

# META-LEARNING ARCHITECTURE FOR KNOWLEDGE REPRESENTATION AND MANAGEMENT IN COMPUTATIONAL INTELLIGENCE

Krzysztof GRĄBCZEWSKI, Norbert JANKOWSKI

*Department of Informatics,  
Nicolaus Copernicus University, Toruń, Poland  
<http://www.is.umk.pl/>  
E-mail: {kgrabcze|norbert}@is.umk.pl*

## Abstract

There are many data mining systems derived from machine learning, neural network, statistics and other fields. Most of them are dedicated to some particular algorithms or applications. Unfortunately, their architectures are still too naive to provide satisfactory background for advanced meta-learning problems.

In order to efficiently perform sophisticated meta-level analysis, we have designed and implemented a very versatile, easily expandable system (in many independent aspects), which uniformly deals with different kinds of models and models with very complex structures (not only committees but also deeper hierarchic models).

We present our requirements and their motivations for an advanced data mining system, and describe some of our solutions facilitating advanced meta-level model management at the scope of each system component, optimization of computation time and memory consumption and much more.

**Keywords:** meta-learning, computational intelligence, learning from data, data mining, intelligent systems, software engineering

## 1 Introduction

*Learning from data* is getting more and more important as a way of knowledge discovery for many real world problems. Today nearly everything is (or

may be) represented in a digital format, hence may be analyzed using computational intelligence (CI) methods. The formalism, we introduce here is a general view of learning, but it is restricted to the most crucial ideas, to illustrate the possibilities of our system architecture. For a broader introduction to learning algorithms see [10, 5, 2, 9, 15, 16].

A *learning problem* can be defined as  $\mathcal{P} = \langle D, \mathcal{M} \rangle$ , where  $D \subseteq \mathcal{D}$  is a *learning dataset* and  $\mathcal{M}$  is a *model space*.

In computational intelligence attractive models  $m \in \mathcal{M}$  are determined with learning process:

$$L^p : \mathcal{D} \rightarrow \mathcal{M}, \quad (1)$$

where  $p$  defines the parameters of the learning machine. This view of learning encircles many different approaches of supervised and unsupervised learning tasks including classification, approximation, clustering, finding associations etc. Such definition does not limit the concept of search to specific kinds of learning methods like neural networks or statistical algorithms, however such reduction (or extension) of model space is possible in practice.

In real life problems, sensible solutions  $m \in \mathcal{M}$  are usually so complex, that it is very advantageous to decompose the problem  $\mathcal{P} = \langle D, \mathcal{M} \rangle$  into sub-problems:

$$\mathcal{P} = [\mathcal{P}_1, \dots, \mathcal{P}_n] \quad (2)$$

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

$$m = [m_1, \dots, m_n], \quad (3)$$

and the model space gets the form

$$\mathcal{M} = \mathcal{M}_0 \times \dots \times \mathcal{M}_n. \quad (4)$$

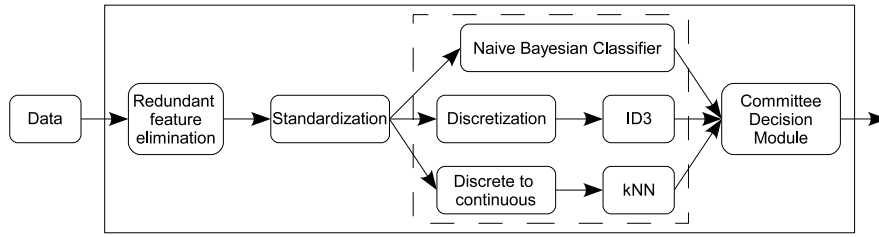
The solution constructed by decomposition is often much easier to find, because of reduction of the main task to a series of simpler tasks: model  $m_i$  solving the subproblem  $\mathcal{P}_i$ , is the result of learning process

$$L_i^{p_i} : \mathcal{D}_i \rightarrow \mathcal{M}_i, \quad i = 1, \dots, n, \quad (5)$$

where

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

and  $K_i \subseteq \{0, 1, \dots, i-1\}$ ,  $\mathcal{M}_0 = \mathcal{D}$ . It means that the learning machine  $L_i^{p_i}$  may take advantage of some of the models  $m_1, \dots, m_{i-1}$  learned by preceding



**Figure 1.** An example of a DM project.

subprocesses and of the original dataset  $D$  of the main problem  $\mathcal{P}$ . Naturally, also parameters  $p = (p_1, \dots, p_n)$ .

