comment their subschemes with the repetition and fold indices),
- special commentator objects may comment machines at any time (this subject is beyond the scope of this article, so we do not describe it in more detail).

The information within the repository has a form of label-value mappings.

Putting the results into the 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.

Queries facilitate collection and analysis of the results of machines in the project, it is not necessary to know the internals of the particular machines. It is sufficient to know the labels of the values deposited to the repository.

A query is defined by:

- the root machine of the query search,
- 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 run time hierarchy of submachines as in Figure 3 with test schemes as in Figure 4. After the repeater is finished, its parent wants to collect all the accuracies of SVM machines, so it runs the following code:

```
1 Query.Series results = Query(repeaterCapsule,
2     new Query.Qualifier.RootSubconfig(1, 3),
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, that 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. The 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 accepts 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. Intemi provides a number of qualifiers and labelers to make formulating and running miscellaneous queries easy and efficient. As a result we obtain a series of four descriptions (descriptions of four nodes) containing mappings of the label Accuracy to the floating point value of the accuracy.

In practice we are usually interested in some derivatives of the collected descriptions, not in the result series being the output of the query. For this purpose, Intemi provides a number of *series transformations* and tools for easy creation of new 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. More advanced predefined transformations perform results grouping, ungrouping, mapping group elements and calculate statistics for hypotheses testing including t-test, Wilcoxon test, McNemar test etc.

For the purpose of meta-learning we have encapsulated machine qualifier, labeler and final series transformation into the class of QueryDefinition. Running a query defined by the three components, in fact means collection of the results according to the qualifier and the labeler, and transforming the collected series with the transformation to obtain the final result of interest.

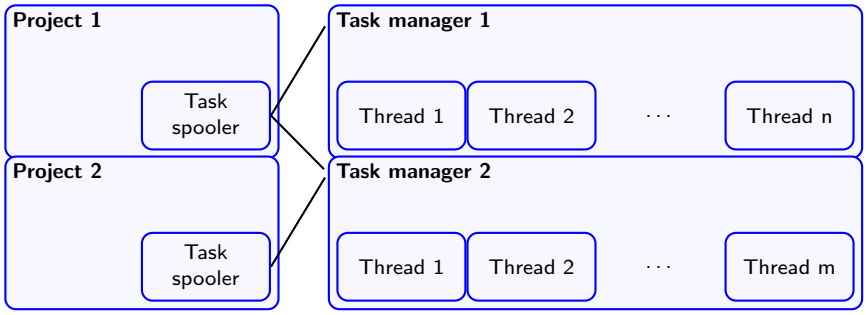


Figure 6: Two projects and two task managers.

3.3 Task Spooling

Before a machine request is fulfilled, i.e. the requested machine is created and its process run, the request context 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 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 from starting many unrelated tasks in parallel i.e. from too large consumption of memory and computation time. As a result, the spooler has the form of tree containing nodes with priorities.

Intemi environment delegates machine creation and running machine processes to separate task management modules. Each project can subscribe to services of any number of *task managers* executed either on local or remote computers (see Figure 6). 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 6, 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 another 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 requires just a single method call.

3.4 Machine Unification and Machine Cache

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 kept two equivalent models in memory. Therefore, Intemi introduced machine contexts as formal objects separate from proper machines. Different contexts may request for the same machine, and may share the machine.

Constructed machines are stored in *machine cache*, where they can be kept even after getting removed from the project. When another request for the same machine occurs, it can be restored from the cache and directly substituted instead of repeated creation and running. To achieve this, each machine request is passed to the machine cache, where the new machine configuration and context are compared to those, deposited in the cache. If unification is successful, the machine cache provides the machine for substitution.

Another unification possibility occurs between the requests pushed to the task spooler. Intemi controls unification also at this level preventing from building the same machine twice.

An illustrative example of machine unification advantages can be a project testing different feature ranking methods. Table 1 shows feature rankings obtained for Wisconsin

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

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

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 (Grąbczewski and Duch, 2000)). 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.

In simple approaches such repetitions can be easily avoided by proper machine process implementation, but in complex projects, it would be very difficult to foresee all the redundancies (especially in very complicated meta-learning projects), so Intemi resolves the problem at the system level by means of machine unification.

4. Parameter Search Machine

One of the very important features of meta-learning is the facility of optimization of given machine configuration parameters, for example, the number of features to be selected from a data table. Parameter optimization may be embedded in the algorithm of given learning machine or the optimization may be performed outside of the learning machine. In some cases, from computational point of view, it is more advantageous to realize embedded optimization—by implementing optimization inside given learning process. However it is rather rare and in most cases the optimization must not be realized in the embedded form without loss of complexity and time of optimization.

As presented in previous sections, in so general purpose system, the machine devoted to optimize parameters of machine configurations must be also very general and ready to realize different search strategies and must be open to extensions by new search strategies in feature. The new search strategies are realized as new modules for optimization machine and extend the possibilities of searching in new directions.

Presented *parameters search machine* (PSM) can optimize any elements of machine configuration[1]. The optimization process may optimize any elements of configuration and any element of subconfigurations (including subsubconfigurations etc.). Also subelements of objects in (sub-)configurations can be optimized. This is important, because so often, machines are very complex and their configurations are complex too, then the elements of optimization process are sometimes deeply nested in complex structures. Thus, a mechanism of pointing such elements is mandatory in flexible optimization definition. The search and optimization strategies are realized as separate modules which realize appropriate functionality. Because of that, the search strategies are ready to provide optimization of any kind of elements (of configurations) even the abstract (amorphic) structures can be optimized. Such structures may be completely unknown for PSM, but given search strategy knows what to do with objects of given structure. Search strategies provides optimization of a single or a set of configuration elements during the optimization process. It is important to notice that in so general system, sometimes even changing simple scalar parameter, the behavior of machine may change very significantly.

The PSM uses test procedures to estimate the objective functions as the quality test to help the search strategy undertake the next steps. The definition of such test must be realized in very open way to enable using of PSM to optimize machines of different kinds and in different ways.

Such general behavior of MPS was obtained by flexible definition of configuration of MPS combined with general optimization procedure presented below. Let's start the description of configuration of MPS. The MPS configuration consists of (not all elements are obligatory):

Test Template: It determines the type of test measuring influence of the parameters being optimized to the quality of the resulting model. The test template may be defined in many different ways and may be defined for different types of problems. The most typical test used in such case for the classification problems is the cross-validation test of chosen classifier (sometimes classifier is defined as complex machine). This is a mandatory part of configuration.

1. It is possible to optimize single or several parameters during optimization, sequentially or in parallel, depending on used search strategy.

Path to the final machine: Variable PathToFinalConfigurationInTemplate in the code presented below, defines a place (a path) of subconfiguration in the test template, which will become the final configuration machine of MPS. For example, in the case of optimizing a classifier this path will point to the classifier in the test template and after optimization process based on configuration pointed by this path in final configuration, the final configuration of classifier will be extracted, and finally the MPS will *play the role* of the classifier (which means that this classifier will be an output of the MPS on finish). This parameter is not obligatory. If this parameter is not defined in configuration of MPS, the MPS will return just the final (optimized) configuration as the result of the optimization procedure.

Query definition: For each machine configuration created in the optimization procedure (see below) the quality test must be computed to advise further process of optimization and final choice of configuration parameters. The query is realized exactly as it was presented in Section 3.2. The test template (for example cross-validation) will produce several submachines with some results like accuracy, which describe the quality of each subsequent test. The query definition specifies which machines (the machine qualifier) have to be asked for results and which labels (the machine labeler) provides interesting values to calculate the final quantity. As a result, series of appropriate values are constructed. The last element of query definition defines how to calculate the result-quantity (single real value) on the basis of previously obtained series of values. Query definition is obligatory in MPS configuration.

Scenario or ConfigPathToGetScenario: These are two alternative ways to define the scenario (i.e. the strategy) of parameter(-s) search and optimization. Either of them must be defined. If the scenario is defined, then it is a direct scenario is defined directly. If the path is defined, it points the configuration which is expected to support auto-reading of the default scenario for this type of configuration.

The *scenario* defined within the configuration of PSM determines the course of the optimization process. Our system contains a number of predefined scenarios and new ones can easily be implemented. The main, obligatory functionalities of the scenarios are:

SetOptimizationItems: each scenario must specify which element(-s) will be optimized. The items will be adjusted and observed in the optimization procedure. This functionality is used at the configuration time of the scenario, not in the optimization time.

NextConfiguration: subsequent calls return the configurations to be tested. The PSM, inside the main loop, calls it to generate a sequence of configurations. Each generated configuration is additionally commented by the scenario to enable further meta-reasoning or just to inform about the divergence between subsequent configurations. The method NextConfiguration returns a boolean value indicating whether a new configuration was provided or the scenario stopped the process of providing next configurations (compare line 8 of the code of MPS shown below).

RegisterObjective: scenarios may use the test results computed for generated configurations when generating next configurations. In such cases, for each configuration

provided by the scenario, the MPS, after the learning process of such task, runs the test procedure, and the value of the quality test is passed back to the scenario (compare above comments on Query definition and line code 12) to inform the *optimization strategy* about the progress.

High flexibility of the elements of MPS, described above, enable creation of optimization algorithm in relatively simple way, as presented below.

