VERSITA

Int. J. Appl. Math. Comput. Sci., 2012, Vol. 22, No. 3, 647–667
DOI: 10.2478/v10006-012-0049-y

amcs

GRAPH–BASED GENERATION OF A META–LEARNING SEARCH SPACE

NORBERT JANKOWSKI

Department of Informatics
Nicolaus Copernicus University, ul. Grudziądzka 5, 87-100 Toruń, Poland
e-mail: norbert@is.umk.pl

Meta-learning is becoming more and more important in current and future research concentrated around broadly defined data mining or computational intelligence. It can solve problems that cannot be solved by any single, specialized algorithm. The overall characteristic of each meta-learning algorithm mainly depends on two elements: the learning machine space and the supervisory procedure. The former restricts the space of all possible learning machines to a subspace to be browsed by a meta-learning algorithm. The latter determines the order of selected learning machines with a module responsible for machine complexity *evaluation*, organizes tests and performs analysis of results. In this article we present a framework for meta-learning search that can be seen as a method of sophisticated description and evaluation of functional search spaces of learning machine configurations used in meta-learning. Machine spaces will be defined by specially defined graphs where vertices are specialized machine configuration generators. By using such graphs the learning machine space may be modeled in a much more flexible way, depending on the characteristics of the problem considered and *a priori* knowledge. The presented method of search space description is used together with an advanced algorithm which orders test tasks according to their complexities.

Keywords: meta-learning, data mining, learning machines, complexity of learning, complexity of learning machines, computational intelligence.

1. Introduction

A major challenge to be faced by computational intelligence clearly regards the nontriviality of model selection (Guyon, 2003; 2006; Guyon *et al.*, 2006; Jankowski and Grąbczewski, 2007). An optimal model usually results from advanced search among learning machines using a number of learning strategies. However, for some dataset benchmarks, e.g., for some of UCI ML Repository benchmarks, simple and accurate (enough) models of a relatively simple structure are known. The conclusion is that before looking for a close-to-optimal model, it is not known how simple the best model will be and how much time will be enough to complete the search process with a satisfactory solution.

One of the approaches to meta-learning develops methods of decision committee construction and various stacking strategies, also performing nontrivial analysis of member models to draw committee conclusions (Chan and Stolfo, 1996; Prodromidis and Chan, 2000; Todorovski and Dzeroski, 2003; Duch and Itert, 2003; Jankowski and Grąbczewski, 2005; Troć and Unold, 2010). Another group of meta-learning enterprises (Pfahringer *et al.*, 2000; Brazdil *et al.*, 2003; Bensusan *et al.*, 2000; Peng *et al.*, 2002) is based on data characterization techniques (characteristics of data like the number of features/vectors/classes, feature variances, information measures on features, also from decision trees, etc.) or on *landmarking* (machines are ranked on the basis of simple machine performances before starting the more power consuming ones) and try to learn the relation between such data descriptions and the accuracy of different learning methods. Duch and Grudziński (1999) present optimization of several addition line metrics or feature selections around the k-Nearest Neighbors (kNN) method. In gating neural networks (Kadlec and Gabrys, 2008), the authors use neural networks to predict the performance of proposed *local experts* (machines preceded by transformations) and decide about the final decision (the best combination learned by regression) of the whole system.

Another application of meta-learning to optimization problems, by building relations between elements which characterize the problem and algorithm performance, can be found in the work of Smith-Miles (2008). Kordík and Černý (2011) combine meta-learning with evolutionary

programming. In another approach (Czarnowski and Jędrzejowicz, 2011), the authors present the usage of a team of agents, which executes several optimization processes by tabu search or simulated annealing. An interesting cooperation based on experience from learning of classifiers in fuzzy-logic approaches can be found in the works of Scherer (2011; 2010), Korytkowski *et al*. (2011) and Łęski (2003). For some other approaches, see the books by Brazdil *et al*. (2009) and Jankowski *et al*. (2011).

Although the projects are really interesting, they still suffer from many limitations and may be extended in a number of ways, especially in composition of the learning machine space which is proposed in this paper.

A *learning problem* can be defined as $\mathcal{P} = \langle D, \mathcal{M} \rangle$, where $D \subseteq \mathcal{D}$ is a *learning dataset* and $\mathcal{M}$ is a *model space*. Then, learning is a function $\mathcal{A}(\mathcal{L})$ of a *learning machine* $\mathcal{L}$:

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

where $\mathcal{K}_{\mathcal{L}}$ represents the space of configuration parameters of a given learning machine $\mathcal{L}$, $\mathcal{D}$ defines the space of data streams (typically, a single data table, sometimes composed of several independent data inputs), which provide the learning material, and $\mathcal{M}$ defines the space of goal models. This means that the *model* is defined as a result of the learning of a given learning machine. Models play different roles (assumed by $\mathcal{L}$), like that of classifier, feature selector, feature extractor, approximator, prototype selector, etc. From a general point of view $\mathcal{M}$, is not limited to any particular kind of algorithms (simple or complex, neural networks or statistical, supervised or unsupervised, etc.).

The *Meta-Learning Algorithm (MLA)* is also a learning algorithm of a machine, although the goal of meta-learning is to find the best way of learning under given conditions. The presented framework for meta-learning search was realized as modules in the Intemi data mining system (Grąbczewski and Jankowski, 2011) (it would be difficult to build it as an element of the Weka (Witten and Frank, 2005), mostly because of the complex approximation framework). Intemi is realized in C#/.Net.

Section 2 describes (in general) the idea of our meta-learning machine. Section 3 presents the main topic of this article which is a functional evolving space of a learning machines used by meta-learning as the decomposition of the learning problem. It is done via special graphs, called *generator flows*. Then, Section 4 briefly presents the way of computing the complexity of learning machines which is used to order test tasks in the meta-learning search loop. Interesting examples of using the described meta-learning are presented in Section 5.

2. Meta-learning algorithm

Equation (1) specifies the processes of (learning) machines. In the case of meta-learning, the learning phase learns how to learn, to learn as well as possible.

A *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. This means that, contrary to the no-free-lunch theorem for non-artificial problems, we may hope that generalization for a given problem $\mathcal{P}$ is possible and our meta-learning may find interesting solutions (such that maximize the defined goal for a given problem $\mathcal{P}$).

However, finding an optimal model for a given data mining problem $\mathcal{P}$ is almost always NP-hard[1]. Because of that, meta-learning algorithms should focus on finding approximations to the optimal solution.

This is why our general goal of meta-learning is *to maximize the probability of finding possibly the best solution within a search space for a given problem $\mathcal{P}$ in a minimal time*.

As a consequence of such a definition of the goal, the construction of meta-learning algorithm should carefully

- produce test-tasks for selected learning machines,
- advise the order of testing the tasks during the progress of the search, and
- build meta-knowledge based on the experience gained from passed tests.

Decomposition of the learning problem. In real life problems, sensible solutions $m \in \mathcal{M}$ are usually complex. Previous meta-learning approaches (mentioned in Introduction) tried to find a final model via selection of one of a number of simple machines or of complex machines but of a fixed structure. This significantly reduces the space of models which could be found as solutions.

Another goal proposed here is based on decomposition of the learning problem $\mathcal{P} = \langle D, \mathcal{M} \rangle$ into subproblems:

$$\mathcal{P} = [\mathcal{P}_1, \ldots, \mathcal{P}_n], \tag{2}$$

where $\mathcal{P}_i = \langle D_i, \mathcal{M}_i \rangle$. In this way, the vector of solutions to the problems $\mathcal{P}_i$ constitutes a model for the main problem $\mathcal{P}$:

$$m = [m_1, \ldots, m_n], \tag{3}$$

and the model space becomes

$$\mathcal{M} = \mathcal{M}_1 \times \cdots \times \mathcal{M}_n. \tag{4}$$

The solution constructed by decomposition is often much easier to find because of reduction of the main task to a series of simpler tasks: model m_i solving the subproblem $\mathcal{P}_i$ is a result of the learning process

$$\mathcal{A}(\mathcal{L}_i) : \mathcal{K}_{\mathcal{L}_i} \times \mathcal{D}_i \rightarrow \mathcal{M}_i, \quad i = 1, \ldots, n, \tag{5}$$

[1]Finding an optimal model does not mean a single learning machine but choosing an optimal model from among all possible ones. For example, the complexity of choosing an optimal subset of features is $O(2^n)$ (n is the number of features).

where

$$\mathcal{D}_i = \prod_{k \in K_i} \mathcal{M}_k, \tag{6}$$

and $K_i \subseteq \{0, 1, \ldots, i-1\}$, $\mathcal{M}_0 = \mathcal{D}$. This means that the learning machine $\mathcal{L}_i$ may take advantage of some of the models $m_1, \ldots, m_{i-1}$ learned by preceding subprocesses and of the original dataset $D \in \mathcal{D}$ of the main problem $\mathcal{P}$.

Thus, the main learning process $\mathcal{L}$ is decomposed to the vector

$$[\mathcal{L}_1, \ldots, \mathcal{L}_n]. \tag{7}$$

Each $\mathcal{L}_i$ may have a particular configuration. Such decomposition is often very natural: standardization or feature selection naturally precedes classification, a number of classifiers precede the committee module, etc.

It is important to see that the decomposition is not a split into *preprocessing* and final (proper) learning. Indeed, we should not talk about preprocessing but about the necessity of data transformation (as an integral part of a complex machine) because not all machines need the same transformation. For example, a classifier committee may contain two classifiers which can learn on continuous data only and on discretized data only. This example clearly shows that there is no single and common preprocessing. Also characteristics of different learning machines differ when used with some filtering transformations, and again such transformations cannot be always shared via many machines in a given decomposition.

