Complexity-based test task ordering for meta-learning algorithms

Norbert Jankowski

Department of Informatics, Nicolaus Copernicus University, Toruń, Poland

Abstract
Researchers working in computational intelligence (CI) have come up with large number of algorithms concerning miscellaneous aspects of data analysis. Not surprisingly the growing number of different algorithms does not lead to simplification of the model selection problem. Finding an optimal model for given data is a hard and very important problem for the machine learning (ML) community. According to Occam’s razor, we should check the simplest methods first and try more complicated ones only if they may provide significantly better models.

In the paper, I present a very advanced meta-learning algorithm in which a search procedure is controlled by estimation of learning machine complexities. This is very important because many algorithms may collapse in case of huge number of instances or attributes. No algorithm before had a possibility to control its own complexity. I propose a complexity measure for learning machines. I introduce special complexity evaluators, estimating several aspects of the complexity of learning machines. Based on this concept, meta-learning test tasks can be ordered and tested according to their complexity. The system was designed to use several types of local and global meta-knowledge.

Example applications prove that such meta-learning is able to control the order of test tasks efficiently and accurately enough.

Keywords: meta-learning, computational intelligence, complexity, machine learning, data mining, knowledge-base systems

1. Introduction

In general the idea of meta-learning is to learn how to learn better. It simply means that a meta-learning machine use the results of learning of other algorithms.

The idea of algorithm selection for a given problem—which touches meta-learning—was first time proposed by Rice [35]. He presented an abstract model of the problem as shown in Figure 1. This is the problem of finding a mapping \( S : \mathcal{D} \rightarrow \mathcal{A} \), such that for given data \( D \in \mathcal{D} \), \( A = S(D) \) is an algorithm maximizing a measure of performance \( \| p(A, D) \| \). In another formulation, the domain of the mapping \( S \) was not the problem space itself, but a space \( \mathcal{F} \) of the features derived from the problem space.

The first group of learning machines that can be called meta-learning is composed of ensembles of learning machines. That group consists of committees of classifiers like bagging or boosting [6, 11, 12, 30] which try to combine machines based on changes in the data distribution and different stacking strategies, which build meta-level decision model to combine committee elements [42, 8, 7, 19, 33, 39, 9, 21] or even stacking with a decision tree [36] with the leaves playing the role of classifiers.

![Figure 1: Rice's model.](image-url)
In gating neural networks [24] authors use paired neural networks, the second, called the local expert is used to predict the performance of the first (which may be proceeded by selection of features or instances). It can be seen as a more complex stacking as well.

Another huge group of meta-learning algorithms at the meta-level of learning build machine(s) that based on the problems characteristics and tries to attach the best learning machine to the captured characteristic of the problem. The sets of features that characterize the problem of learning was defined in a variety of ways [34, 13, 29, 25, 32, 18, 5, 1, 31, 4]. The characteristics of the problem are used to learn the relation between problem characteristics and the performance of different learning methods. The most common data characteristics were numbers of features/classes, feature variances, statistical information, information theory-based measures, information from histograms, etc. In some cases authors use regression in building of the meta-module, while sometimes they use classification strategy. The concept of landmarking [32, 3, 2] was used to predict performance of the more time-consuming machines by using prediction based on problem characteristics and performance of the simpler learning machines and their relation to accuracy of the more complex machines.

Another application of meta-learning to optimization problems, conducted by building relations between elements which characterize the problem and algorithms performance, can be found in Smith-Miles [37]. For some other approaches please see Brazdil et al. [4].

Although the projects are really interesting, they still suffer from many limitations. One of major disadvantages was focusing only on choosing of the decision model (a classifier) neglecting the proceeding preprocessing of data. In recent years it was clearly presented [15, 17, 22] that finding appropriate data transformation is more important than optimization of the process of choosing the classifier. It can be observed in the results of serious data mining challenges like NIPS 2003 Challenge in Feature Selection [15, 17] or WCCI Performance Prediction Challenge [16] how sophisticated machine configurations must be constructed to provide maximum performance. The top ranked algorithms, in such contests are almost always complex machines built of several data transformation techniques and often decision making ensembles.

Neither does the presented approach enforce the choice between decision machines nor restricts the structural complexity of machines. The presented meta-learning mimics the human experts behavior by performing a search in the space of algorithms, adjusting the algorithm structure and detail parameters to the results of proper performance tests. The adaptive process strongly depends on the problem being solved, so it looks different for each individual input data set. Keeping Occam’s razor in mind, I aim at the proper order of testing candidate machines (simple machines first, then more and more complex). Therefore I have introduced a complexity measure and made it a very serious element of my meta-learning algorithm. Complexity measures introduced in Sec. 4 are not related to VC dimension introduced by [40, 41]. My complexity measures are used to calculate time and memory complexities.

To facilitate easy and versatile machine construction and testing, one needs a versatile and efficient computational intelligence (CI) system with uniform machine handling, equipped with quite diverse a set of machines. In recent years, I have developed a versatile CI system Intemi [14], that fulfills all the requirements mentioned above. On the basis of Intemi, I have constructed a general meta-learning algorithm capable of searching for attractive machines [23].

Outline of the paper:. In this paper, I focus mainly on the mechanism of complexity control for my meta-learning algorithms, which is the crucial element of the approach to efficient, automated meta-search. The idea of controlling meta-learning by complexity of the machine learning opens new possibilities of research concentrated on meta-learning.

Section 2 describes fundamental definitions important for further parts, among others the restricted learning problem or meta-learning, and also information about the importance of test-task ordering. Section 3 first presents a general scheme of meta-learning and after that continues with rough insights into specific parts of my algorithm: the definition of the space of the learning machines searched during meta-learning, the procedure of starting test tasks and the analysis of the results performed within the meta-learning process.

Section 4 is devoted to describing how to estimate the necessary complexity aspects of learning machines to build proper order in the machines heap which determines the test task order in meta-learning search. First, a theoretical investigation is presented in which I comment on known complexity definitions and propose new ones. After that, I present an approximation framework for constructing complexity evaluators with appropriate learning. The evaluators calculate, in automated way, complexity of the given learning machines (simple or complex).
Next, section 5 depicts the advantages of complexity controlled meta-learning by some example applications.

2. Learning and Meta-learning

To investigate learning machines complexities we have to come back to the roots of learning problem.

Definition 2.1 (Learning problem \( \mathcal{P} \)). The learning problem \( \mathcal{P} \) is represented by the pair of data \( D \in \mathcal{D} \) and model space \( \mathcal{M} \), and consists in finding possibly best model within \( \mathcal{M} \).

Finding the best solution for \( \mathcal{P} \) is usually an NP-hard problem and we must be satisfied with suboptimal solutions found by learning machines defined as processes

\[
L : \mathcal{K}_L \times \mathcal{D} \to \mathcal{M}
\]

where \( \mathcal{K}_L \) is the space of configuration parameters of \( L \).

The task of finding possibly best model for fixed data \( D \), can be replaced by the task of finding possibly best learning machine \( L \). The latter can be done manually or by meta-learning algorithms.

The above definition describes what data analysts usually do, but it neglects one very important aspect: the time which can be spent on solving the problem is in fact, always limited. We never have infinite time for finding satisfactory solutions, even in the case of research computations. This is why I propose to analyze learning problems with special respect to time limits.

Definition 2.2 (Restricted learning problem). The restricted learning problem is a learning problem \( \mathcal{P} \) with defined time limit for providing the solution.

Such definition of the problem of learning from data is much more realistic in comparison to the original one. It should be preferred also for research purposes. In a natural way, it reflects the reality of data analysis challenges, where deadlines are crucial, but it is also more adequate for real business applications, where solutions must be found in appropriate time and any other applications with explicit or implicit time constraints.

Note that the learning machines may be simple or complex. Usually, a learning machine is composed of several other learning machines and learning machine may be seen as vector of submachines:

\[
L = [L_1, L_2, \ldots, L_k]
\]

where output of some machines constitutes the inputs for other machines, and their input-output connections compose direct acyclic graphs. For example, a composition of a data transformation machine and a classifier machine is just another learning machine. In quite a similar way, each committee is also a complex learning machine. What’s more, the committee of complex machines is also a learning machine.

Moreover I do not distinguish between learning and non-learning machines as it was done in [14]. Thanks to that, test procedures estimating some measures of learning machines accuracies, adaptation etc. can also be seen as other machines.

Meta-learning is just another type of learning (with another data and model spaces). The most characteristic feature of meta-learning is that it learns how to learn. Usually, this task is done by conclusions based on conclusions from other learning tasks formulated and evaluated during the run of meta-learning.

The goal of meta-learning. In general, the goal of my approach to meta-learning is to maximize the probability of finding possibly best solution within a search space of given restricted problem \( \mathcal{P} \) in as little time as possible.