```
4  function MetaParamSearch(TestTemplate, Scenario,
5    PathToFinalConfigurationInTemplate, QueryDefinition);
6    Scenario.Init(TestTemplate);
7    ListOfChangeInfo = {};
8    while (Scenario.NextConfiguration(config, changes))
9    {
10     t = StartTestTask(config);
11     qt = computeQualityTest(t, QueryDefinition);
12     Scenario.RegisterObjective(config, qt);
13     ListOfChangeInfo.Add(<qt, changes>);
14     RemoveTask(t);
15   }
16   if (defined PathToFinalConfigurationInTemplate)
17   {
18     confM = config.GetSubconfiguration(
19       PathToFinalConfigurationInTemplate);
20     c = CreateChildMachine(confM);
21     SetOutput(c);
22   }
23   return <config, ListOfChangeInfo>;
24 end
```

In line 6 the scenario is initialized with the configuration to be optimized. It is important to note that the starting point of the MPS is not a default configuration of given machine type but strict configuration (testing template with strict configuration of *adjustable* machine) which may be a product of another optimization. As a consequence, starting the same MPS machine with different optimized configurations may finish in different final configurations—for example tuning of the SVM with linear kernel or tuning of the SVM with Gaussian kernel finish in completely different states.

Every configuration change is *commented* by the chosen scenario. This is useful in further analysis of optimization results—see line 7.

The main loop of MPS (line 8) works as long as any new configuration is provided by the scenario. When NextConfiguration returns **false**, the variable config holds the final configuration of the optimization process and this configuration is returned by the last line of the MPS code. After a new configuration is provided by the scenario, a test task is started (see line 10) to perform the test procedure, for example the cross-validation test of a classifier. After the test procedure, the QueryDefinition is used to compute the quality test. The quality test may be any type of attractiveness measure (attractiveness ≡ reciprocal of error). For example the accuracy or negation of the mean squared error. The

resulting quality is sent to the scenario, to inform it about the quality of the configuration adjustments (line 12). Additionally, the resulting quality value and the comments on the scenario's adjustments are added to the queue ListOfChangeInfo.

Finally the test task is removed and the loop continues.

When the PathToFinalConfigurationInTemplate is configured, the MPS builds the machine resulting from the winner configuration—see code lines 16–22. The type of the machine is not restricted in any way—it depends only on the configuration pointed by PathToFinalConfigurationInTemplate.

The MPS algorithm finishes by returning the final (the best) configuration and the comments about the optimization process.

4.1 Examples of Scenarios

Intemi provides a number of predefined scenarios. At any time, the system may be extended by new scenarios. Below, we present some simple, not nested scenarios and then, more complex, nested ones.

The most typical scenario, used for optimization of many learning machines' parameters, is the *StepScenario* based on quite simple idea to optimize single chosen element of a configuration. Such element is defined (not only in the case of that scenario) by two paths which constitute the scenario configuration. The first is the subconfiguration path, which goal is to define in which (sub-)configuration the optimization parameter lies (at any depth). When the path is empty, it points the main configuration. The second path is the property path, pointing (in configuration pointed by first path) to the property (or sub-sub-...property) which is the element to be optimized. The property path may not be empty (something must be pointed!). The search strategy is very simple, it generates configurations with the element set to values from a sequence defined by start value, step value and the number of steps. Step-type of StepScenario may be linear, logarithmic or exponential. This is very convenient because of different behaviors of configuration elements. This scenario may be used with real or integer value type. It may optimize, for example, the number of selected features (for feature selection machine) or the SVM's C parameter or the width of the Gaussian kernels.

The *SetScenario* is powerful in the cases, when the optimized parameter is not continuously valued. It can be used with every enumerable type (including real or integer). Because this scenario is type independent, the examples of using it are quite diverse. In the case of k nearest neighbors, the metric parameter may be optimized by setting the configuration of metric to each of the elements of a set of metrics, for example the set of Euclidean, Manhattan and Chebyshev metrics. In cases where a few fixed values should be checked as the values of given configuration element, the step scenario may be configured to the fixed set of values, for example: 2, 3, 5, 8, 13, 21. The determination of the optimization parameter is done in the same way as in the case of StepScenario: by two paths which point the subconfiguration and the subproperty.

SetAndSetSubScenario is, in some way, a more general version of the previous scenario. It also checks a set of enumerable values attached to given configuration element, but additionally it is able to start sub-scenario declared for all or selected values from the mentioned enumerable set. Configuration of this scenario consists of a set of pairs: $<$

$value_k, scenario_k >$. For each pair, an element of configuration is set to $value_1$ and the subscenario is started to optimize another element of configuration by $scenario_1$, if only $scenario_1$ is not null. The scenario can be used, for example, to chose metric (between some metric from a set as before) and, in the case of Minkovsky metric, additionally tune (via nested scenario) the parameter which represents the power in the Minkovsky metric.

Another very useful scenario is the *StackedScenario* which facilitates building a sequence or a combination of scenarios. This scenario may be used in one two modes. The first mode, the *sequence mode*, enables starting scenarios one by one, according to the order in the sequence. In the *grid mode* each value generated by the first scenario is tried against each value provided by second scenario and so on. For example, assume that we need to optimize the C parameter of the SVM and also the width of the Gaussian kernel. Using the sequence mode it is possible to optimize the C first, and after that, the width or in the reversed order. Also, it is possible to optimize the C first, then the width and then again the C, and so on. In grid mode, every pair of parameters will be checked (every C against every width). In general, the grid may be composed of more than two scenarios, then such scenario will try each triple, each four and so on. Note that the stacked scenario may be composed of any scenarios, including stacked scenario or set scenario, however too complex optimizations are not recommended because of growing complexity.

SimpleMaximumScenario may be used to look for maximum, assuming the single maximum problem (or looking for local minimum), observing the quality test returned to the scenario. When using this scenario it is recommended to put a limit on the number of steps beside the limit on progress of optimization. Of course this scenario may be extended in several simple ways to more sophisticated versions.

4.2 Auto-scenario

If the optimization of given machine or its single chosen parameter is to be performed typically it is done, in most cases, in the same way. This suggests that it is not necessary (and not recommended) to rediscover the method of parameters tuning in each optimization of given machine. The auto-scenarios is the idea to ,,dedicate" the behavior of optimization for configuration parameters and whole learning machines. In the presented system it is done using the scenario-comment attributes which define the default way of optimization. Scenario-comments are used with configuration elements and also with the whole configurations of learning machines. The difference is that, for the whole machine, it is responsible for optimization of the whole machine configuration in possibly best way, while for single configuration elements it is responsible to optimize the pointed element of configuration without any change of other elements.

Such idea of auto-scenarios is very convenient, because it simplifies the process of optimization without loss of quality. Nobody is obliged to know the optimal way of parameter optimization for all machines. The scenario comments compose a brilliant base of meta-knowledge created by experts—using auto-scenarios, meta-learning does not have to rediscover the optimal way of optimization of each particular machine. In meta-learning auto-scenarios are used together with machine generators to auto-generate optimization strategies of chosen machine configuration. It is also possible to define a few default optimization ways for one machine. This enables using a simpler and a deeper optimiza-

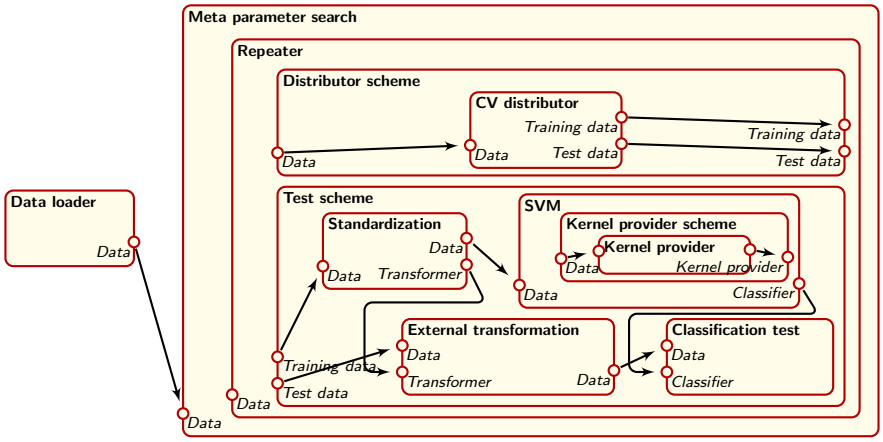


Figure 7: A meta parameter search project configuration.

tion scenarios and depending on the circumstances, one may be chosen before the other. Auto-scenarios can be attached to any machines, both the simple ones and the complex hierarchical ones.

4.3 Parameter Search and Machine Unification

As mentioned in the preceding section, machine unification is especially advantageous in meta-learning. Even one of the simplest meta-learning approaches, a simple meta parameter search, is a good example. Imagine a project configuration depicted in Figure 7, where the MPS machine is designed to repeat 5 times 2-fold CV of the test template scenario for different values of the SVM C and kernel σ parameters. Testing the C parameter within the set $\{2^{-12}, 2^{-10}, \ldots, 2^{2}\}$ and σ within $\{2^{-1}, 2^{1}, \ldots, 2^{11}\}$ we need to perform the whole 5×2 CV process 8×7 times. As enumerated in Table 2, such a project contains (logically)

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

Table 2: Numbers of machines that exist in the project logically and physically.

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

5. Meta-learning Algorithm Elements

This section provides description of all the parts of the presented meta-learning algorithm. First, machine configuration generators, which represent and define the functional form of meta-learning search space, are described. Next, a clear distinction between the configuration of the algorithm and the main part of the algorithm is stated. After that, other elements of the main part of the algorithm will be presented.

5.1 Machine Configuration Generators and Generators Flow

In the simplest way, the machine space browsed by meta-learning may be realized as a set of machine configurations. However such solution is not flexible and strongly limits the meta-learning possibilities to fixed space. To overcome this disadvantage the *machine configuration generators* (MCG) are introduced.

The main goal of MCGs is to provide/produce machine configurations. Each meta-learning may use several machine configuration generators nested in a **generators flow** (a graph of generators). Each MCG may base on different meta-knowledge, may reveal different behavior, which in particular may even change in time (during the meta-learning progress). The simplest generators flow is presented in Figure 8.

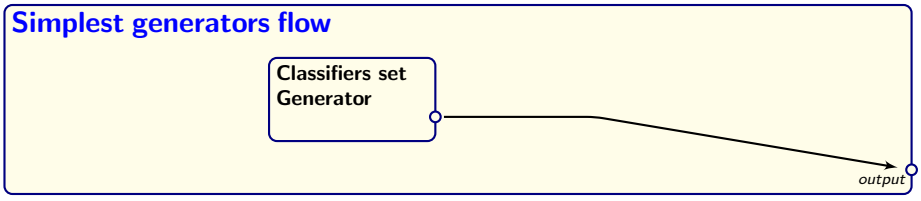


Figure 8: Example of simplest generator flow.

Each generator provides machine configurations through its output. Generators may also have one or more inputs, which can be connected to outputs of other generators, similarly to machines and their inputs and outputs. If the output of generator A is connected to an input number 3 of generator B, then every machine configuration generated by the generator A will be provided to the input 3 of the generator B. If a generator output is attached to more inputs, then each of the inputs will receive the output machine configurations.

The inputs-outputs connections between generators, compose a generators flow graph, which must be a directed acyclic graph. A cycle would result in infinite number of machine configurations. Some of the outputs of generators can be selected as the output of the generators flow—the providers of configurations to the meta-learning. In the run time

of the meta-learning algorithm, the configurations returned by the generators flow, are transported to a special heap, before the configured machines are tested.

The streams of configurations provided by generators may be classified as fixed or non-fixed. *Fixed* means that the generator depends only on its inputs and configuration. The non-fixed generators depend also on the learning progress (see the advanced generators below).

Another important feature of generators is that, when a machine configuration is provided by a generator, information about the *origin* of configuration and some comments about it are attached to the machine configuration. This is important for further meta-reasoning. It may be useful to know information on how the configuration was created in further meta-reasoning.

In general, generators behavior is not limited except the fact, that they should provide a finite series of configurations. Below we describe some generators examples.

5.1.1 Set-based Generators

The simplest and very useful machine generator is based on the idea to provide just an arbitrary sequence of machine configurations (thus the name *set-based generator*). Usually, it is convenient to have a few set-base generators in single generators flow—a single set per a group of machines of similar functionality like feature rankings or classifiers. An example of set-base generator providing classifier machine configurations was presented in Figure 8.

5.1.2 Template-based Generators

Template-based generators are used to provide complex configurations based on given machine configuration template (described in Section 3.1) and generators connected to them. For example, if a meta-learner is to search through combinations of different data transformers and kNN classifier, it can easily do it with a template-based generator. The combinations may be defined by a machine template with a placeholder for data transformer and fixed kNN classifier. Such a template-based generator may be connected to a set-based generator with a sequence of data transformations, which would make the template-based generator provide a sequence of complex configurations resulting from replacing the placeholder with subsequent data transformation configurations. Please, note that, in the example, the machine template is to play the role of a classifier. Because of that, we can use the Transform and Classify machine template shown in Figure 9 on the left. This machine learning process starts with learning the data transformation first and then the classifier.

The generator's template may contain more than one placeholder. In such a case the generator needs more than one input. The number of inputs must be equal to the number of placeholders. The role of a placeholder is defined by its inputs and outputs declarations. So, it may be a classifier, approximator, data transformer, ranking, committee, etc. Of course the placeholder may be filled with complex machine configuration too.

Replacing the kNN from the previous example by a classifier placeholder (compare Figure 9 on the right), we obtain a template that may be used to configure a generator with two inputs. One designated for a stream of transformers, and the other one for a stream of classifiers.

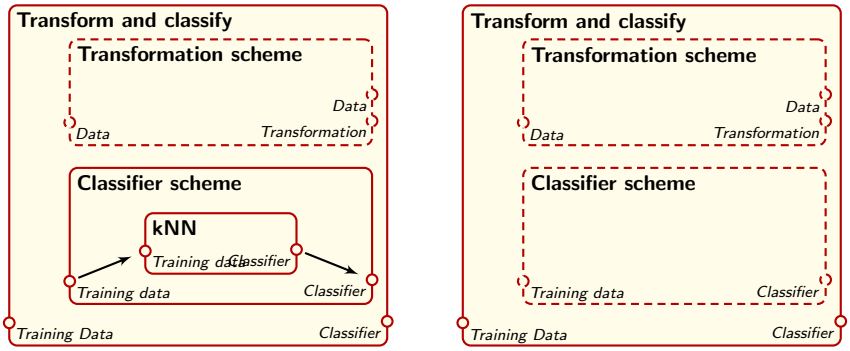


Figure 9: A Transform and classify machine configuration template. LEFT: placeholder for transformer and fixed classifier (kNN). RIGHT: two placeholders, one for the transformer and one for the classifier

The template-base generator can be configured in one of two modes: *one-to-one* or *all-to-all*. In the case of the example considered above, mode 'one-to-one' makes the template-based generator get one transformer and one classifier from appropriate streams and put them into the two placeholders to provide a result configuration. The generator repeats this process as long as both streams are not empty. In 'all-to-all' mode the template-based generator combines each transformer from the stream of transformers with each classifier from the classifiers stream to produce result configurations..

Figure 10 presents a simple example of using two set-based generators and one template-based generator. The set-based generators provide transformers and classifiers to the two

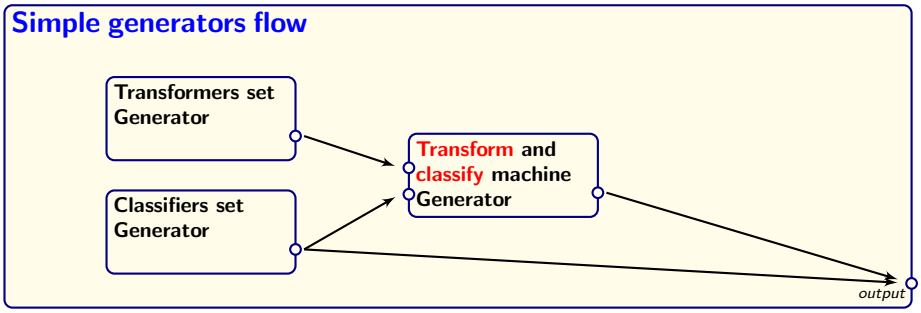


Figure 10: A simple generator flow.

inputs of the template-based generator, which puts the configurations coming to its inputs to the template providing fully specified configurations. Different mixtures of transformations and classifiers are provided as the output, depending on the mode of the generator: one-to-one or all-to-all. Please, note that the generator's output gets configuration for both the set-based classifiers generator and the template-based generator, so it will provide both the

classifiers taken directly from the proper set and the classifiers preceded by the declared transformers.

Another interesting example of a template-based generator is an instance using a template of ParamSearch machine configuration (the ParamSearch machine is described in Section 4). The template containing test scheme with a placeholder for a classifier is very useful here. When the input of the template-based generator is connected to a stream of classifiers, the classifiers will fill the placeholder, producing new configurations of the ParamSearch machine. Configuring the ParamSearch machine to use the *auto-scenario* option makes the meta-learner receive configurations of ParamSearch to realize the auto-scenario for given classifier. It means that such a generator will provide configurations for selected classifiers to auto-optimize their parameters.

An more complex generator flow using the generators presented above, including template-based generator with ParamSearch machine is presented in Figure 11. This generator flow contains three set-based generators, which provide classifiers, transformers and ranking configurations to other (template-based) generators. Please, see that the classifier generator

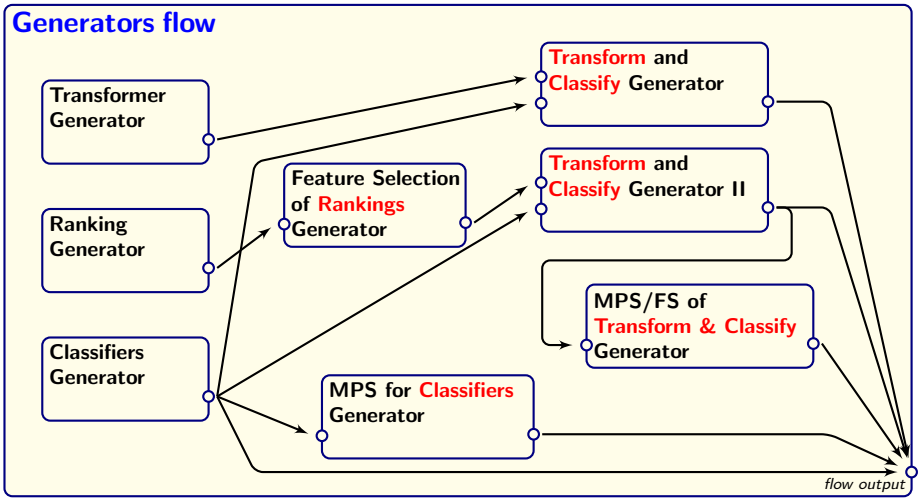


Figure 11: Example of generators flow.

sends its output configurations directly to the generator flow output and additionally to three template-based generators: two combining transformations with classifiers and the ParamSearch generator (MPS for Classifiers Generator). It means that each classifier configuration will be send to the meta-learning heap and additionally to other generators to be nested in other configurations (generated by the Transform and Classify and the MPS generators).

The two Transform and Classify generators combine different transformations with the classifiers obtained from the Classifiers Generator. The configurations of transformation machines are received by proper inputs. It is easy to see, that the first Transform and Classify generator uses the transformations output by the Transformer Generator while the second one receives configurations from another template-based generator which generated feature selection configurations with the use of different ranking algorithms received through

the output-input connection with the Ranking Generator. The combinations generated by the two Transform and Classify generators are also sent to the meta-learning heap through the output of the generators flow.

Additionally, the Transform and Classify Generator II sends its output configurations to the ParamSearch generator (MPS/FS of Transform & Classify generator). This generator produces ParamSearch configurations, where the number of features is optimized for configurations produced by the Transform and Classify Generator II. The output of the ParamSearch generator is passed to the output of the generators flow too.

In such a scheme, a variety of configurations may be obtained in a simple way. The control of template-based generators is exactly convergent with the needs of meta-search processes.

There are no a priori limits on the number of generators and connections used in generator flows. Each generator flow may use any number of any kind of generators. In some cases it is fruitful to separate groups of classifiers in independent set-based generators to simplify connection paths in the generators flow graph. The same conclusion is valid also for transformations which grouped into separate sets may facilitate more flexibility, for example when some machines have to be preceded by a discretization transformation or should be used with (or without) proper filter transformations.

5.1.3 Advanced Generators

Advanced generators are currently under development. Their main advantage, over the generators mentioned above, is that they can make use of additional meta-knowledge. A meta-knowledge, including the experts' knowledge, may be embedded in a generator for more intelligent filling of placeholders in the case of template-based generators. For example, generators may *know* that given classifier needs only continuous or discrete features.

Advanced generators are informed each time a test task is finished and analyzed. The generators may read the results computed within the test task and the current machine configuration ranking (compare code line 67 in Section 5.5). The strategy enables providing machine configurations derived from an observation of the meta-search progress. Advanced generators can learn from meta-knowledge, half-independently of the heart of the meta-learning algorithm, and build their own specialized meta-knowledge. It is very important because the heart of meta-learning algorithm can not (and should not) be responsible for specific aspects of different kind of machines or other *localized* type of knowledge.

5.2 Configuring Meta-learning Problem

It is crucial to see the meta-learning not as a wizard of everything but as a wizard of consistently defined meta-learning task. The meta-learning does not discover the goal itself. The meta-learning does not discover the constraints on the search space. Although the search space may be adjusted by meta-learning. To define a meta-learning algorithm that can be used for different kinds of problems and that can be adjusted not by reimplementing but through configuration changes, it must be designed very flexibly.

5.2.1 Stating of Functional Searching Space

A fundamental aspect of every meta-learning is determination of the search space of learning machines. The simplest solution is just a set of configurations of learning machines. Although it is acceptable that the configurations can be modified within the meta-learning process, a single set of configurations still offers strongly limited possibilities. A functional form of configurations of learning machines is much more flexible. This feature is realized by the idea of *machine configuration generators* and the *machine generators flow* described in Section 5.1. Using the idea of machine configuration generators and their flows, defining the search space for meta-learning is much more powerful and intuitive. One of the very important features is, that generators may describe non-fixed spaces of configurations. This means that the space may continuously evolve during the meta-search procedure. The graph of generators may be seen as continuously working factory of machine configurations. Defining meta-search space by means of machines generators fulfill all or almost all the needs of applications to miscellaneous types of problems.

5.2.2 Defining the Goal of Meta-learning

Another very important aspect of configuration of a meta-learner is the possibility of strict and flexible definition of the goal. In the context of our general view of MLA (please look back at Figure 1), we focus on the step *start some test tasks*. Creation of test tasks is not learning a machine according to the chosen configuration, but preparation of the *test procedure* of the chosen configuration. From technical point of view, it is nothing else than building another complex machine configuration, performing the test procedure i.e. validating the chosen machine configuration. This is why the definition of the test procedure may be encapsulated in a functional form of *machine configuration template* (compare Section 3.1) with nested placeholder for chosen configuration to be validated. The meta-learning takes the test machine template, fills the placeholder with the chosen machine configuration and starts the test task. For example, when the MLA is to solve a classification problem, a classifier test must be defined as the *test machine template*—in particular it may be the commonly used repeated cross-validation test.

After normal finish of a test task the MLA needs to *evaluate the results* i.e. to determine a quantity estimating, in some way, the quality of the test results. When the test task is finished, the MLA has the possibility to explore it. The test task has a form of a tree of machines (compare Figure 3). In our approach, to compute the *quality of the results* it is necessary to define:

- which nodes of the machine tree preserve the values with the source information for computing the *quality-test*,
- which values at given nodes are the source of required information,
- how to use the values collected from the tree to measure the quality of given test.

And this is exactly the same information type as those presented in Section 3.2, where to define a *query* we had to define: the *root machine* (the root of the tree to search—the test machine in this case), *qualifier* which selects the nodes of the tree containing the information, the *labeler* which describes the selected nodes with labels corresponding to the

desired values and the *series transformation*, which determines the final value of the quality estimate. The example of a query, presented in Section 3.2, clearly shows that it is a simple and minimal mean to compute qualities of given test tasks. Thanks to such a form of the quality test definition, the MLA can be simply adapted to very different types of quality tests, independently from the problem type.

5.2.3 Defining the Stop Condition

Another important part of the MLA configuration is the specification, when to stop. The stop condition can be defined in several ways, for example:

- stop after given *real time limit*,
- stop when given *CPU time limit* is achieved,
- stop after given *real time limit* or no important progress in the quality test is observed for given amount of time (defined by progress threshold and progress tail length).

Of course the meta-learning may be stopped in other ways—by implementing another stop-function. An interesting extension may be intelligent stop functions, i.e. functions, which make use of meta-knowledge in their decisions.

5.2.4 Defining the Attractiveness Module

If the attractiveness module is defined, then the corrections to the complexity of machines will be applied, to give additional control of the order of tests in the machine space exploration. It is described in more detail in Section 6.1.

5.2.5 Initial Meta-knowledge.

A meta-knowledge is almost always passed to the MLA by means of the configuration items such as the machine configuration generators, the attractiveness module or the complexity computation module. It is very important include appropriate meta-information at the initialization stage of the search process, since it may have very important influence on the results of the meta-learning search.

Summing up the above section, beside the attractiveness module, all the configuration elements, explained above, compose the minimal set of information to realize the meta-learning search. It is impossible to search in undefined space or search without strictly defined goal. All the parts of the configuration can be defined in very flexible way. At the same time, defining the search space by the machine configuration generators, specifying the goal by the query system, determining the stop condition, are also quite simple and may be easily extended for more sophisticated tasks.

5.3 Initialization of Meta-learning Algorithm

The general scheme of Figure 1, can be written in a simple meta-code as:

```
25 procedure Meta_learning;
26   initialization;
```

```
27   while(stopCondition != true)
28   {
29     start tasks if possible
30     wait for finished task or break delayed task
31     analyze finished tasks
32   }
33   finalize;
34 end
```

At the initialization step, the meta-learning is set up in accordance to the configuration. Among others, the goal (testing machine template and query definition) and search space and the stop-condition of meta-learning are defined. The machine search space is determined by machine configuration generators embedded in a generator flow. See lines 36–41 in the code below.

The machinesRanking serves as the ranking of tested machines according to the quality test results obtained by applying the queryDefinition to the test task. machinesRanking keeps information about the quality of each configuration, and about their origin (it is necessary to understand how it was evolving). The heap named machinesHeap (line 44) will keep machine configurations organized according the complexity i.e. will decide about the order of test tasks starting. The machineGeneratorForTestTemplate is constructed as a machine generator based on the testing template. The goal of using this generator is to nest each machine configuration provided by the generators flow inside the testing machine template (see line 46) which is then passed through the machineHeap to start, and later to test the quality of provided machine. This is why machineGeneratorForTestTemplate has to be connected to the machinesHeap (line 47).

```
35 procedure Initialization;
36   read configuration of ML and
37   set machineGeneratorsFlow
38   set testingTemplate
39   set queryDefinition
40   set stopCondition
41   set attractivenessModule
42   machinesRanking = {};
43   priority = 0;
44   machinesHeap = {};
45   machineGeneratorForTestTemplate =
46     MachineGenerator(testingTemplate);
47   machineGeneratorForTestTemplate.ConnectTo(machinesHeap);
48 end
```

5.4 Test Tasks Starting

Candidate machine configurations are passed through a heap structure (machineHeap), from which they come out in appropriate order, reflecting machine complexity. The heap and complexity issues are addressed in detail, in Section 6. According to the order decided within

the heap, procedure *start_tasks_if_possible*, sketched below, starts the simplest machines first and than more and more complex ones.