Now, the role of *meta-learning* can be seen as searching for possibly the best decomposition in an automated way. In our work, this has been done by *generator flows* used to provide machine configurations. Of course, it may happen that the best decomposition found consists of a single machine, as it is *good* enough. However, even such a solution must be found and validated. The claim must follow a thorough analysis of many machines of different kinds and structures.

To make meta-learning as universal as possible, the items listed below should be included in the configuration of a meta-learning algorithm.

• **Functional search space definition**. A fundamental aspect of every meta-learning is determination of the search space of learning machines and the corresponding final configurations of learning machines which will be considered by meta-learning. In the algorithm presented here, the initial state of the generator flow is this configuration element. Much more details are presented below and in Section 3.

• **Definition of the goal of meta-learning**. Formulation and formalization of the problem $\mathcal{P}$ is another obligatory element to be defined. Definition of the goal has crucial influence on what meta-learning finds. Within our algorithm, the goal is defined by two items:

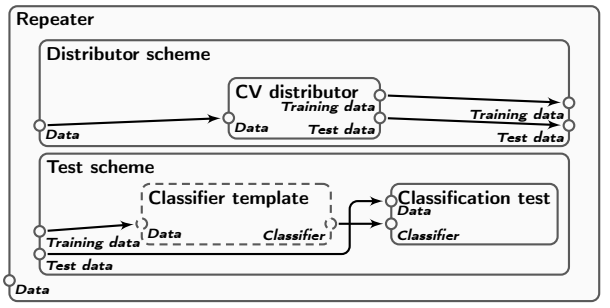

Fig. 1. Template of the test task scenario: to define the goal of learning for a classification problem, a CV test can be defined by a repeater machine with a CV distributor and a test scheme composed of the classifier template and the classifier test module.

– *template of test scenario*: it defines the test procedure in which a given candidate machine will be embedded, learned and verified. See an example in Fig. 1 devoted to classification problems tested by cross-validation. The expectation of this step is to obtain (a series of) test results which are further analyzed to estimate the candidate's eligibility.

– *quality estimation measure*: it is a formal query in a special language that is executed to collect results from selected machines (within the test scenario) and transform the collected series of results into a final quantity measuring quality of the machine embedded within the test task.

• **Stop condition**. "When to stop?" needs an answer. We always have some time limits or expect an appropriate level of quality of the goal machine.

• **Other elements**. In the case of advanced algorithms, some other items may also be included in the configuration. This increases the generality and flexibility of the meta-learning algorithm. An example may be meta-knowledge of different kinds that may improve the search process.

The heart of our meta-learning algorithm is depicted in Fig. 2. The initialization step is a *link* between a given configuration of meta-learning responsible for preparing initial states of several structures like the generator flow or machine ranking. The meta-learning algorithm, after some initialization, starts the main loop, where up to the given *stop condition* new test tasks are prepared and analyzed to *conclude* from their gains.

In each loop repetition, first the algorithm defines and starts a number of test tasks for constructed configurations of learning machines. 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

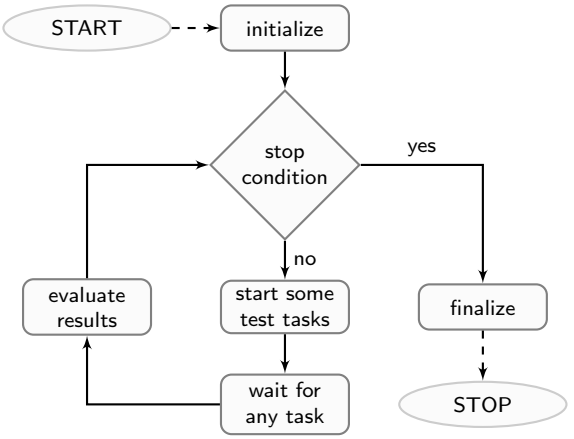

Fig. 2. General meta-learning algorithm.

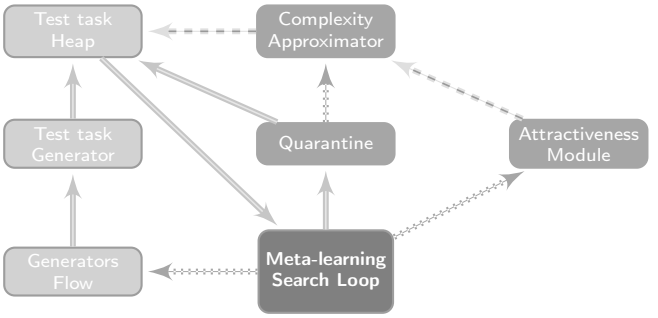

Fig. 3. Relations between modules of meta-learning algorithms.

to some exception (different types of errors or broken by meta-learning because of an exceeded time limit). After a task is finished, its results are analyzed and *evaluated*. In this step some results may be accumulated (for example, saving information about the best machines) and new knowledge items created (e.g., about cooperation between different machines). Such knowledge may have crucial influence on further parts of meta-learning (tasks formulation and the control of the search through the space of learning machines). When the *stop condition* is satisfied, the MLA returns the final result: the configuration of the best learning machine or, in a more general way, a ranking of learning machines (ordered according to obtained test results).

Starting test tasks. Candidate machine configurations are constructed by a graph of machine generators called the *generator flow*. In general, the goal of the generator flow is to provide machine configurations on its output which will be considered potential solutions for a given problem $\mathcal{P}$ represented by given data D. Generator flows should provide a variety of machine configurations of a given type. Such machines may be of a simple or a complex structure and may strongly differ in complexity.

Next, each machine configuration formulated by the generator flow is nested in the *test task template* defined for a given problem $\mathcal{P}$ (for a description of templates, see Section 3.2 or Fig. 1). It is made by a *test task generator* (see Fig. 3, double solid lines denote the test task exchange paths, double dotted lines mean that one module informs another about something, dashed lines mean that one module uses another one). Test task generator uses the test task template, which is a part of the configuration of meta-learning and defines the goal of meta-learning. Each machine configuration is nested in a single test task configuration.

Test task configurations produced by a test task generator are sent directly to the *test task heap*. In the test task heap, tasks are continuously ordered by approximated complexity of machines. The complexity is defined in a way designed for computational intelligence tasks. Except the time and space, its definition may be influenced by the meta-learning process as it will be seen further on. Also Section 4 presents some information about the approximation of machine complexity. Thanks to ordering by complexity, simpler machines are favored and tested before more complex ones.

This feature is crucial because, otherwise, meta-learning would be vulnerable to losing much time without guaranteeing that a given machine process will end. It would be natural if we assumed that each machine configuration provided by the generator flow was equally promising. In the cases when our *a priori* knowledge is *deeper*, it may be transformed in a formal way as a special correction of complexity (see comments on the attractiveness of the machine in Section 2 and a complexity definition in Section 4). If the conditions to start a task (line 3 of the code) are satisfied, then a pair of the machine configuration mc and its corresponding complexity description cmplx is extracted from test task heap (see line 5).

According to the order decided within the heap, the procedure startTasksIfPossible, sketched below, starts the simplest machines first and then more and more complex ones by extracting subsequent test tasks from the heap.

```
1  procedure startTasksIfPossible;
2      while (¬ testTaskHeap.Empty() ∧
3              ¬ mlTaskSpooler.Full()) {
4          <mc, cmplx>
5          = test task heap.ExtractMinimum();
6          timeLimit = τ · cmplx.time / cmplx.q
7          mlTaskSpooler.Add(mc,
8              limiter(timeLimit), priority−−);
9      }
10 end
```

Tasks are taken from test task heap, so when it is empty, no task can be started, but still some tasks may already be running. Additionally, the task-spooler of meta-

learning must not be full if we want to start a new task.

Since we use only an approximation of machine complexity, 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 the convergence of learning). To circumvent 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 6 of the code, the time limit is set up according to predicted time consumption (cmplx.time) of the test task and the current reliability of the machine (cmplx.q and Eqn. (9)). 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 below). Here τ is a constant (in our experiments equal to 2) to protect against too early test task breaking.