So, the main learning process  $L^p$  is decomposed to the vector

$$[L_i^{p_i}, \dots, L_n^{p_n}]. \quad (7)$$

Such decomposition is often very natural: a standardization or feature selection naturally precedes classification, a number of classifiers precede committee module etc. Note that, the subprocesses need not to be dependent on all preceding subprocesses, so such decomposition has natural consequences in the possibility of parallelization of problem solving, what will get even clearer a little below.

*Meta-learning* algorithms are also learning machines, however the goal of meta-learning is to find the best decomposition Eq. 2 in an automated way.

A real life example of learning process decomposition is presented in figure 1, where a classification committee is constructed, but member classifiers need the data transformed before they can be applied. The structure of the project directly corresponds to both learning process and final model decomposition. The rectangle including all others but “Data” depicts the whole learning process, which given dataset is expected to provide a classification routine. For different kinds of analysis like testing classification accuracy etc. it must be treated as a whole, but from the formal point of view each inner rounded rectangle is a separate process solving its task and providing its model.

Because each DM process is a directed acyclic graph, it is easy to show the composite process and composite model it corresponds to. The model of

figure 1 may be decomposed as

$$\begin{aligned}
m_{rfe} &= L_{rfe}(Data) \\
m_{std} &= L_{std}(m_{rfe}) \\
m_{nbc} &= L_{nbc}(m_{std}) \\
m_{discr} &= L_{discr}(m_{std}) \\
m_{id3} &= L_{id3}(m_{discr}) \\
m_{dtc} &= L_{dtc}(m_{std}) \\
m_{knn} &= L_{knn}(m_{dtc}) \\
m_{comm} &= L_{comm}(m_{nbc}, m_{id3}, m_{knn})
\end{aligned} \tag{8}$$

The subscripts are easy to decode when compared to the figure 1. Each of the components learns some part of the final model, which has a corresponding structure.

Such general and uniform foundations of our DM system facilitate solving problem of any kind, requiring any structural complexity, provided appropriate components. It is especially important when undertaking meta-learning challenges, where we must try many different methods, from simple ones to that of large complexity. Our system architecture is eligible for answering questions about which method, how configured and why, we should use to solve a given task. Such questions remain without satisfactory answers, although there are thousands of articles devoted to learning methods in computational intelligence and their modifications.

Nontriviality of model selection is evident when browsing the results of NIPS 2003 Challenge in Feature Selection [7, 6] or WCCI Performance Prediction Challenge [8] in 2006. The competitions results are an evidence that in real applications, optimal solutions are often complex models and require atypical ways of learning. Problem complexity is even more clear when solving more difficult problems in text mining or bioinformatics. Then, only when a good cooperation of submachines are obtained, we may hope for a reasonable solution. This means that for example before classification we have to prepare some transformation(-s) (and/or their ensembles) which play crucial role in further classification.

Some meta-learning approaches [14, 3, 1, 17] base mainly on data characterization techniques (characteristics of data like number of features/vectors/-classes, features variances, information measures on features, also from decision trees etc.) or on *landmarking* (machines are ranked on the basis of simple machines performances before starting the more power consuming ones). Although the projects are really interesting, they still may be done in different ways or at least may be extended in some aspects. The whole space of pos-

sible and interesting models is not browsed so thoroughly by the mentioned projects, thereby some types of solutions can not be found with them.