```
49 procedure startTasksIfPossible;
50   while (¬ machinesHeap.Empty() ∧ ¬ mlTaskSpooler.Full())
51   {
52     <mc, cmplx> = machinesHeap.ExtractMinimum();
53     timeLimit = τ · cmplx.time / cmplx.q
54     mlTaskSpooler.Add(mc, limiter(timeLimit), priority−−);
55   }
56 end
```

Tasks are taken from the machinesHeap, so when it is empty, no task can be started. Additionally, the task-spooler of meta-learning must not be full. MLAs use the task spooler described in Section 3.3 but via additional spooler which controls the width of the beam of tasks waiting for run.

If the conditions to start a task (line 50 of the code) are satisfied, then a pair of machine configuration mc and its corresponding complexity description cmplx is extracted from machinesHeap (see line 52).

As commented above, the complexity is approximated for given configuration. Since it is only an approximation, the meta-learning algorithm must be ready for cases when this approximation is not accurate or even the test task is not going to finish (according to the halting problem or problems with convergence of learning). To bypass the halting problem and the problem of (the possibility of) inaccurate approximation, each test task has its own time limit for running. After the assigned time limit the task is aborted. In line 53 of the code, the time limit is set up according to predicted time consumption (cmplx.time) of the test task and current reliability of the machine (cmplx.q). The initial value of the reliability is the same (equal to 1) for all the machines, and when a test task uses more time than the assigned time limit, the reliability is decreased (it can be seen in the code and its discussion presented in Section 5.5). τ is a constant (in our experiments equal to 2) to protect against too early test task braking.

The time is calculated in universal seconds, to make time measurements independent of the type of computer on which the task is computed. The time in universal seconds is obtained by multiplication of the real CPU time by a factor reflecting the comparison of the CPU power with a reference computer. It is especially important when a cluster of different computers is used.

Each test task is assigned a priority level. The MLAs use the priorities to favor the tasks that were started earlier (to make them finish earlier). Therefore the tasks of the smallest complexity are finished as first. Another reason of using the priority (see code line 54) is to inform the task spooler (compare Section 3.3) to favor not only the queued task but also each child machine. This is very important, because it saves memory resources. In the case of really complex machines which MLAs have to deal with, it is crucial.

The *while* loop in line 50 saturates the task spooler. This concept works in harmony with the priority system, yielding a rational usage of memory and CPU resources.

It is a good place to point out, that even if the generators flow provides a test task which has already been provided earlier, it will not be calculated the next time. Thanks to

the unification engine, described in Section 3.4, the machine cache keeps all the computed solutions (if not directly in RAM memory, then in a disc cache). Of course, it does not mean that the generators flow should not care for the diversity of the test tasks configurations. The machine cache is also important to save time when any sub-task is requested repeatedly.

5.5 Analysis of Finished Tasks

After starting apropriate number of tasks, the MLA is waiting for a task to finish (compare the first code in Section 5.3). A task may finish normally (including termination by an exception) or halted by time-limiter (because of exceeding the time limit).

```
57 procedure analyzeFinishedTasks;
58   foreach (t in mlTaskSpooler.finishedTasks)
59   {
60     mc = t.configuration;
61     if (t.status = finished_normally)
62     {
63       qt = computeQualityTest(t, queryDefinition);
64       machinesRanking.Add(qt, mc);
65       if(attractivenessModule is defined)
66         attractivenessModule.Analyze(t, qt, machinesRanking);
67       machineGeneratorsFlow.Analyze(t, qt, machinesRanking);
68     }
69     else // task broken by limiter
70     {
71       cmplx = mc.cmplx;
72       cmplx.q = cmplx.q / 4;
73       machinesHeap.Quarantine(mc);
74     }
75     mlTaskSpooler.RemoveTask(t);
76   }
77 end
```

The procedure runs in a loop, to serve all the finished tasks as soon as possible (also those finished while serving other tasks.

When the task is finished normally, the quality test is computed basing on the test task results (see line 63) extracted from the project with the query defined by queryDefinition. As a result a quantity qt is obtained. The machine information is added to the machines ranking (machinesRank) as a pair of quality test qt and machine configuration mc.

Later, if the attractiveness module is defined (see lines 65–66), it gets the possibility to analyze the new results and, in consequence, may change the attractiveness part of machines complexities. In this way, the MLA may change the complexity of machines already deposited in the heap (machinesHeap) and the heap is internally reorganized according to the new complexities (compare Eq. 7). Attractiveness modules may learn and organize meta-knowledge basing on the results from finished tasks.

Next, the generators flow is called (line 67) to analyze the new results. The flow passes the call to each internal generator to let the whole hierarchy analyze the results. Gener-

ators also may learn by observation of results to provide new, more interesting machine configurations (only in the case of advanced generators).

When a task is halted by time-limiter, the task is moved to the *quarantine* for a period not counted in time directly but determined by the complexities. Instead of constructing a separate structure responsible for the functionality of a quarantine, the quarantine is realized by two naturally cooperating elements: the machines heap and the reliability term of the complexity formula (see Eq. 7). First, the reliability of the test task is corrected—see code line 72, and after that, the test task is resend to the machine heap as to quarantine—line 73. The role of quarantine is very important and the costs of using the quarantine are, fortunately, not too big. MLAs restart only these test task for which the complexity was badly approximated. To better see the costs, assume that the time complexity of a test task was completely badly approximated and the real universal time used by this task is t. In the above scheme of the quarantine, the MLA will spend, for this task, a time not greater than $t + t + \frac{1}{4}t + \frac{1}{16}t + \ldots = \frac{7}{3}t$. So the maximum overhead is $\frac{4}{3}t$, however it is the worst case—the case where we halt the process just before it would be finished (hence the two t's in the sum). The best case gives only $\frac{1}{3}t$ overhead which is almost completely insignificant. The overhead is not a serious hamper, especially, when we take to the account that the MLA with the quarantine is not affected by the halting-problem of test-task. Moreover, the cost estimation is pessimistic also from another point of view: thanks to the unification mechanism, each subsequent restart of the test may reuse significant number of submachines run before, so in practice, the time overhead is usually minimal.

5.6 Meta-learning Results Interpretation

The final machine ranking returned by meta-learning may be interpreted in several ways. The first machine configuration in the ranking is the best one according to the measure represented by queryDefinition.

But the runners-up may not be significantly worse. Using statistical tests, like McNemmar test, a set of machines of insignificant quality differences, can be caught. Next, the final solution may be chosen according to another criterion, for example, the simplest one from the group of the best solutions or the one with the smallest variance etc.

The results may be explored in many different ways as well. For example, one may be interested in finding solutions using alternative (to the best machine) feature sets or using the smallest number of features or instances (in the case of similarity based machines), etc. Sometimes, comprehensive models may be preferred, like decision trees, if only they work sufficiently well.

In some cases it may be recommended to repeat some tests, but with little "deeper" settings analysis, before the final decision is made.

During the results ranking analysis, the commentary parts[2] of the results can also be used as a fruitful information to support the choice.

2. Parts with information about derivation of configuration and other comments on quality test.

6. Machine Complexity Evaluation

Since the complexity has to determine the order of performing test tasks, its computation is extremely important. The complexity can not be computed from learning machines, but from configurations of learning machines and descriptions of their inputs, because the information about complexity is needed before the machine is ready for use. In most cases, there is no direct analytical way of computing the machine complexity on the basis of its configuration. Therefore, we introduce an approximation framework for automated complexity approximation.

6.1 Complexity in the Context of Machines

The Kolmogorov complexity definition is not very useful in real tasks especially in computational intelligence problems. The problem of finding a minimal program is unsolvable—the search space of programs is unlimited and the time of program execution is unlimited. In the case of Levin's definition (Eq. 3) it is possible to realize the Levin Universal Search (LUS) (Jankowski, 1995; Li and Vitányi, 1993) but the problem is that this algorithm is NP-hard. This means that, in practice, it is impossible to find an exact solution to the optimization problem.

The strategy of meta-learning is different than the one of LUS. Meta-learning uses the functional definition of the search space, which is not infinite, in the finite meta-learning process. This means that the search space is, indeed, strongly limited. The generators flow is assumed to generate machine configurations which are "rational" from the point of view of given problem $\mathcal{P}$. Such solution restricts the space to the most interesting algorithms and makes it strictly dependent on the configuration of the MLA.

In our approach to meta-learning, the complexity controls the order of testing machine configurations collected in machine heap. Ordering programs only on the basis of their length (as it was defined in Kolmogorov Eq. 2) is not rational. The problem of using Levin's additional term of time, in real applications, is that it is not rigorous enough in respecting time. For example, a program running 1024 times longer than another one may have just a little bigger complexity (just +10) when compared to the rest (the length). This is why we use the following definition, with some additions described later:

$$c_a(p) = l(p) + t^p / \log(t^p). \tag{5}$$

Naturally, we use an approximation of the complexity of a machine, because the actual complexity is not known before the real test task is finished. The approximation methodology is described in Section 6. Because of this approximation and because of the halting problem (we never know whether given test task will finish) an additional penalty term is added to the above definition:

$$c_b(p) = [l(p) + t^p / \log(t^p)]/q(p), \tag{6}$$

where $q(p)$ is a function term responsible for reflecting an estimate of reliability of p. At start the MLAs use $q(p) = 1$ (generally $q(p) \leq 1$) in the estimations, but in the case when the estimated time (as a part of the complexity) is not enough to finish the program p (given test task in this case), the program p is aborted and the reliability is decreased.

The aborted test task is moved to a *quarantine* according to the new value of complexity reflecting the change of the reliability term. This mechanism prevents from running test tasks for unpredictably long time of execution or even infinite time. Otherwise the MLA would be very brittle and susceptible to running tasks consuming unlimited CPU resources. More details on this are presented in Section 5.4.

Another extension of the complexity measure is possible thanks to the fact that MLAs are able to collect meta-knowledge during learning. The meta-knowledge may influence the order of test tasks waiting in the machine heap and machine configurations which will be provided during the process. The optimal way of doing this, is adding a new term to the $c_b(p)$ to shift the start time of given test in appropriate direction:

$$c_m(p) = [l(p) + t^p / \log(t^p)]/[q(p) \cdot a(p)]. \tag{7}$$

$a(p)$ reflects the attractiveness of the test task p.

6.1.1 Complexities of What Machines are We Interested in?

As described in Section 5.3, the generators flow provides machine configurations to machineGeneratorForTestTemplate and after nesting the configurations inside the test template, the whole test configurations are sent to the machinesHeap. The machinesHeap uses the complexity of the machine of given configuration, as the priority key. It is not accidental, that the machine configuration which comes to the machinesHeap is the configuration of the whole test machine (where the proposed machine configuration is nested). This complexity really well reflects complete behavior of the machine: a part of the complexity formula reflects the complexity of learning of given machine and the rest reflects the complexity of computing the test (for example classification or approximation test). The costs of learning are very important, because trivially, without learning there is no model. The complexity of the testing part is also very important, because it reflects the simplicity of further use of the model. Some machines learn quickly and require more effort to make use of their outputs (like kNN classifiers), while others learn for a long time and after that may be very efficiently exploited (like many neural networks). Therefore, the test procedure should be as similar to the whole life cycle of a machine as possible (and of course as trustful as possible).

To understand the needs of complexity computing we need to go back to the task of learning. To provide a learning machine, regardless of whether it is a simple one, a complex machine or a machine constructed to help in the process of analysis of other machines, its configuration and inputs must be specified (compare Section 3). Complexity computation must reflect the information from configuration and inputs. The recursive nature of configurations, together with input–output connections, may compose quite complex information flow. Sometimes, the inputs of submachines become known just before they are started, i.e. after the learning of other machines[3] is finished. This is one of the most important reasons why determination of complexity, in contrary to actual learning processes, must base on *meta-inputs*, not on exact inputs (which remain unknown). Assume a simple scene, in which a classifier TC is built from a data transformer T and a classifier C (compare Figure 9). It

3. Machines which provide necessary outputs.

would be impossible to compute complexity of the classifier C basing on its inputs, because one of the inputs is taken from the output of the transformer T, which will not be known before the learning process of T is finished. Complexity computation may not be limited to a part of TC machine or wait until some machines are ready. To make complexity computation possible we use proper *meta-inputs* descriptions. Meta-inputs are counterparts of inputs in the "meta-world". Meta-inputs contain descriptions (as *informative* as possible) of inputs which "explain" or "comment" every useful aspect of each input which could be helpful in determination of the complexity.

Because machine inputs are outputs of other machines, the space of meta-inputs and the space of meta-outputs are the same.

To facilitate recurrent determination of complexity—which is obligatory because of basing on a recurrent definition of machine configuration and recurrent structure of real machines—the functions, which compute complexity, must also provide meta-outputs, because such meta-outputs will play crucial role in computation of complexities of machines which read the outputs through their inputs.

In conclusion, a function computing the complexity for machine $\mathcal{L}$ should be a transformation

$$\mathcal{D}_{\mathcal{L}} \ : \ \mathcal{K}_{\mathcal{L}} \times \mathcal{M}_{+} \to R^2 \times \mathcal{M}_{+}, \tag{8}$$

where the domain is composed by the configurations space $\mathcal{K}_{\mathcal{L}}$ and the space of meta-inputs $\mathcal{M}_{+}$, and the results are the time complexity, the memory complexity and appropriate meta-outputs. It is important to see the similarity with the definition of learning (Eq. 1), because computation of complexity is a derivative of the behavior of machine learning process.

The problem is not as easy as the form of the function in Eq. 8. Finding the right function for given learning machine $\mathcal{L}$ may be impossible. This is caused by unpredictable influence of some configuration elements and of some inputs (meta-inputs) to the machine complexity. Configuration elements are not always as simple as scalar values. In some cases configuration elements are represented by functions or by subconfigurations. Similar problem concerns meta-inputs. In many cases, meta-inputs can not be represented by simple chain of scalar values. Often, meta-inputs need their own complexity determination tool to reflect their functional form. For example, a committee of machines, which plays a role of a classifier, will use other classifiers (inputs) as "slave" machines. It means that the committee will use classifiers' outputs, and the complexity of using the outputs depends on the outputs, not on the committee itself. This shows that sometimes, the behavior of meta-inputs/outputs is not trivial and proper complexity determination requires another encapsulation.

6.2 Meta Evaluators

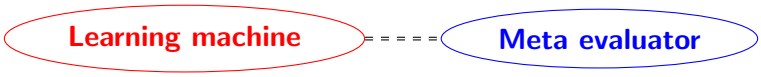
To enable so high level of generality, the concept of *meta-evaluators* has been introduced. The general *goal of meta-evaluator* is

- to evaluate and exhibit *appropriate aspects* of complexity representation basing on some meta-descriptions like meta-inputs or configuration[4].

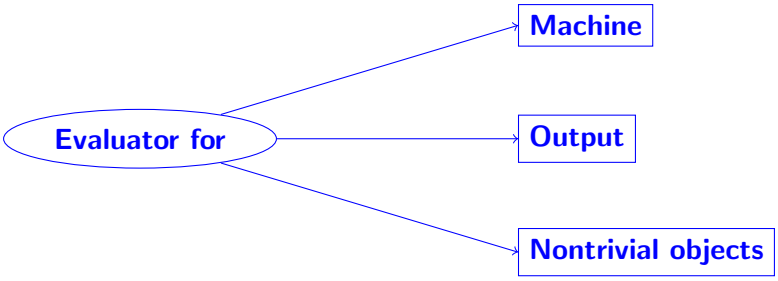
4. In case of a machine to exhibit complexity of time and memory.

- to exhibit a functional description of complexity aspects (comments) useful for further reuse by other meta evaluators[5].

To enable complexity computation, every learning machine gets its own meta evaluator.



Because of recurrent nature of machine (and machine configuration) and because of nontriviality of inputs behavior (which sometimes have complex functional form), meta evaluators are constructed not only for machines, but also for outputs and other elements with "nontrivial influence" on machine complexity.



Each evaluator will need *adaptation*, which can be seen as an initialization and can be compared to the learning of machine. In such meaning the process $D_{\mathcal{L}}$ (Eq. 8) will be the typical adaptation of evaluator devoted on the machine $\mathcal{L}$. It means that before using given evaluator, it has to be adapted. Then, evaluator can be used to calculate aspects of complexity devoted for given evaluator (compare, presented below, typical evaluators type and their functionality).

It is sometimes necessary to estimate complexity on the basis of machine configuration and real inputs (not meta-inputs as in Eq. 8). In such case, we would need an adaptation of machine evaluator in the form:

$$\mathcal{D}'_{\mathcal{L}} \ : \ \mathcal{K}_{\mathcal{L}} \times \mathcal{I}_{+} \rightarrow R^2 \times \mathcal{M}_{+}, \tag{9}$$

where $\mathcal{I}_{+}$ is the space of machine $\mathcal{L}$ inputs. Such approach would require construction of two evaluators for each machine: for the forms presented in Eq. 8 and Eq. 9. But it is possible to resign from the Eq. 9 form. The solution is to design output evaluators and their adaptation as:

$$\mathcal{D}_o \ : \ \mathcal{I}_1 \rightarrow \mathcal{M}_1, \tag{10}$$

where $\mathcal{I}_1$ is a space of (single) output and $\mathcal{M}_1$ is the space of meta-output. And now we can seen that meta-input (or meta-output) is nothing else than special case of evaluator, the output evaluator.

Using output evaluators, the "known" inputs can be transformed to meta-inputs ($\mathcal{K}_{\mathcal{L}} \times \mathcal{I}_{+} \rightarrow \mathcal{K}_{\mathcal{L}} \times \mathcal{M}_{+}$), and after that, the machine evaluator of the form of Eq. 8 can be used. This finally reduces the needs of adaptation in the form of Eq. 9.

5. In case of a machine the meta-outputs are exhibited to provide complexity information source for their inputs readers.

Sometimes, machine complexity depends on nontrivial elements (as it was already mentioned), typically some parts of the configuration. Then, the behavior of machine changes according to changes of nontrivial part of the machine configuration. For example, configurations of machines like kNN or SVM are parameterized by metric. The complexity of metric (the time needed to calculate single distance between two instances) does not depend on the kNN or SVM machine, but on the metric function. Separate evaluators for such nontrivial objects, simplify creation of machine evaluators, which may use subevaluators for the nontrivial objects. Every evaluator may order creation of any number of (nested) evaluators. Adaptation of evaluators for nontrivial objects may be seen as:

$$\mathcal{D}_{obj} \ : \ \mathcal{OBJ} \rightarrow \mathcal{M}_{obj}, \tag{11}$$

where $\mathcal{OBJ}$ is the space of nontrivial objects and $\mathcal{M}_{obj}$ is their evaluators space (which is subspace of all evaluators, of course).

The adaptation process is the major functionality of each evaluator and depends on the type of the evaluator and parameters of the adaptation function. Adaptation is realized by EvaluatorBase method:

```
EvaluatorBase(object[] data);
```

In general, the goal of this method is to use the data as the "source of information" for given evaluator.

The data are different in type, goal and other aspects, depending on the type of evaluator (compare Eq. 8, 10, and 11):

- if an evaluator is defined for a machine, then the data may be a real machine or a configuration and meta-inputs,
- evaluators constructed for outputs, in the data get a real output,
- in other cases, data depend on the needs of particular evaluators.

When evaluators may be defined in analytical way (quite rare cases), the evaluators need only to be adapted via EvaluatorBase. In other cases, the *approximation framework* is used to construct evaluators (see Figure 12 and Section 6.3). The precess of creation of evaluators is presented in Section 6.3.2.

Further functionality of meta evaluators depends on their types. Some examples are presented in the following subsections.

6.2.1 Machine Evaluator

In the case of any machine evaluator, the additional functionality consists of:

Declarations of output descriptions:
: If given machine provides outputs, then also the output evaluators, devoted to this machine type, must provide meta-descriptions of the outputs. The descriptions of outputs are meta evaluators of appropriate kind (for example meta-classifier, meta-transformer, meta-data etc.). Output description may be the machine evaluator itself

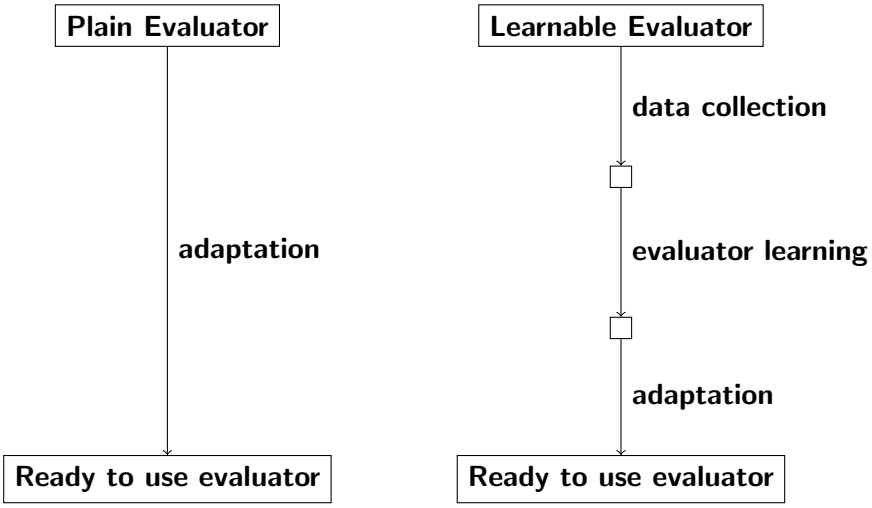


Figure 12: Creation of ready to use evaluator for plain evaluator and evaluator constructed with approximation framework.

or a subevaluator produced by the machine evaluator or the evaluator provided by one of submachine evaluators constructed by the machine evaluator (machine may create submachines, evaluator may create evaluators of submachines basing on their configuration and meta-inputs).

Time & Memory:
The complexities defined by Eq. 3, 5–7 make use of *program* length and time. Here, the two quantities must be provided by each machine evaluator to enable proper computation of time and memory complexity.

Child Evaluators:
for advanced analysis of complex machines complexities, it is useful to have access to meta evaluators of submachines. Child evaluators are designed to provide this functionality.

6.2.2 Classifier Evaluator

Evaluator of a classifier output, has to provide the time complexity of classification of an instance:

```
real ClassifyCmplx(DataEvaluator dtm);
```

Apart from the learning time of given classifier, the time consumed by the instance classification routine is also very important in calculation of complexities. To estimate time requirements of a classifier test machine, one needs to estimate time requirements of the calls to the machine classification function. The final time estimation depends on the classifier and on the data being classified. The responsibility to compute the time

complexity of the classification function, belongs to the meta classifier side (the evaluator of the classifier). Consider a classification committee: to classify data, it needs to call a sequence of classifiers to get the classification decisions of the committee members. The complexity of such classification, in most natural way, is a sum of the costs of classification using the sequence of classifiers, plus a (small) overhead which reflects the scrutiny of the committee members' results to make the final decision. Again, the time complexity of data classification is crucial to estimate the complexity and must be computable.

6.2.3 Approximator Evaluator

Evaluator of an approximation machine has exactly the same functionality as the one of a classifier, except that approximation time is considered in place of classification time:

```
real ApproximationCmplx(DataEvaluator dtm);
```

6.2.4 Data Transformer Evaluator

Evaluator of a data transformer has to provide two estimation aspects. The first one is similar to the functionality of the evaluators described above. Here it represents the time complexity of transformation of data instances. The second requirement is to provide a meta-description of data after transformation: the data evaluator. It is of highest importance—the quality of this meta-transformation of data-evaluator is transfered to the quality of further complexity calculations.

6.2.5 Metric Evaluator

The machines that use metrics, usually allow to set the metric at the configuration stage (e.g. kNN or SVM). As parameters of machine configurations, metrics have nontrivial influence on the complexity of the machine while not being separate learning machines. The most reasonable way to enable complexity computation, in such cases, is to reflect the metric-dependence inside the evaluators (one evaluator per one metric). The meta-evaluators for metrics provide the functionality of time complexity of distance computation and are used by the evaluators of proper machines or outputs:

```
float DistanceTimeCmplx();
```

6.2.6 Data Evaluators

Another evaluators of crucial meaning are data evaluators. Their goal is to provide information about data structure and statistics. Data evaluator has to be as informative as possible, to facilitate accurate complexity determination by other evaluators. In the context of data tables, the data evaluators should provide information like the number of instances, the number of features, descriptions of features (ordered/unordered, number of values, etc.), descriptions of targets, statistical information per feature, statistical information per data and others that may provide useful information to compute complexities of different machines learning from the data.

6.2.7 Other Evaluators

The number of different types of meta evaluators is not determined. Above, only a few examples are presented of many instances available in the system. During future expansion of the system, as the number of machine types grows, the number of evaluators will also increase.

6.3 Learning Evaluators

Defining manually the functions to compute time and memory complexities inside evaluators for each machine (as well as other complexity quantities for evaluators of other types) is very hard or even impossible. Often, analytical equation is not known, and even if it is known or determinable, there is still a problem with conversion of the analytical formula or the knowledge about the dependencies into estimation of real time measured in universal seconds.

In any case, it is possible to build approximators for elements of evaluators which estimate complexity or help in further computation of such complexity. We have defined an *approximation framework* for this purpose. The framework is defined in very general way and enables building evaluators using approximators for different elements like learning time, size of the model, etc. Additionally, every evaluator that uses the approximation framework, may define special functions for estimation of complexity (MethodForApprox). This is useful for example to estimate time of instance classification etc. It was constructed to fulfill needs of different kinds of evaluators.

The complexity control of task starting in meta-learning does not require very accurate information about tasks complexities. It is enough to know, whether a task needs a few times more of time or memory than another task. The differences of several percent are completely out of interest here. Assuming such level of accuracy of complexity computation, we do not loose much, because meta-learning is devoted to start many test tasks and small deviations from the optimal test task order are irrelevant. Moreover, although for some algorithms the approximation of complexity is not so accurate, the quarantine (see Section 5.5) prevents from capturing too much resources by a single long-lasting task.

Using the approximation framework, meta evaluator can learn as many aspects of machine behavior as necessary. Evaluator using approximation framework can estimate an unlimited set of quantities that may be useful for determination of complexities of some elements or some quantities for further computation of complexities. Typically, a single evaluator using the approximation framework creates several approximators. For example, evaluator of each machine has to provide time and memory complexities. The evaluator will realize it with two approximators. Additionally, in the case, when machine corresponding to given evaluator is also a classifier, the classification time may be learned as well, within the same framework (another dedicated approximator may be constructed). The approximators are constructed, learned and used (called to approximate) automatically, according to appropriate declarations in the evaluators, as it will be seen later (in the examples of evaluators). There is no manual intervention needed in the approximator building process.

Of course, before an evaluator is used by a meta-learning process, all its approximators must be trained. The learning of all evaluators may be done once, before the first start of meta-learning tasks. Typically, learned evaluators reside in an *evaluators project* which is

loaded before the MLA starts its job. If the system is extended by a new learning machine and a corresponding evaluator, the evaluator (if it uses the approximation framework) has to learn and also will reside in the evaluators project for further use. This means that every evaluator using the approximation engine, has to be trained just once.

Before learning of evaluator approximation models, appropriate data tables must be collected (as learning data). This process will be described later. First we will present the evaluator functionality extension, facilitating usage of the approximation framework.

6.3.1 Approximation Framework of Meta Evaluators

The construction of a *learnable* evaluator (an evaluator making use of approximation framework) differs from construction of a plain evaluator (compare Figure 12).

The approximation framework enables to construct series of *approximators* for single evaluators. The *approximators* are functions approximating a real value on the basis of a vector of real values. They are learned from examples, so before the learning process, the learning data have to be collected for each approximator.

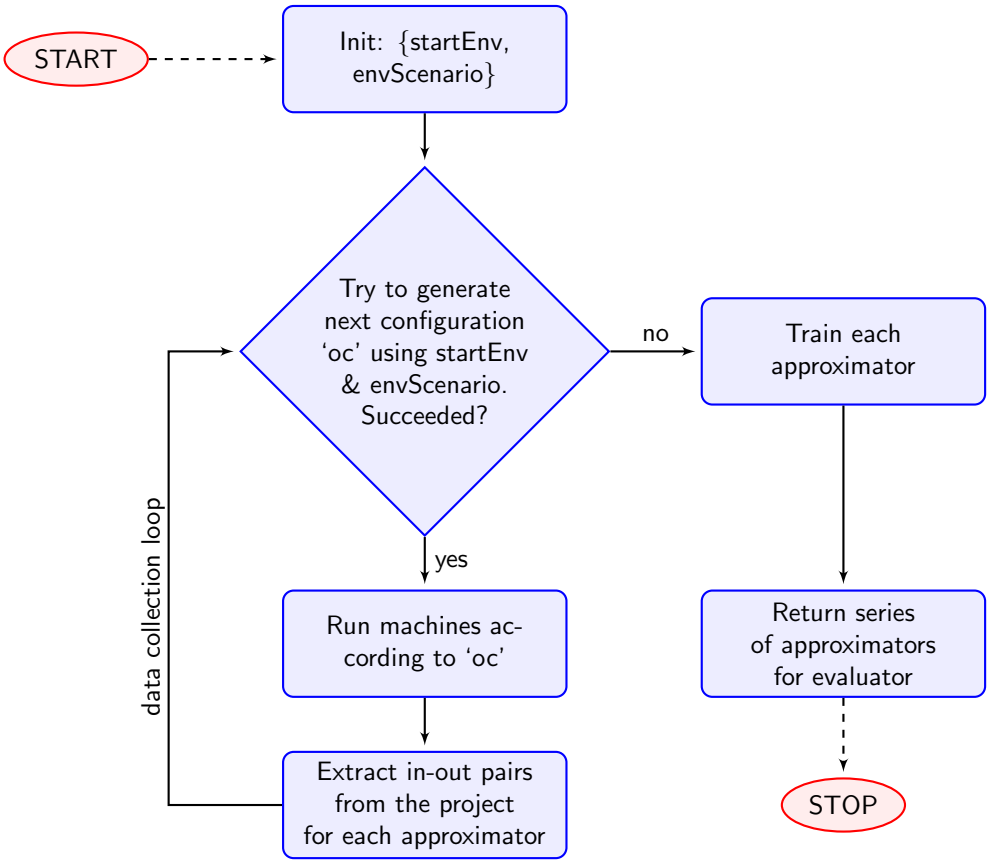


Figure 13: Process of building approximators for single evaluator.

Figure 13 presents the general idea of creating the approximators for an evaluator. To collect the learning data, proper information is extracted form observations of "machine behavior". To do this an "environment" for machine monitoring must be defined. The environment configuration is sequentially adjusted, realized and observed (compare the data collection loop in the figure). Observations bring the subsequent instances of the training data (corresponding to current state of the environment and expected approximation values). Changes of the environment facilitate observing the machine in different circumstances and gathering diverse data describing machine behavior in different contexts.

The environment changes are determined by initial representation of the environment (the input variable startEnv) and specialized scenario (compare Section 4), which defines how to modify the environment to get a sequence of machine observation configurations i.e. configurations of the machine being examined nest in a more complex machine structure. Generated machine observation configurations should be as realistic as possible—the information flow similar to expected applications of the machine, allows to better approximate desired complexity functions. Each time, a next configuration 'oc' is constructed, machines are created and run according to 'oc', and when the whole project is ready, the learning data are collected. Full control of data acquisition is possible thanks to proper methods implemented by the evaluators. The method EvaluatorBase is used to prepare the evaluator for analysis of the environment being observed, while GetMethodsForApprox declares additional approximation tasks and ApproximatorDataIn, ApproximatorDataOut prepare subsequent input–output vectors from the observed environment.

When generation of new machine observation configurations in the data collection loop fails, the data collection stage is finished. Now each approximator can be learned from the collected data and after that the evaluator may use them for complexity prediction. Before the predicition, the evaluator is provided with appropriate configuration and/or meta-inputs (depending on the evaluator type)—this adaptation is performed by the EvaluatorBase method (see Eq. 8, 10, 11).

Note that the learning processes of evaluators are conducted without any advise from the user (compare Section 6.3.2).

From the description given above, we may conclude that the approximation framework gives two types of functionality:

- to define information necessary to collect training data for learning approximators,
- to use learned approximators.

The latter enables using approximators (after they are built within the approximation framework) by calling the function:

```
real[] Approximate(int level);
```

The input vector for given approximator is obtained by ApproximatorDataIn, described in detail below. Note that ApproximatorDataIn is used to provide input vectors to approximate and to collect learning input vectors. It is possible thanks to the extended EvaluatorBase.

The following code presents an example of using the Approximate. It is used to calculate time complexity of classification of data basing on their meta-input (dtm is an evaluator of the data).

```
78 function ClassifyCmplx(Meta.Data.DataTable_Meta dtm)
79   return Approximate(classifyCmplxLevel)[0] * dtm.DataMetaParams.InstanceCount;
80 end
```

Line 79 calls the approximator with index 0, from level classifyCmplxLevel (0-th element in the output vector is responsible for time complexity; the idea of levels and layers is addressed below).

Extended EvaluatorBase. There are two roles of the EvaluatorBase functionality:

- To prepare the adaptation process (as it was presented in Section 6.2 and in Eq. 8, 10 and 11), which has to provide necessary variables/quantities to become ready to use complexity *items* inside the evaluator. Also to prepare elements needed by ApproximatorDataIn to build the input vectors for particular approximators,
- To provide elements necessary for data collection for approximator learning basing on observed environments. EvaluatorBase has to provide elements, necessary and sufficient to build learning in-out pairs with ApproximatorDataIn, ApproximatorDataOut and GetMethodsForApprox.

The approximators can be constructed to estimate:

- time and memory complexity of machine,
- time and memory complexity of particular machine methods (declared for the approximation by GetMethodsForApprox),
- other quantities.

To embed the above three types of approximation, the approximator learners are placed in separate *levels* in three *layers*:

Level	**Layer**
level 1 ... level k	1 — Approximators of other quantities
level $k+1$ ... level $n-1$	2 — Approximators for specially defined methods
level n	3 — Approximators of time & memory complexity of machine

The order of levels reflects the order of data collection (in each iteration of data collection loop) and of further learning of approximators after data collection.

Using of approximators from the first layer may be helpful for composing input vectors for the next two layers. This functionality is used only for advanced evaluators (not too common, but sometimes very helpful) and will not be described here in more detail.

For each of the three layers, another set of evaluator methods is used to prepare the in-out learning pairs. In case of the first layer, two functions are used:

real[] ApproximatorDataIn(**int** level) — provides the input vector for level *level*,

real[] ApproximatorDataOut(**int** level) — provides the output vector for level *level*.

For the purpose of the second layer, we need:

real[] ApproximatorDataIn(**int** level) — provides the input vector for complexity of the corresponding method,

MethodForApprox[] GetMethodsForApprox() — provides table of methods being subjects to complexity checking.

To define approximation of the *machine* layer (the third one), there is only a need for

real[] ApproximatorDataIn(**int** level).

Functions of each layer, have the same general goal: to collect a single input-output pair of data for learning appropriation basing on the information extracted by EvaluatorBase from the observed environment.

Collection of the learning data for the first layer is easy to interpret: the function ApproximatorDataIn composes the input vector and ApproximatorDataOut composes the output vector.

The number of levels in the second layer is auto-defined by the number of methods returned by the GetMethodsForApprox method. When empty sequence is returned, the layer is not used by the evaluator. Otherwise, for each of the returned methods, the approximation framework, automatically approximates the time of execution and sizes of all the returned (by the method) objects. Therefore, only the input part of the learning data (ApproximatorDataIn) must be provided by the evaluator—the output is determined automatically by the system. Each approximation method MethodForApprox is defined as function:

object[] MethodForApprox(**int** repetitionNr);

The parameter repetitionNr is used by the approximation framework to ask for a number of repetitions of the test performed by the method (to eliminate the problem of time measurement for very quick tests). For example, let's see the code shown below, where a classifier is called to classify repetitionNr instances. The aim of the function is to measure time complexity of the classification routine.

```
81 function ClassifyTimeChecking(int repetitionNr)
82   IDataSet ds = mi.OpenInput("Dataset") as IDataSet;
83   IDataSet ds2 = RandomChainOfInstances(ds, repetitionNr);
84   IClassifier c = mi.OpenOutput("Classifier") as IClassifier;
85   IOneFeatureData d = c.Classify(ds2);
86   return null;
87 end
```

It is important to realize that the call of ClassifyTimeChecking is preceded by a call to EvaluatorBase, which sets up the mi to facilitate opening appropriate inputs (dataset and classifier).

The last layer is designed for learning time complexity and memory complexity of the machine. This layer is used only for machine evaluators. In this case, the approximation framework automatically tests the learning time and memory usage. These quantities compose the output vectors, so that definition of ApproximatorDataOut is not used here (as in the case of the second layer). The input vector is obtained, as for both previous layers, by calling the ApproximatorDataIn function, which is the only requirement for this layer.

Very important is the role of EvaluatorBase, responsible for two types of adaptation—during data collection and during complexity estimation by evaluators. The method has to provide all the information necessary to collect proper data with ApproximatorDataIn and ApproximatorDataOut methods, and to realize the tests of each associated MethodForApprox.

The input vector for each level may be different and should be as useful as possible, to simplify the process of approximator learning. Any useful element of machine description which can help to learn the characteristics of complexity should be placed inside the input vector. The same ApproximatorDataIn method is called also before the evaluator estimates the complexity of a machine. After the valuator adapts to given machine configuration and meta-inputs, ApproximatorDataIn prepares data for all the approximators to predict their targets and final complexities are estimated.

Environments for approximators learning. As shown in Figure 13 and described above, to build input data tables, necessary for training the approximators, the machine is observed in changing environment. Each change in the environment results in a single input–output pair of data for each of approximators. Therefore, to construct successful complexity evaluators, apart from specification of the necessary approximators, one needs to define the environment and the way of its manipulation.

To share the ways of handling environments, some groups of common properties are defined and each evaluator has to assign itself to one of the groups or to define a new group and assign to it. For example, to learn estimation of machine complexities, the machine should be trained using different configurations with different input data to explore the space of significantly different situations of the machine training and exploitation of its model.

The machine observation configurations, generated in the data collection loop are determined by the following items:

IConfiguration ApproximationConfig — defines the initial configuration of the machine closest environment for observations, to be nested in the ApproximationGroupTemplate defined for the group (see below). ApproximationConfig is needed because not always, it is enough to learn and observe a single machine. Sometimes it is necessary to precede the learning of the machine by a special procedure (some necessary data transformation, etc.). However sometimes the machine may be used directly (then the property is just a configuration instance of the machine).

int[] Path — determines the placement of the machine, being observed, in the environment defined above. For machines, it points the machine. For outputs, it points the machine which provides the output.

IScenario Scenario — defines the scenario (see Section 4), which goal is to provide different configurations derived from the ApproximationConfig to explore the space of machine

observation configurations. For example, in the case of the kNN machine, the scenario may browse through different values of the number of neighbors k and different metric definitions.

MachineGroup Group; — encapsulates a few functionalities, which extend the space of observed configurations. The groups of functionalities are shared between evaluators of similar type, which simplifies the process of defining evaluators. Each group is characterized by:

string[] DataFileNames; — defines file names of learning data which will be used to observe behavior of the learning process.

IConfiguration ApproximationGroupTemplate; — defines the procedure of using given type of machines. For example, it may consist of two elements in a scheme: a data multiplier which constructs learning data as a random sequence of instances and features from a dataset provided as an input, and a placeholder for a classifier (an empty scheme to be replaced by a functional classification machine).

int[] Path; — points the placeholder within the ApproximationGroupTemplate, to be filled with the observed machine, generated by Scenario used on ApproximationConfig (compare with the Path described above).

IScenario GroupScenario; — the scenario of configuration changes (see Section 4) to the ApproximationGroupTemplate. The environment is subject to changes of a data file DataFileNames and configuration changes defined by the GroupScenario. For example, this scenario may cooperate with a machine randomizing the learning data within the ApproximationGroupTemplate as it was already mentioned.

All the functionalities described above, used together, provide very flexible approximation framework. Evaluators can be created and functionally-tuned, according to the needs, supplying important help in to successful complexity computation.

The functions discussed above, are used in the meta-code of the next section, to present some aspects of the proposed meta-learning algorithm.

6.3.2 Creation and Learning of Evaluators.

After presenting the idea of the approximation framework for evaluators, here, we present the algorithms constructing evaluators, in more detail.

Before any meta-learning algorithm is run, the function CreateEvaluatorsDictionary builds a dictionary of evaluators which are constructed by the function CreateEvaluator. In fact, CreateEvaluatorsDictionary creates the evaluators project, which is used inside any meta-learning task.