Analysis of finished tasks. After starting an appropriate number of tasks, the MLA is waiting for a task to finish. A task may finish normally (including termination by an exception) or be halted by a time-limiter (because of exceeding the time limit).

```
11 procedure analyzeFinishedTasks;
12     foreach (t in mlTaskSpooler.finishedTasks) {
13         mc = t.configuration;
14         if (t.status = finished_normally) {
15             qt = computeQualityTest(t,
16               queryDefinition);
17             machinesRanking.Add(qt, mc);
18             machineGeneratorsFlow.Analyze(t, qt,
19                     machinesRanking);
20         } else { // task broken by limiter
21             mc.cmplx.q = mc.cmplx.q / 4;
22             testTaskHeap.Quarantine(mc);
23         }
24         mlTaskSpooler.RemoveTask(t);
25     }
26 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 (which is a part of the meta-learning configuration) is computed based on the test task results (see line 16) 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 the quality test qt and the machine configuration mc.

Next, the generator flow is called (line 19) to analyze the new results (it can be seen in Fig. 3, too). The flow passes the call to each internal generator to let the whole hierarchy analyze the results. Influenced machine generators inside the generator flow may provide new machine configurations or at least change meta-knowledge.

When a task is halted by a time-limiter, it is moved to the *quarantine* for a period not counted in time directly but determined by the complexities (see again Fig. 3). Instead of constructing a separate structure responsible for the functionality of a quarantine, the quarantine is realized by two naturally cooperating elements: the test task heap and the reliability term of the complexity formula (see Eqn. (9)). First, the reliability of the test task (its quality of complexity approximation) is corrected (see code line 21), and after that the test task is resent to the test task heap as to the quarantine (line 22). This means that such a task will be started again if the meta-learning algorithm starts running tasks of such (corrected) complexity.

3. Functional form of the MLA search space

In the simplest way, the machine space browsed by meta-learning may be restricted to a set of machine configurations. So far, meta-learning projects have concentrated on selection of algorithms from a fixed set of parameters of known learning machines. The most important limitation of such approaches is that their MLAs cannot autonomously find appropriate data transformations before application of final decision machines, which is almost always necessary in real applications.

An interesting solution to overcome these limitations is decomposition of the learning problem (Eqn.(2)) as presented at the beginning of Section 2. In general, such decomposition is NP-hard and cannot be solved in a simple way. Additionally, it cannot be solved in a direct analytical way. But this does not mean that we cannot do much more than the previous meta-learning projects.

Finding attractive decompositions can be done quite naturally when using knowledge about the usefulness of particular computational intelligence algorithms in different contexts. This helps in construction of reasonable classifiers, approximators, combined with data transformations, etc. and augmented by complexity control may constitute very powerful meta-search algorithms. Based on meta-knowledge and attainable prior information about the problem, it is possible to construct functional form of the search space for meta-learning algorithm. We define such search spaces in the form of a *graph* designed to provide a series of learning machine configurations for building test tasks for further browsing by the main part of the meta-learning algorithm presented in the previous section.

The graph is composed of *Machine Configuration Generators* (MCGs) as nodes. The main goal of each MCG is also to provide machine configurations through its output. But each MCG is free to do it in its own way—there are MCGs of different types and new ones may be developed and extend the system at any time. MCGs may obtain on their inputs (input edges) a series of machine configurations from other MCGs. The graph (called the

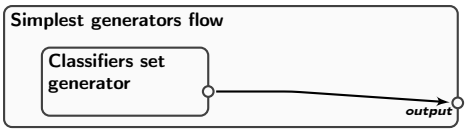

Fig. 4. Example of the simplest generator flow.

generator flow) is directed and acyclic[2]. Each MCG may use (different) meta-knowledge, and reveal different behavior, which in particular may even change in time (during the MLA run). By adding generators (with proper connections where adequate) we unfold the meta-learning space in proper directions and can control the unfolding actively during the meta-learning process.

The decomposition is performed by the graph in a *natural* way by means of connections between its nodes. This will be seen soon in examples. Connections facilitate the exchange of machine configurations between nodes, which can be appropriately combined within the nodes. With such a concept, simple and complex machine configurations can be built easily with or without deep knowledge about particular MCGs.

Some of the outputs of generators in the graph can be bound to the output of the generator flow—the actual provider of configurations to meta-learning. In the run time of the meta-learning algorithm, the configurations returned by the generator flow are nested in the test task and transported to the machine heap before the MLA starts tests of selected (by ordering) configurations.

The simplest possible generator flow is presented in Fig. 4. It contains a single MCG with no inputs. The output of the flow is exactly that of the only MCG (it is realized by the connection between the MCG output and the flow output).

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 (each generator may have a configuration, similarly to learning machines). This means no influence of the process of the MLA on the output of the (fixed) MCG. Non-fixed generator outputs depend also on the learning progress (see the advanced generators below).

When a machine configuration is provided by a generator, information about the *origin* of the 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 how the configuration was created—the *production line* can be restored and analyzed to gather more meta-knowledge.

The only assumption about generators behavior is that they provide finite series of configurations. As can be seen in line 19 of the code in Section 2, the generator flow receives information about the progress in the meta-search process. Such information is propagated to each of the MCG. This means that each MCG has access to the current state of the MLA and their further behavior may depend on that process. More and more sophisticated machine generators can optimize time of searching for the most attractive solutions. Below, we describe examples of machine configuration generators and examples of generator flows.

3.1. Set-based generators. The simplest but very useful type of machine generator is aimed at providing just an arbitrary sequence of chosen machine configurations (thus the name *set-based generator*). Usually, it is convenient to have a few set-based generators in a single generator flow, each devoted to machines of different functionality, for example,

- set-based generator of simple classifiers,
- set-based generator of classifiers,
- set-based generator of approximators,
- set-based generator of feature ranking,
- set-based generator of decision trees,
- set-based generator of prototype selectors,
- set-based generator of committees,
- set-based generator of base data transformers,
- set-based generator of data filters (outlayers, redundant attributes, etc.).

An example of a generator flow with a set-based generator providing classifier machine configurations is presented in Fig. 4.

Sometimes it is even more attractive to disjoin some groups of methods because of their specific needs. For example, when different machines expect data with different types of features, they may be grouped according to their preferences. For example, in the case of feature ranking methods, we may need the following three generators:

- set-based generator of feature ranking (for discretized/symbolic data),
- set-based generator of feature ranking (for continuous data),
- set-based generator of feature ranking (for any data).

Thanks to the split, it is easy to precede the groups of methods by appropriate data transformation—an example is provided in Fig. 9. Some other examples of using set-based generators will also be shown below.

[2]Cycles would result in an infinite number of configurations provided by the graph.

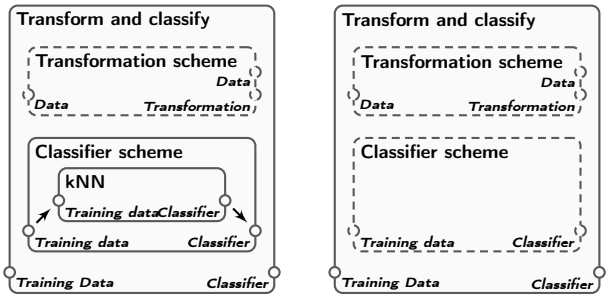

Fig. 5. Transform and classify machine configuration template: placeholder for the transformer and the fixed classifier (kNN) (left), two placeholders—one for the transformer and one for the classifier (right).