In our approach the term *meta-learning* encompasses the whole complex process of model construction including adjustment of training parameters for different parts of the model hierarchy, construction of hierarchies, combining miscellaneous data transformation methods and other adaptive processes, performing model validation and complexity analysis, etc.

Currently such tasks are performed by humans. Our long-range goal is to eliminate human interactivity in the processes and obtain meta-learning algorithms which will outperform human-constructed models. Here we present the framework facilitating dealing with complex learning machines in a simple and efficient manner. Section 2 enumerates the general advantages of our system over other available systems which are not eligible for advanced meta-learning tasks. In section 3 we present different aspects of machines and models management in our system and in section 4 of their complex structures. Section 5 is devoted to the ways of inter-models information extraction and exchange, and section 6 to the ease of adding components to the system: both are more technical to show not only its possibilities but also precise means. Section 7 mentions some other interesting possibilities of our architecture and summarizes presented ideas.

## **2 Why the new architecture for meta-learning was indispensable**

There are plenty of data mining software systems available today, but to provide satisfactory tools for meta-level model manipulation, we had to design and implement a new architecture. Software packages like free Weka, Yale, commercial SPSS Clementine, GhostMiner etc.—see [4, 11] for a comprehensive list—are designed to prepare and validate different computational intelligence models, but they lack most of the features listed below, which are substantial for effective meta-learning. Thereby these systems may be used like calculators in computational intelligence rather than systems which discover models in really automated and autonomous way.

Among the features of advanced systems for complex model construction we need the following (all supplied by our architecture):

- A unified view of most aspects of handling CI models, (including complex model structures) like model creation and removal, defining adaptive methods inputs and outputs and their connections, adaptive pro-

cesses execution, etc. Project management module responsible for easy control of the graph of interconnected models. Unified model and machine representation facilitating exploration of the hierarchy of models without deep knowledge about the intrinsics of its particular elements (it is crucial in advanced meta-learning).

- Easy and uniform access to learners' parameters; each method implementation is assisted by its configuration class with a standard way to adjust its fields and a possibility to describe the characteristic of the fields (linear, exponential, etc.), the scopes of sensible values, etc.
- Easy and uniform mechanisms for representation of machine inputs and outputs so that they can be managed from outside including information interchange between the processes.
- Easy and uniform access to exhaustive browsing of results of training. A repository of learners' results, providing uniform access to this information, independent of particular models, available to other learners (fundamental for meta-learners) and interactive user.
- Tools for estimation of model *relevance* (according to the goal, it may be accuracy, MSE, MAP or one of many other measures [10, 5, 2, 13]) together with an analysis of reliability, complexity and statistical significance of differences [12] to other solutions,
- Templates for configuration of complex method structures with exchangeable parts, instantiated during meta-learning.
- Versatile time and memory management to guarantee optimal usage of the resources (especially when dealing with very complex model hierarchies), also when run on a computer cluster; this includes parallel execution of independent methods, model cache systems and unification framework preventing from repeated calculations, which are very probable in massive meta-level calculations ('probable' not because of chaotic meta-search but same models can be used as parts of others more complex systems).
- Rich library of fundamental methods providing high versatility by facilitating dealing with different kinds of data and standard tools for their analysis. Loaders for "tabular" data, text data, bioinformatic sequences, microarrays, etc., miscellaneous data including standardization, feature

selection and aggregation, reference vector extraction, Principal Components Analysis, multidimensional scaling and many others. Knowledge extraction methods derived from statistics, machine learning, neural networks, etc. and solving different optimization problems (classification, regression, clustering, etc.). Fundamental methods for validation and testing, eligible also for effortless, reliable testing and comparison of complex model structures.

