Int. J. Appl. Math. Comput. Sci., , Vol. , No. , –
DOI:

GRAPH-BASED GENERATION OF META-LEARNING SEARCH SPACE

NORBERT JANKOWSKI

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

Meta-learning becomes more and more important in current and future research concentrated around widely understood data mining or computational intelligence. It can solve problems that can not be solved by any single, narrowly specialized algorithm.
The overall characteristic of each meta-learning algorithm mainly depends on two elements: learning machine space and supervisory procedure. The first element restricts the space of all possible learning machines to a subspace to be browsed by meta-learning algorithm. The supervisory procedure determines the order of selected learning machines with a module responsible for machine complexity *evaluation*, organizes tests and performs results analysis.
In this article I 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 much more flexible way, depending on the characteristics of considered problem and a-priori knowledge. Presented method of search space description is used together with 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

Challenges organized around computational intelligence in recent years clearly present nontriviality of model selection (Guyon, 2003; Guyon *et al.*, 2006; Guyon, 2006; Jankowski and Grąbczewski, 2007). Optimal model usually results from advanced search among learning machines using a number of learning strategies. However it happens that for some dataset benchmarks, for example for some of UCI ML Repository benchmarks, simple and accurate (enough) models of 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 finish the search process with satisfactory solution.

One of the approaches to meta-learning develops methods of decision committees construction, different 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) 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. Authors of (Duch and Grudziński, 1999) presents optimization of several additions line metric or feature selection around k nearest neighbors method. In gating neural networks (Kadlec and Gabrys, 2008) authors use neural networks to predict performance of proposed *local experts* (machines preceded by transformations) and decide about 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 algorithms performance, can be found in (Smith-Miles, 2008). Authors of other research (Kordík and Černý, 2011) combines meta-learning with evolutionary programming. In another approach (Czarnowski and Jędrzejowicz, 2011) authors presents usage of team of agents, which executes several optimizations by tabu search or simulated annealing. An interesting cooperations based on experience from learning of classifiers in fuzzy-logic approaches can be found (Scherer, 2011; Scherer, 2010; Korytkowski *et al.*, 2011; Łęski, 2003). For some other approaches please see following books (Brazdil *et al.*, 2009; 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 learning machine space which is proposed in this paper.

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* $\mathcal{L}$:

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

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. It means that *model* is defined as result of learning of given learning machine. Models play different roles (assumed by $\mathcal{L}$) like the role of classifier, feature selector, feature extractor, approximator, prototype selector, etc. The $\mathcal{M}$ from general point of view is not limited to any particular kinds of algorithms (simple or complex, neural networks or statistical, supervised or unsupervised, etc.).

The *Meta-learning algorithm (MLA)* is also learning algorithm of a machine, however the goal of meta-learning is to find the best way of learning under given conditions.

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 complexity approximation framework). Intemi is realized in C#/.Net.

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

2. Meta-learning Algorithm

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

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

However finding an optimal model for 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 my general goal of the meta-learning is *to maximize the probability of finding possibly best solution within a search space of given problem $\mathcal{P}$ in as short time as possible*.

As a consequence of such 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 learning problem. In real life problems, sensible solutions $m \in \mathcal{M}$ are usually complex. Previous meta-learning approaches (mentioned in introduction) have been trying to find a final model via selection of one of a number of simple machines or of complex machines but of fixed structure. It has significantly reduced the space of models which could be found as solutions.

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

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

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

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

and the model space gets the form

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

The solution constructed by decomposition is often much easier to find, because of reduction of the main task

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

to a series of simpler tasks: model m_i solving the subproblem $\mathcal{P}_i$, is the result of learning process

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

where

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

and $K_i \subseteq \{0, 1, \ldots, i-1\}$, $\mathcal{M}_0 = \mathcal{D}$. It 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 $\mathcal{P}$.

So, 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 particular configuration. Such decomposition is often very natural: a standardization or feature selection naturally precedes classification, a number of classifiers precede 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 necessity of data transformation (as an integral part of complex machine) because not all machines need the same transformation. For example, a classifier committee may contain two classifiers which can be learned on continuous only data and on discretized only data respectively. 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 can not be always shared via many machines in given decomposition.

Now, the role of *meta-learning* can be seen as searching for possibly best decomposition in an automated way. In my work, it has been done by *generators 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 solution must be found and validated. The claim must follow a thorough analysis of many machines of different kinds and structures.

To provide the meta-learning as universal as possible the items listed below, should be included in the configuration of meta-learning algorithm:

• **Functional Search Space definition**: A fundamental aspect of every meta-learning is determination of the search space of learning machines and corresponding final configurations of learning machines which will be considered by the meta-learning. In the algorithm presented here, initial state of generators 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 my algorithm the goal is defined by two items:

– template of test scenario—it defines the test procedure in which given candidate machine will be embedded, learned and verified. See an example on Fig. 1 devoted to classification problems tested by cross-validation. The expectation of this step is to obtain (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 collected series of results into final quantity measuring quality of the machine embedded within the test task.

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

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

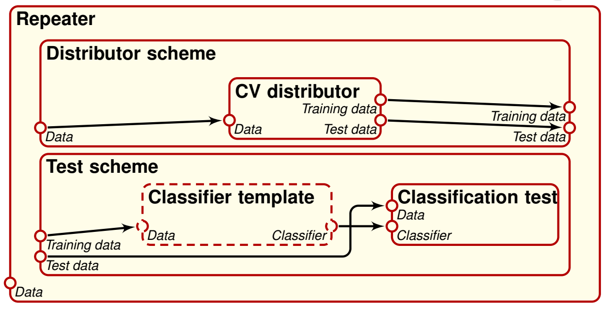

Fig. 1. A template of the test task scenario: to define the goal of learning for classification problem a CV test can be defined by a repeater machine with CV distributor and with test scheme composed of the classifier template and classifier test module.