Pursuing the goal, meta-learning algorithms should carefully order testing tasks during the progress of the search and build meta-knowledge based on experience from past tests. They need to avoid test tasks, that could accumulate too large amount of time, since it would decrease the probability of finding more interesting solutions. During the search process, meta-knowledge of different kinds may be extracted and may help in choosing further steps.

In my meta-learning approach, the algorithms search not only among the basic learning machines, but also produce and test complex machines like compositions of (several) transformations and classifiers (or other final decision
making machines), committees of different types of machines, including complex ones (e.g. compositions of a data transformer and a classifier as a single complex classifier inside the committee). Also, the data transformations may be nested or compose chains. However the order of the test tasks is not accidental but depends on the complexity of the machines, as it will be described in section 4. This feature will maximize the probability of finding a solution the of given problem.

3. Meta-learning Algorithm

Meta-learning algorithms learn by observation and testing of learning machines. The algorithms differ in the strategy of selecting, which learning machines to observe and what to observe in the machines to find possibly best or at least satisfactory conclusions. From theoretical point of view, the meta-learning algorithm is not limited in any way. In practice, the process must be adapted to limitations of memory and time.

I propose a unified scheme of meta-learning algorithms (MLAs) which base on learning from observations. It is depicted in Figure 2. Indeed many existing meta-learning algorithms fulfill this scheme (except the red blocks). The red blocks are specialized modules devoted on presented meta-learning, and each of these modules will be described below and in next sections.

After initialization step, the meta-learning algorithm starts its main loop, which up to the given stop condition, runs different learning processes, monitors them and concludes from their gains.

In each repetition, module start some tasks defines a number of tasks which test the behavior of appropriate learning configurations (i.e. configurations of single or complex learning machines). In other words, at this step it is decided, which (whether and when) machines are tested and how it is done (the strategy of given MLA).

Figure 2: Scheme of presented meta-learning algorithm.
In the next step (wait for any task) the MLA waits until any test task is finished, so that the main loop may be continued. A test task may finish in a natural way (at the assumed end of the task) or due to some exception (different types of errors, broken by meta-learning because of exceeded time limit and so on).

After a task is finished, its results are analyzed and evaluated. In this step some results may be accumulated (for example saving information about best machines) and new meta-knowledge items created (e.g. about different machines cooperations). Such knowledge may have crucial influence on further parts of the meta-learning (tasks formulation and the control of the search through the space of learning machines). Precious conclusions may be drawn, even if a task is finished in a non-natural way.

When the stop condition becomes satisfied, the MLA prepares and returns the final result: the configuration of the winner learning machine or, more generally, a ranking of learning machines (ordered by a degree of goal satisfaction), comments on chosen learning machines and their interactions, etc. For the restricted learning problem the stop condition may be simply defined by the time limit.

Each of the key steps of this general meta-learning algorithm may be realized in different ways yielding different meta-learning algorithms: from simple committees like boosting to sophisticated meta-learning algorithms.

The following subsections provide short descriptions of the most important parts of my meta-learning algorithm, to make the goals of complexity control easier to understand.

3.1. Configuring the meta-learning

Each meta-learning pretending to be a solver of a broad type of problems must exhibit flexible configuration possibilities. In other case the algorithm may be used just with specific type of problems and a determined procedure of estimating the quality of the solution. Construction of a really general meta-learning must not be too complex, but it needs to be configurable by:

- definable search space—defines the broadness of meta-learning penetration (see more in Section 3.2),
- stop criterion—already mentioned,
- definable goal of the problem—definition of the test procedure and an appropriate quality estimation procedure is necessary to control the generalization for the given type of problems and specific needs. The goal in the context of this meta-learning may be defined in flexible way by choosing an appropriate test procedure which is actually a configuration template of one of test machines.
- initial meta-knowledge is also an element of meta-learning configuration. In case of the presented meta-learning techniques the initial knowledge consists of complexity evaluators used and an ontology of optimization procedures for different types of learning machines (this is used by a later-described Meta-parameter search machine).

3.2. The Space of Learning Machines Used in Meta-learning

There are two main aspects of meta-learning algorithms definition having the most important influence on final results:

- formulation of the search space of learning machines—the definition of the set of learning machines that can be tested inside the MLA,
- definition of the order within the test tasks that can be performed by the MLA—the time limits do not allow for performing all the tests, so the order strongly affects the availability of attractive results.

The problem of test task ordering is the main problem of this article and will be investigated further on. Here, I shortly present the idea of the search space definition in MLA because it is strongly connected with ordering based on complexity measures. In my MLA the search space is defined in the functional form of generator flow. It is a specialized type of graph which vertices are machine configuration generators (MCG). The edges are used to send machines (configurations) provided by one generator to another.

Each MCG may be based on different meta-knowledge, may reveal different behavior, which in particular may even change in time (during the progress of meta-learning). The simplest generator flow may be defined by a single
machine configuration generator. For example, it can generate a set of classifiers, however I have never used such simple cases, because they are too limited.

To understand the idea of the generator flow, I describe the one presented in Figure 3. This generator flow contains two set-based generators, which provide rankings and classifiers configurations to other (template-based) generators.

By configuration template (shortly template) I mean complex machine configuration which has placeholder(-s) (unfilled subconfigurations of predefined goal). The goal of a placeholder is defined by the types of its inputs and outputs, for example a classifier placeholder has one dataset input and single classifier output. By template-based generator I mean each machine generator based on a template configuration.

The goal of the template-based generators is to provide filled template configuration at the outputs, based on their inputs and a strategy of template filling.

Please, notice that the classifiers generator sends its output configurations directly to the generator flow output and additionally to two template-based generators: one combining transformations with classifiers and one constructing meta-parameter search (MPS) machines for classifiers. Meta-parameter search machine is a sophisticated learning machine capable of optimizing parameters of any learning machine (either simple or complex) based on declarations of the suggested optimization strategies for given learning machine. The strategy descriptions are deposited in a specialized knowledge database (for more see page 32).

Additionally, the transform and classify generator sends its output configurations to the MPS/FS of transform & classify generator. This generator produces meta parameter search machine configurations, where the number of features is optimized for configurations produced by the transform and classify generator. The output of the MPS/FS generator is passed to the output of the generator flow too.

In such a scheme, a variety of configurations may be obtained in a natural way (see Section 5). The control of template-based generators is exactly convergent with the needs of meta-search processes.

Each generator flow may use any number of any kind of generators. There are no a priori limits on the number of connections between the generators. In some cases it is fruitful to separate groups of classifiers into independent set-based generators to simplify the connection paths in the generator flow graph. The same conclusion is valid also for transformations.

3.3. Test Tasks Starting

In the main loop of the general scheme of meta-learning presented in Figure 2, test tasks are started to evaluate the candidate machine configurations. The machine configurations provided by the generator flow, described above, are passed to the test task generator (compare with Figure 4), where they get nested into appropriate test template resulting in the test task configuration. For example, machine configurations may be nested in the cross-validation or any other test. The test configuration template is a part of the test task generator parameterization, which in turn, is a part of configuration of the MLA.

The prepared test tasks are sent to the test task heap (testTaskHeap in the code below). The test tasks come out from the testTaskHeap in appropriate order, reflecting their complexities estimated by the complexity approximator module. According to the order decided within the heap, the procedure startTasksIfPossible starts the simplest machines first and than more and more complex ones. The heap and complexity issues are addressed in detail, in Section 4.
If there is a task to be run and the task spooler is not full—see line 2 of the code, then a pair of machine configuration \( mc \) and its corresponding complexity description \( cmplx \) is extracted from the testTaskHeap (see line 4).

Since the machine complexity it is only an approximation, the meta-learning algorithm must be ready for cases when this approximation is not accurate or even the test task is not going to finish (e.g. according to problems with convergence of learning). To bypass these problems each test task has its own time limit for running. After the limit is reached, the task is aborted. In line 5 of the code, the time limit is set up according to predicted time consumption (\( \text{cmplx.time} \)) of the test task and current reliability of the machine (\( \text{cmplx.q} \)). The initial value of the reliability is the same (equal to 1) for all the machines, and when a test task uses more time than the assigned time limit, the reliability is decreased (see code line 24 and its discussion). \( \tau \) is a coefficient (in experiments equal to 2) to protect against too early test task braking.

### 3.4. Analysis of Finished Test Tasks and the Quarantine

After starting appropriate number of tasks, the MLA will wait until any task is finished. A task may finish normally (including termination by an exception) or halted by time-limiter (because of exceeding the time limit).

```plaintext
procedure analyzeFinishedTasks;
foreach (t in mlTaskSpooler.finishedTasks)
{
    mc = t.configuration;
    if (t.status = finished_normally)
    {
        qt = computeQualityTest(t, queryDefinition);
        machinesRanking.Add(qt, mc);
        machineGeneratorsFlow.Analyze(t, qt, machinesRanking);
        if (attractivenessModule is defined)
        attractivenessModule.Analyze(t, qt, machinesRanking);
    }
    else // task broken by limiter
```
The procedure runs in a loop, to serve all the finished tasks as soon as possible (also those finished while serving other tasks).