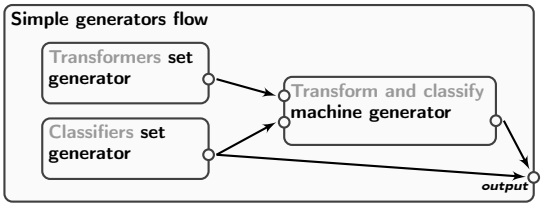

Fig. 6. Simple generator flow.

3.2. Template-based generators. Template-based generators are used to provide complex configurations based on machine configuration templates and machine configurations generators providing items to replace template placeholders. A machine configuration with an empty placeholder (or placeholders) as a child machine configuration is called a *machine template* (or, more precisely, a *machine configuration template*). Each placeholder in a given template can be filled with a machine configuration or with a hierarchy (DAG graph) of machines configurations. For example, if a meta-learner is to search for combinations of different data transformers and the kNN as a classifier, it can easily do it with a template-based generator. The combinations may be defined by a machine template with a placeholder for the data transformer and a fixed kNN classifier. Such a template-based generator may be connected to a set-based generator providing data transformations. As a result, the template-based generator will 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 (TnC) machine template shown in Fig. 5 on the left. The TnC machine learning process starts with learning data transformation and then runs 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 that of placeholders. The role of a placeholder is always defined by its inputs and outputs declarations. Thus, it may be a classifier, approximator, data transformer, ranking, committee, etc. Of course, the placeholder may be filled with a complex machine configuration as well as a simple one.

Replacing the kNN from the previous example by a classifier placeholder (cf. Fig. 5, 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.

In general, the template-based generator is defined by

- a template with one or more placeholders,
- a series of paths to placeholders, and
- the operation mode.

The order within the series of paths to placeholders defines the order of inputs of the generator. The main goal of a template-based generator can be seen as production of series of machine configurations by filling the template with appropriate machine configurations from inputs according to the series of the path. Similarly, the role of each machine configuration produced by this generator is always the same as the role of template.

The template-based generator can operate in one of two modes: *one-to-one* or *all-to-all*. In the case of the example considered above, the 'one-to-one' 'mode makes the template-based generator obtain' 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 the '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. Naturally, the mode affects the operation of multi-input generators only.

Figure 6 presents the discussed example of using two set-based generators and one template-based generator. The set-based generators provide transformers and classifiers (respectively) to the two 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 on the output, depending on the mode of the generator: one-to-one or all-to-all. Please note that the generator flow's output obtains configuration from both the set-based classifiers generator and the template-based generator, so it will provide both the classifiers taken directly from the declared set and the classifiers preceded by the declared transformers.

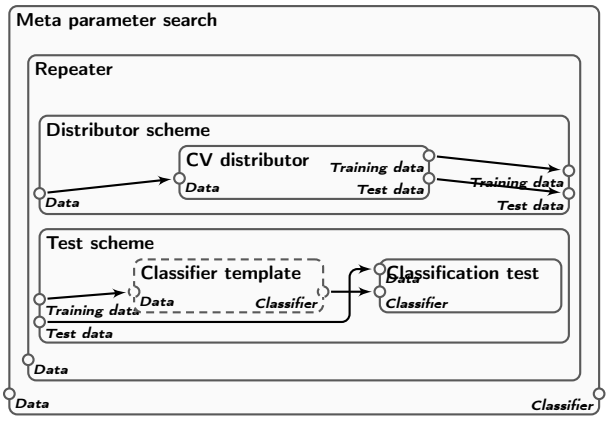

Fig. 7. Template of the meta parameter search machine with a classifier placeholder and a classifier in the role of the parameter search machine.

Another interesting example of a template-based generator is using a template of the Meta-Parameter Search (MPS) machine configuration. The MPS machine can optimize any elements of machine configuration[3]. The search and optimization strategies are realized as separate modules which carry appropriate functionality. Because of that, search strategies are ready to provide optimization of any kind of elements (of configurations). Even abstract (amorphic) structures can be optimized. Such structures may be completely unknown for MPS, but a given search strategy knows what to do with objects of a given structure. The template containing a test scheme with a placeholder for a classifier is very useful here (see Fig. 7). Note that the placeholder of that template plays the role of the classifier, and finally the MPS machine will also play that role. Configuring the MPS machine to use the *auto-scenario* option makes the meta-learner receive configurations of MPS to realize the auto-scenario[4] for a given classifier. This means that such a generator will provide configurations for selected classifiers to auto-optimize their parameters.

A more complex generator flow using template-based generator with the MPS machine is presented in Fig. 8. This generator flow contains three set-based generators, which provide transformers, rankings and classifiers configurations to other (template-based) generators. Please note that the classifier generator sends its output configurations directly to the generator flow output and additionally to three template-based generators: two combining transformations with classifiers and an MPS generator. This means that each classifier configuration will be sent to the meta-learning heap and additionally to other

[3]It is possible to optimize several parameters during optimization, sequentially or in parallel, depending on the search strategy used.

[4]Auto-scenario is realized thanks to the ontology of optimization procedures for each machine.

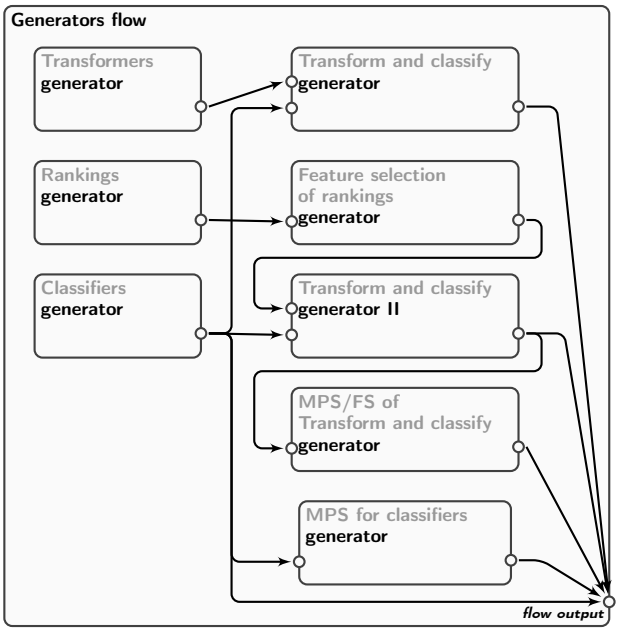

Fig. 8. Example of a generator flow.

generators to be nested in other configurations (generated by the Transform and classify and MPS generators).

The two Transform and classify generators combine different transformations with the classifiers obtained from 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 Transformers generator while the second one receives configurations from another template-based generator and generates feature selection configurations with the use of different ranking algorithms received through the output-input connection with Rankings generator. The combinations generated by the two Transform and classify generators are also sent to the output of the generator flow.

Additionally, Transform and classify generator II sends its output configurations to MPS/FS of Transform and classify generator. This generator produces MPS configurations, where the number of features is optimized for configurations produced by Transform and classify generator II. The output of the MPS generator is passed to the output of the generator flow, too.

In such a scheme, a variety of configurations is 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 generators of any kind. As discussed above, it is often fruitful to separate groups of classifiers (data transformations, etc.) into independent

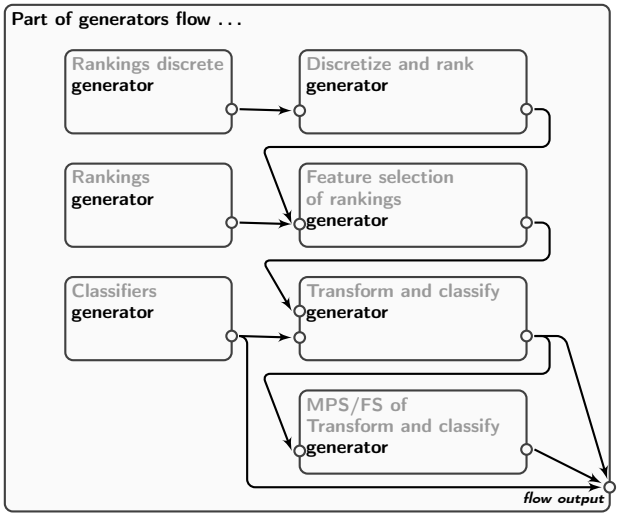

Fig. 9. Part of the generator flow with two separate ranking generators to reflect different roles and different application contexts.

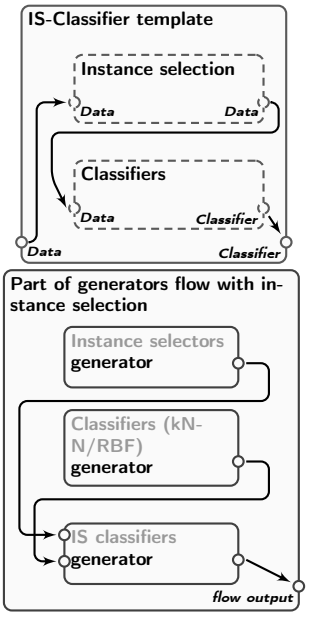

Fig. 10. Part of the generator flow providing combinations of instance selection methods with classifiers trained on selected data instances.

set-based generators to reflect different application contexts of the machines. Such a case can be seen in Fig. 9, where, for more flexibility, feature ranking methods are separated into those operating on discrete features (to be preceded by a discretization transformation, e.g., information theory based rankings) and others, which may be used without discretization. The generator of ranking machines which need discretized data sends its configurations to another generator which nests configurations ranking in a template, where the ranking machine is preceded by a discretization transformer. The feature selection generator has two inputs and all algorithms finish within better data propagation scenarios, as natural as it is.

The generator flow may be easily extended by adding proper template-based generators. One useful extension creates machine configurations exploiting instance selection. The instance selection algorithms (Jankowski and Grochowski, 2005; 2004) are used for several reasons: to filter out noise instances, to shrink the dataset or to find prototype instances (as a prototype-based explanation of the problem considered). The most typical way of using instance selection algorithms is to combine them with classifiers like kNN or RBF. To do it with a machine generator we just need a proper template as presented in Fig. 10. The template contains two placeholders: one for an instance selection algorithm and the other for a classifier. The generator based on this template needs two inputs: one with configurations of instance selection algorithms and the other with classifiers. The required generators and their bindings are presented in Fig. 10 (bottom). The IS classifiers generator provides combinations of instance selection methods with classifiers. Note that the *combinations* are so simple because of using appropriately composed templates in machine generators. Besides the definition of the template, it is sufficient to provide proper connections between generators reflecting the roles played by particular machine components.