The heart of my meta-learning algorithm is depicted in Fig. 2. The initialization step is a *link* between given configuration of meta-learning responsible to prepare initial states of several structures like generators 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 to some exception (different types of errors or broken

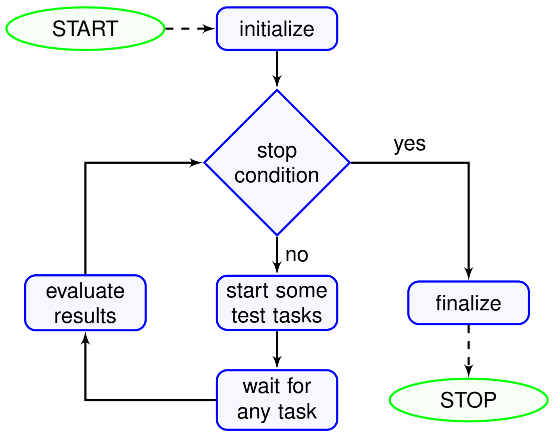

Fig. 2. General meta-learning algorithm.

by meta-learning because of 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 best machines) and new knowledge items created (e.g. about cooperation between different machines). 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). When the *stop condition* gets satisfied, the MLA returns the final result: configuration of the best learning machine or, in 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 *generators flow*. In general the goal of generators flow is to provide machine configurations on its output which will be considered as potential solutions for given problem $\mathcal{P}$ represented by given data D. Generators flows should provide a variety of machine configurations of given type. Such machines may be of simple or complex structure and may strongly differ in complexity.

Next, each machine configuration formulated by generators flow is nested in the *test tasks template* defined for given problem $\mathcal{P}$ (for description of templates see section 3.2 or figure 1). It is made by *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 configuration of meta-learning and defines the goal of meta-learning. Each machine configurations is nested in single test task configuration.

Test task configurations produced by test task generator are sent directly to the *test task heap*. In the test

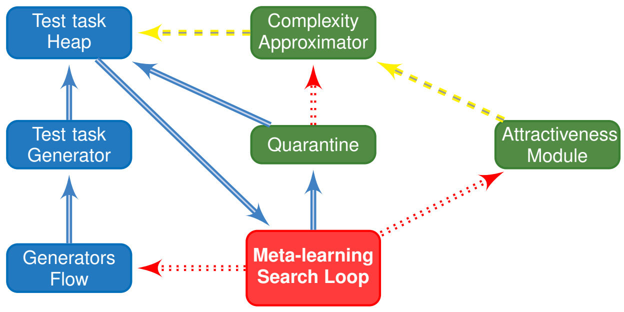

Fig. 3. Relations between modules of the meta-learning algorithms.

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 meta-learning process as it will be seen further. Also section 4 presents some information about approximation of machine complexity. Thanks to ordering by complexity simpler machines are favored and are tested before more complex ones.

This feature is crucial because otherwise, meta-learning would be vulnerable to losing much time without guarantee that given machine process will end. It would be natural if I assumed that each machine configuration provided by generators flow were equally promising. In cases when our *a priori* knowledge is *deeper*, it may be transformed in formal way as a special correction of complexity (see further parts about attractiveness of machine, page 5 and complexity definition on page 11). If the conditions to start a task (line 3 of the code) are satisfied, then a pair of machine configuration mc and its corresponding complexity description cmplx is extracted from test task heap (see line 4).