When the task is finished normally, the quality test is computed based on the test task results (see line 15) 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 qt and machine configuration mc. Machine ranking is ordered by the quality estimate qt from code line 15. The machine ranking is the main output of presented meta-learning.

If the attractiveness module is defined (see lines 18–19), it gets the possibility to analyze the new results and, in consequence, may change the attractiveness part of machines complexities (compare Section 4.2). In this way, the MLA may change the complexity of machines already deposited in the test task heap (testTaskHeap) and the heap is internally reorganized according to the new complexities (compare Section 4.2 and especially Eq. 10). Attractiveness modules may learn and organize meta-knowledge on the basis of the finished tasks results.

Next, the generator flow is called (line 17) to analyze the new results to build meta-knowledge and use it in future.

When a task is halted because of exceeding the time-limit (which was set up with respect to the complexity approximation), the task is moved to the quarantine for a period not counted in time directly, but determined by the complexities. Instead of constructing a separate structure responsible for the functionality of a quarantine, the quarantine is realized by two naturally cooperating elements: the test task heap and the reliability term of the complexity formula (see Eq. 10). First, the reliability of the test task is corrected—see code line 24, and after that, the test task is resent to the test task heap—line 25. The role of quarantine is very important and the costs of using the quarantine are, fortunately, not too big. MLAs restart only those test task for which the complexity was badly approximated.

To better see the costs, assume that a test task was completely badly approximated and visited quarantine maximum number of times. If the real universal time used by this task is \( t \), then, in the above scheme of the quarantine, the MLA may spend for this task, a time not greater than \( t + 1/4t + 1/16t + \ldots = 7/3t \). So the maximum overhead is \( 4/3t \), however it is the worst case—the case where I halt the process just before it would be finished (hence the two whole \( t \)'s in the sum). The best case gives only \( 1/3t \) overhead which is almost completely insignificant. The overhead is not a serious hamper, especially, when we take to the account that the MLA with the quarantine is not affected by the halting-problem of test-task. Moreover, the cost estimation is pessimistic also from another point of view: thanks to the unification mechanism, each subsequent restart of the test may reuse significant number of submachines run before, so in practice, the time overhead is usually minimal.

4. Machine Complexity Evaluation

In the past, I have postulate the idea that meta-learning algorithms should favor simple solutions (learning machines of simple structures—no submachines etc.) and start the machines providing them before more complex ones. It means that MLAs should start with maximally plain learning machines, then they should test some plain compositions of machines (plain transformations with plain classifiers), after that more and more complex structures of learning machines (complex committees, multi-transformations etc.). But the problem is that the order of such generated tasks does not reflect real complexity of the tasks in the context of problem \( \mathcal{P} \) described by data \( D \). Let’s consider two testing tasks \( T_1 \) and \( T_2 \) of computational complexities \( O(mf^2) \) and \( O(m^2f) \) respectively. Assume the data \( D \) is given in the form of data table and \( m \) is the number of instances and \( f \) is the number of features. In such case, it is not possible to compare time consumption of \( T_1 \) and \( T_2 \) until the final values \( m \) and \( f \) are known. What’s more, sometimes a composition of a transformation and a classifier may be indeed of smaller complexity than the classifier without transformation. It is true because when using a transformation, the data passed to the learning process of the classifier
may be of smaller complexity and, as a consequence, classifier’s learning is simpler and the difference between the
classifier learning complexities, with and without transformation may be bigger than the cost of the transformation.
This proves that real complexity is not reflected directly by structure of learning machine.

4.1. Complexity Measures by Kolmogorov and Levin

To obtain the right order in the searching queue of learning machines, a complexity measure should be used.
Kolmogorov complexity \[27, 28\] is very well defined from theoretical point of view:

\[ C_K(P) = \min_p \{l_p : \text{program } p \text{ prints } P \}, \]

(3)

where \(l_p\) is the length of program \(p\). The Kolmogorov complexity is not useful in real tasks (in particular in computa-
tional intelligence problems) also because the problem of finding a minimal program is unsolvable—the search space
of programs is unlimited and the time of program execution is unlimited. Levin’s definition \[28\] introduced a term
responsible for time consumption:

\[ C_L(P) = \min_p \{c_L(p) : \text{program } p \text{ prints } P \text{ in time } t_p \} \]

(4)

where

\[ c_L(p) = l_p + \log t_p. \]

(5)

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

In the case of Levin’s definition (Eq. 4) it is possible to realize the Levin Universal Search (LUS) \[20, 28\] but the
problem is that this algorithm is NP-hard. This means that, in practice, it is impossible to find an exact solution to the
optimization problem.

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

4.2. Complexity in the Context of Machines

Each meta-learning algorithm may be seen in the following way: it postulates some learning machines and pro-
vides appropriate tests to estimate the quality of each learning machine. But in case of time-restricted learning there is
a time limit \(t\). In such case, testing all postulated learning machines may be impossible because summed testing times
of all the candidates may exceed the time limit \(t\). It is really big problem, because huge number of learning machines
can be easily proposed. The problem of meta-learning algorithms is defining which learning machines should be
chosen for testing and which should be rejected.

Assume that we have restricted learning problem \(\mathcal{P}\) with time limit \(t\) and a meta-learning algorithm postulates
learning machines \(m_1, m_2, \ldots, m_q\), with testing times of \(t_1, t_2, \ldots, t_q\) respectively. When not all tests can be done within
the time limit (\(\sum t_i > t\)) and if we assume that each test is equally promising, then to maximize the expected value of
the best test result obtained within the time, we need to run as many tests as possible. Therefore, an optimal order
of the tests is an order \(i_1, i_2, \ldots, i_m\) of nondecreasing testing times:

\[ t_{i_1} \leq t_{i_2} \leq \ldots \leq t_{i_m}, \]

(6)

and the choice of the first \(m\) shortest tests such that

\[ m = \arg \max_{1 \leq k \leq q} \left[ \sum_{i=1,...,k} t_i \leq t \right], \]

(7)
is an optimal choice of respective learning machines. The proof of this property is trivial.

This property is very important because it clearly answers the main problem of MLA about the choice and order of learning machine tests.

Even if we do not assume that learning machines are not equally promising, the complexity measure may be adjusted to maximize the expected value of test results (see Eq. 10).

The problem of using Levin’s additional term of time, in real applications, is that it is not rigorous enough in respecting time. For example, a program running 1024 times longer than another one may have just a little bigger complexity (just +10) when compared to the rest (the length).

On the other hand, total rejection of the length \( l_p \) is also not recommended because always machines have to fit in memory, which is limited too. Besides, the length control is in compliance with the commonly accepted rule of Occam’s razor.

In learning machines, typically time complexity exceeds significantly memory complexity (the length of machine data). This is why I compose new measure of complexity combining time and memory complexity:

\[
c_a(p) = l_p + t_p / \log t_p.
\] (8)

Although the time part may seem to be respected less than the memory part, it is not really so, because, as mentioned above, almost always time complexity exceeds significantly memory complexity. Thus, in fact, memory part dominates anyway, so the order of performing tests is very sensitive to time requirements. What’s more, the order of machines based on the time \( t_p \) and based on \( t_p / \log t_p \) are equivalent. The length component \( l_p \) just keeps in mind the limited resources of memory and the whole \( c_a \) defines some balance between the two complexities.

Why not to use the \( O(\cdot) \) complexity? If we analyze the complexity of an algorithm in computer science, typically, the word complexity is bound with the \( O(\cdot) \) (bigO) as the pessimistic complexity, whose definition is straightforward. In the context of the presented problem this complexity is not enough because \( O(n) = O(100n) \) (and so on). For practical reasons it means that the computation of classification by two different machines may have the same bigO complexity but the real time will differ significantly.

Complexity measure and quarantine: 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. The approximation methodology is described in Section 4. Because of this approximation and because of the halting problem (we never know whether given test task will finish) an additional penalty term is added to the above definition:

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

where \( q(p) \) is a function term responsible for reflecting reliability of \( c_a(p) \) estimate and is used to correct the approximation in case it was wrong. In other case it is equal to 1.

At start the MLAs use \( q(p) = 1 \) (generally \( q(p) \leq 1 \)) in the estimations, but in the case when the estimated time (as a part of the complexity) is not enough to finish the program \( p \) (given test task in this case), the program \( p \) is aborted and the reliability is decreased. The aborted test task is moved to a quarantine according to the new value of complexity reflecting the change of the reliability term. This mechanism prevents from running test tasks for unpredictably long time of execution or even infinite time. Otherwise the MLA would be very brittle and susceptible to running tasks consuming unlimited CPU resources. More details on this are presented in Section 3.3.

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

\[
c_m(p) = [l_p + t_p / \log t_p] / [q(p) \cdot a(p)].
\] (10)
a(p) reflects the attractiveness of the test task p.