Committees can be composed using template-based generators in a similar way. In this case we use another flexible feature of generators input–output connections: the possibility of propagation of not only series of machine configurations but also series of sequences of machine configurations. Let us consider committee templates as in Fig. 11. The top one has a placeholder for a sequence of classifiers (not a single classifier). The bottom one has an additional placeholder for a committee decision module, so that committees can be composed of different sequences of classifiers and different decision modules (voting, weighting, etc.). It can be incorporated into a generator flow as in Fig. 12.

3.3. Advanced generators. The advantage of advanced generators over the generators mentioned above is that they can make use of additional meta-knowledge and can observe the progress of meta-learning to actively provide configurations and exhibit their own strategy. Meta-knowledge, including 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 a given classifier needs only continuous or discrete features.

Advanced generators are informed each time a test task is finished and analyzed. The generators may access

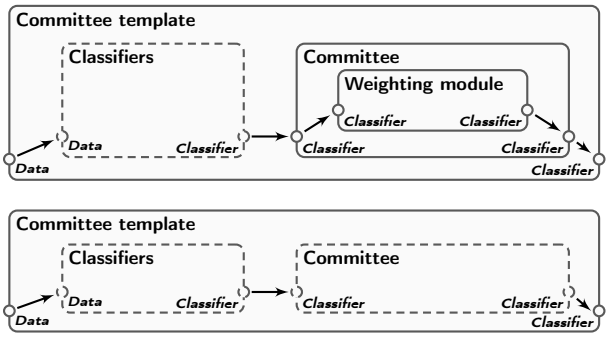

Fig. 11. Templates of committees.

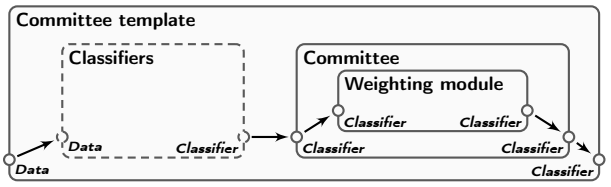

Fig. 12. Part of the generator flow devoted to construction of committees.

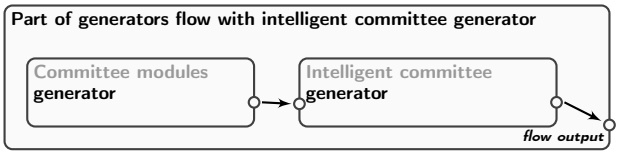

Fig. 13. Part of the generator flow with an intelligent committee generator.

the results computed within the test task and the current machine configuration ranking (compare code line 19 in Section 2). The strategy enables providing machine configurations derived from observations 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 the meta-learning algorithm cannot (and should not) be responsible for specific aspects of different kinds of machines or other sort of *local* knowledge. This feature is very welcome because, as everybody believes, there is no one universal knowledge that could advise all known (and even not known) learning algorithms. Generator modules should be independent of the main part of the algorithm and should cooperate with the MLA and other generators. On the other hand, the MLA should not be responsible for all aspects of each learning machine; meta-learning cannot be "aware" even about all types of learning machines.

Another worthwhile example of an advanced generator is the *intelligent committee generator* presented in Fig. 13. It continuously observes the progress of meta-learning, especially changes in the ranking of machines. On the basis of this information, it builds new configurations of (promising) committees. New committees are built only under restricted conditions—when we expect it is rational to do it. Shortly, it is based on the progress in ranking and on the diversity of committee members which are selected from among models of known quality and results that can be compared with the McNemmar test. It is really simple to realize it in our Intemi data mining system (Grąbczewski and Jankowski, 2011). Additionally, even if several committee decision modules are tested, the real time of computations does not grow with the factor of the number of modules because our system uses a specialized machine cache so that committee members are computed once and never recomputed—they are reused when appropriate.

Simple meta-knowledge about the influence of data transformation methods on further learning may be the foundation of another advanced generator: the *filter generator*. It may provide filter transformations depending on not so complex analysis of the dataset representing the problem. For example, in the cases of very large data (huge number of instances or features) or too large for a particular machine (quite complex classifier or feature selector), the generator can provide methods to select instances or features depending on the characteristic of the data. In a similar way, some other dataset transformation may be output by the filter provider, for example, pruning non-informative features, redundant features, outliers, etc. The advantage of such solution lies in the fact that filtering methods are not provided obligatorily but optionally after shallow or deeper analysis.

Such a generator brilliantly fits the idea of complexity based control of tasks order in the heart of the meta-learning algorithm. For example, in the above-mentioned case of too huge data, the filter generator provides a machine for reduction of data complexity, which, used prior other machines may yield significant reduction of the complexity of the whole complex configuration. This way, the task of too huge computational time requirements is reduced to a computable task, while the solution may still be attractive from the point of view of the original problem.

Summing the main features of the generator flow:

- The generator flow is a direct acyclic graph which may be composed of many machine generators. Each generator has zero or more inputs which are used to receive machine configurations from other generators.

- The generator flow should reflect the functional domain of a given problem. In other words, the search

space should be as adequate for a given problem as possible. This may be done in a very natural way by composing the relation between adequately chosen machine generators and their connections.

- The generator flow is informed by the MLA about the progress in meta-learning. This information is propagated to each generator and can be used to control the operation of the generator. The results of performed tests are valuable meta-knowledge that can be collected, analyzed and used to construct a new machine configuration and then passed to the machine heap for testing. In consequence, the product of the generator flow depends not only on its configuration but also on the progress of meta-learning.

- The generator graph may also be changed (nodes and their connection, which in configuration are initialized) during meta-learning.

- The meta-learning algorithm may be continuously extended by a new version of more and more sophisticated machine generators. This is, currently, our main goal.

It is important to see that each type of modules (generators) has its own strategy and can be used as an element of an arbitrarily huge generator flow. The graph defines the space of meta-learning search. Intelligent behavior of machine configuration generators within a modular generator flow facilitates creation of more and more intelligent learning algorithms. The meta-learning algorithm depicted in Fig. 2 may get more and more advanced by extending its generator flow with more and more advanced generators. In consequence, the algorithm will gather more meta-knowledge and exhibit more general artificial intelligence contrary to many known algorithms of AI or CI.

4. Machine complexity evaluation

A detailed description of computation complexity is beyond the scope of this paper. We mostly focus on search space modeling and its relations with other parts of a large meta-learning environment. Only the most important aspects of complexity control are provided below.

To obtain the right order of learning machines within the search queue, a complexity measure should be used. We use the following definition of complexity:

$$c_a(p) = l_p + t_p/\log t_p. \tag{8}$$

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. Because of this approximation and because of the halting problem (we never know whether a given test task ends), an additional penalty term is introduced to the above definition:

$$c_b(p) = [l_p + t_p/\log t_p]/q(p), \tag{9}$$

where $q(p)$ is a function term responsible for reflecting an estimate of reliability of p. At the beginning, the MLAs use $q(p) = 1$ (generally $q(p) \leq 1$) in the estimation, but in the case when the estimated time (as a part of the complexity) is not enough to finish the program p (a 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 (cf. comments in Section 2). This mechanism prevents the running of test tasks for an unpredictably long time or even an infinite time. Otherwise the MLA would be very brittle and susceptible to running tasks consuming unlimited CPU resources.

The test task heap uses the complexity of the machine of a given configuration as the priority key. It is not accidental that the machine configuration which comes to the test task heap is the configuration of the whole machine test (where the proposed machine configuration is nested). This complexity really well reflects the complete behavior of the machine: a part of the complexity formula reflects the complexity of the learning of a given machine and the rest reflects the complexity of computing the test (for example, a classification or an approximation test).