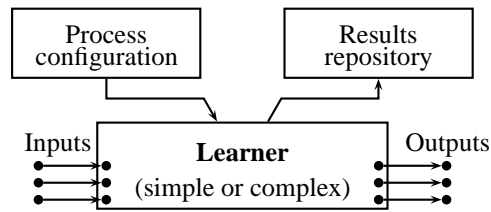
- Simple and highly versatile Software Development Kit (SDK) for programming system extensions. SDK users should define just the essential parts of their methods with as little code as possible and with minimum system-specific overhead.
- Tools for fast and easy on-line definition of some small extensions of the system like new metrics, new feature ranking algorithms etc.
- Readable and intuitive graphical user interface. Boxes with clearly marked inputs and outputs, arrows displaying the flow of information and context-dependent way of setting machines parameters.

All these ideas (and some other) constituted our strong need for a new system and have been carefully incorporated into our system. Therefore, it is adequate not only for simple model building and testing, but also for advanced meta-learning approaches, very efficient and versatile at different levels of abstraction.

### **3 Machines and models**

As defined in the introduction the term *learning machine* (*machine* or *learning method*) denotes *adaptive algorithms*, while *model* means the final result of adaptation process of such algorithm. Although the term model is often understood as a representation of some “fully-functional” model performing approximation, classification or other tasks (e.g. a neural network, a decision tree, a k Nearest Neighbors model, etc.), there is no point in such a restriction. In accordance with the introductory formalism, any algorithmic part with clear inputs and outputs fits the definition of learner, so that it includes the processes of testing other machines, loading and transforming data as well as any separate part of a machine or any hierarchy of learning machines.

For example a cross-validation test can be treated as an learning machine, because it also performs some calculations to gather some information which



**Figure 2.** Abstract view of an adaptive process.

as a result of a method may be called a model—in this case it is the information about series of results obtained with some adaptive processes. The output it generates can also be an input for other models: for example some algorithms controlling statistical significance of differences between different methods results.

The very popular technique of distinguishing the stages of data preprocessing and final model construction within knowledge acquisition process does not find any confirmation in the unified theoretical view of the process. The border between data transformations and final model is vague and gets completely blurry when we exploit meta-learning techniques. Moreover, the distinction often leads to a misuse of learning strategies, for example a supervised discretization is performed as a data preprocessing, and then method capabilities are evaluated with a cross-validation performed on the discretized data, which obviously leads to overoptimistic results.

### **Learning machine abstraction**

An abstract view of a learning machine, according to our approach is presented in figure 2. Given a configuration the adaptive process can be run to obtain the model i.e. the results which may be exhibited to other methods by means of outputs or deposited in a special results repository.

During the adaptive process, the machine can create instances of other machines, specify their parameters and inputs, run them, collect their results and use the results in further part of learning. For example a Support Vector Machine (SVM) can be organized in such a way, that a separate machine may perform kernels calculations and give the SVM access to them through proper output. In this way we obtain a submachine of SVM—it is a proper machine in the sense of our approach, because it precisely defines the input data, kernels parameters, and yields outputs in the form of a table of kernel values. Such solution is very attractive from the point of view of the efficiency of calculations.

If we start another adaptive process of the SVM, which does not differ from the first one with respect to the kernels, then the kernel part may be shared between the two SVM machines and this way we obtain significant savings in both memory and time consumption. The unification of the kernel submachine can be performed automatically by appropriate design of the project management part of the system engine, provided that the inputs and parameters of machines are also uniform, so can be handled in the same manner on a high level of abstraction.

Also when we use the same data transformation technique as a preprocessing stage for two different learning machines, there is no reason to perform the transformation twice and occupy twice as much memory. If the data transformation is implemented as a separate machine, then the machine management routines will notice the unification possibility and will use the same transformation model for both algorithms.

Another spectacular example of memory and power savings are the families of feature selection and vector selection methods. We do not need to copy data, when we select a subset of features or vectors. Thanks to submachine extraction we may obtain the same submachine representing the whole data set for each of the machines, and the subsets may be defined by sets of indices which usually occupy significantly less memory than the corresponding data subset. Although the access to such selected features or vectors must be a bit more expensive than in the case of copied data, proper definition of the enumerators makes the difference not too large, and savings which result from not copying the data will usually compensate the loss.