While c_a(p) was a measure aimed at such order of the tasks that maximizes the expected value of results, the factor 1/a(p) is a correction to the initial assumption of equal a priori probabilities in optimal ordering of different programs (or machines).

4.2.1. Complexity based test task ordering vs. generalization and the bias-variance dilemma

The complexity based task ordering is not in contrary with the generalization control in the presented system. The presented meta-learning approach separates the general definition of the problem goal (see section 3.1) from the root of the meta-learning, so that the meta-learning is parametrized by the goal of the problem. The test template, together with the quality estimation method (as an element of the meta-learning configuration) defines how to estimate the quality for each analyzed learning machine. Therefore to obtain trustful results we need a trustful definition of the test procedure. However these two elements (the test template and the quality estimation method) are elements of the meta-learning configuration, not an integral element of meta-learning. Thanks to this feature, the criterion of quality estimation may be changed without altering the meta-learning algorithm.

4.2.2. Complexity of the test task in contrary to the complexity of learning

The generator flow provides machine configurations for the test task generator, and after nesting the configurations inside the test template, all the test configurations are sent to the test task heap. The test task heap uses the complexity of the machine (for a given machine configuration), as the priority key.

It is not accidental, that the machine configuration which is inserted into the test task heap is actually the configuration of the whole machine test (in which the proposed machine configuration is nested). This complexity reflects the complete behavior of the machine very well; a part of the complexity formula reflects the complexity of the learning of the given machine, and the rest reflects the complexity of computing the test (e.g. a classification or approximation test). The costs of learning are very important, as without learning there is no model. The complexity of the testing part is also very important, because it reflects the simplicity of the further use of the model. Some machines learn quickly and require more effort to make use of their outputs (like kNN classifiers), while others learn for a long time and after that may be exploited very efficiently (as e.g. many neural networks). Therefore, the test procedure should be as similar to the whole life cycle of a machine as possible (and of course as trustful as possible).

From inputs and outputs to meta-inputs and meta-outputs. To understand the needs of the complexity computing we need to go back to the task of learning. To provide a learning machine, regardless of whether it is a simple one, a complex machine or a machine constructed to help in the process of analysis of other machines, its configuration and inputs must be specified (then a machine is a function mapping the inputs to the outputs). By a specified input, I mean the input bound to an output of another machine. Complexity computation must reflect the information from the configuration and inputs. The recursive nature of configurations, together with input–output connections, may compose quite complex information flow. Sometimes, the inputs of the submachines become known just before they are started, i.e. after the learning of other machines1 is finished. This is one of the most important reasons why the determination of complexity, in contrary to the actual learning processes, must be based on meta-inputs, not on exact inputs (which remain unknown). Assume a simple scene, in which a classifier TC is built from two parts: a data transformer T and a classifier C. It would be impossible to compute the complexity of the classifier C based on its inputs, because one of the inputs is taken from the output of the transformer T, which will not be known until the learning process of T is finished. Complexity computation cannot be limited to a part of the machine TC, or wait until some other machines are ready. To make the complexity computation possible we use proper meta-inputs descriptions. Meta-inputs are counterparts of inputs in the “meta-world”. Meta-inputs contain as informative descriptions of inputs as possible. The descriptions “explain” or “comment” every useful aspect of each input which could be helpful in determining of the complexity.

Learning machines (as any algorithms) use input (data set(-s) or output(-s) of other machines) to construct useful outputs (which play the roles of classifiers, transformers, etc.), compare with Figure 5. To evaluate complexity, the meta evaluator will use meta-inputs that describe the characteristics of inputs and will provide meta-outputs, which

---

1Machines which provide the necessary outputs.
describe the characteristics of outputs that would be produced by the learning machine. Additionally, the meta-evaluator returns the time and memory complexity of the learning process.

Going back to the example of TC machine, the computation of complexity of the classifier C will be possible on the basis of its meta-inputs. Meta-input of the meta-evaluator of the classifier C is the meta-output of the meta-evaluator of the transformer T. First, the complexity of the transformer T is calculated and then the complexity of the classifier C can be computed as well. And finally, overall complexity of the classifier TC is a sum of those complexities. Additionally, the meta-evaluator of TC has to provide a meta-output which is responsible for calculating the complexity of the classification using TC, but now this task is also simple: it is a sum of the complexity of the transformation by T and the complexity of the classification by C.

![Figure 5: A machine has inputs and outputs. The meta evaluator needs meta-inputs and meta-outputs.](image)

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

To facilitate recurrent determination of complexity—which is obligatory due to the choice of a recurrent definition of a machine configuration and a recurrent structure of the real machines—the functions which compute complexity, must also provide meta-outputs, because such meta-outputs will play a crucial role in the computation of the complexities of other machines (those that read the outputs of the assessed machine).

In conclusion, a function computing the complexity of a machine \( L \) should be a transformation

\[
P_L : \mathcal{K}_L \times \mathcal{M}_+ \rightarrow \mathbb{R}^2 \times \mathcal{M}_+ ,
\]

where the domain is composed of the configuration space \( \mathcal{K}_L \) and the space of meta-inputs \( \mathcal{M}_+ \), and the results are the time complexity, the memory complexity and the appropriate meta-outputs.

The problem is not as easy as the form of the function in Eq. 11. Finding the right function for a given learning machine \( L \) might be impossible. This is caused by the possibly unpredictable influence of some configuration elements and of some inputs (meta-inputs) on the machine complexity. Configuration elements are not always as simple as scalar values. In some cases the configuration elements are represented by functions or by subconfigurations. A similar problem concerns meta-inputs. In many cases, meta-inputs can not be represented by a simple chain of scalar values. Often, meta-inputs need their own complexity determination tool to reflect their functional form. For example, a committee of machines, which plays a role of a classifier, will use other classifiers (inputs) as “slave” machines. It means that the committee will use the classifiers’ outputs, and the complexity of using the outputs will depend on the outputs, not on the committee itself. This shows that sometimes, the behavior of meta-inputs/outputs is not trivial and that proper complexity determination requires yet another encapsulation.

4.3. Meta Evaluators

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

- to evaluate and exhibit the appropriate aspects of complexity representation, based on some meta-descriptions like meta-inputs or configuration\(^2\).
- to exhibit a functional description of complexity aspects (comments) eligible for further reuse by other meta evaluators\(^3\).

---

\(^2\)In case of a machine to exhibit complexity of time and memory.

\(^3\)In case of a machine the meta-outputs are exhibited to provide a source of information about complexity for their outputs’ readers.
To enable complexity computation, every learning machine gets its own meta evaluator.

Because of the recurrent nature of machines (and machine configurations) and because of the sophistication of the behaviour of inputs (which sometimes have complex functional forms), the meta evaluators are constructed not only for machines, but also for outputs and other elements with significant influence on the machine’s complexity.

Each evaluator will need adaptation, which can be seen as an initialization and can be compared to the learning of a machine. In such meaning the process \( D_{L'} \) (Eq. 11) will be the typical adaptation of an evaluator devoted to the machine \( L' \). It means that before using the given evaluator, it has to be adapted. Then, the evaluator can be used to calculate the aspects of complexity devoted to the given evaluator (compare with the typical evaluators type and their functionality presented below).

It is sometimes necessary to estimate complexity on the basis of the machine configuration and real inputs (not meta-inputs as in Eq. 11). In such case, we will need an adaptation of machine evaluator in the form:

\[
D_{L'} : \mathcal{X} \times \mathcal{I}_+ \rightarrow \mathbb{R}^2 \times \mathcal{M}_+,
\]

where \( \mathcal{I}_+ \) is the space of machine \( L' \)’s inputs. Such approach would require the construction of two evaluators for each machine: for the forms presented in Eq. 11 and Eq. 12. But it is possible to resign from the Eq. 12 form. The solution is to construct output evaluators and their adaptation as:

\[
D_o : \mathcal{I}_1 \rightarrow \mathcal{M}_1,
\]

where \( \mathcal{I}_1 \) is a space of outputs and \( \mathcal{M}_1 \) is the space of meta-outputs. Now we can see that meta-input (or meta-output) is nothing more than a special kind of an evaluator, the output evaluator.

Using the output evaluators, the “known” inputs can be transformed into meta-inputs (\( \mathcal{X} \times \mathcal{I}_+ \rightarrow \mathcal{X} \times \mathcal{M}_+ \)), and after that, the machine evaluator of the form of Eq. 11 can be used. This, in the end, reduces the need of adaptation in the form of Eq. 12.

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

\[
D_{obj} : \mathcal{O} \mathcal{B} \mathcal{I} \rightarrow \mathcal{M}_{obj},
\]

where \( \mathcal{O} \mathcal{B} \mathcal{I} \) is the space of nontrivial objects and \( \mathcal{M}_{obj} \) is their evaluators space (which is, naturally, the subspace of all evaluators).