According to the order decided within the heap, 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> = test task heap.ExtractMinimum();
5     timeLimit = $\tau \cdot$ cmplx.time / cmplx.q
6     mlTaskSpooler.Add(mc,
7       limiter(timeLimit), priority−−);
8   }
9 **end**
```

Tasks are taken from the 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 I want to start a new task.

Since I 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 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 5 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 and Eq. 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 on page 5). τ is a constant (in my experiments equal to 2) to protect against too early test task braking.

Analysis of Finished Tasks

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

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

Next, the generators flow is called (line 17) 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 generators flow may provide new machine configurations or at least may change meta-knowledge.

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 (see again the scheme 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 Eq. 9). First, the reliability of the test task (its quality of complexity approximation) is corrected—see code line 19, and after that, the test task is resent to the test task heap as to quarantine—line 20.

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 been 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 can not 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 (Eq. 2) as it has been presented at the beginning of section 2. In general, such decomposition in NP-hard and can not be solved in a simple way. Additionally, it can not be solved in a direct analytical way. But it does not mean that I can not do much more than the previous meta-learning projects.

Finding attractive decompositions can be done quite naturally when using knowledge about usefulness of particular computational intelligence algorithms in different contexts. It helps in construction of reasonable classifiers, approximators, combined with data transformations, etc. and augmented by complexity control may constitute very powerful meta-search algorithms. Basing on meta-knowledge and attainable prior information about the problem, it is possible to construct functional form of search space for meta-learning algorithm. I define such search spaces in a form of *graph* designed to provide series of learning machine configurations for building test tasks for further browse by main part of 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) series of machine configurations from other MCGs. The graph (called *generators flow*) is directed and acyclic[2]. Each MCG may use (different) meta-knowledge, may reveal different behavior, which in particular may even change in time (during

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

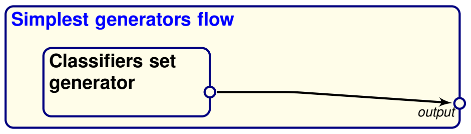

Fig. 4. Example of simplest generator flow.

the MLA run). By adding generators (with proper connections where adequate) I 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. It will be seen soon by examples. Connections facilitate 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 build 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 generators flow—the actual provider of configurations to the meta-learning. In the run time of the meta-learning algorithm, the configurations returned by the generators flow, are nested in test task and transported to the machine heap, before the MLA will start tests of selected (by ordering) configurations.

The simplest possible generators flow is presented in Fig. 4. It contains a single MCG with no inputs. The output of the flow is exactly the output of the only MCG (it is realized by the connection between 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 configuration, similarly to learning machines). It means no influence of the process of MLA on the output of the (fixed) MCG. Non-fixed generators 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 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 it can be seen in line 17 of the code on page 5, generators flow receives information about progress in meta-search process. Such information is propagated to each of MCG. It means that each MCG has access to current state of MLA and their further behavior may depend on that process. More and more sophisticated machine generators can optimize time of searching for most attractive solutions. Below, I describe examples of machine configuration generators and examples of generators 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 single generators 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 generator flow with set-based generator providing classifier machine configurations was 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 case of feature ranking methods, I 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 will be provided later in Fig. 9. Some other examples of using set-based generators will also be shown below.