```
88 function CreateEvaluatorsDictionary(Type[] allMachineTypes);
89   foreach (machineType in allMachineTypes)
90     evalDict[machineType] = CreateEvaluator(machineType);
91   return evalDict;
92 end
```

Creation of an evaluator starts with the creation of an instance (object) of given class corresponding to the type of given learning machine (machineType, see code line 94).

```
93 function CreateEvaluator(Type machineType);
94   eval = getEvalutorInstanceFor(machineType);
95   if(eval is ApproximableEvaluator)
96   {
97     sequenceOfDatasets = CreateDataTablesForApprox(machineType);
98     listOfApprox = {};
99     for (level=1 to eval.LevelsCount)
100    {
101      <TRS, TES> = GetTrainTestDataTablesFor(level);
102      approxTab = TrainApproximatorTab(<TRS, TES>);
103      listOfApprox.Append(approxTab);
104    }
105    eval.Approximators = listOfApprox;
106  }
107  return eval;
108 end
```

If the evaluator does not use the approximation framework, then it is ready (without learning) and may be called to estimate complexities. Otherwise, learning of appropriate approximators is performed. Line 97 calls a function (described later) which creates a sequence of learning data tables, according to the meta-description of the evaluator.

Next lines (99–104), for each level, prepare data tables and start learning of a vector of approximators. The vector of approximators is appended to the list listOfApprox and finally assigned to the evaluator (in line 105).

The function CreateDataTablesForApprox plays a crucial role in the complexity approximation framework, as it constructs learning data tables for the approximators. To start with, it needs an instance (object) of the evaluator (line 111) and the meta-description of its requirements related to the approximation framework.

```
109 function CreateDataTablesForApprox(Type machineType);
110   sequenceOfDatasets = {};
111   eval = getEvalutorInstanceFor(machineType);
112   machineGroup = eval.Group;
113   foreach (DataFileName in machineGroup.DataFileNames)
114   {
115     dataMachine = CreateDataLoader(DataFileName);
116     groupScenario = machineGroup.GroupScenario;
117     groupScenario.Init(machineGroup.ApproximationGroupTemplate);
118     foreach (gconfig in groupScenario)
119     {
120       scenario = eval.Scenario;
121       scenario.Init(eval.ApproximationConfig);
122       foreach (sconfig in scenario)
123       {
124         c = PlaceConfigInTemplate(gconfig, sconfig, machineGroup.Path);
125         t = StartTask(c, dataset);
```

```
126           m = t.GetSubmachine(machineGroup.Path + eval.Path);
127           eval.EvaluatorBase(m);
128           for (level=1 to eval.InnerApproximatorLevels) // layer 1 (other quantities)
129           {
130             dtIn[level].AddVector(eval.ApproximatorDataIn(level));
131             dtOut[level].AddVector(eval.ApproximatorDataOut(level));
132           }
133           foreach (meth in eval.GetMethodsForApprox())   // layer 2 (methods)
134           {
135             level++;
136             <time, objectsTab> = CheckMethod(meth, t);
137             objectsSizes = CheckSizes(objectsTab);
138             dtIn[level].AddVector(eval.ApproximatorDataIn(level));
139             dtOut[level].AddVector(<time, objectsSizes>);
140           }
141           if (eval is assigned to a machine)    // layer 3 (machine)
142           {
143             level++;
144             dtIn[level].AddVector(eval.ApproximatorDataIn(level));
145             dtOut[level].AddVector(<t.time, t.memorySize>);
146           }
147         }
148       }
149     }
150     sequenceOfDatasets = resplit(<dtIn, dtOut>);
151     return sequenceOfDatasets;
152   end
```

The observations are performed for each dataset (the loop in line 113), for each group configuration gconfig generated by the group scenario (the loop in line 118), for each machine configuration sconfig generated by the scenario (the loop in line 122). The scenarios are initialized before they provide the configurations (see lines 117 and 121, and Section 4).

In line 124, the configuration sconfig is placed within the group configuration gconfig in the location defined by machineGroup.Path. This composes a configuration c which defines the observation task t (next line). The following line extracts a link to the observed machine m from the observation task.

Next, basing on the link m, the function EvaluatorBase is called to provide information on the observed machine to ApproximationData(In/Out).

Then, new instances are added to the learning data tables for approximators of subsequent levels (first those of the first layer, then the methods layer and finally the machine layer).

As described before, the method ApproximationDataIn is called for each layer, while ApproximationDataOut just for the levels of the first layer—the system automatically prepares the output data for both methods layer (the outputs are time and proper object sizes) and machine layer (the outputs are time and memory complexities).

At the end (line 150), the input and target parts are transformed to appropriate data tables to be returned by the function.

6.4 Example of Evaluators

To better see the practice of evaluators, we present key aspects of evaluators for some particular machines.

6.4.1 Evaluator for K Nearest Neighbors Machine

This evaluator is relatively simple. Nevertheless, the requires functionality must be defined (according to the description of the preceding sections).

EvaluatorBase

In the case of kNN, in this function, the evaluator saves the kNN configuration and the evaluator of the input data. Another goal is to prepare the output description of the classifier's evaluator.

```
153 function EvaluatorBase(object[] data);
154   Config = GetConfiguration(data);
155   Outputs_Meta inputsMeta = GetOutput_MetaFrom(data);
156   DataEvaluator = inputsMeta["Dataset"][0];
157   DeclareOutputDescription("Classifier", this);
158 end
```

Time

kNN machines do not learn, so the time of learning is 0.

```
159 function Time()
160   return 0;
161 end
```

Memory

The model uses as much memory as the input data, i.e. DataEvaluator.MemoryCmplx():

```
162 function Memory()
163   return DataEvaluator.MemoryCmplx();
164 end
```

ClassifyCmplx

This function approximates the complexity of classification using the approximator:

```
165 function ClassifyCmplx(DataEvaluator dtm);
166   return Approximate(classifyCmplxLevel)[0] * dtm.InstanceCount;
167 end;
```

classifyCmplxLevel points appropriate approximators level.

ApproximatorDataIn
: The method provides training data items for approximation of the classification complexity (line 171) and of the machine learning process complexity (line 174):

```
168 function ApproximatorDataIn(int level)
169   switch (level)
170   {
171     case classifyCmplxLevel:
172       return { Config.K,
173         DataEvaluator.InstanceCount * Metric_Meta.DistanceTimeCmplx };
174     case machineLevel:
175       return { Config.K, Metric_Meta.DistanceTimeCmplx,
176         DataEvaluator.InstanceCount, DataEvaluator.FeatureCount };
177   }
178 end
```

ApproximationConfig
: For kNN, it is just the configuration of kNN machine.

```
179 function ApproximationConfig()
180   return new kNNConfig();
181 end
```

Scenario
: The scenario manipulates the ,,k" (the numbers of neighbors) and the metric.

Path
: The kNN configuration is not nested in another machine configuration (it constitutes the ApproximationConfig itself), so the path does not need to point any internal configuration, hence is empty.

```
182 function Path()
183   return null;
184 end
```

GetMethodsForApprox
: Returns the function ClassifyTimeChecking devoted to computing the time of classification with the kNN model:

```
185 function GetMethodsForApprox()
186   return new MethodForApprox[]{ ClassifyTimeChecking };
187 end
```

The kNN evaluator is assigned to a machine group prepared for classifiers. The group definition includes:

ApproximationGroupTemplate
: This template is a scheme configuration with two subconfigurations. The first is the

RandomSubset machine, which provides data sets consisting of different numbers of randomly selected instances and features taken from some source data set. The second subconfiguration is the placeholder for a classifier. At runtime, the placeholder is filled with proper classifier (in this case with the kNN configuration). The classifier gets data input from the RandomSubset machine output.

GroupScenario
: It randomizes the configuration of RandomSubset machine to obtain more observations for learning the approximation targets.

6.4.2 Example of Evaluator for Boosting Machine

Boosting is an example of machine using many submachines. Intemi implementation of boosting machine, repeatedly creates triples of submachines consisting of data distributor machine, classifier machine and a test of the classifier. All of the submachines have their own influence on the complexity of the boosting machine.

EvaluatorBase
: Boosting evaluator requires more complex EvaluatorBase then the one of kNN:

```
188 function EvaluatorBase(params object[] data)
189   Config = GetConfiguration(data);
190   Outputs_Meta inputsMeta = GetOutput_MetaFrom(data);
191   DataEvaluator = inputsMeta["Dataset"][0];
192   Outputs_Meta d = { {"Dataset", DataEvaluator} };
193   boostDistributor = EvaluatorEngine.EvaluateMachine(
194     BoostingDistributor, d);
195   classifier = EvaluatorEngine.EvaluateMachine(
196     Config.Subconfigurations[0], d);
197   Outputs_Meta d2 = new Outputs_Meta();
198   d2.Add("Dataset", DataEvaluator);
199   d2.Add("Classifier", classifier.GetOutputDescription("Classifier"));
200   classTest = EvaluatorEngine.EvaluateMachine(
201     new Intemi.Testing.ClassTestConfig(), d2);
202   DeclareOutputDescription("Classifier", this);
203 end
```

Lines 189 and 191 determine the configuration of boosting machine and the data evaluator, similarly as for the evaluator of kNN. In line 193 the data distributor evaluator is created using the EvaluatorEngine, which enables creation of evaluators by other evaluators.

The classifier evaluator (see line 195) is constructed in a similar way to the data evaluator. Here, the classifier configuration is extracted from the structure of subconfigurations, because it may be any classifier defined as the first subconfigration of the boosting configuration.

Line 200 constructs the evaluator of a classification test. The evaluator gets meta-inputs descriptions of the data evaluator (DataEvaluator) and the classifier evaluator (classifier).

Because boosting is a classifier, the last line of the code of EvaluatorBase declares output meta-description of the classifier.

Time

The time of boosting machine training is the sum of time amounts necessary to build the sequence of distributors, classifiers and test machines plus the time of the boosting-only part of learning:

```
204 function Time
205   return JustThisMachineTime + Config.NrOfClassifiers *
206       (boostDistributor.Time + classifier.Time + classTest.Time);
207 end
```

Memory

Calculation of the occupied memory is analoguous to that of time consumption:

```
208 function Memory
209   return JustThisMachineMemory + Config.NrOfClassifiers *
210       (boostDistributor.Memory + classifier.Memory + classTest.Memory);
211 end
```

ClassifyCmplx

The costs of classifying by boosting models are nearly equal to the sum of classifying given data (dtm evaluator) by each of the subclassifiers:

```
212 function ClassifyCmplx(DataEvaluator dtm)
213   subclass = classifier.GetOutputDescription("Classifier");
214   return Config.NrOfClassifiers *
215       subclass.ClassifyCmplx(dtm) * 1.1;
216 end
```

ApproximatorDataIn

The input data for the approximators is quite easy to determine. Boosting complexity (excluding learning of submachines) depend mostly on the number of submachines (the cost of creation not of the learning) and on the size of data. Thus:

```
217 function ApproximatorDataIn(int level)
218   return { Config.NrOfClassifiers, DataEvaluator.InstanceCount };
219 end
```

ApproximationConfig

The tested configuration is just a boosting configuration with proper classifier configuration inside (here, the naive Bayes classifier):

```
220 function ApproximationConfig
221   BoostingConfig c;
222   c.ClassfierTemplate = NBCConfig();
223   return c;
224 end
```

The evaluator of boosting will work properly not only with naive Bayes, because in code line 195, appropriate evaluator for inner classifier is constructed (at the time of complexity estimation, it is the evaluator of the classifier defined in the configuration).

Scenario

The scenario, simply builds configurations with different numbers of submachines.