The adaptation process is the major functionality of each evaluator and depends on the type of the evaluator and the parameters of the adaptation function. In general, the goal of this functionality is to use appropriate initialization data as the “source of information” for a given evaluator.
The data varies in type, goal and other aspects, depending on the type of the evaluator (compare Eq. 11, 13, and 14):

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

When the evaluators may be defined in an analytical way (quite rare cases), the evaluators only need to be adapted. In other cases, the approximation framework is used to construct the evaluators (see Figure 6 and Section 4.4). The process of creation of the evaluators is presented in Section 4.4.2.

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

4.3.1. Machine Evaluator

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

- **Declarations of output descriptions**: If a given machine provides outputs, then also the output evaluators, devoted to this machine type, must provide meta-descriptions of the outputs. The descriptions of outputs are meta evaluators of an appropriate kind (for example meta-classifiers, meta-transformers, meta-data etc.). Output description may be the machine evaluator itself or a subevaluator produced by the machine evaluator, or the evaluator provided by one of submachine evaluators constructed by the machine evaluator (machine may create submachines, evaluators may create evaluators of submachines based on their configuration and meta-inputs).

- **Time & Memory**: The complexities defined by Eq. 4, 8–10 make use of the program’s length and time. Here, the two quantities must be provided by each machine evaluator to enable proper computation of time and memory complexity.

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

4.3.2. Classifier Evaluator

The evaluator of a classifier output has to provide the time complexity of classification of an instance. Apart from the learning time of the given classifier, the time consumed by the instance classification routine is also very important in the calculation of the complexities. To estimate the time requirements of a classifier test machine, one needs to estimate the time requirements of the calls to the machine classification function. The final time estimation depends on the classifier and on the data being classified. The responsibility to compute the time complexity of the classification function lays on the meta classifier’s side (the evaluator of the classifier). Consider a classification committee: to classify data, it needs to call a sequence of classifiers to get the classification decisions of the committee members. The complexity of such classification, in the most natural way, is a sum of the costs of classification using the sequence of classifiers, and a (small) overhead reflecting the scrutiny of the committee members’ results to make the final decision. Again, the time complexity of data classification is crucial to estimate the complexity and must be computable.
4.3.3. Approximator Evaluator

The evaluator of an approximation machine has exactly the same functionality as the evaluator of a classifier, except for the fact that the approximation time is considered instead of the classification time.

4.3.4. Data Transformer Evaluator

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

4.3.5. Metric Evaluator

The machines that use metrics usually allow to set the metric at the configuration stage (e.g. kNN or SVM). As parameters of machine configurations, metrics have significant influence on the complexity of the machine while not being separate learning machines. In such cases the most reasonable way to enable complexity computation is to reflect the dependence on the metric within the evaluators (one evaluator per metric). The meta-evaluators for metrics provide the functionality of time complexity computation. Such evaluators are utilized by machine evaluators for machines using distance computation, or meta-outputs corresponding to outputs providing a metric.

4.3.6. Data Evaluators

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

4.3.7. Other Evaluators

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

4.4. Learning Evaluators

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

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

The complexity control of task starting in meta-learning does not require very accurate information about tasks complexities. It is enough to know whether a task needs a few times more of time or memory than another task. The differences of several percent are completely out of interest here. Assuming such level of accuracy of complexity computation, we do not lose much, because meta-learning is devoted to start many test tasks and small deviations from the optimal test task order are irrelevant. Moreover, although for some algorithms the approximation of complexity is not absolutely accurate, the quarantine (see Section 3.4) prevents from capturing too much resources by a single long-lasting task.
Using the approximation framework, a meta evaluator can learn as many aspects of the machines behaviour as necessary. An evaluator that uses the approximation framework can estimate an unlimited set of quantities that may be useful for the determination of complexities of some elements or some quantities for further computation of complexities. For example, the evaluator of each machine has to provide time and memory complexities. The evaluator will realize it with two approximators. Additionally, in the case, when the machine corresponding to a given evaluator is also a classifier, the classification time may be learned as well, within the same framework. The approximators are constructed, learned and used (called to approximate) automatically, according to appropriate declarations in the evaluators, as it will be seen later (in the examples of evaluators). There is no manual intervention needed in the approximator building process.

Naturally, before an evaluator is used by a meta-learning process, all its approximators must be trained. The learning of all evaluators may be done once, before the first start of meta-learning tasks. Typically, the learned evaluators reside in an evaluators project which is loaded before the MLA starts its job. If the system is extended by a new learning machine and a corresponding evaluator, the evaluator (if it uses the approximation framework) has to learn, and will also reside in the evaluators project for further use. This means that every evaluator that uses the approximation engine has to be trained only once.

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

4.4.1. Meta Evaluators’ Approximation Framework

The construction of a learnable evaluator (an evaluator making use of approximation framework) differs from the construction of a plain evaluator (compare with Figure 6). The approximation framework enables the construction of series of approximators for single evaluators. The approximators are functions that approximate a real value on the basis of a vector of real values. They are learned from examples, so before the learning process, the learning data has to be collected for each approximator.

Figure 7 presents the general idea of creating the approximators for an evaluator. To collect the learning data, appropriate information is extracted from the observations of the machines’ “behavior”. To do this an “environment” for machine monitoring must be defined. The environment configuration is sequentially adjusted, executed and observed (compare with the data collection loop in the figure). Observations bring the subsequent instances of the training data (corresponding to the current state of the environment and expected approximation values). Changes of the environment facilitate the observations of the machine in various circumstances and gathering diverse data describing machine behavior in different contexts.

The environment changes are determined by an initial representation of the environment (the input variable startEnv) and a specialized configurable scenario, which defines how to modify the environment to get a sequence of machine observation configurations i.e. configurations of the machine being examined is nested in a more complex machine structure. The scenarios produce series of complex machine configurations by changing the starting configuration.

Generated machine observation configurations should be as realistic as possible—the information flow similar to the expected applications of the machine, which enables better approximation of the desired complexity functions. Each time, a next configuration ‘oc’ is constructed, machines are created and ran according to ‘oc’, and when the whole project is ready, the learning data is collected.

When a generation of a new machine observation configuration in the data collection loop fails, the data collection stage is finished. Now, each approximator can be learned from the collected data and after that, the evaluator can use them for complexity prediction. Now the prediction will depend on an appropriate configuration and/or meta-inputs (depending on the evaluator type)—according to Eq. 11, 13, 14.

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

- defining the information necessary to collect training data for the learning of approximators,
- using the learned approximators.
In consequence each evaluator has to provide elements (functions) necessary for data collection (sequence of in-out pairs) for approximators learning based on observed environments. On the base of such functionalities the approximation framework prepares the adaptation process of approximators which will be a base for the construct of functionality as it was presented in Section 4.3 and in Eq. 11, 13 and 14.

The approximators can be constructed to estimate:

- time and memory complexity of a machine,
- time and memory complexity of particular machine methods (like classification, information selection, data transformation, etc.),
- other quantities (not known at general level, but necessary for the description of specific behaviours of a machine).

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

<table>
<thead>
<tr>
<th>Level</th>
<th>Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>level 1</td>
<td>1 — Approximators of other quantities</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>level k</td>
<td></td>
</tr>
<tr>
<td>level k+1</td>
<td>2 — Approximators of time &amp; memory complexity for specially defined methods</td>
</tr>
<tr>
<td>level n-1</td>
<td></td>
</tr>
<tr>
<td>level n</td>
<td>3 — Approximators of time &amp; memory complexity of machine</td>
</tr>
</tbody>
</table>
The order of levels reflects the order of data collection (in each iteration of the data collection loop) and of the further learning of approximators (after data collection). Using the approximators from the first layer may be helpful for composing input vectors for the next two layers. This functionality is used only for advanced evaluators (not too common, but sometimes very helpful) and will not be described here in more detail.

The functions from each layer have the same general goal: to collect a single input-output pair of data for learning appropriation based on the information extracted from a snapshot of the observed environment.

The collection of the learning data for the first layer is easy to interpret: they compose $k$-sequences of input–output pairs.

The number of levels in the second layer is auto-defined by the number of methods used to observe. For each level of the second layer, the approximation framework automatically approximates the time of execution and sizes of all returned data.

The last layer is designed for learning the time and memory complexity of the machine. This layer is used only for machine evaluators. In this case, the approximation framework automatically tests the learning time and memory usage. These quantities compose the output vectors. The input vector is obtained, as for both previous layers.

Environments for approximators learning. As shown in Figure 7 and described above, to build input data tables, necessary for training the approximators, the machine is observed in a changing environment. Each change in the environment results in a single input–output pair of data for each approximator. Therefore, to construct successful complexity evaluators, apart from specifying the necessary approximators, one needs to define the environment and the way of its manipulation.

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

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

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

**Observation scenario:** defines a scenario, which goal is to provide different configurations derived from the Approximation configuration to explore the space of machine observation configurations. For example, in the case of the kNN machine, the scenario may browse through different values of the number of neighbors $k$ and different metric definitions.