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 empty placeholder (or placeholders) as 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 kNN as classifier, it can easily do it with a template-based generator. The combinations may be defined by machine template

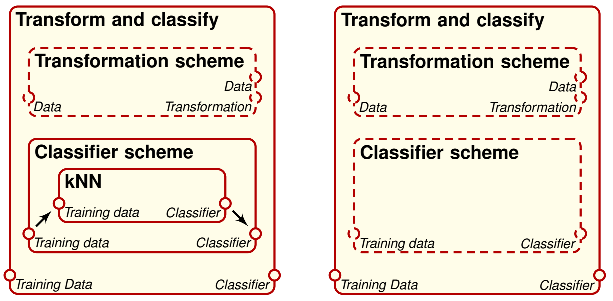

Fig. 5. A Transform and classify machine configuration template. Left: placeholder for transformer and fixed classifier (kNN). Right: two placeholders, one for the transformer and one for the classifier

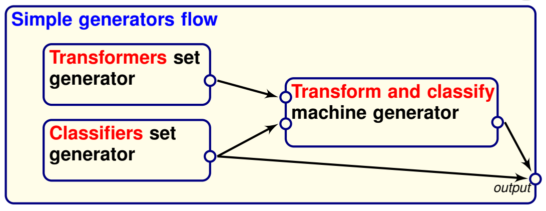
Simple generators flow
Transformers set generator
Classifiers set generator
Transform and classify machine generator
output

Fig. 6. A simple generator flow.

with a placeholder for data transformer and fixed kNN classifier. Such 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, I can use the Transform and classify (TnC) machine template shown in Fig. 5 on the left. 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 the number of placeholders. Always 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 a complex machine configuration as well as with simple one.

Replacing the kNN from the previous example by a classifier placeholder (compare Fig. 5 right part), I 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
– template with one or more placeholders,
– series of paths to placeholders and
– operation mode.

The order within the series of paths to placeholders defines the order of inputs of the generator. The main goal of template-based generator can be seen as production of series of machine configurations by filling template with appropriate machine configurations from inputs according to the series of path. By the same 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, 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. Naturally, the mode affects operation of multi-input generators only.

Fig. 6 presents the considered 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 gets configuration from both the set-based classifiers generator and the template-based generator, so it will provide both the classifiers taken directly from declared set and the classifiers preceded by declared transformers.

Another interesting example of a template-based generator is an instance using a template of Meta-parameter search (MPS) machine configuration. MPS machine can optimize any elements of machine configuration[3]. The search and optimization strategies are realized as separate modules which realize 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 given search strategy knows what to do with objects of given structure. The template containing 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 classifier, and finally the MPS machine will also play the role of classifier. Configuring the MPS machine to use the *auto-scenario* option makes the

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

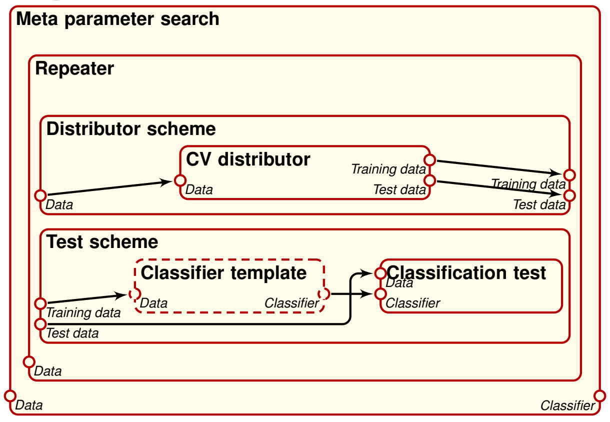

Fig. 7. A template of meta parameter search machine with classifier placeholder and classifier as a role of parameter search machine.

meta-learner receive configurations of MPS to realize the auto-scenario[4] for given classifier. It 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 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, see 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 a MPS generator. It means that each classifier configuration will be sent 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 Transformers generator while the second one receives configurations from another template-based generator and generate feature selection configurations with the use of different ranking algorithms received through the output-input connection with the Rankings generator. The combinations generated by the two Transform and classify generators are also sent to the output of the generators flow.

Additionally, the 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 the Transform and classify generator II. The output of the MPS generator is passed to the output of the generators flow too.

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

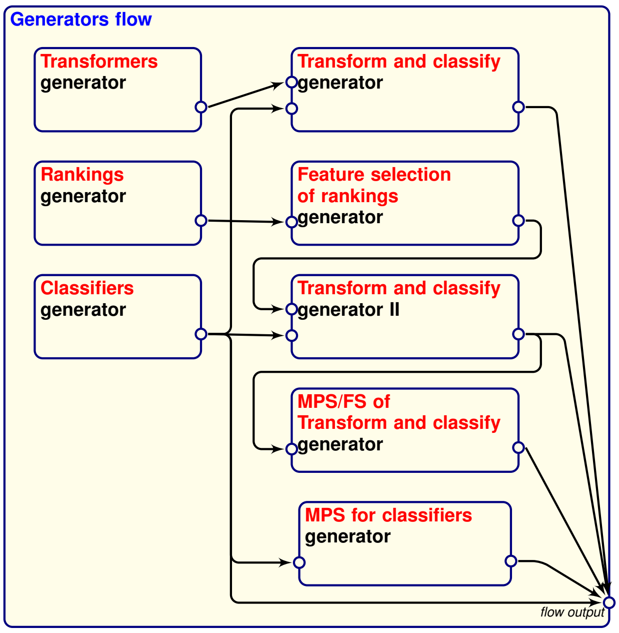
Generators flow
Transformers generator
Transform and classify generator
Rankings generator
Feature selection of rankings generator
Classifiers generator
Transform and classify generator II
MPS/FS of Transform and classify generator
MPS for classifiers generator
flow output

Fig. 8. Example of generators flow.

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 generators flows. Each generators flow may use any number of any kind of generators. As discussed above, it is often fruitful to separate groups of classifiers (data transformations etc.) in independent set-based generators, to reflect different application contexts of the machines. Such case can be seen in Fig. 9, where for more flexibility, feature ranking methods are separated into ones operating on discrete features (to be preceded by a discretization transformation, e.g. information theory based rankings) and others, that may be used without discretization. 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.

Generators 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; Jankowski and Grochowski, 2004) are

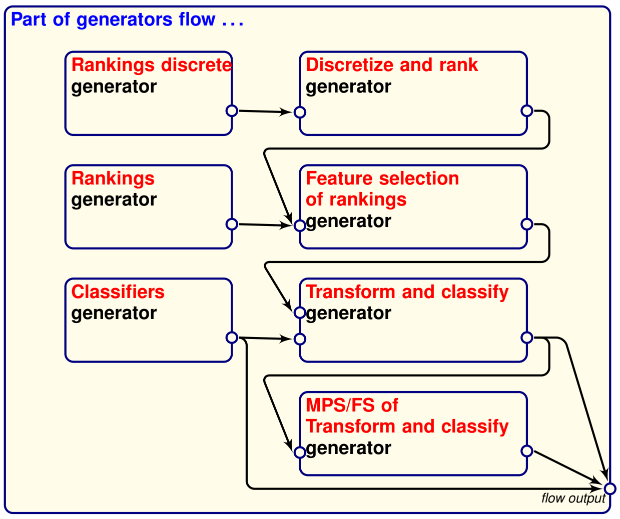
Part of generators flow ...