Because complexity depends on the configuration and inputs, the complexity computation must reflect the information from the configuration and inputs. The recursive nature of the configuration, together with input–output connections, may compose quite a complex information flow. Sometimes, the inputs of submachines become known just before they are started, i.e., after the learning of other machines is finished (machines that provide necessary outputs). This is one of the most important reasons why determination of complexity, contrary to actual learning processes, must be based on *meta-inputs*, not on exact inputs (which remain unknown).

To facilitate recurrent determination of complexity, the functions which compute complexity must also provide meta-outputs, because such meta-outputs will play a crucial role in computation of complexities of machines which read the outputs through their inputs.

In conclusion, a function computing the complexity for machine $\mathcal{L}$ is a transformation in the form

$$\mathcal{D}_{\mathcal{L}} \; : \; \mathcal{K}_{\mathcal{L}} \times \mathcal{M}_{+} \rightarrow \mathbb{R}^2 \times \mathcal{M}_{+}, \tag{10}$$

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 time complexity, memory complexity and appropriate meta-outputs. Please refer to the definition of learning (Eqn. (1)), because computation of complexity is a derivative of the behavior of the machine learning process.

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

- to evaluate and exhibit *appropriate aspects* of complexity representation based on some meta-descriptions like meta-inputs or configuration[5];
- to exhibit a functional description of complexity aspects (comments) useful for further reuse by other meta evaluators[6].

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

Evaluators are almost always constructed with the help of a series of approximators. The number of approximators per evaluator depends on the number of functionalities which has to be provided by a given evaluator. Typically, each machine evaluator has approximators for learning time and space consumption approximation and others depend on the type of evaluator.

Construction of learnable evaluators. We have defined a common framework for building approximators for evaluators to simplify the process of complexity approximation. The framework is defined in a very general way and enables building evaluators using dedicated approximators for different aspects of complexity like learning time, the size of the model, classification time, transformation time, etc.

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 time or memory than another task.

Figure 14 presents the general idea of creating approximators for an evaluator. To collect 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, and an appropriate test scheme is created and observed (compare the data collection loop in the figure). Observations bring the subsequent instances (vectors) of the training data (corresponding to the 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 a specialized scenario (cf. Section 3.2),

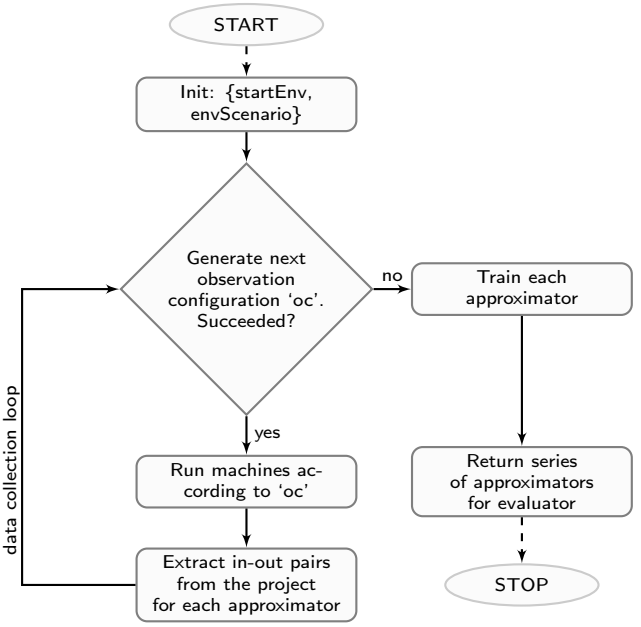

Fig. 14. Process of building approximators for a single evaluator.

which defines how to modify the environment to get a sequence of machine observation configurations, i.e., configurations of the machine being examined nested in a more complex machine structure. The generated machine observation configurations should be as realistic as possible—the information flow similar to expected applications of the machine allows us to better approximate the desired complexity functions. For each observation configuration 'oc', machines are created and run according to 'oc', and when the whole project is ready, proper learning data items are collected.

When the data collection loop fails to generate new machine observation configurations, the data collection stage is finished. Next, each approximator can be trained from the collected data. After that, the evaluator may use the approximators for complexity prediction. Note that the learning processes of evaluators are conducted without any advice from the user.When the process of building approximators for a given evaluator is finished, the evaluator is deposited in a repository of evaluators. Then, meta-learning can use the evaluator to approximate quantities concerning complexity of corresponding learning machines.

5. Examples of application

In this section we show how the generator flows defined in Section 3 expand the search space for the meta-learning algorithm. Note that the goal here is not to present how optimal accuracies can be achieved, but how generator flows unfold the search space.

[5]In the case of a machine, to exhibit the complexity of time and memory.

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

Machine configuration notation. To present complex machine configurations in a compact way, special notation is introduced that allows sketching complex configurations of machines inline as a single term. The notation does not present the input–output interconnections, so it does not allow reconstructing the full scenario in detail but shows a simplified machine structure by presenting a single configuration via its hierarchy.

With square brackets we denote the 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 names), the machines are placed within a scheme machine. Parentheses embrace significant parts of machine configurations. For example, the following text:

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

denotes a complex machine, where a feature selection submachine (FeatureSelection) selects features from the top of a correlation coefficient based ranking (RankingCC), and next, the dataset composed of the feature selection is an input for a kNN with an Euclidean metric—the combination of feature selection and the kNN classifier is controlled by a TransformAndClassify machine. Similar 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 an MPS (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. In a machine denoted as

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

only the number of chosen features is optimized because this configuration is provided by MPS/FS of Transform and classify generator (see Fig. 15), where the ParamSearch configuration is set up to optimize only the parameters of the 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 become too complex for the assumed time limit.

5.1. Meta-learning configuration. The most important elements of our meta-learning algorithm configuration are the meta-learning test template, the query test, the stop criterion and the generator flow.

Meta-learning test template. The test template must be adequate to the goal of learning. 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 the distributor set up to the CV-distributor and the inner test scheme containing a placeholder for the 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 presented in Fig. 1. When used as the MLA test template, it will be repeatedly converted to different feasible configurations by replacing the classifier placeholder inside the template with classifier configurations generated by the generator flow.

Query test. The second part which defines the goal of the problem is the query test used to calculate the quality of tested configurations based on results obtained from a series of test templates. To test 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. The stop criterion was defined to become true when all the configurations provided by the generator flow are tested.

5.2. Configuration of the generator flow and its consequences for the search space of the MLA. The generator 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, a more complex generator graph should be used. Anyway, it will be seen that using even such a basic generator flow, the results ranked high by the MLA can be very good. The generator flow used in our experiments is presented in Fig. 15. A very similar generator flow was explained in detail in Section 3.

To know what exactly will be generated by this generator 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 the Euclidean metric,

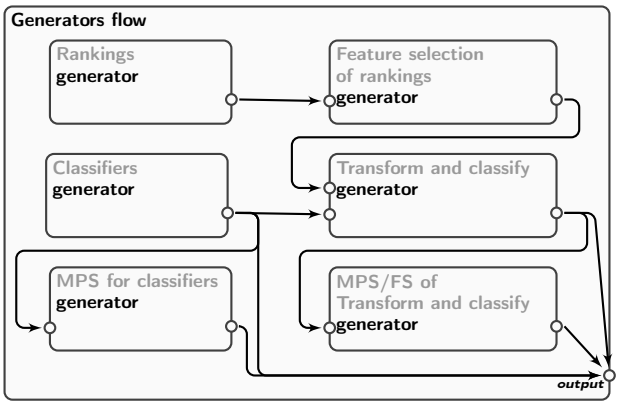

Fig. 15. Generator flow used in tests.

– kNN [MetricMachine (EuclideanOUO)]: kNN with the Euclidean metric for ordered features and the Hamming metric for unordered ones,

– kNN [MetricMachine (Mahalanobis)]: kNN with the Mahalanobis metric,

– NBC: naive Bayes classifier,

– SVMClassifier: support vector machine with a Gaussian kernel,

– LinearSVMClassifier: SVM with linear kernel,

– [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]]: first, the – ExpectedClass[7] machine transforms the original dataset, then the transformed data become the learning data for kNN,