**Machine group:** encapsulates a few functionalities, which extends the space of observed configurations. The groups of functionalities are shared between evaluators of the same type, which simplifies the process of defining evaluators. Each group is characterized by:

**Several data sets:** defines file names of learning data which will be used to observe behavior of the learning process.

**Approximation-group template:** defines the procedure of using a given type of machines as a configuration template with a placeholder for an approximation configuration. For example, it may consist of two elements in a scheme: a data multiplier which constructs learning data as a random sequence of instances and features from a data set provided as an input, and a placeholder for a classifier (an empty scheme to be replaced by a functional classification machine).

**Group-scenario:** the scenario of configuration changes for the approximation-group template. The environment is subject to changes of the data sets and configuration changes defined by the Group-scenario.
For example, this scenario may cooperate with a machine randomizing the learning data within the Approximation-group template as it was already mentioned.

All the functionalities described above, used together, provide a very flexible approximation framework. Evaluators can be created and functionally-tuned, according to the needs, supplying important help for successfully computing complexities.

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

4.4.2. The Creation and Learning of Evaluators

After presenting the idea of the approximation framework for evaluators, here I present the algorithms which construct evaluators. Before any meta-learning algorithm is run, a dictionary of evaluators must be created.

The creation of a (ready-to-use) evaluator starts with the creation of an instance (object) the of given evaluator class, in accordance with the type of the given learning machine (\texttt{machineType}, see code line 31).

```plaintext
30 function CreateEvaluator(machineType);
31 evaluator = getEvaluatorInstanceFor(machineType);
32 if(evaluator is ApproximableEvaluator)
33 {
34    sequenceOfDatasets = CreateDataTablesForApprox(machineType);
35    listOfApprox = {};
36    for (level=1 to evaluator.LevelsCount)
37    {
38        <TRS, TES> = GetTrainTestDataTablesFor(level, sequenceOfDatasets);
39        approxArray = TrainApproximatorArray(<TRS, TES>);
40        listOfApprox.Append(approxArray);
41    }
42    evaluator.Approximators = listOfApprox;
43 }
44 return evaluator;
45 end
```

If the evaluator does not use the approximation framework, then it is ready for use (without learning) and may be called to estimate complexities. Otherwise, learning of appropriate approximators is performed. Line 34 calls a function (described later) which creates a sequence of learning data tables, according to the meta-description of the evaluator.

Next lines (36–41), for each level, prepare data tables and start learning a vector of approximators. The vector of approximators is appended to the list \texttt{listOfApprox} and finally assigned to the evaluator (in the line 42).

The function \texttt{CreateDataTablesForApprox} plays a crucial role in the complexity approximation framework, as it constructs learning data tables for the approximators. In the line 47, the machine type is converted to information about \textit{Machine group}, \textit{Observation scenario}, \textit{Approximation configuration} and \textit{Approximation methods}.

```plaintext
46 function CreateDataTablesForApprox(machineType);
47 [machineGroup, Scenario, ApproximationConfig, ApproximationMethods] = machineType;
48 foreach (dataset in machineGroup.Several_data_sets)
49 {
50    groupScenario = machineGroup.GroupScenario;
51    groupScenario START FROM machineGroup.ApproximationGroupTemplate;
52    foreach (gconfig in groupScenario)
53    {
54        scenario = Scenario;
55        scenario START FROM ApproximationConfig;
56        foreach (sconfig in scenario)
57        {
58            c = PLACE sconfig in gconfig;
59            t = StartTestTask(c, dataset);
60        }
61    }
62}
```
foreach (level in layer1) // layer 1 (other quantities)
dtInOut[level] += InOut_Pair(t, level);

foreach (meth in ApproximationMethods) // layer 2 (methods)
{
  <time, memorySizes> = CheckMethod(meth, t);
dtInOut[level++] += InOut_Pair(t, time, memorySizes);
  if (evaluator is a machine) // layer 3 (machine)
dtInOut[level++] += InOut_Pair(t.time, t.memorySize);
}

sequenceOfDatasets = resplit(dtInOut);
return sequenceOfDatasets;

The observations are performed for each dataset (the loop in line 48), for each group a configuration gconfig generated by the appropriate group scenario (the loop in line 52), for each machine a configuration sconfig generated by the scenario (the loop in line 56). The scenarios are initialized before they provide the configurations (see lines 51 and 55).

In the line 58, the configuration sconfig is placed within the group configuration gconfig. This composes a configuration c which defines the observation test task t (next line).

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

At the end (line 72), the input and target parts are transformed into appropriate data tables, to be returned by the function.

Examples of evaluators: The composition of evaluators for specific machines is very interesting from the point of view of meta-learning. It may be very successful while reflecting deeper knowledge about the specificity of a given type of a learning machine. A few examples of evaluators are provided in http://www.is.umk.pl/norbert/publications/11-njCmplxEval-Evals.pdf. I'm convinced that inquiring readers will find them valuable. In case the reader would be interested in details about implementation of other parts of presented ideas, please contact via email: norbert@is.umk.pl. The whole system is too complex to describe every detail in an article or even in supplementary materials. The presented meta-learnings were realized inside the data mining system Intemi [14].

5. Meta-learning in Action

The meta-learning algorithm, or rather meta-learning system presented here may be used in various ways. The generator flow may be defined as a simple graph, but usually, for advanced problems, it is a quite nontrivial graph, which, in effect, produces the observation test task t (next line).

The goal of meta-learning, which reflects the problem type, may also be defined in several ways, according to the needs. Similarly, the stop criterion should reflect the preferences about the conditions of declaring the meta-search finished.

The main goal of this section is to prove that a meta-learning system can approximate complexity with satisfactory precision and that the order based on complexity helps to find good models within short time.

An outline of the further part of this section: First part of this section concentrates on the configuration of a meta-learning system (the meta-learning test template, quality estimation measure, stop-criterion and generator flow). Later the benchmark data is shortly described, followed by a table which presents machines generated with the presented generator flow and ordered according to their complexities. The further part describes several details from the course of the meta-learning system work which help to see the predicted and the real complexity of the generated machines and their accuracies.

To present meta-learning in action, I have used a few well known problems from the UCI Machine Learning repository [10]. All the benchmarks, presented below, are classification problems. All the results presented below
have been computed using the same configuration of meta-learning (obviously except for the specification of the benchmark dataset).

First, I have to present the meta-learning configuration. The configuration consists of several elements: the meta-learning test template, query test, stop criterion and the generator flow.

**Meta-learning test template.** The test template exhibits the goal of the problem. Since the chosen benchmarks are classification problems, we may use cross-validation as the strategy for estimation of the classifiers’ capabilities.

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

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

**Generator flow.** 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 see the very interesting details of its cooperation with the complexity control mechanism more clearly. To find more sophisticated configuration machines, a more complex generators graph should be used. Anyway, it will be seen that using even such basic a generator flow, the results ranked high by the MLA can be very good. The generator flow used in experiments is presented in Figure 3 on page 6.

To know what exactly will be generated by this generator flow, the configurations (the sets) of classifiers generator and rankings generator need to 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 ExpectedClass⁴ machine transforms the original dataset, then the transformed data become the learning data for kNN,
- [LVQ, kNN (Euclidean)] — first, Learning Vector Quantization algorithm [26] 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 generator flow presented in Figure 3 (Page 6), 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 2. The square brackets, used there, denote submachine relations. 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.

⁴ExpectedClass 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.
To make the notation easier to read, I explain some entries of the table. The notation does not present the input–output interconnections, so it does not allow to reconstruct the full scenario in detail, but shows machine structure, which is sufficient here, and significantly reduces the occupied space.

The following notation:

\[
\text{[[[RankingCC], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]}
\]

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

Notation:

\[
\text{[[[RankingCC], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]}
\]

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

The following notation represents the meta-parameter search machine (ParamSearch) which optimizes parameters of a kNN machine:

ParamSearch [kNN (Euclidean)]

The meta-parameter search machine was shortly described on page 6.

In the case of

ParamSearch [LVQ, kNN (Euclidean)]

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

However, in the case of a notation

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

only the number of chosen features is optimized because this configuration is provided by the MPS/FS of Transform & Classify Generator (see Figure 3), where the ParamSearch configuration is set up to optimize only the parameters of feature selection machine. Of course, it is possible to optimize all parameters of all submachines, but this is not the goal of the example and, moreover, the optimization of too many parameters may result in producing overly complex machines (sometimes uncomputable in a rational time).