Rankings discrete generator
Discretize and rank generator
Rankings generator
Feature selection of rankings generator
Classifiers generator
Transform and classify generator
MPS/FS of Transform and classify generator
flow output

Fig. 9. A part of generators flow with two separate ranking generators to reflect different roles and different application contexts.

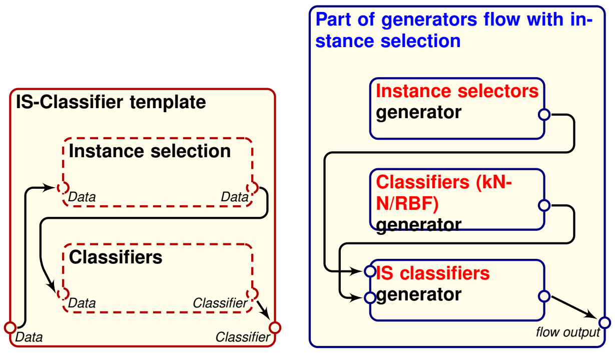

Fig. 10. Part of generator flow providing combinations of instance selection methods with classifiers trained on selected data instances.

used for several reasons: to filter out noise instances, to shrink the dataset or to find prototype instances (as a prototype-based explanation of considered problem). 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 I just need proper template as it is presented on the top of Fig. 10. The template contains two placeholders: one for an instance selection algorithm and another one for a classifier. The generator based on this template needs two inputs: one with configurations of instance selection algorithms and another with classifiers. Required generators and their bindings are presented in Fig. 10 on the 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. Beside the definition of the template it

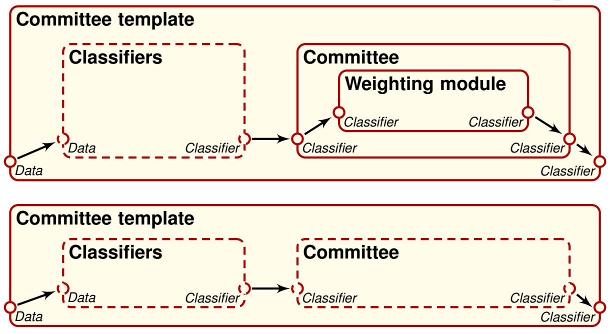

Fig. 11. Templates of committees.

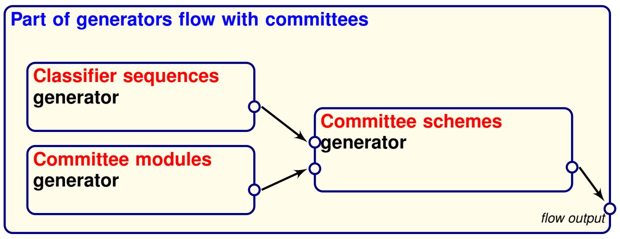
Part of generators flow with committees
Classifier sequences generator
Committee schemes generator
Committee modules generator
flow output

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

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 I use another flexible feature of generators input–output connections: possibility of propagation of not only series of machine configurations but also series of sequences of machine configurations. Let's 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 additional placeholder for committee decision module, so that committees can be composed of different sequences of classifiers and of different decision modules (voting, weighting, etc). It can be incorporated into a generators flow as in Fig. 12.

3.3. Advanced generators. 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, exhibits their own strategy. A 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 given classifier needs only continuous or discrete features.

Advanced generators are informed each time a test

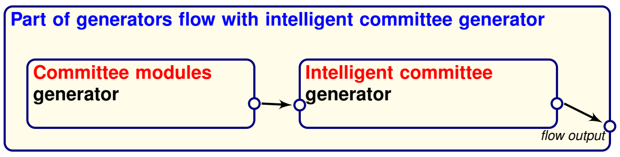
Part of generators flow with intelligent committee generator
Committee modules generator
Intelligent committee generator
flow output

Fig. 13. Part of generators flow with intelligent committee generator.

task is finished and analyzed. The generators may access the results computed within the test task and current machine configuration ranking (compare code line 17 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 meta-learning algorithm can not (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 from the main part of the algorithm and should cooperate with MLA and with other generators. On the other hand, MLA should not be responsible for all aspects of each learning machine, meta-learning can not be “conscious” even about all types of learning machines.

Another worthwhile example of advanced generator is the *intelligent committee generator* presented in Fig. 13. It continuously observes the progress of meta-learning, especially changes in ranking of machines. On the basis of this information, it builds new configurations of (promising) committees. New committees are build only under restricted conditions—when I expect it is rational to do it. Shortly it bases on the progress in ranking and on diversity of committee members which are selected between models of known quality and results that can be compared with 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 my system uses specialized machine cache so that committee members are computed once and never recomputed—they are reused when appropriate.

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

Such generator brilliantly fits the idea of complexity based control of tasks order in the heart of meta-learning algorithm. For example, in above-mentioned case of too huge data, the filter generator provides a machine for reduction of data complexity, which used before other machines may yield significant reduction of complexity of the whole complex configuration. This way, a 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 generators flow:

- 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.
- Generators flow should reflect functional domain of given problem. In other words the search space should be as adequate for given problem as possible. It may be done in very natural way by composing relation between adequately chosen machine generators and their connections.
- Generators flow is informed by MLA about progress in meta-learning. This information is propagated to each generator and can be used to to control operation of the generator. The results of performed tests are valuable meta-knowledge that can be collected, analyzed and used to construct new machine configuration and pass then to the machine heap for testing. In consequence, the product of generators flow depends not only on its configuration but also on the progress of meta-learning.