– [LVQ, kNN (Euclidean)]: first, the learning vector quantization algorithm (Kohonen, 1986) 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.

[7]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 a vector of the means (for ordered features) or expected values (for unordered features) for a 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.

The base classifiers and ranking algorithms, together with the generator flow presented in Fig. 15, produce 54 configurations that are nested (one by one) within the meta-learning test-scheme and sent to the meta-learning heap for a complexity controlled run. All the configurations provided by the generator flow are presented in Table 1.

Depending on the changes in the generator flow, the sequence of machine configurations may change significantly. Assume that the generator flow is defined by the graph presented in Fig. 16. Please note that the difference between this graph and the one in Fig. 15 is two additional generators: IT based ranking generator and Discretize and rank generator. Additionally, assume that the IT based ranking generator is based on the set of two configurations of machines for ranking features on the basis of information theory measures (see the work of Duch *et al.* (2004) for details about the algorithms):

- entropy based ranking,
- Mantaras distance based ranking.

The sequence of machine configurations output by such generator flow is the one presented in Table 1, extended by $2{\cdot}2{\cdot}9 = 36$ items including rankings based on information theory. The additional rows can be easily determined by converting all the items of the form

[* RankingCC *]

into the corresponding items of the form

[* [EntropyRank], DiscretizeAndRank *]

and

[* [MantarasRank], DiscretizeAndRank *] .

For example, in analogy to

[[[RankingCC], FeatureSelection], [NBC], TransformAndClassify],

we get two additional feature selections for NBC:

[[[EntropyRank], DiscretizeAndRank], FeatureSelection], [NBC], TransformAndClassify]

and

[[[MantarasRank], DiscretizeAndRank], FeatureSelection], [NBC], TransformAndClassify].

The full collection of 90 configurations will be analyzed by the MLA: the tests will be ordered by approximated complexity and run in appropriate order.

5.3. Benchmark results analysis. Because complexity analysis is not the main thread of this article, we have presented examples on two datasets, "vowel" and "image", selected from the UCI machine learning repository (Frank and Asuncion, 2010).

The results obtained for the benchmarks are presented in the form of diagrams. The diagrams are very

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]

Table 1. Machine configurations produced by the generator flow of Fig. 15 and the enumerated sets of classifiers and rankings.

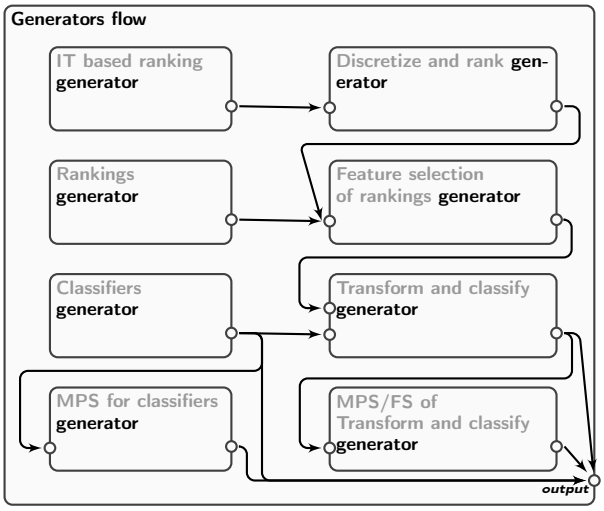

Fig. 16. Extended generator flow.

specific and present many properties of the meta-learning algorithm. They show information about the times of starting, stopping and breaking of each task, the complexities (global, time and memory) of each test task, the order of the test tasks (according to their complexities, cf. Table 1) and the accuracy of each tested machine.

In the middle of the diagram—see the first diagram in Fig. 17—there is a column with task IDs (the same IDs as in Table 1). But the order of rows in the diagram reflects the complexities of test tasks. This means that the most complex tasks are placed at the top and the task of the smallest complexities is visualized at the bottom. Machine complexity is approximated in the context of particular input data, so the order of the same set of machine configurations may be quite different in different applications. For example, in Fig. 17, at the bottom, we can see the task IDs 4 and 31, which correspond to the naive Bayes classifier and the ParamSearch [NBC] classifier. At the top, We can see the task IDs 54 and 45 as the most complex ParamSearch test tasks of this benchmark. The task order in the second example (Fig. 18) is completely different.

On the right-hand side of the *Task id* column, there is a plot presenting the starting, stopping and breaking times of each test task. As was presented in Section 2, the tasks are started according to the approximation of their complexities, and when a given task does not reach the time limit (which corresponds to the time complexity—see Section 4), it finishes normally, otherwise, the task gets broken and restarted according to the modified complexity (see Section 4). For an example of a restarted task, please see Fig. 17, 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. Two different diagrams (in Figs. 17 and 18) easily bring the conclusion that the amount of inaccurately predicted time complexity is quite small (there are very 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 a $[0, 1]$ interval 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 cannot see any two bars overlapping in time. If we ran the projects on more than one CPU, a number of bars would be "active" 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 greater predicted complexity), the longer bars can be seen. This confirms the adequacy of the complexity estimation framework because the relations between the predictions correspond very well to the relations between real time consumed by the tasks. When browsing other diagrams, similar behavior can be observed—the simple tasks are started at the beginning and then, the more and more complex ones are run.

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

The leftmost column of the diagram presents ranks of the test tasks (the ranking of the accuracies). In the case of vowel data, the machine of the best performance is the kNN machine with default parameters (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 the Mahalanobis metric, which is a more complex task.

Between the columns with task IDs and the accuracy-ranks, on top of the gray bars corresponding to the accuracies some thin solid lines can be seen. The lines start on the right-hand side (just like the accuracy bars) and go to the right according to proper magnitudes. For each task, the three lines correspond to total complexity (the upper line), memory complexity (the middle line) and time com-

plexity (the lower line)[8]. All three complexities are approximated complexities (see Eqn. (9)). Approximated complexities presented on the left-hand side of the diagram can be easily compared visually to the time-schedule obtained in real time on the right-hand side of the diagram. Longer lines mean higher complexities. It can be seen that sometimes the time complexity of a task is smaller while the total complexity is greater and vice versa. For example, see tasks 42 and 48 again in Fig. 17.

The meta-learning illustration diagrams clearly show that the behavior of different machines changes between benchmarks. Even the standard deviation of accuracies is highly diverse. Simple solutions are started before the complex ones, to support finding simple and accurate solutions as soon as possible. For the presented benchmark vowel, very simple and accurate models were found quite early—close to the beginning of the meta-learning process (see Fig. 17, task IDs 1 and 2), while for benchmark image (Fig. 18) there are no solutions with such a high accuracy found first. The best one is found as one of most complex machines: see the task ID 32, which is tuned by MPS with the SVM with a Gaussian kernel (for ionosphere-ALL), and the best one for the image was the task IDs 28 and 29, which are tuned by MPS with the kNN machine with the Euclidean and Euclidean/Hamming distance metric.

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 a deeper parameter search.

6. Summary

To efficiently solve different problems with algorithms around computational intelligence, we need a really flexible higher order algorithm (meta-learning) that can make use of different algorithms (meta-learning can), also by analyzing them at the meta level. Without advanced meta-learning algorithms, the space of problems that can be satisfactorily solved dramatically shrinks.

In previous approaches to meta-learning, researches used only "flat" spaces (just small sets of machines were tested and compared). The framework for meta-learning search presented here, featuring a very flexible way of defining the functional search space of meta-learning, can explore broad spectra of machines and may be used for very different kinds of problems. It mimics human expert behavior in searching for an interesting decomposition of learning machines with the advantage of general meta-knowledge about reasonable combinations of machine components and specific knowledge gathered during the search (about the fitness of different methods to the problem being solved). With this MLA, browsing through complex machine structures is as easy as testing several simple configurations. The MLA can be easily extended with new machine generators specializing in particular types of machine configurations. Moreover, thanks to advanced machine generators, the search space may change during learning.