Data benchmarks. Table 1 summarizes the properties of data benchmarks (from the UCI repository) selected for the presentation. I do not present an exhaustive set of results, as it is not the goal of the paper. Here, I do NOT focus on the obtained accuracy of the models, but on the rightness of complexity-based task ordering.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># classes</th>
<th># instances</th>
<th># features</th>
<th># ordered f.</th>
</tr>
</thead>
<tbody>
<tr>
<td>mushroom</td>
<td>2</td>
<td>8124</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>german-numeric</td>
<td>2</td>
<td>1000</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>glass</td>
<td>6</td>
<td>214</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>splice</td>
<td>3</td>
<td>3190</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>thyroid-all</td>
<td>3</td>
<td>7200</td>
<td>21</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 1: Benchmark data discussed here.

Table 2 presents exact complexities (see Eq. 9) for each test machine configuration obtained for the mushroom data. The table has three columns: the first one contains the task id which reflects the order in which the configurations are provided by the generator flow, the second column is the task configuration description, and the third column shows the task complexity. The rows are sorted according to the complexity.
Table 2: Complexities of the tasks produced by the generator flow for mushroom data.

<table>
<thead>
<tr>
<th>Task Description</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

22 [[[RankingFScore], FeatureSelection], [NFC], TransformAndClassify]            | 5.99E+006      |
| 13 [[[RankingCC], FeatureSelection], [NFC], TransformAndClassify]               | 5.91E+006      |
| 4 NFC                                                                            | 6.50E+006      |
| 31 ParamSearch [NFC]                                                            | 6.72E+006      |
| 7 ExpectedClass, kNN [MetricMachine (EuclideanOUO)]                              | 8.79E+006      |
| 25 [[[RankingFScore], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify] | 9.21E+006      |
| 16 [[[RankingCC], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify] | 9.22E+006      |
| 19 [[[RankingFScore], FeatureSelection], [kNN (Euclidean)], TransformAndClassify] | 1.12E+007      |
| 10 [[[RankingCC], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]  | 1.13E+007      |
| 26 [[[RankingFScore], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify] | 2.11E+007      |
| 17 [[[RankingCC], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify] | 2.11E+007      |
| 27 [[[RankingFScore], FeatureSelection], [Boosting (10x) NBC]], TransformAndClassify] | 5.41E+007      |
| 18 [[[RankingCC], FeatureSelection], [Boosting (10x) NBC]], TransformAndClassify] | 5.41E+007      |
| 34 ParamSearch [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]]               | 8.91E+007      |
| 8 [LVQ, kNN (Euclidean)]                                                        | 9.40E+007      |
| 9 [kNN (Euclidean)]                                                             | 1.04E+008      |
| 36 ParamSearch [Boosting (10x) NBC]                                              | 1.04E+008      |
| 20 [[[RankingFScore], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify] | 1.79E+008      |
| 11 [[[RankingCC], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify] | 1.79E+008      |
| 49 ParamSearch [[[RankingFScore], FeatureSelection], [NFC], TransformAndClassify] | 1.81E+008      |
| 40 ParamSearch [[[RankingCC], FeatureSelection], [NFC], TransformAndClassify]  | 1.81E+008      |
| 52 ParamSearch [[[RankingFScore], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify] | 2.03E+008      |
| 43 ParamSearch [[[RankingCC], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify] | 2.03E+008      |
| 21 [[[RankingFScore], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify] | 2.91E+008      |
| 12 [[[RankingCC], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify] | 2.91E+008      |
| 46 ParamSearch [[[RankingFScore], FeatureSelection], [kNN (Euclidean)], TransformAndClassify] | 3.17E+008      |
| 37 ParamSearch [[[RankingCC], FeatureSelection], [kNN (Euclidean)], TransformAndClassify] | 3.18E+008      |
| 1 [kNN (Euclidean)]                                                             | 3.56E+008      |
| 2 [kNN [MetricMachine (EuclideanOUO)]]                                           | 3.61E+008      |
| 53 ParamSearch [[[RankingFScore], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify] | 5.04E+008      |
| 44 ParamSearch [[[RankingCC], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify] | 5.04E+008      |
| 24 [[[RankingFScore], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify] | 1.01E+009      |
| 15 [[[RankingCC], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify] | 1.01E+009      |
| 23 [[[RankingFScore], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify] | 1.08E+009      |
| 14 [[[RankingCC], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify] | 1.08E+009      |
| 54 ParamSearch [[[RankingFScore], FeatureSelection], [Boosting (10x) NBC]], TransformAndClassify] | 1.08E+009      |
| 45 ParamSearch [[[RankingCC], FeatureSelection], [Boosting (10x) NBC]], TransformAndClassify] | 1.08E+009      |
| 6 LinearSVMClassifier [LinearKernelProvider]                                    | 1.65E+009      |
| 33 ParamSearch [LinearSVMClassifier [LinearKernelProvider]]                     | 1.65E+009      |
| 3 [kNN [MetricMachine (Mahalanobis)]]                                           | 1.97E+009      |
| 5 SVMClassifier [KernelProvider]                                                | 2.40E+009      |
| 29 ParamSearch [kNN [MetricMachine (EuclideanOUO)]]                             | 2.48E+009      |
| 47 ParamSearch [[[RankingFScore], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify] | 3.75E+009      |
| 38 ParamSearch [[[RankingCC], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify] | 3.75E+009      |
| 48 ParamSearch [[[RankingFScore], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]]], TransformAndClassify] | 6.06E+009      |
| 39 ParamSearch [[[RankingCC], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify] | 6.06E+009      |
| 35 ParamSearch [LVQ, kNN (Euclidean)]                                            | 8.19E+009      |
| 30 ParamSearch [kNN [MetricMachine (Mahalanobis)]]                              | 1.35E+010      |
| 28 ParamSearch [kNN (Euclidean)]                                                | 1.87E+010      |
The results obtained for the benchmarks are presented in the form of diagrams. The diagrams are very specific and present many properties of the meta-learning algorithm. The diagrams present information about the times of starting, stopping and breaking of each task, about complexities (global, time and memory) of each test task, about the order of the test tasks (according to their complexities, compare Table 2) and about the accuracy of each tested machine.

In the middle of the diagram—see the first diagram in Figure 8—there is a column with task ids (the same ids as in table 2). Row ordering reflects the complexities of test tasks, but is inversed in respect to that of table 2: at the top, the most complex tasks are placed and at the bottom the task of the smallest estimated complexity (ran first by the system).

On the right side of the Task-id column, there is a plot presenting the starting, stopping and breaking times of each test task. As it was presented in Section 3.3 the tasks are started according to the approximation of their complexities, and with time limits defined on the basis of the time complexity (see Section 4.2). When the task does not end within the limit, it is broken and restarted according to the modified complexity (see Section 4.2). For an example of a restarted task please look at Figure 8, at the third topmost task-id 50—there are two horizontal bars corresponding to two periods of the task run. The break means that the task was started, broken because of exceeding the allocated time and restarted when the tasks of larger complexities got their turn. A survey of different diagrams (in Figures 8–12) easily brings the conclusion that the amount of inaccurately predicted time complexities is quite small (there are quite few broken bars). Note that, when a task is broken, its subtasks that have already been computed are not recalculated during the test-task restart (thanks to the machine unification mechanism and machine cache). It can be perfectly seen by the example of above-mentioned task 50: when the task was restarted it reused all the subtasks finished during the first pass so that the second bar is very thin, meaning that very little time was consumed after the restart.

At the bottom, the Time line axis can be seen. The scope of the time is the interval [0, 1] to show the times relative to the start and the end of the whole MLA computations. To make the diagrams clearer, the tests were performed on a single CPU, so only one task was running at a time and we can not see any two bars overlapping in time. If I had run the projects on more than one CPU, a number of test tasks 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 well with the relations between real time consumed by the tasks—the number of restart is small and the overall length of blue rectangle increase from bottom to top. While browsing other diagrams, a similar behavior can be observed—the simple tasks are started at the beginning and then, the more and more complex ones.

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

The leftmost column of the diagram presents ranks of the test tasks (the ranking of the accuracies). In the case of the mushroom data, it is not difficult to obtain 100% accuracy, so there are many machines with rank 1.

Between the columns of Task-id and accuracy-ranks, on top of the gray bars corresponding to the accuracies, some thin solid lines can be seen. The lines start at the right side as the accuracy bars and go to the right according to proper magnitudes. For each task, the three lines correspond to the total complexity (the upper line), the memory complexity (the middle line) and the time complexity (the lower line)\(^5\). All three complexities are the approximated complexities.

\(^5\)In the case of time complexity the \(t/\log t\) is plotted, not the time \(t\) itself.
(see Eq. 9 and 10). Approximated complexities presented on the left side of the diagram can be easily compared visually to the time-schedule obtained in the real time on the right side of the diagram. Longer lines mean higher complexities. The longest line is spread to maximum width. The others are proportionally shorter. So the complexity lines at the top of the diagram are long while the lines at the bottom are almost invisible. It can be seen that sometimes the time complexity of a task is smaller while the total complexity is larger and vice versa. For example see tasks 28 and 51 again in Figure 8.