- Generators graph may also be changed (nodes and their connection, which in configuration are initialized) during meta-learning.
- Meta-learning algorithm may be continuously extended by new version of more and more sophisticated machine generators. It is, currently, my 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 arbitrarily huge generators flow. The graph defines the space of meta-learning search. Intelligent behavior of machine configuration generators within modular generators flow facilitates creation 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 in contrary to many known algorithms of AI or CI.

4. Machine Complexity Evaluation

Detailed description of complexity computation is out of the scope of this paper. I mostly focus on search space modeling and its relations with other parts of 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. I use the following definition of complexity:

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

Naturally, I 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 (I never know whether 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 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 (compare comments on page 5). This mechanism prevents from running test tasks for unpredictably long time or even 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 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 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).

Because complexity depends on configuration and inputs, the complexity computation must reflect the information from configuration and inputs. The recursive nature of configuration, 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 is finished (machines that provide necessary outputs). 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).

To facilitate recurrent determination of complexity 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 $\mathcal{L}$ is a transformation in form

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

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 (Eq. 1), because computation of complexity is a derivative of the behavior of machine learning process.

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[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.

Almost always evaluators are constructed with help of series of approximators. Number of approximators per evaluator depend on the number of functionalities which has to be provided by 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: I have defined a common framework for building approximators for evaluators to simplify the process of complexity approximation. The framework is defined in very general way and enables

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

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

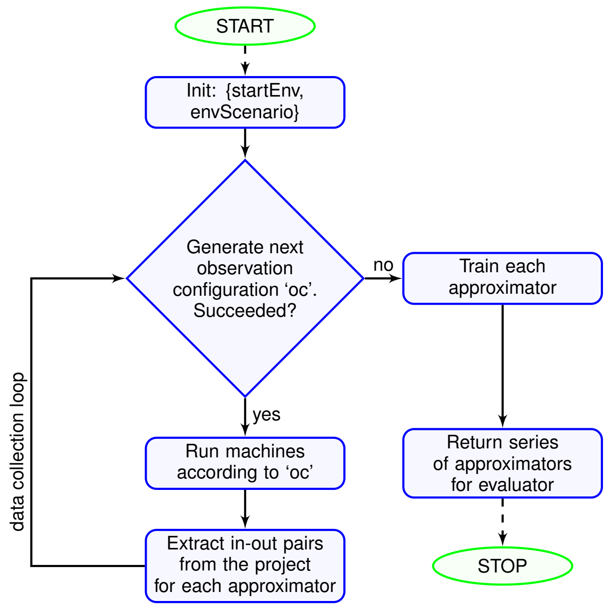
START
Init: {startEnv, envScenario}
Generate next observation configuration 'oc'. Succeeded?
no
Train each approximator
yes
Run machines according to 'oc'
Return series of approximators for evaluator
Extract in-out pairs from the project for each approximator
STOP
data collection loop

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

building evaluators using dedicated approximators for different aspects of complexity like learning time, 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 of time or memory than another task.

Fig. 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, 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 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 page 7), 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. 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. 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 advise from the user.When the process of building approximators for 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 I show how generators flows defined in section 3 expand the search space for meta-learning algorithm. Note that the goal here is not to present how optimal accuracies can be achieved but how generators flows unfold the search space.

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

With square brackets are denoted 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 Euclidean metric—the combination of feature selection and 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 machine denoted as

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

only the number of chosen features is optimized because this configuration is provided by the MPS/FS of Transform and classify generator (see Fig. 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 become too complex for assumed time limit.

5.1. Meta-learning configuration. The most important elements of my meta-learning algorithm configuration are: meta-learning test template, query test, stop criterion and generators flow.

Meta-learning test template. The test template must be adequate to the goal of learning. Since the chosen benchmarks are classification problems, I 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 was 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 generators flow.