Advanced generators give the possibilities of providing intelligent behavior at different levels of abstraction. This will give outstanding possibilities in future.

In addition to the flexible search space, our approach uses complexity control for test task ordering. This feature helps the MLA perform test tasks in the most reasonable order—first the simplest learning machines are tested and then more and more complex ones are tried. Such an approach increases the probability of finding solutions of an attractive balance between machine simplicity and accuracy. It is important to see that the MLA provides approximation of a simple and a complex machine configuration in the same way.

The next step toward more sophisticated meta-learning is to design more and more advanced machine configuration generators, able to collect different kinds of knowledge and use it in a suitable way. The independence of machine generators gives opportunity to spread the responsibility of knowledge distribution. There is no need to have a single machine generator which "knows everything". Machine generators can specialize in a chosen type of behavior/subproblems.

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

References

Bensusan, H., Giraud-Carrier, C. and Kennedy, C.J. (2000). A higher-order approach to meta-learning, *in* J. Cussens and A. Frisch (Eds.), *Proceedings of the Work-in-Progress Track at the 10th International Conference on Inductive Logic Programming*, Springer-Verlag, Berlin/Heidelberg, pp. 33–42.

Brazdil, P., Giraud-Carrier, C., Soares, C. and Vilalta, R. (2009). *Metalearning: Applications to Data Mining*, Springer, Berlin/Heidelberg.

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

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

Czarnowski, I. and Jędrzejowicz, P. (2011). Application of agent-based simulated annealing and tabu search procedures to solving the data reduction problem, *International Journal of Applied Mathematics and Computer Science* **21**(1): 57–68, DOI: 10.2478/v10006-011-0004-3.

Duch, W. and Grudziński, K. (1999). Search and global minimization in similarity-based methods, *International Joint Conference on Neural Networks, Washington, DC, USA*, p. 742.

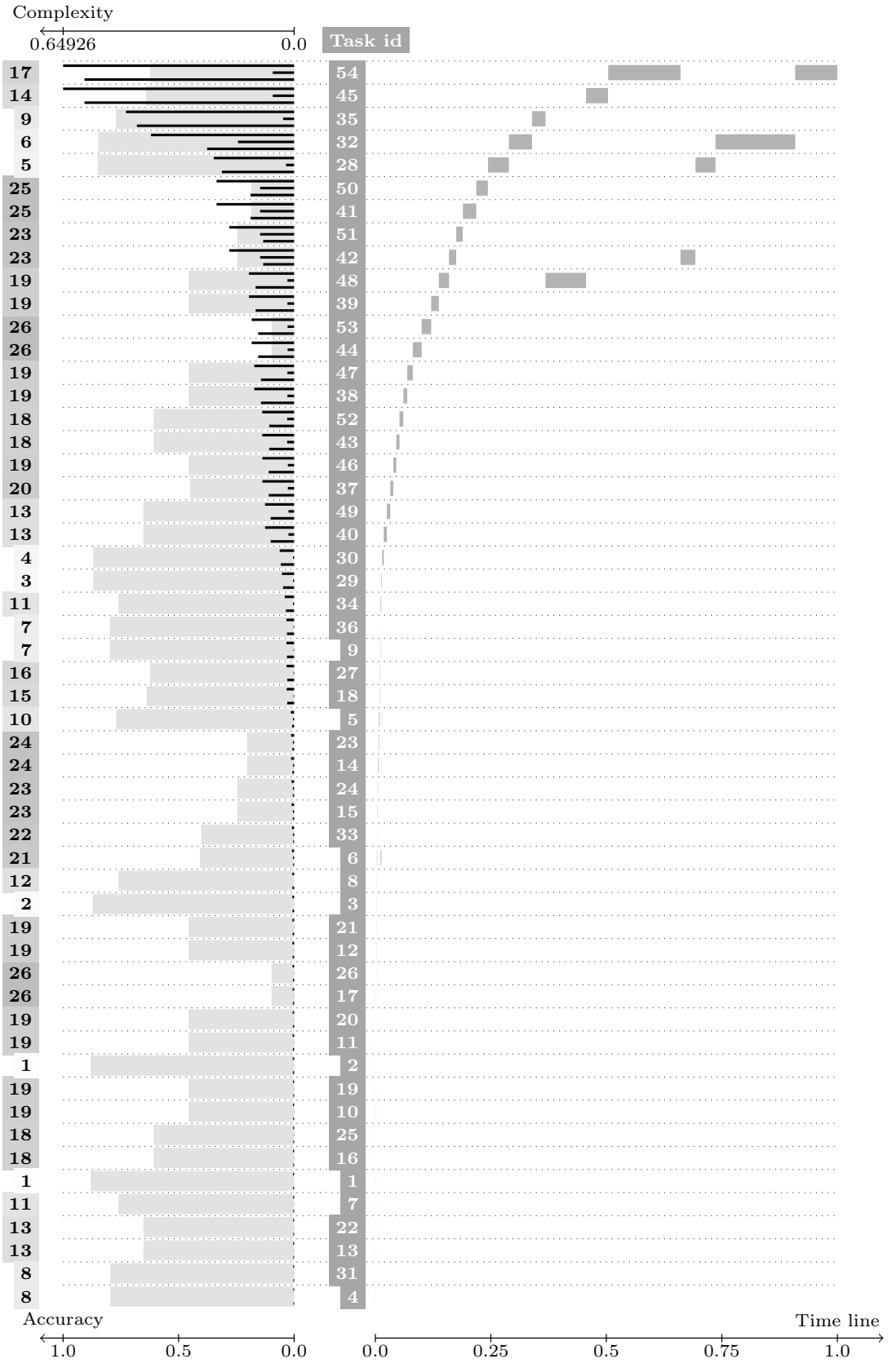

Fig. 17. Machine search space and test task ordering for vowel data.

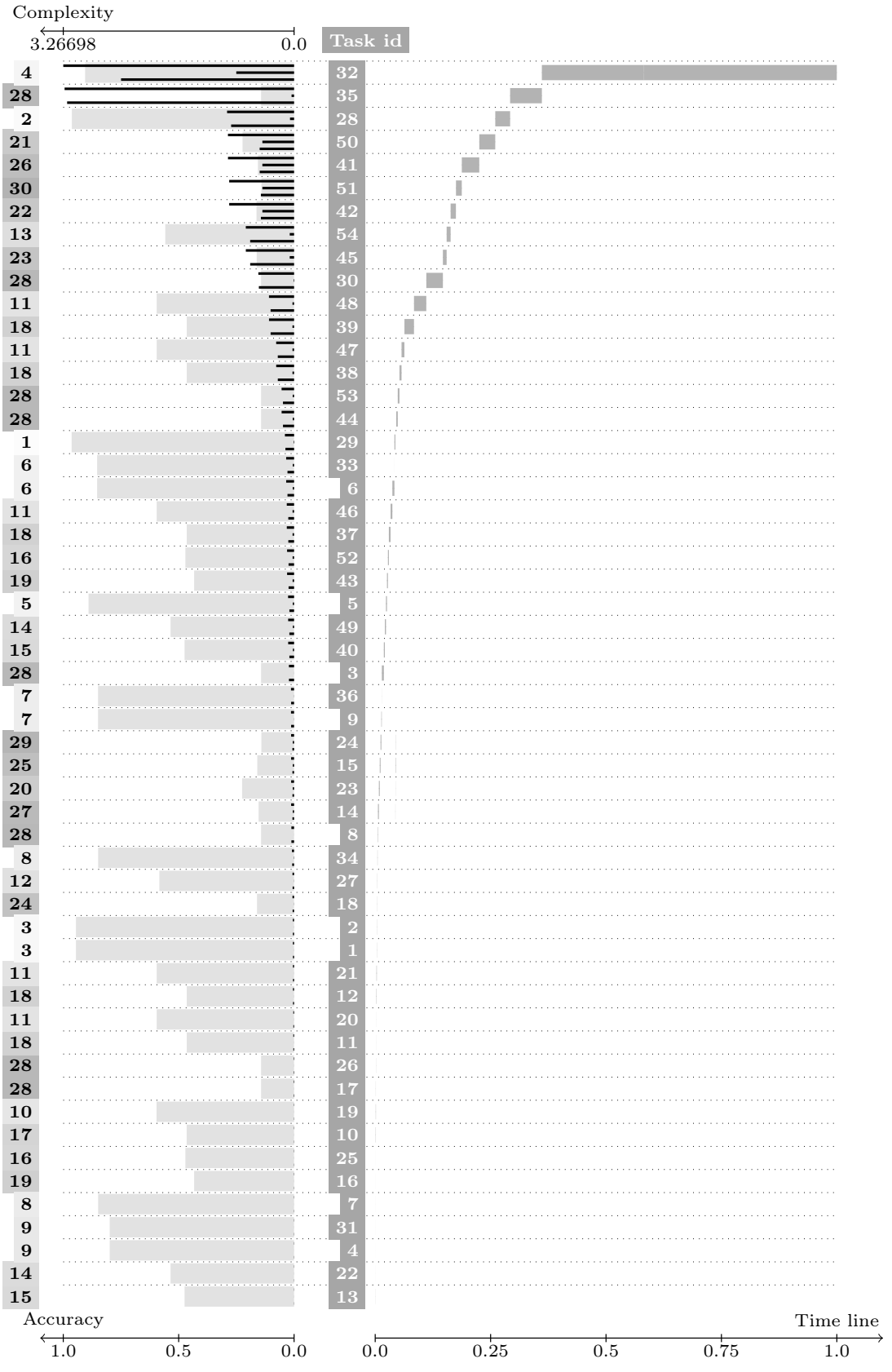

Fig. 18. Machine search space and test task ordering for image data.

Duch, W. and Itert, L. (2003). Committees of undemocratic competent models, *Proceedings of the Joint International Conference on Artificial Neural Networks (ICANN) and the International Conference on Neural Information Processing (ICONIP), Istanbul, Turkey*, pp. 33–36.

Duch, W., Wieczorek, T., Biesiada, J. and Blachnik, M. (2004). Comparison of feature ranking methods based on information entropy, *Proceedings of International Joint Conference on Neural Networks, Budapest, Hungary*, pp. 1415–1420.

Frank, A. and Asuncion, A. (2010). UCI machine learning repository, University of California, School of Information and Computer Science, Irvine, CA, `http://archive.ics.uci.edu/ml`.

Grąbczewski, K. and Jankowski, N. (2011). Saving time and memory in computational intelligence system with machine unification and task spooling, *Knowledge-Based Systems* **24**(5): 570–588.

Guyon, I. (2003). NIPS 2003 workshop on feature extraction, `http://www.clopinet.com/isabelle/Projects/NIPS2003`.

Guyon, I. (2006). Performance prediction challenge, `http://www.modelselect.inf.ethz.ch`.

Guyon, I., Gunn, S., Nikravesh, M. and Zadeh, L. (Eds.) (2006). *Feature Extraction: Foundations and Applications*, Springer, Berlin/Heidelberg.

Jankowski, N., Duch, W. and Grąbczewski, K. (Eds.) (2011). *Meta-learning in Computational Intelligence*, Studies in Computational Intelligence, Vol. 358, Springer, Berlin/Heidelberg.

Jankowski, N. and Grąbczewski, K. (2005). Heterogenous committees with competence analysis, *in* N. Nedjah, L. Mourelle, M. Vellasco, A. Abraham and M. Köppen (Eds.), *5th International Conference on Hybrid Intelligent Systems, Rio de Janeiro, Brazil*, IEEE Press, New York, NY, pp. 417–422.

Jankowski, N. and Grąbczewski, K. (2007). Handwritten digit recognition—Road to contest victory, *IEEE Symposium Series on Computational Intelligence,* IEEE Press, New York, NY, pp. 491–498.

Jankowski, N. and Grochowski, M. (2004). Comparison of instances selection algorithms I: Algorithms survey, *in* L. Rutkowski, I. Siekmann, R. Tadeusiewicz and L.A. Zadeh (Eds.), *Artificial Intelligence and Soft Computing*, Lecture Notes in Artifical Intelligence, Vol. 3070, Springer-Verlag, Berlin/Heidelberg pp. 598–603.

Jankowski, N. and Grochowski, M. (2005). Instances selection algorithms in the conjunction with LVQ, *in* M.H. Hamza (Ed.), *Artificial Intelligence and Applications*, ACTA Press, Innsbruck, pp. 453–459.

Kadlec, P. and Gabrys, B. (2008). Learnt topology gating artificial neural networks, *IEEE World Congress on Computational Intelligence, Hong Kong, China*, pp. 2605–2612.

Kohonen, T. (1986). Learning vector quantization for pattern recognition, *Technical Report TKK-F-A601*, Helsinki University of Technology, Espoo.

Kordík, P. and Černý, J. (2011). Self-organization of supervised models, *in* N. Jankowski, W. Duch and K. Grąbczewski (Eds.), *Meta-learning in Computational Intelligence*, Studies in Computational Intelligence, Vol. 358, Springer, Berlin/Heidelberg, pp. 179–223.

Korytkowski, M., Nowicki, R., Rutkowski, L. and Scherer, R. (2011). AdaBoost ensemble of DCOG rough-neuro-fuzzy systems, *in* P. Jędrzejowicz, N.T. Nguyen and K. Hoang (Eds.), *ICCCI (1)*, Lecture Notes in Computer Science, Vol. 6922, Springer, Berlin/Heidelberg, pp. 62–71.

Łęski, J. (2003). A fuzzy if-then rule-based nonlinear classifier, *International Journal of Applied Mathematics and Computer Science* **13**(2): 215–223.

Peng, Y., Falch, P., Soares, C. and Brazdil, P. (2002). Improved dataset characterisation for meta-learning, *5th International Conference on Discovery Science, Luebeck, Germany*, pp. 141–152.

Pfahringer, B., Bensusan, H. and Giraud-Carrier, C. (2000). Meta-learning by landmarking various learning algorithms, *International Conference on Machine Learning, Stanford, CA, USA*, pp. 743–750.

Prodromidis, A. and Chan, P. (2000). Meta-learning in distributed data mining systems: Issues and approaches, *in* H. Kargupta and P. Chan (Eds.), *Book on Advances of Distributed Data Mining*, AAAI Press, Menlo Park, CA.

Scherer, R. (2010). Designing boosting ensemble of relational fuzzy systems, *International Journal of Neural Systems* **20**(5): 381–388.

Scherer, R. (2011). An ensemble of logical-type neuro-fuzzy systems, *Expert Systems with Applications* **38**(10): 13115–13120.

Smith-Miles, K.A. (2008). Towards insightful algorithm selection for optimization using meta-learning concepts, *IEEE World Congress on Computational Intelligence, Hong Kong, China*, pp. 4117–4123.

Todorovski, L. and Dzeroski, S. (2003). Combining classifiers with meta decision trees, *Machine Learning Journal* **50**(3): 223–249.

Troć, M. and Unold, O. (2010). Self-adaptation of parameters in a learning classifier system ensemble machine, *International Journal of Applied Mathematics and Computer Science* **20**(1): 157–174, DOI: 10.2478/v10006-010-0012-8.

Witten, I.H. and Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufmann, Amsterdam.

Norbert Jankowski has been an assistant professor at Nicolaus Copernicus University in Toruń, Poland, since 2000. He received Ph.D. in computer science at the Institute of Biocybernetic and Biomedical Engineering, Polish Academy of Sciences, Warsaw, and an M.Sc. in computer science at the Institute of Computer Science, University of Wrocław, Poland. He is the author of 67 scientific articles and two books. Norbert Jankowski is a reviewer for numerous scientific journals and major conferences. He organized special sessions and tutorials on meta-learning at conferences. His main research areas are computational intelligence, meta-learning, machine learning, neural networks, data mining, pattern recognition, complexity, algorithms and data structures, modeling of brain function, neuroscience, complexity of information, graph theory. He won the competition for the Best Classifier of Handwritten Digits Recognition at the *8th International Conference on Artificial Intelligence and Soft Computing*, Zakopane Poland, 2006, and was third in the Feature Selection Challenge at *Neural Information Processing Systems*, 2003. In 2011, his co-edited book on *Meta-learning in Computational Intelligence* (Springer) was published. He is the author of another book, *Ontogenic Neural Networks* (EXIT, 2003).

Received: 21 June 2011
Revised: 12 December 2011
Re-revised: 15 February 2012