The meta-learning illustration diagrams (Figures 8–12) clearly show that the behavior of different machines changes between benchmarks. Even the standard deviation of accuracies is highly diverse. When looking at accuracies within some test, groups of machines of similar accuracy may be seen, however, for another benchmark, within the same group of machines the accuracies are quite variant. Of course, the complexity of a test task for a given configuration may change significantly from benchmark to benchmark. However it can be seen that in the case of benchmarks of similar properties (by similar I mean similar number of instances and features in benchmark), the permutations of task ids in the diagrams are partially similar (e.g. see the bottoms of Figures 8 and 12).

The most important feature of the presented MLA is that it facilitates the search for accurate solutions in the order of increasing complexity. Simple solutions are started before the complex ones, to maximize the probability that a simple and accurate solution is found as soon as possible. It is confirmed by the diagrams in Figures 8–12. Thanks to this property, in the case of a strong stop-condition (significant restriction on the running time) we are able to find really good solution(-s) because of starting the test tasks in a proper order. Even if some tasks get broken and restarted, it is not a serious hindrance to the main goal of algorithm.

For a few of the benchmarks, very simple and accurate models were found just at the beginning of the meta-learning process. Please see Figure 11 task ids 4 and 31, Figure 12 task id 19. The machines of the first three diagrams, are all single machines of relatively low complexities. But not only single machines may be of small complexity. The most accurate machine (of the 54 machines being analyzed) for the thyroid data is the combination of feature selection based on F-score with kNN machine (task id 19). Even nontrivial combinations of machines (complex structures) may provide low time and memory complexity while single machines do not guarantee small computational complexity. In the case of very huge datasets (with huge numbers of instances and features) almost no single algorithm works successfully in rational time. However classifiers (or approximators) preceded by not-overly complex data transformation algorithms (like feature selection or instance selection) may be calculated in a very short time. The transformations may reduce the costs of classifier learning and testing, resulting in significant decrease of the overall time/memory consumption.

In some of the benchmarks (see Figures 8 and 9) the most accurate machine configurations were not of as small complexity as in the cases mentioned above. The MLA running on the mushroom data, has found several alternative configurations of very good performance: the simplest is a boosting of naive bayes classifier (task id 9), the second simplest is the kNN [MetricMachine (EuclideanOUO)], followed by SVM (with Gaussian kernel). Some of the most complex machines applying ParamSearch to kNN and SVM have also finished with 100% accurate models. For the german-numeric benchmark, the best machines are SVM’s with linear and Gaussian kernels (task ids 6, 33\textsuperscript{6} and 5). The winner machines for this benchmark are of average complexity, and are placed in the middle of the diagram.

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

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

Without repeating the experiments, one can think of the results obtained with the stop criterion set to a time-limit constraint. For example, assume that the time limit was set to 1/5 of the time really used by a given MLA run. In such cases, the linear SVM itself. The small difference is a result of additional memory costs for ParamSearch encapsulation.

\textsuperscript{6}Note that 33 means ParamSearch [LinearSVMClassifier [LinearKernelProvider]] where a linear SVM is nested within a ParamSearch, but the auto-scenario for linear SVM is empty, which means that ParamSearch machine does not optimize anything and indeed it is equivalent to the linear SVM itself. The small difference is a result of additional memory costs for ParamSearch encapsulation.
a case, some of the solutions described above would not be reached, but still, for several datasets the optimal solutions would be obtained, and for other benchmarks some slightly worse solutions would be the winners. This feature is crucial, because in real life problems the time is always limited and we are interested in finding as good solutions as possible within the meaning of a restricted learning problem, where the time is always limited.

If you like to see more results please download the document at address:

5.1. Presented meta-learning approach vs. other approaches

First important disadvantage of other projects is that they do not work in the regime of time-restricted learning problems, and in consequence such systems do not guarantee any success. The presented system is the only one which will consciously starts only the test tasks which are predicted to have finished in the restricted time. It is of highest importance, as in a real application the time devoted to the search for a model is always strictly finite.

Almost all of known approaches around meta-learning neglect data transformation (preprocessing and active embedded processing), while it is known that only for simple data set data transformation is not necessary. Basing on the experience from several competitions (already mentioned in the introduction) it is straightforward that choosing appropriate data transformations is more important then the choice of a classifier.

Not only ensemble machines tend to find very complex models, while it is not rare that simple and really accurate models can be found. This is also of highest importance in case of medical applications, where invasive or too expensive tests should not by applied if not necessary. What’s more, if only the data can be characterized by simple relation we should find them. Just imagine that almost all committees will never prepare feature selection even if one of them if useful and sufficient.

Most of the known systems are also prepared in a specific way for fixed type of application and it is hard to adapt such systems to other problems. Among others, it is caused by the embedding of quality estimation in heart of system or by observation of fixed and problem-specific features.

Typically, meta-learning just recommends a model (recommendation is based on some characteristics) as a single learning machine from a fixed machine set. Even the parameters of machine configuration are not optimized. Parameters optimization is sometimes investigated by intelligent data assistant projects, but they often bear most of the above disadvantages. And, in other projects, when the search space looks more optimistic it is, it is very simple to meet combinatorial explosion of all possibilities.

Flexibility of the configuration of meta-learning process in case of the already known meta-learning systems is also strongly limited. Usually, they were designed for fixed, single type of a problem. For example those devoted to classification cannot be used (without strong reconstruction) for approximation.

Also, the type of meta-knowledge in all known projects was fixed to single type—the relation between characteristics and the classifier (if any). While more advanced meta-learning should be able to use and update different kinds of meta-knowledge (as it was introduced here).

All above mentioned problems were positively solved in the presented meta-learning system.

6. Summary and future perspectives

In recent years, my research has been concentrated on new advanced methodology of meta-learning. Now, I have put forward an advanced meta-learning algorithm performing a controlled search through the space of simple and complex learning machines. One of the most important aspects of my approach, concerns the order in which candidate machine configurations are tested. The order corresponds with the machine complexities, estimated by meta evaluators constructed with the presented algorithm. The meta evaluators are a brilliant example of meta-knowledge. An important feature of such evaluators is that they approximate complexity of simple and complex machines, including huge testing tasks hierarchies.

The complexity control I have described, is very advantageous in meta-search, because it gives the opportunity to find the most interesting, simple solutions at the beginning of the process and then proceed to the examination of more complex ones. Such control protects us from spending much time on testing some machines, while simpler and more accurate models may be easily found with the presented MLA.
Figure 8: mushroom
Figure 9: german-numeric
Figure 10: glass
Figure 12: thyroid-all
The results I have obtained with the help of the complexity control module are very successful and confirm the advantages of the approach.

Another unique feature of the presented meta-learning system is that it makes use of several types of meta-knowledge. Different kinds of knowledge cooperate successfully with different modules of the MLA. Meta-knowledge can be found in meta evaluators, in the ontology of the optimization scenarios, in machine templates, in machine generators and in the attractiveness module. In my approach it is very important that the meta-knowledge is created, verified and exploited independently in many modules of the MLA.

Thanks to the machine generator flows, which direct the search through the enormously huge space of complex machines, the MLA may be simply and efficiently configured for different learning problems. Moreover, it may be simply extended by new, more and more sophisticated machine generators, resulting in more and more sophisticated meta-search procedures. The strategy divides complex modeling of meta-learning into simpler meta-subtasks. It is simpler to compose an intelligent machine generator devoted to a chosen type of tasks than to build a single hyper-module which knows everything. As in the brain, the modules of different tasks cooperate to provide more advanced functionality.

To realize presented meta-learning system, I have created versatile and specialized data mining system Intemi, which was successfully used.

7. Appendix: Meta Parameter Search Machine

I have created a meta parameter search (MPS) machine, that can optimize any elements of the machine configuration\(^7\). The optimization process may optimize any elements of the configuration and any element of any subconfigurations (including subsubconfigurations etc.). Also subelements of objects in (sub-)configurations can be optimized. The search and optimization strategies called scenarios are realized as separate modules providing appropriate functionality.

Ontology database of optimization scenarios was made for all learning machines. Based on the collected meta-knowledge each learning machine (classifiers, approximators, transformers, etc.) has one or two dedicated optimization scenarios which are used by the MPS machine to search for the best parameters. The first scenario is of shallow search and second is deeper search.

The system contains a number of predefined scenarios. The most typical scenario, used for optimization of many learning machines’ parameters, is the StepScenario based on quite a simple idea to optimize single chosen element of a configuration. The search strategy is very simple, StepScenario generates configurations with optimized element set to values from a sequence defined by a start value, step value and the number of steps. The SetScenario is powerful in the cases where the optimized parameter is not continuously valued but its type is enumerable. Another very useful scenario is StackedScenario which facilitates building a sequence or a combination of scenarios. The MPS is also equipped with several more sophisticated scenarios.

Bibliography


\(^7\)It is possible to optimize a single parameter or several parameters during optimization, sequentially or in parallel, depending on the search strategy applied.