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

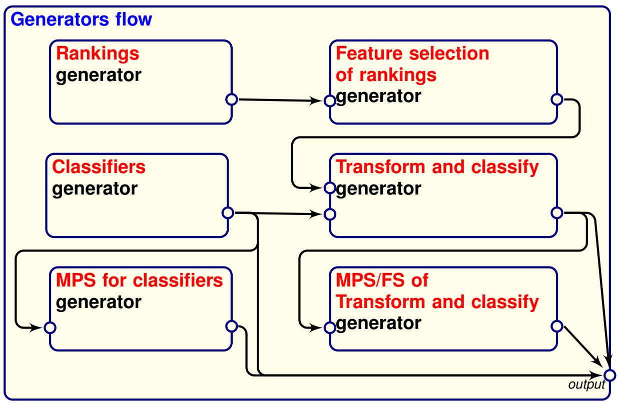
Generators flow
Rankings generator
Feature selection of rankings generator
Classifiers generator
Transform and classify generator
MPS for classifiers generator
MPS/FS of Transform and classify generator
output

Fig. 15. Generators flow used in tests.

5.2. Configuration of generators flow and its consequences for the search space of MLA. 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 my experiments is presented in Fig. 15. Very similar generators flow was explained in detail in section 3.

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, I use the following:

Classifier set:

– kNN (Euclidean) — k Nearest Neighbors with Euclidean metric,

– 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[7] machine transforms the original dataset, then the transformed data become the

[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 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.

learning data for kNN,
– [LVQ, kNN (Euclidean)] — first, 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.

The base classifiers and ranking algorithms, together with the generators 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 complexity controlled run. All the configurations provided by the generators flow are presented in Table 1.

Depending on the changes in the generators flow, the sequence of machine configurations may change significantly. Assume that the generators 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 (Duch *et al.*, 2004) for details about the algorithms):

- Entropy based ranking,
- Mantaras distance based ranking.

The sequence of machine configurations output by such generators 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 corresponding items of the form:

[* [EntropyRank], DiscretizeAndRank *]

and

[* [MantarasRank], DiscretizeAndRank *] .

For example, in analogy to

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

I get two additional feature selections for NBC:

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

and

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

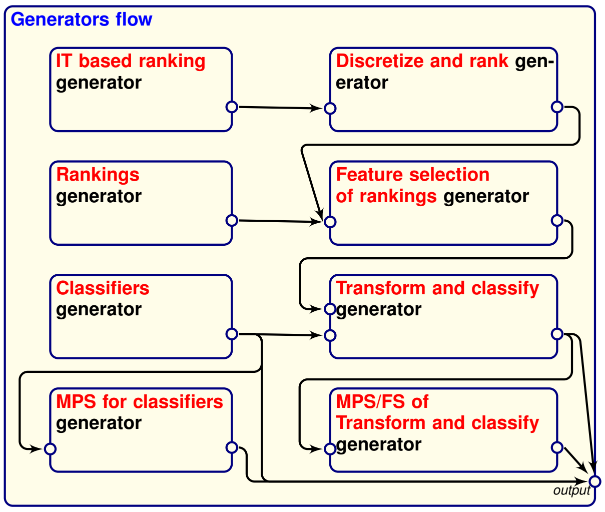
Generators flow
IT based ranking generator
Discretize and rank generator
Rankings generator
Feature selection of rankings generator
Classifiers generator
Transform and classify generator
MPS for classifiers generator
MPS/FS of Transform and classify generator
output

Fig. 16. Extended generators flow.

The full collection of 90 configurations will be analyzed by 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, I 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 specific and present many properties of the meta-learning algorithm. The diagrams present information about 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 1) and about 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 complexities of test tasks. It 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, I can see task ids 4 and 31 which correspond to the Naive Bayes Classifier and the ParamSearch [NBC] classifier. At the top, I can see task ids 54 and 45, as the most complex ParamSearch test tasks of this benchmark. Task order in the second example (Fig. 18) is completely different.

On the right side of the *Task id* column, there is a plot presenting starting, stopping and breaking times of

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 generators flow of Fig. 15 and the enumerated sets of classifiers and rankings.

each test task. As it 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 restarted task please look at Fig. 17, 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. Two different diagrams (in Fig. 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 $[0, 1]$ 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 I can not see any two bars overlapping in time. If I 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 *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. However, remember that the experiments were not tuned to obtain the best accuracies possible, but to illustrate the behavior of the generators 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 the 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 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 at the right 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 complexity (the lower line)[8]. All three complexities are the approximated complexities (see Eq. 9). 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. 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 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 presented benchmark vowel, very simple and accurate models were found quite early—close to the beginning of the meta-learning process. Please see Fig. 17 task ids 1 and 2. While for benchmark image (Fig. 18) there are no solution with so high accuracy found first. And the best one is found as one of most complex machines: see task id 32, which is tuned by MPS machine the SVM with Gaussian kernel (for ionosphere-ALL) and the best one for image was the taks Id 28 and 29 which are tuned by MPS the kNN machine with 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 deeper parameter search.