```
225 function Scenario()
226   return new Meta.ParamSearch.StepScenario_I(null,
227     new string[] { "NrOfClassifiers" },
228     Meta.ParamSearch.StepScenario_I.StepTypes.Linear, 10, 10, 3);
229 end
```

Path and GetMethodsForApprox

These properties are exactly the same as in the evaluator of the kNN machine.

Boosting machine is a classifier, so its evaluator is also attached to the group devoted to classifiers. Therefore, the group items are the same as in the case of kNN.

7. Meta-learning in Action

Presented meta-learning algorithm, or rather meta-learning system, may be used in variety of ways. Generators flow may be defined as a simple graph, but usually, for advanced problems, it is quite nontrivial graph, which in effect produces many test configurations. The goal of meta-learning which reflects the problem type may also be defined in several ways, according to the needs. Similarly, the stop criterion should reflect the preferences about the conditions of regarding the meta-search as finished.

To present meta-learning in action, we have used a few well known problems from the UCI Machine Learning repository (Merz and Murphy, 1998). All the benchmarks, presented below, are classification problems. All the following results are computed using the same configuration of meta-learning (obviously except the specification of the benchmark dataset).

First, we have to present the meta-learning configuration, according to the information presented in Sections 5.2 and 5.3. The configuration consists of several elements: the meta-learning test template, query test, stop criterion and the generators flow.

Meta-learning Test Template:
The test template exhibits the goal of the problem. Since, the chosen benchmarks are classification problems, we may use cross-validation as the strategy for estimation of classifiers capabilities. The repeater machine may be used as the test configuration with distributor

set up to the CV-distributor and the inner test scheme containing a placeholder for classifier and a classification test machine configuration, which will test each classifier machine and provide results for further analysis. Such a repeater machine configuration template is

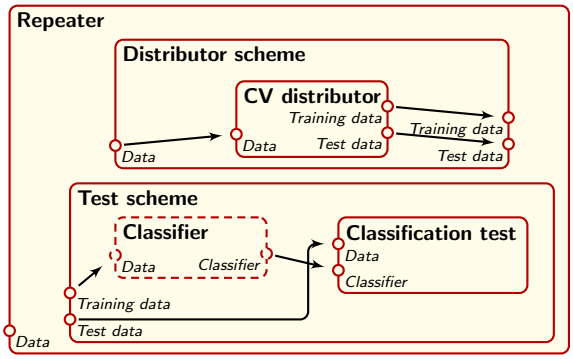


Figure 14: Meta-learning test template: Repeater machine configuration for cross-validation test with placeholder for classifier.

presented in Figure 14. When used as the ML test template, it will be repeatedly converted to different feasible configurations by replacing the classifier placeholder inside the template with classifiers configurations generated by the generators flow.

Query test:
To test a classifier quality, the accuracies calculated by the classification test machines may be averaged and the mean value may be used as the quality measure.

Stop criterion:
In the tests, the stop criterion was defined to become true when all the configurations provided by the generators flow are tested.

Generators flow:
The generators flow used for this analysis of meta-learning is rather simple, to give the opportunity to observe the behavior of the algorithm. It is not the best choice for solving classification problems, in general, but lets us better see the very interesting details of its cooperation with the complexity control mechanism. To find more sophisticated configuration machines, more complex generators graph should be used. Anyways, it will be seen that using even so basic generators flow, the results ranked high by the MLA, can be very good. The generators flow used in experiments is presented in Figure 15. Very similar generators flow was explained in detail in Section 5.1.

To know what exactly will be generated by this generators flow, the configurations (the sets) of classifiers generator and rankings generator must be specified. Here, we use the following:

Classifier set:
kNN (Euclidean) — k Nearest Neighbors with Euclidean metric,

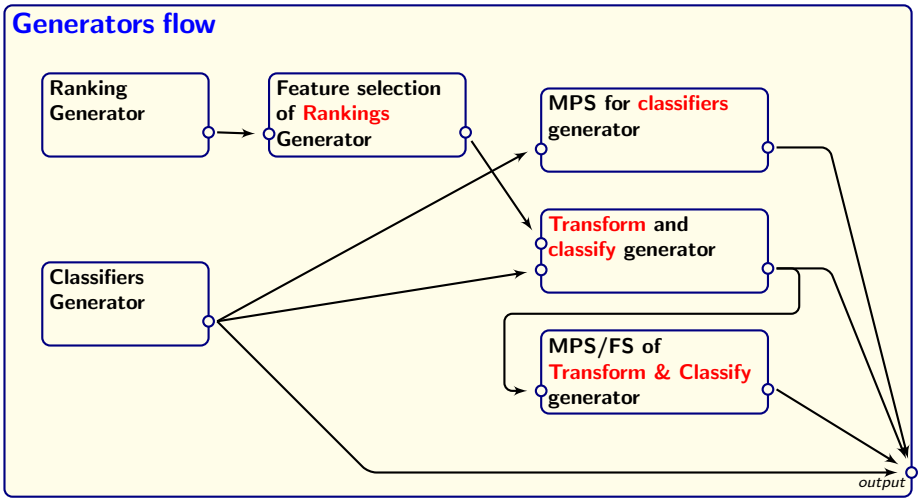


Figure 15: Generators flow used in tests.

kNN [MetricMachine (EuclideanOUO)] — kNN with Euclidean metric for ordered features and Hamming metric for unordered ones,
kNN [MetricMachine (Mahalanobis)] — kNN with Mahalanobis metric,
NBC — Naive Bayes Classifier
SVMClassifier — Support Vector Machine with Gaussian kernel
LinearSVMClassifier — SVM with linear kernel
[ExpectedClass, kNN [MetricMachine (EuclideanOUO)]] — first, the ExpectedClass[6] machine transforms the original dataset, then the transformed data become the learning data for kNN,
[LVQ, kNN (Euclidean)] — first, Learning Vector Quantization algorithm is used to select prototypes, then kNN uses them as its training data (neighbor candidates),
Boosting (10x) [NBC] — boosting algorithm with 10 NBCs.

Ranking set:

RankingCC — correlation coefficient based feature ranking,
RankingFScore — Fisher-score based feature ranking.

The base classifiers and ranking algorithms, together with the generators flow presented in Figure 15, produce 54 configurations, that are nested (one by one) within the meta-learning test-scheme and sent to the meta-learning heap for complexity controlled run.

All the configurations provided by the generators flow are presented in Table 3. The square brackets, used there, denote submachine relation. A machine name standing before the brackets is the name of the parent machine, and the machines in the brackets are the submachines. When more than one name is embraced with the brackets (comma-separated

6. ExpectedClass is a transformation machine, which outputs a dataset consisting of one "super-prototype" per class. The super-prototype for each class is calculated as vector of the means (for ordered features) or expected values (for unordered features) for given class. Followed by a kNN machine, it composes a very simple classifier, even more "naive" than the Naive Bayes Classifier, though sometimes quite successful.

names), the machines are placed within a scheme machine. Parentheses embrace significant parts of machine configurations.

To make the notation easier to read, we explain some entries of the table. The notation does not present the input–output interconnections, so it does not allow to reconstruct the full scenario in detail, but shows machine structure, which is sufficient, here, and significantly reduces the occupied space.

The following notation:

[[[RankingCC], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]

means that a feature selection machine selects features from the top of a correlation coefficient based ranking, and next, the dataset composed of the feature selection is an input for a kNN with Euclidean metric—the combination of feature selection and kNN classifier is controlled by a TransformAndClassify machine.

Notation:

[[[RankingCC], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]

means nearly the same as the previous example, except the fact that between the feature selection machine and the kNN is placed an LVQ machine as the instance selection machine.

The following notation represents the ParamSearch machine which optimizes parameters of a kNN machine:

ParamSearch [kNN (Euclidean)]

In the case of

ParamSearch [LVQ, kNN (Euclidean)]

both LVQ and kNN parameters are optimized by the ParamSearch machine.

However in the case of notation

ParamSearch [[[RankingCC], FeatureSelection], kNN (Euclidean)]

only the number of chosen features is optimized because this configuration is provided by the MPS/FS of Transform & Classify Generator (see Figure 15), where the ParamSearch configuration is set up to optimize only the parameters of feature selection machine. Of course, it is possible to optimize all the parameters of all submachines, but this is not the goal of the example and, moreover, the optimization of too many parameters may provide to very complex machines (sometimes uncomputable in a rational time).

Table 3: Machine configurations produced by the generators flow of Figure 15 and the enumerated sets of classifiers and rankings.

1 kNN (Euclidean)
2 kNN [MetricMachine (EuclideanOUO)]
3 kNN [MetricMachine (Mahalanobis)]

4	NBC
5	SVMClassifier [KernelProvider]
6	LinearSVMClassifier [LinearKernelProvider]
7	[ExpectedClass, kNN [MetricMachine (EuclideanOUO)]]
8	[LVQ, kNN (Euclidean)]
9	Boosting (10x) [NBC]
10	[[[RankingCC], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]
11	[[[RankingCC], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]
12	[[[RankingCC], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify]
13	[[[RankingCC], FeatureSelection], [NBC], TransformAndClassify]
14	[[[RankingCC], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify]
15	[[[RankingCC], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify]
16	[[[RankingCC], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]
17	[[[RankingCC], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]
18	[[[RankingCC], FeatureSelection], [Boosting (10x) [NBC]], TransformAndClassify]
19	[[[RankingFScore], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]
20	[[[RankingFScore], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]
21	[[[RankingFScore], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify]
22	[[[RankingFScore], FeatureSelection], [NBC], TransformAndClassify]
23	[[[RankingFScore], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify]
24	[[[RankingFScore], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify]
25	[[[RankingFScore], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]
26	[[[RankingFScore], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]
27	[[[RankingFScore], FeatureSelection], [Boosting (10x) [NBC]], TransformAndClassify]
28	ParamSearch [kNN (Euclidean)]
29	ParamSearch [kNN [MetricMachine (EuclideanOUO)]]
30	ParamSearch [kNN [MetricMachine (Mahalanobis)]]
31	ParamSearch [NBC]
32	ParamSearch [SVMClassifier [KernelProvider]]
33	ParamSearch [LinearSVMClassifier [LinearKernelProvider]]
34	ParamSearch [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]]
35	ParamSearch [LVQ, kNN (Euclidean)]
36	ParamSearch [Boosting (10x) [NBC]]
37	ParamSearch [[[RankingCC], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]

38	ParamSearch [[[RankingCC], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]
39	ParamSearch [[[RankingCC], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify]
40	ParamSearch [[[RankingCC], FeatureSelection], [NBC], TransformAndClassify]
41	ParamSearch [[[RankingCC], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify]
42	ParamSearch [[[RankingCC], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify]
43	ParamSearch [[[RankingCC], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]
44	ParamSearch [[[RankingCC], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]
45	ParamSearch [[[RankingCC], FeatureSelection], [Boosting (10x) [NBC]], TransformAndClassify]
46	ParamSearch [[[RankingFScore], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]
47	ParamSearch [[[RankingFScore], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]
48	ParamSearch [[[RankingFScore], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify]
49	ParamSearch [[[RankingFScore], FeatureSelection], [NBC], TransformAndClassify]
50	ParamSearch [[[RankingFScore], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify]
51	ParamSearch [[[RankingFScore], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify]
52	ParamSearch [[[RankingFScore], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]
53	ParamSearch [[[RankingFScore], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]
54	ParamSearch [[[RankingFScore], FeatureSelection], [Boosting (10x) [NBC]], TransformAndClassify]

Data benchmarks

Table 4 summarizes the properties of data benchmarks (from the UCI repository) used in the tests.

Dataset	# classes	# instances	# features	# ordered f.
appendicitis	2	106	7	7
german-numeric	2	1000	24	24
glass	6	214	9	9
ionosphere-ALL	2	351	34	34
mushroom	2	8124	22	0
splice	3	3190	60	0
thyroid-all	3	7200	21	6
vowel	6	871	3	3

Table 4: Benchmark data used for the tests.

Table 5 presents exact complexities (see Eq. 6 for each test machine configuration obtained for the vowel data. The table has three columns: the first one contains the task id which corresponds to the order of configurations providing by the generators flow (the same as the ids in Table 3), the second column is the task configuration description, and the third column shows the task complexity. The rows are sorted according to the complexity.