6. Summary

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

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

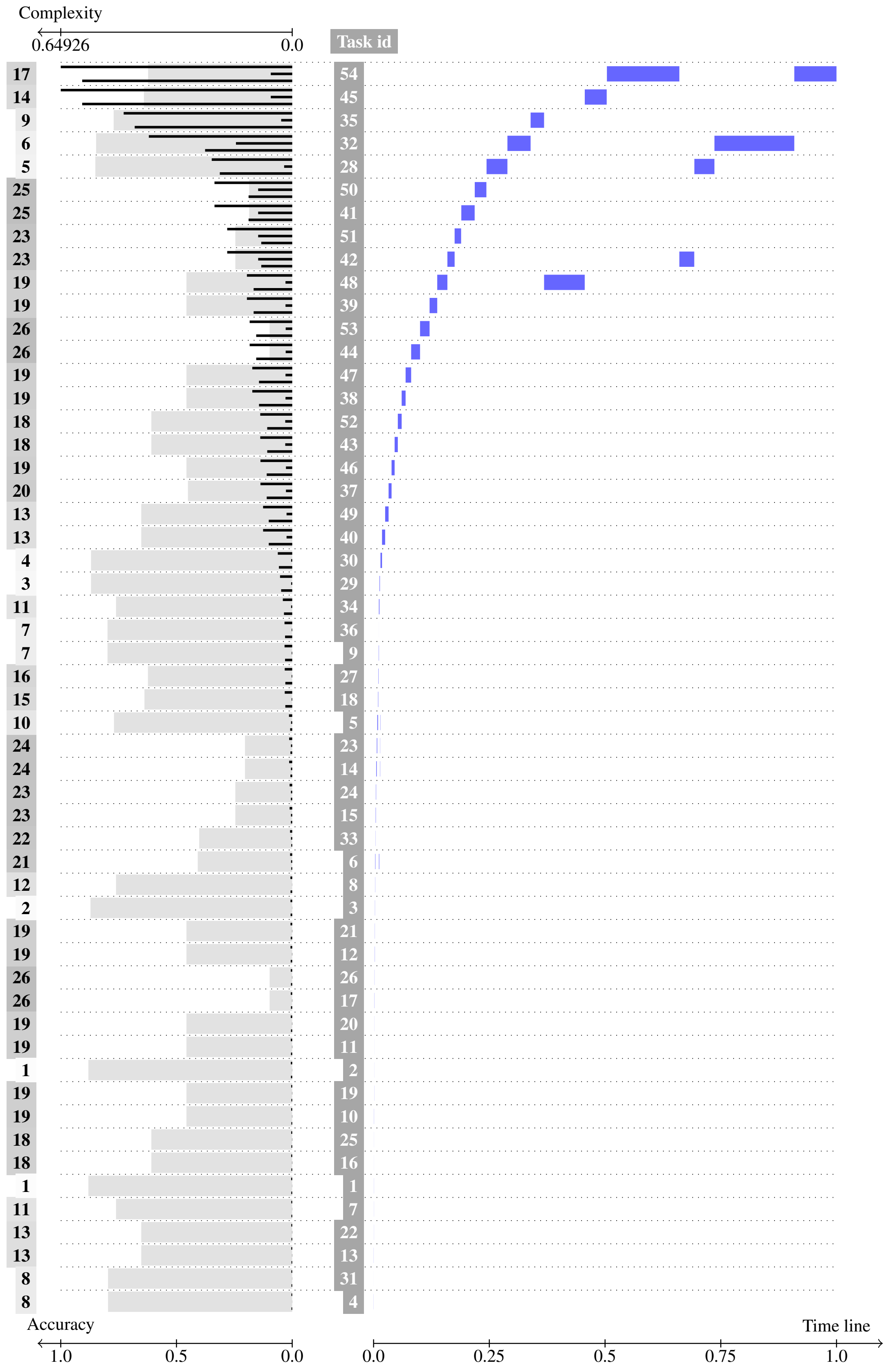

Fig. 17. Machine search space and test task ordering for vowel data.

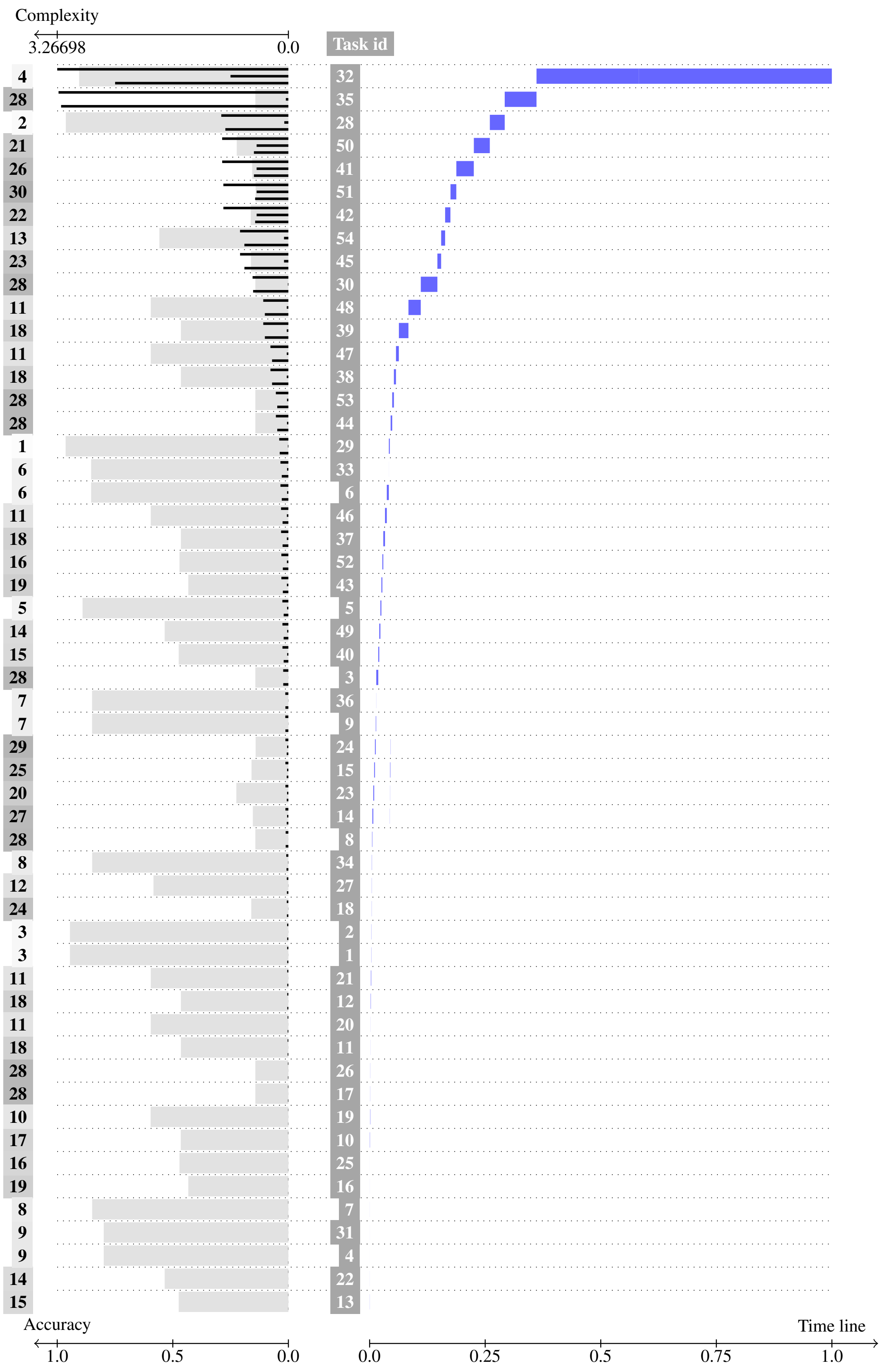

Fig. 18. Machine search space and test task ordering for image data.

In previous approaches to meta-learning, researches used only “flat” spaces (just small sets of machines were tested and compared). Framework for meta-learning search presented here, featuring very flexible way of defining 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 interesting decomposition of learning machines with advantage of general meta-knowledge about reasonable combinations of machine components and specific knowledge gathered during the search (about fitness of different methods to the problem being solved). With this MLA, browsing through complex machine structures is similarly 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 the learning.

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

In addition to the flexible search space, my approach uses complexity control for test task ordering. This feature helps the MLA perform the test tasks in most reasonable order—first the simplest learning machines are tested and then more and more complex ones are tried. Such approach increases the probability of finding solutions of attractive balance between machine simplicity and accuracy. It is important to see that MLA provides approximation of simple and 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 suitable way. Independence of machine generators gives opportunity to spread responsibility of knowledge-distribution. There is no need to have a single machine generator which “knows everything”. Machine generators can specialize in chosen type of behavior/subproblems.

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*, pp. 33–42.

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

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**: 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.

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

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

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

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

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. (2006). *Feature extraction, foundations and applications*, Springer.

Jankowski, N., Duch, W. and Grąbczewski, K. (Eds) (2011). *Meta-learning in computational intelligence*, Studies in Computational Intelligence, Springer.

Jankowski, N. and Grąbczewski, K. (2005). Heterogenous committees with competence analysis, *in* N. Nedjah, L. Mourelle, M. Vellasco, A. Abraham and M. Köppen (Eds), *Fifth International conference on Hybrid Intelligent Systems*, IEEE, Computer Society, Brasil, Rio de Janeiro, pp. 417–422.

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

Jankowski, N. and Grochowski, M. (2004). Comparison of instances selection algorithms: I. Algorithms survey, *Artificial Intelligence and Soft Computing*, Lecture Notes in Computer Science, Springer-Verlag, 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, Austria, pp. 453–459.

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

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

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, Springer, pp. 179–223.

Korytkowski, M., Nowicki, R., Rutkowski, L. and Scherer, R. (2011). Adaboost ensemble of dcog rough-neuro-fuzzy systems, *in* P. Jedrzejowicz, N. T. Nguyen and K. Hoang (Eds), *ICCCI (1)*, Vol. 6922 of *Lecture Notes in Computer Science*, Springer, 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, *The 5th International Conference on Discovery Science*, Springer-Verlag, 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*, Morgan Kaufmann, 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.

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*, IEEE Press, 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.

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

Norbert Jankowski is an assistant professor at Nicolaus Copernicus University in Torun Poland since 2000. He made PhD in computer science at Institute of Biocybernetic and Biomedical Engineering Polish Academy of Science, Warsaw. The M.Sc. he made in computer science at Institute of Computer Science University of Wrocşaw, Poland. He is author of 66 scientific articles and one book. Norbert Jankowski is a reviewer of scientific journals and major conferences. He organized special sessions and tutorials about 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 was at 1st place in Competition for the best Classifier of Handwritten Digits Recognition at The Eight International Conference on Artificial Intelligence and Soft Computing, Zakopane Poland, 2006 and at 3rd place in Feature selection challenge at Neural Information Processing Systems, 2003. In 2011 was published edited book "Meta-learning in computational intelligence" (Springer, Computational Intelligence Series). He is author of two other books “Ontogenic neural networks" (2003) and “Meta-learning in computational intelligence" (2011).