Table 5: Complexities of the tasks produced by the generators flow for vowel data.

4	NBC	4.77E+006
31	ParamSearch [NBC]	4.99E+006
13	[[[RankingCC], FeatureSelection], [NBC], TransformAndClassify]	5.25E+006
22	[[[RankingFScore], FeatureSelection], [NBC], TransformAndClassify]	5.26E+006
7	[ExpectedClass, kNN [MetricMachine (EuclideanOUO)]]	5.29E+006
1	kNN (Euclidean)	5.78E+006
16	[[[RankingCC], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]	5.81E+006
25	[[[RankingFScore], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]	5.81E+006
10	[[[RankingCC], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]	5.84E+006
19	[[[RankingFScore], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]	5.84E+006
2	kNN [MetricMachine (EuclideanOUO)]	7.82E+006
11	[[[RankingCC], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]	8.09E+006
20	[[[RankingFScore], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]	8.09E+006
17	[[[RankingCC], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]	8.18E+006
26	[[[RankingFScore], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]	8.18E+006
12	[[[RankingCC], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify]	9.60E+006
21	[[[RankingFScore], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify]	9.60E+006
3	kNN [MetricMachine (Mahalanobis)]	9.70E+006
8	[LVQ, kNN (Euclidean)]	1.00E+007
6	LinearSVMClassifier [LinearKernelProvider]	1.19E+007
33	ParamSearch [LinearSVMClassifier [LinearKernelProvider]]	1.21E+007
15	[[[RankingCC], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify]	1.46E+007
24	[[[RankingFScore], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify]	1.46E+007
14	[[[RankingCC], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify]	1.72E+007

23	[[[RankingFScore], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify]	1.72E+007
5	SVMClassifier [KernelProvider]	1.82E+007
18	[[[RankingCC], FeatureSelection], [Boosting (10x) [NBC]], TransformAndClassify]	4.20E+007
27	[[[RankingFScore], FeatureSelection], [Boosting (10x) [NBC]], TransformAndClassify]	4.20E+007
9	Boosting (10x) [NBC]	4.31E+007
36	ParamSearch [Boosting (10x) [NBC]]	4.33E+007
34	ParamSearch [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]]	5.27E+007
29	ParamSearch [kNN [MetricMachine (EuclideanOUO)]]	6.84E+007
30	ParamSearch [kNN [MetricMachine (Mahalanobis)]]	8.09E+007
40	ParamSearch [[[RankingCC], FeatureSelection], [NBC], TransformAndClassify]	1.63E+008
49	ParamSearch [[[RankingFScore], FeatureSelection], [NBC], TransformAndClassify]	1.63E+008
37	ParamSearch [[[RankingCC], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]	1.78E+008
46	ParamSearch [[[RankingFScore], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]	1.78E+008
43	ParamSearch [[[RankingCC], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]	1.79E+008
52	ParamSearch [[[RankingFScore], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]	1.79E+008
38	ParamSearch [[[RankingCC], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]	2.24E+008
47	ParamSearch [[[RankingFScore], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]	2.24E+008
44	ParamSearch [[[RankingCC], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]	2.39E+008
53	ParamSearch [[[RankingFScore], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]	2.39E+008
39	ParamSearch [[[RankingCC], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify]	2.54E+008
48	ParamSearch [[[RankingFScore], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify]	2.54E+008
42	ParamSearch [[[RankingCC], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify]	3.65E+008
51	ParamSearch [[[RankingFScore], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify]	3.65E+008
41	ParamSearch [[[RankingCC], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify]	4.36E+008
50	ParamSearch [[[RankingFScore], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify]	4.36E+008
28	ParamSearch [kNN (Euclidean)]	4.52E+008
32	ParamSearch [SVMClassifier [KernelProvider]]	8.04E+008

35	ParamSearch [LVQ, kNN (Euclidean)]	9.46E+008
45	ParamSearch [[[RankingCC], FeatureSelection], [Boosting (10x) [NBC]], TransformAndClassify]	1.30E+009
54	ParamSearch [[[RankingFScore], FeatureSelection], [Boosting (10x) [NBC]], TransformAndClassify]	1.30E+009

The results obtained for the benchmarks are presented in the form of diagrams. The diagrams are very specific and present many properties of the meta-learning algorithm. The diagrams present information about the times of starting, stopping and breaking of each task, about complexities (global, time and memory) of each test task, about the order of the test tasks (according to their complexities, compare Table 3) and about the accuracy of each tested machine.

In the middle of the diagram—see the first diagram in Figure 16—there is a column with task ids (the same ids as in tables 3 and 5). But the row order in diagram reflects the complexities of test task, not the order of machine creation. It means that at the top, the most complex tasks are placed and at the bottom the task of the smallest complexities. For example, in Figure 16, at the bottom, we can see task ids 4 and 31 which correspond to the Naive Bayes Classifier and the ParamSearch [NBC] classifier. At the top, task ids 54 and 45 are the most complex ParamSearch test tasks of this benchmark.

On the right side of the *Task id* column, there is a plot presenting starting, stopping and breaking times of each test task. As it was presented in Section 5.4 the tasks are started according to the approximation of their complexities, and when a given task does not reach the time limit (which correspond to the time complexity—see Section 6.1) it finishes normally, otherwise, the task is broken and restarted according to the modified complexity (see Section 6.1). For an example of restarted task please look at Figure 16, at the topmost task-id 54—there are two horizontal bars corresponding to the two periods of the task run. The break means that the task was started, broken because of exceeded allocated time and restarted when the tasks of larger complexities got their turn. The breaks occur for the tasks, for which the complexity prediction was too optimistic. A survey of different diagrams (in Figure 16–23) easily brings the conclusion that the amount of inaccurately predicted time complexity is quite small (there are quite few broken bars). Note that, when a task is broken, its subtasks, that have already been computed are not recalculated during the test-task restart (due to the machine unification mechanism and machine cache). At the bottom, the *Time line* axis can be seen. The scope of the time is the interval $[0, 1]$ to show the times relative to the start and the end of the whole MLA computations. To make the diagrams clearer, the tests were performed on a single CPU, so only one task was running at a time and we can not see any two bars overlapping in time. If we ran the projects on more than one CPU, a number of bars would be "active" at almost each time, which would make reading the plots more difficult.

The simplest tasks are started first. They can be seen at the bottom of the plot. Their bars are very short, because they required relatively short time to be calculated. The higher in the diagram (i.e. the larger predicted complexity), the longer bars can be seen. It confirms the adequacy of the complexity estimation framework, because the relations between the predictions correspond very good to the relations between real time consumed by the tasks. When browsing other diagrams a similar behavior can be observed—the simple tasks are started at the beginning and then, the more and more complex ones.

On the left side of the *Task-id* column, the accuracies of classification test tasks and their approximated complexities are presented. At the bottom, there is the the *Accuracy* axis with interval from 0 (on the right) to 1 (on the left side). Each test task has its own gray bar starting at 0 and finished exactly at the point corresponding to the accuracy. So the accuracies of all the tasks are easily visible and comparable. Longer bars show higher accuracies. However remember that the experiments were not tuned to obtain the best accuracies possible, but to illustrate the behavior of the complexity controlled meta-learning and the generators flows.

The leftmost column of the diagram presents ranks of the test tasks (the ranking of the accuracies). In the case of the vowel data, the machine of the best performance is the kNN machine (the task id is 1 and the accuracy rank is 1 too) ex equo with kNN [MetricMachine (EuclideanOUO)] (task id 2). The second rank was achieved by kNN with Mahalanobis metric which is a more complex task.

Between the columns of *Task-id* and the accuracy-ranks, on top of the gray bars corresponding to the accuracies, some thin solid lines can be seen. The lines start at the right side as the accuracy bars and go to the right according to proper magnitudes. For each task, the three lines correspond to the total complexity (the upper line), the memory complexity (the middle line) and the time complexity (the lower line)[7]. All three complexities are the approximated complexities (seeEq. 6 and 7). Approximated complexities presented on the left side of the diagram can be easily compared visually to the time-schedule obtained in the real time on the right side of the diagram. Longer lines mean higher complexities. The longest line is spread to maximum width. The others are proportionally shorter. So the complexity lines at the top of the diagram are long while the lines at the bottom are almost invisible. It can be seen that sometimes the time complexity of a task is smaller while the total complexity is larger and vice versa. For example see tasks 42 and 48 again in Figure 16.

The meta-learning illustration diagrams (Figures 16–23) clearly show that the behavior of different machines changes between benchmarks. Even the standard deviation of accuracies is highly diverse. When looking at accuracies within some test, groups of machine of similar accuracy may be seen, however for other benchmark, within the same group of machines the accuracies are very variant. Of course the complexity of a test task for given configuration may change significantly from benchmark to benchmark. However it can be seen that in the case of benchmarks of similar properties, the permutations of task ids in the diagrams are partially similar (e.g. see the bottoms of Figures 21 and 23).

The **most important feature** of the presented MLA is that it facilitates finding accurate solutions in the order of increasing complexity. Simple solutions are started before the complex ones, to maximize the probability that an accurate solution is found as soon as possible. It is confirmed by the diagrams in Figures 16–23. Thanks to this property, in the case of a strong stop-condition (significant restriction on the running time) we are able to find really good solution(-s) because of starting test tasks in proper order. Even if some tasks get broken and restarted, it is not a serious hindrance to the main goal of algorithm.

For a few of the benchmarks, very simple and accurate models were found just at the beginning of the meta-learning process. Please see Figure 16 task ids 1 and 2, Figure 17 task

7. In the case of time complexity the $t/\log t$ is plotted, not the time t itself.

ids 1 and 2, Figure 19 task ids 1 and 2, Figure 22 task ids 4 and 31, Figure 23 task id 19. The machines of the first four diagrams, are all single machines of relatively low complexities. But not only single machines may be of small complexity. The most accurate machine (of the 54 machines being analysed) for the thyroid data is the combination of feature selection based on F-score with kNN machine (task id 19). Even nontrivial combinations of machines (complex structures) may provide low time and memory complexity while single machine do not guarantee small computational complexity. In the case of very huge datasets (with huge number of instances and features) almost no single algorithm works successfully in rational time. However classifiers (or approximators) preceded by not too complex data transformation algorithms (like feature selection or instance selection) may be calculated in quite short time. The transformations may reduce the costs of classifier learning and testing, resulting in significant decrease of the overall time/memory consumption.

In some of the benchmarks (see Figures 18, 20 and 21) the most accurate machine configurations were not of as small complexity as in the cases mentioned above. For the german-numeric benchmark, the best machines are SVM's with linear and Gaussian kernels (task ids 6, 33[8] and 5). The winner machines, for this benchmark, are of average complexity and are placed in the middle of the diagram. For the ionosphere benchmark, the most accurate machine is the SVM with Gaussian kernel but nested in a ParamSearch machine tuning the SVM parameters. This is one of the most complex test tasks. The MLA running on the mushroom data, has found several alternative configurations of very good performance: the simplest is a boosting of naive bayes classifier (task id 9), the second is the kNN [MetricMachine (EuclideanOUO)], followed by SVM (with Gaussian kernel), ParamSearch [kNN [MetricMachine (EuclideanOUO)]], other two configuration of kNN and ParamSearch [SVMClassifier [KernelProvider]].

Naturally, in most cases, more optimal machine configuration may be found, when using more sophisticated configuration generators and larger sets of classifiers and data transformations (for example adding decision trees, instance selection methods, feature aggregation, etc.) and performing deeper parameter search.

Note that the approximated complexity time is not in perfect compatibility with real time presented on the right side of the diagrams. The differences are due to not only the approximation inaccuracy, but also the machine unifications and some deviations in real CPU time consumption which sometimes is different even for two runs of the same task (probably it is caused by the .Net kernel, for example by garbage collection which, from time to time, must use the CPU to perform its own tasks).

Without repeating the experiments, one can think of the results obtained with the stop criterion set to a time-limit constraint. For example, assume that the time limit was set to 1/5 of the time really used by given MLA run. In such a case, some of the solutions described above would not be reached, but still, for several datasets the optimal solutions would be obtained and for other benchmarks, some slightly worse solutions would be the winners. This feature is crucial, because in real life problems the time is always limited and we are interested in finding as good solutions as possible within the time limits.

8. Note that 33 means ParamSearch [LinearSVMClassifier [LinearKernelProvider]] where a linear SVM is nested within a ParamSearch, but the auto-scenario for linear SVM is empty, which means that ParamSearch machine does not optimize anything and indeed it is equivalent to the linear SVM itself. The small difference is a result of additional memory costs for ParamSearch encapsulation.

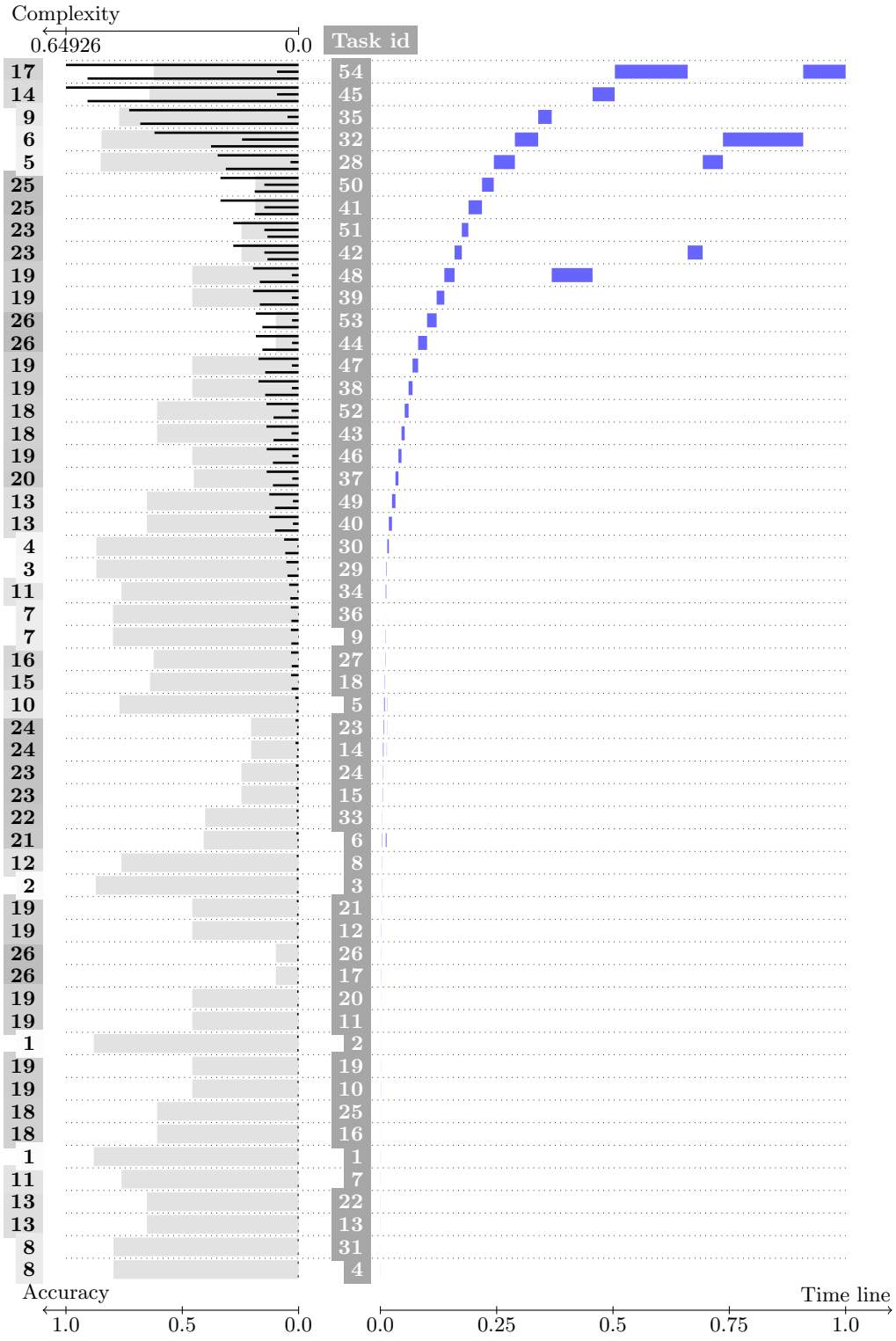


Figure 16: vowel

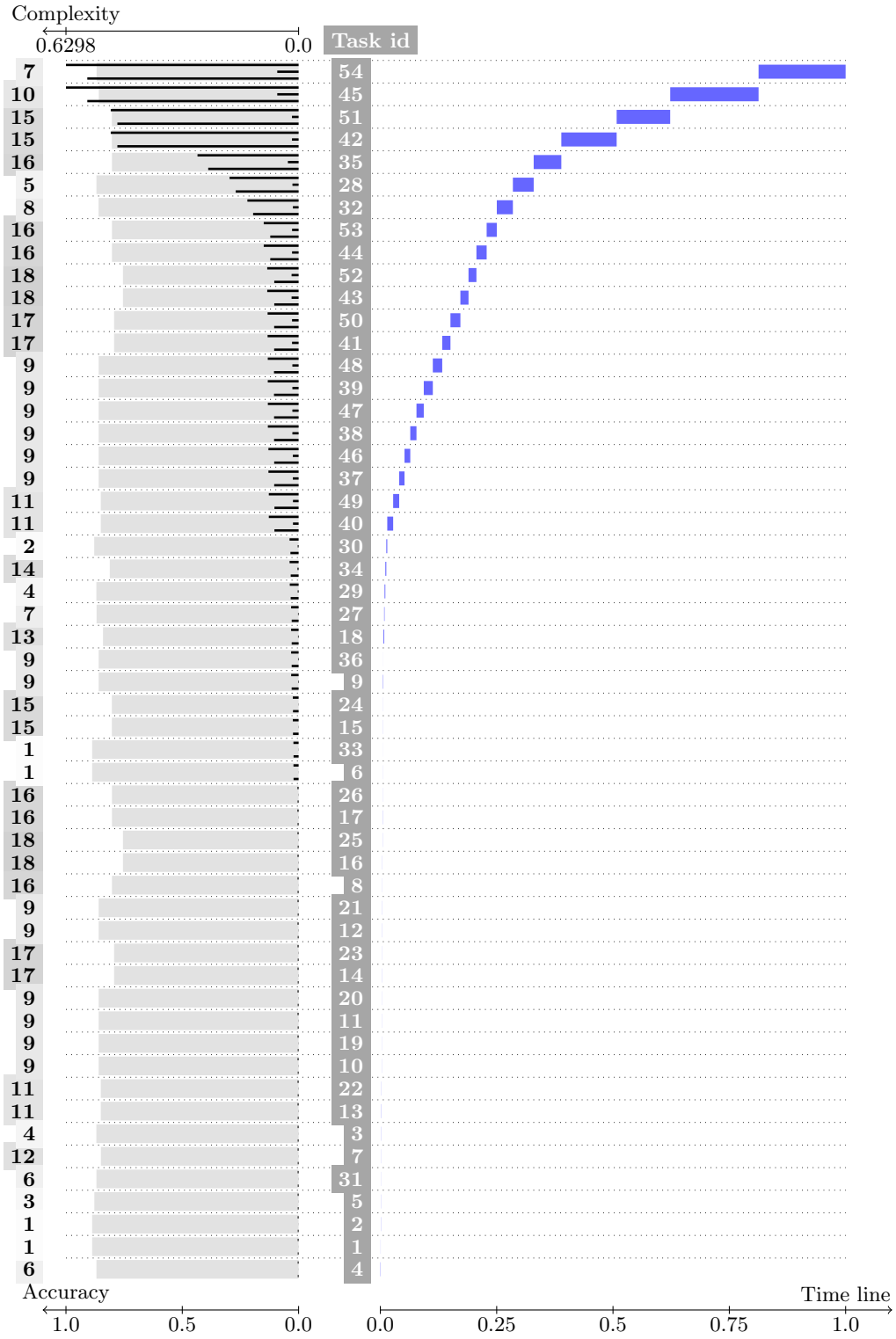


Figure 17: appendicitis

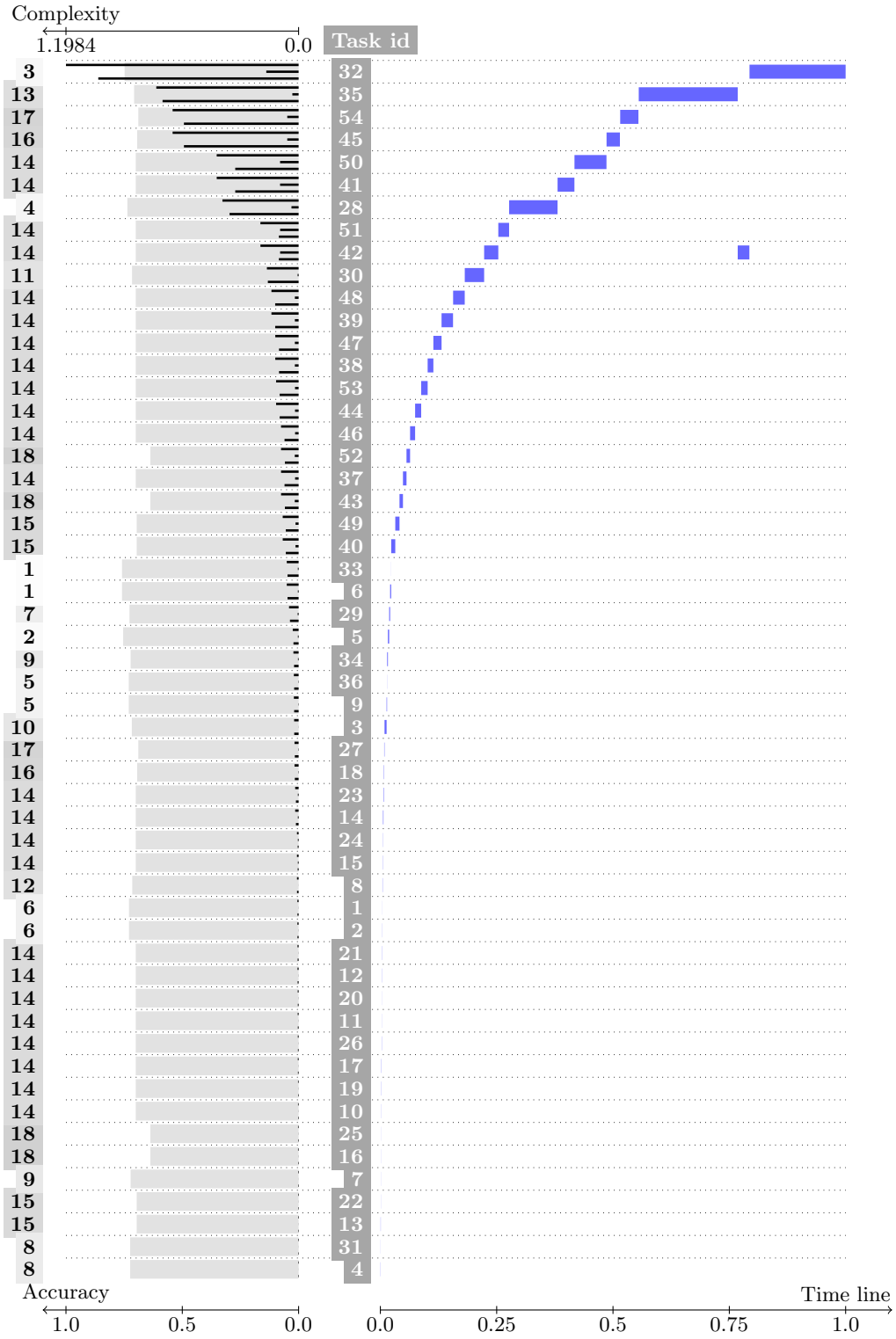


Figure 18: german-numeric

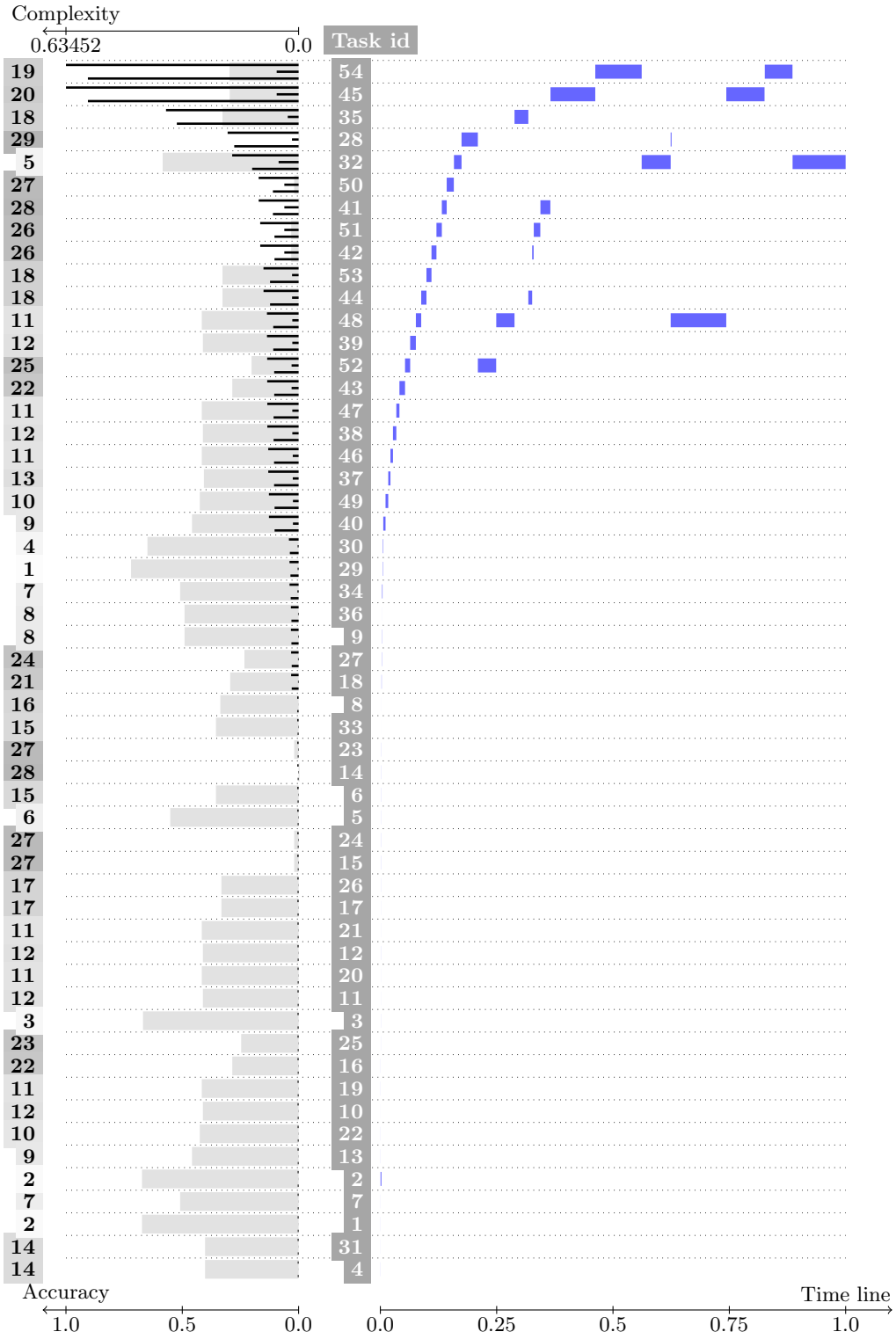


Figure 19: glass

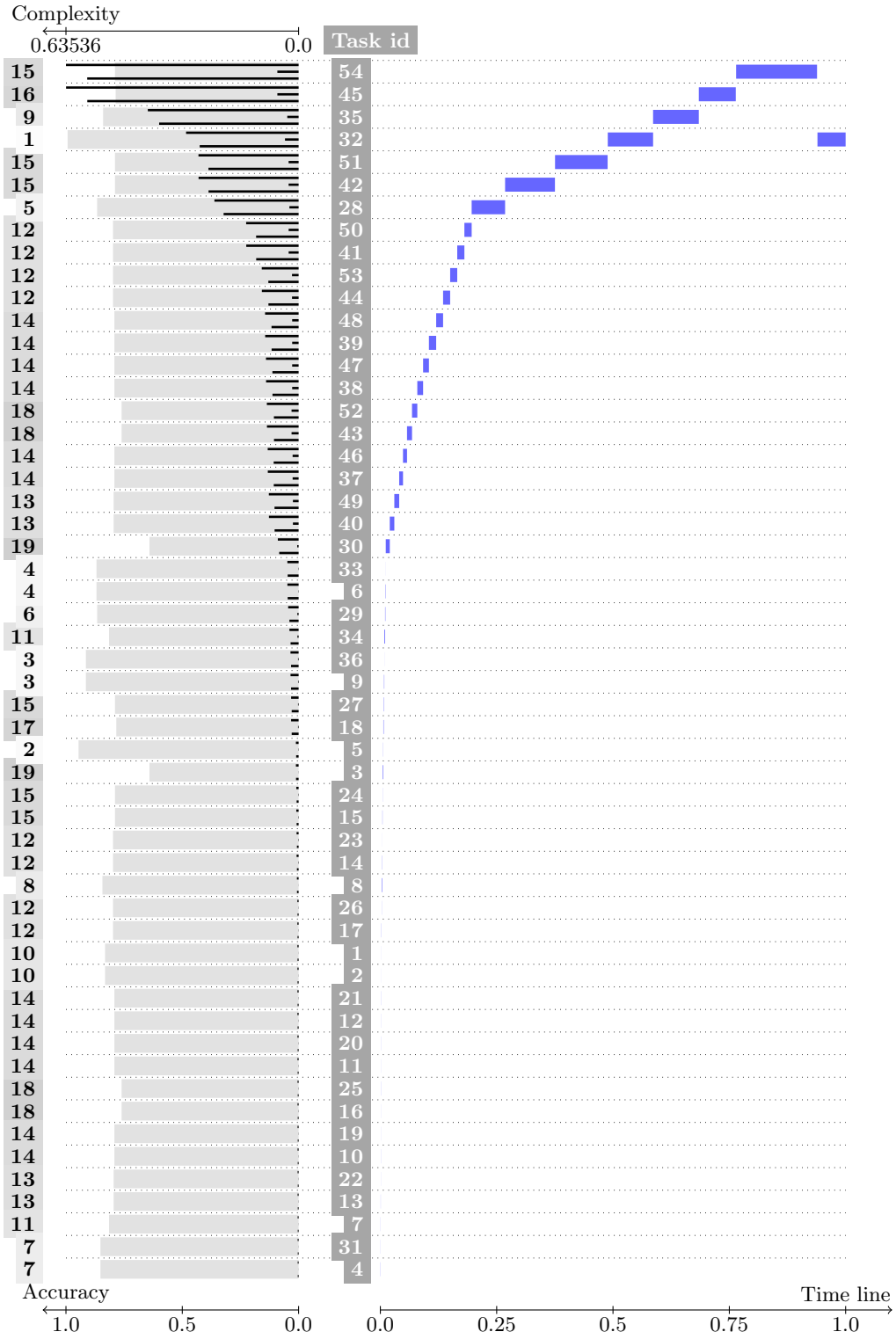


Figure 20: ionosphere-ALL

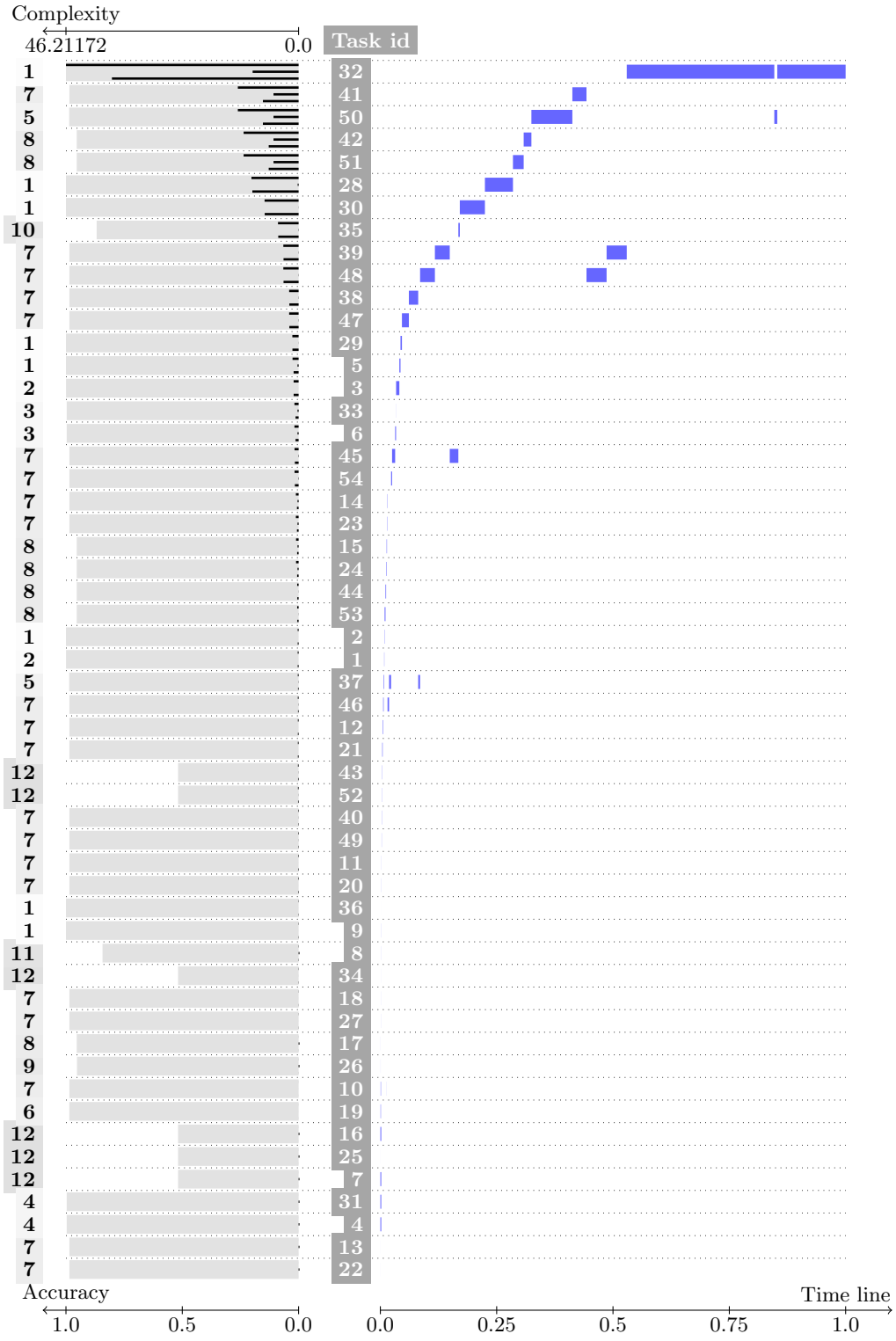


Figure 21: mushroom

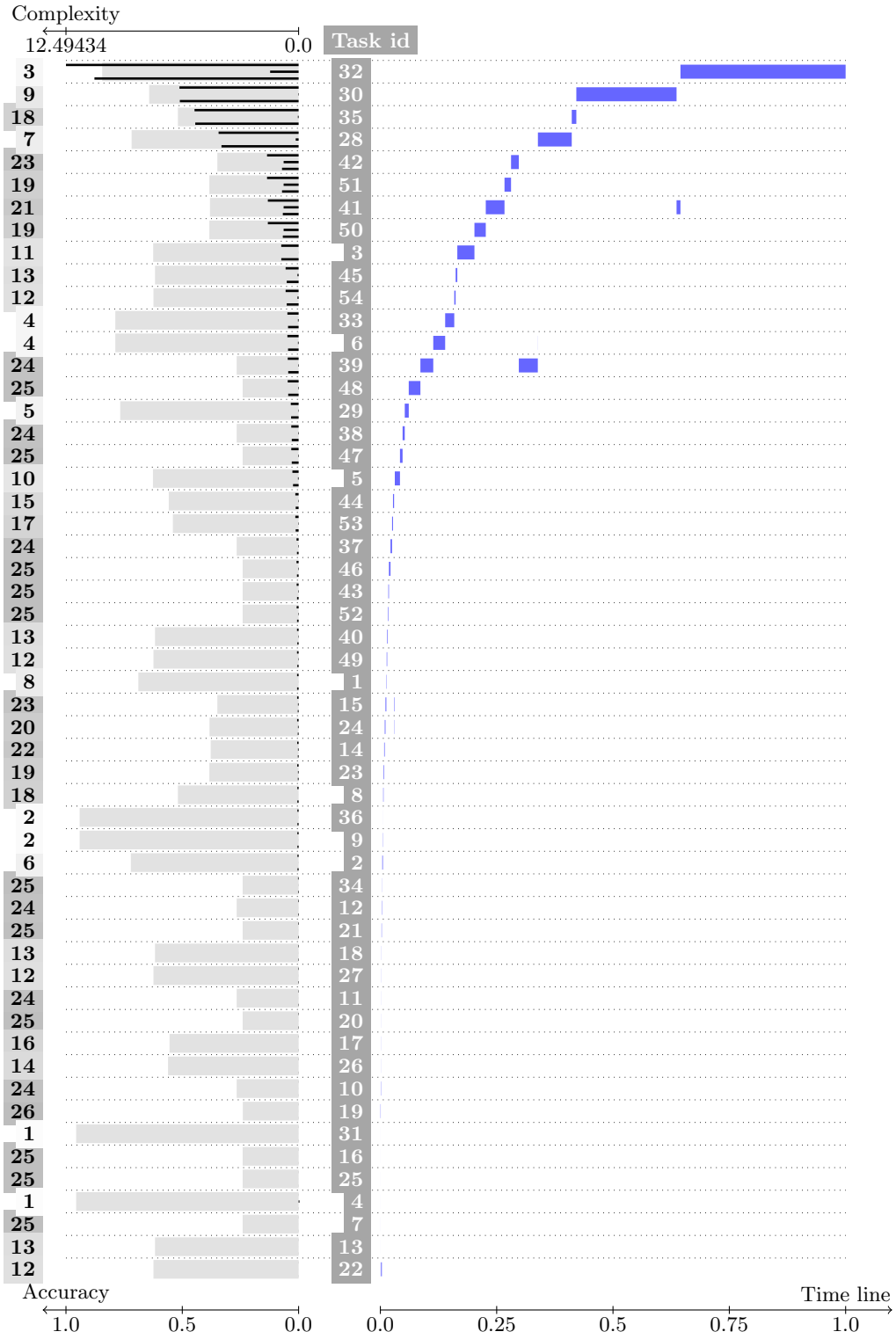


Figure 22: splice

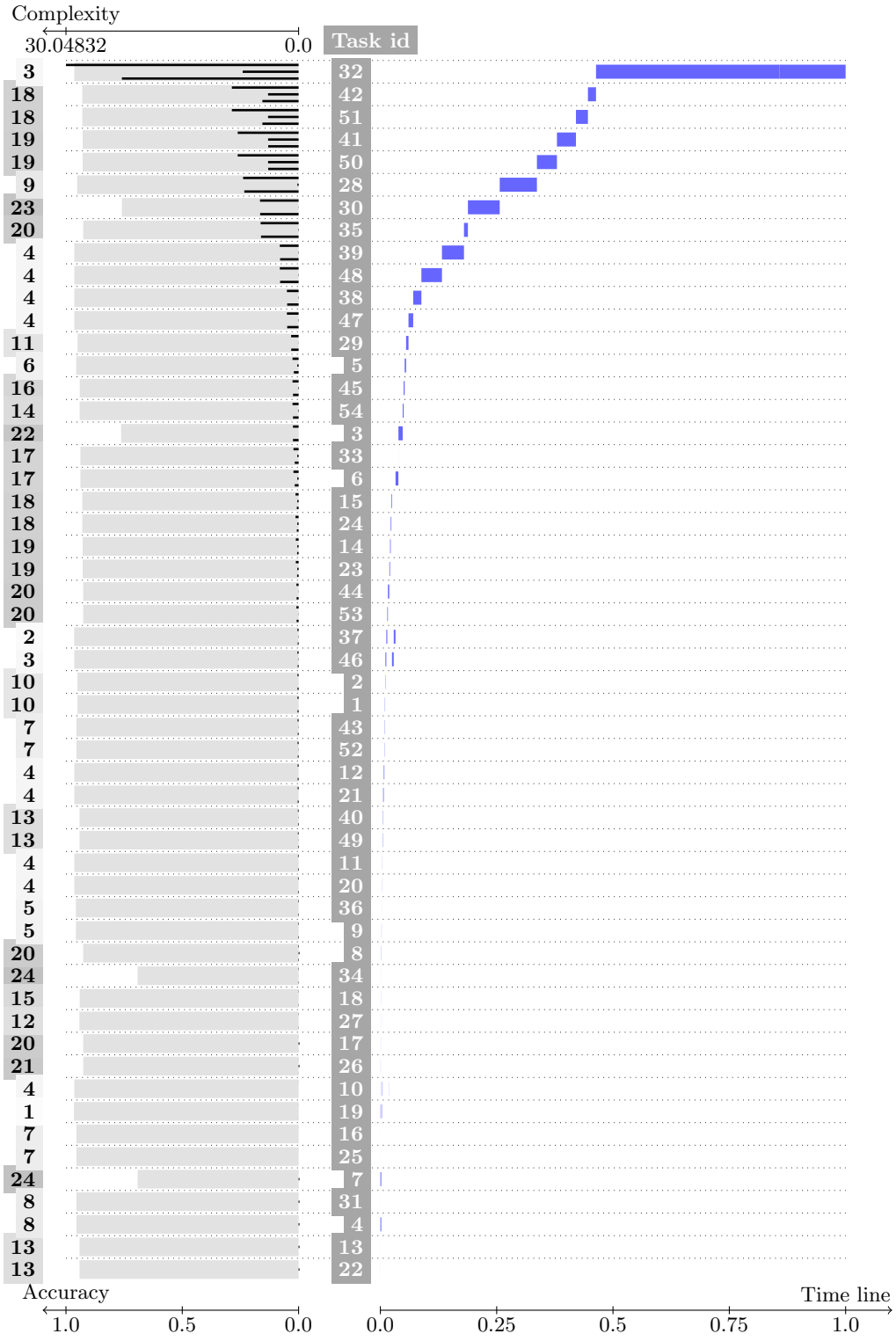


Figure 23: thyroid-all

8. Future and Current Research

The next step toward more sophisticated meta-learning is to design advanced machine configuration generators, able to use and gather meta-knowledge of different kinds. We have already started some efforts in this direction. For example, we are working on using meta-knowledge to advise composition of complex machines and to get rid of ineffective machine combinations. Meta-knowledge will also help produce and advise new data transformations and check their influence on the results of the meta-search. Advanced generators can learn and collect experience from information about the most important events of the meta-search (starting, stopping and breaking test tasks). Using a number of specialized machine generators will facilitate composition of a variety of machine configurations, while enabling *smart* control over the generators results, by means of the part of complexity definition, responsible for machine attractiveness.

9. Summary

The meta-learning algorithm presented in the paper opens new gates of computational intelligence. The algorithm may be used to solve problems of various types, in the area of computational intelligence. Defining the goal is every flexible and may fulfill different requirements.

Also the search space, browsed during meta-learning is defined by means of a general mechanism of generators flow, which enables defining different combinations of base machines in a flexible way. As a result, the meta-learning searches among simple machines, compositions of transformers and classifiers or approximators, and also among more complex structures. It means that we look for more and more optimal combinations of transformations and final classifiers or approximators. What's more, this meta-learning is able to find independent (more optimal) transformations for different classifiers and then, use the complex models in committees.

The criterion of choosing the best model may be defined up to the needs, thanks to the query system. The focus may be put on accuracy, balanced accuracy or some result of a more complex statistical tests.

The most important job is made by the complexity control module which organizes the order of test task analysis in the loop of meta-learning. In most cases, the complexities are learned by approximation techniques. This approach may and be used to any type of machines in the system. Even the machines that will be added in future, may work as well with the scheme of complexity control. The biggest advantage of complexity based test task order is its independence of particular problem and used generators flow. Without such a mechanism, meta-learning is condemned to a serious danger of yielding no results because of starting a long lasting test task, which can not be finished in available time.

There is no problem to search for solutions among different complex machine structures exploiting feature selection algorithms, instance selection algorithms, other data transformation methods, classification machines, approximation machines, machine committees etc. MLAs do not need to know much about the nature of different machine components, so as to be able to run the tasks from the simplest to the most time and memory consuming.

They can not *loose* simple and accurate solution, even when they are given little time for the search.

Proposed methodology allows to collect meta-knowledge and use it in further work (of the same MLA or other MLAs). Complexity estimation may be augmented, in a variety of ways, by defining corrections based on the knowledge gained during meta-learning.

Presented MLAs are able to autonomously and effectively search through functional model spaces for close to optimal solutions which sometimes are simple and sometimes really complex. They present very universal and powerful tools for solving really non-trivial problems in computational intelligence.

References

H. Bensusan, C. Giraud-Carrier, and C. J. Kennedy. A higher-order approach to meta-learning. In J. Cussens and A. Frisch, editors, *Proceedings of the Work-in-Progress Track at the 10th International Conference on Inductive Logic Programming*, pages 33–42, 2000. URL citeseer.ist.psu.edu/article/bensusan00higherorder.html. 1

C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995. 2

B. E. Boser, I. M. Guyon, and V. Vapnik. A training algorithm for optimal margin classifiers. In D. Haussler, editor, *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, Pittsburgh, PA, 1992. ACM Press. 3.1

Pavel Brazdil, C. Soares, and J. Pinto da Costa. Ranking learning algorithms: Using IBL and meta-learning on accuracy and time results. *Machine Learning*, 50(3):251–277, 2003. 1

Philip Chan and Salvatore J. Stolfo. On the accuracy of meta-learning for scalable data mining. *Journal of Intelligent Information Systems*, 8:5–28, 1996. 1

T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *Institute of Electrical and Electronics Engineers Transactions on Information Theory*, 13(1):21–27, January 1967. 3.1

W. Duch and Ł. Itert. Committees of undemocratic competent models. In *Proceedings of the Joint Int. Conf. on Artificial Neu-ral Networks (ICANN) and Int. Conf. on Neural Information Processing (ICONIP)*, pages 33–36, Istanbul, Turkey, 2003. 1

R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley, 2 edition, 2001. 2

K. Grąbczewski and W. Duch. The Separability of Split Value criterion. In *Proceedings of the 5th Conference on Neural Networks and Their Applications*, pages 201–208, Zakopane, Poland, June 2000. 3.4

I. Guyon, S. Gunn, M. Nikravesh, and L. Zadeh. *Feature extraction, foundations and applications*. Springer, 2006. 1

Isabelle Guyon. Nips 2003 workshop on feature extraction. http://www.clopinet.com/isabelle/Projects/NIPS2003, December 2003. 1

Isabelle Guyon. Performance prediction challenge. http://www.modelselect.inf.ethz.ch, July 2006. 1

T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, 2001. 2

N. Jankowski. Applications of Levin's universal optimal search algorithm. In E. Kącki, editor, *System Modeling Control'95*, volume 3, pages 34–40, Łódź, Poland, May 1995. Polish Society of Medical Informatics. 2, 6.1

N. Jankowski and K. Grąbczewski. Learning machines. In I. Guyon, S. Gunn, M. Nikravesh, and L. Zadeh, editors, *Feature extraction, foundations and applications*, pages 29–64. Springer, 2006. 1

Norbert Jankowski and Krzysztof Grąbczewski. Heterogenous committees with competence analysis. In N. Nedjah, L.M. Mourelle, M.M.B.R Vellasco, A. Abraham, and M. Köppen, editors, *Fifth International conference on Hybrid Intelligent Systems*, pages 417–422, Brasil, Rio de Janeiro, November 2005. IEEE, Computer Society. 1

Petr Kadlec and Bogdan Gabrys. Learnt topology gating artificial neural networks. In *IEEE World Congress on Computational Intelligence*, pages 2605–2612. IEEE Press, 2008. 1

A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Prob. Inf. Trans.*, 1:1–7, 1965. 2

M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Text and Monographs in Computer Science. Springer-Verlag, 1993. 2, 2, 2, 6.1

C. J. Merz and P. M. Murphy. UCI repository of machine learning databases, 1998. http://www.ics.uci.edu/∼mlearn/MLRepository.html. 7

T. Mitchell. *Machine learning*. McGraw Hill, 1997. 2

Y.H. Peng, P.A. Falch, C. Soares, and P. Brazdil. Improved dataset characterisation for meta-learning. In *The 5th International Conference on Discovery Science*, pages 141–152, Luebeck, Germany, January 2002. Springer-Verlag. 1

Bernhard Pfahringer, Hilan Bensusan, and Christophe Giraud-Carrier. Meta-learning by landmarking various learning algorithms. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 743–750. Morgan Kaufmann, June 2000. 1

A.L. Prodromidis and P.K. Chan. Meta-learning in distributed data mining systems: Issues and approaches. In Hillol Kargupta and Philip Chan, editors, *Book on Advances of Distributed Data Mining*. AAAI press, 2000. URL citeseer.ist.psu.edu/article/prodromidis00metalearning.html. 1

J. Rissanen. Modeling by shortest data description. *Automatica*, 14:445–471, 1978. 2

Kate A. Smith-Miles. Towards insightful algorithm selection for optimization using meta-learning concepts. In *IEEE World Congress on Computational Intelligence*, pages 4117–4123. IEEE Press, 2008. 1

L. Todorovski and S. Dzeroski. Combining classifiers with meta decision trees. *Machine Learning Journal*, 50(3):223–249, 2003. URL citeseer.ist.psu.edu/article/todorovski03combining.html. 1

V. Vapnik. *Statistical Learning Theory*. Wiley, New York, NY, 1998. 3.1

P. J. Werbose. *Beyond regression: New tools for prediction and analysis in the bahavioral sciences*. PhD thesis, Harvard Univeristy, Cambridge, MA, 1974. 3

Contents