The model reuse may be much broader, when we supply the system with model cache. The models released from the project may be kept in the cache for some time, and possibly be reused in the future. Different types of cache may be implemented. The simplest one keeps models in memory during a single session (obviously with additional conditions determining when to finally release the model). Another cache module may keep the models in a database stored on a disk, which allows for models reuse among sessions. Yet another cache system could be designed as a network server and provide mechanisms for sharing models by many users of the system. This would allow for reliable comparison of results obtained with different models for popular data sets without the need for recalculating results of all the models used in the comparison.

The only effort necessary from the side of learning machine developer is that the method configuration class provides a method to compare two configuration objects. The system can not have such knowledge about all possible

methods, so they are expected to provide such information, which is usually one very simple function.

### **Inputs vs parameters**

One of the ideas mentioned above, that require some additional comment is the distinction between inputs and parameters. Formally the function of inputs is to provide means for exploiting outputs of other learners, while parameters do not interfere with external world, but specify how the adaptive process of the model will operate on inputs to generate outputs and results.

Although from the formal point of view there is no difference between parameters and inputs it is very advantageous to introduce such distinction in a data mining system, because it makes project management much more intuitive.

### **Outputs vs results**

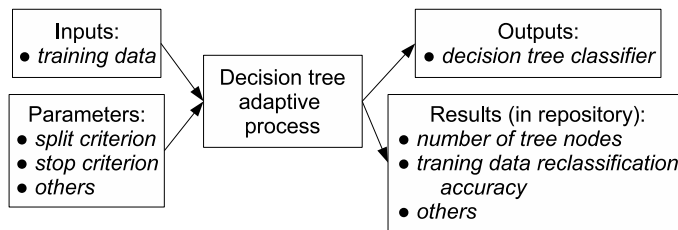
The distinction between outputs and results is subtler and concerns the way they can be used by external learners. Both are the effects of the adaptive process and together represent the model, but the results are deposited in a special repository, which makes them available even after the model itself is released (for example when we create large number of models and to avoid getting out of memory we want to release most of them, but keep some information about them).

Outputs and results also differ in the kind of information expected there. The nature of results repository is to gather some “static” information about the model, while outputs are the adequate place to put also methods, which perform the task of the model (e.g. classification). Although the methods of the result objects may also provide extended functionality, it is not recommended to mix the solutions this way. But it is the matter of a convention, not a technical restriction.

Some tools to easily explore the results repository and manipulate with extracted information are presented in section 5.

### **Decision tree example**

An example scenario with inputs, parameters, outputs and results is shown in figure 3. It shows a decision tree learner with single input of training data and some parameters of the adaptive process. The resulting model exhibits



**Figure 3.** Decision tree learner structure

classification routine as its output and deposits some numbers in the results repository.

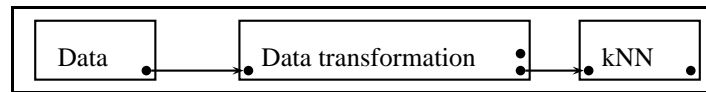
We can use the output to classify other data sets. This operation makes sense only while the decision tree model is fully available. When the method is tested within a cross-validation, where in order to save memory we do not keep in memory all the models built in each fold, the classification routines of the released models are not available, however it is still possible to analyze the results in the repository, for example to check the numbers of tree nodes obtained in each fold, calculate their averages, standard deviations, etc.

## 4 Complex machine structures

The information exchange between machines is a crucial feature of an effective CI system. Indeed, almost everything data analysis systems do, is an information exchange or preparation because of information exchange. Separate machines are not satisfactory even in the simplest cases. For example, when planning a cross-validation we need some machines to learn and others to perform the tests. There are many reasons, for which the possibility of building complex structures of machines is obligatory for contemporary data analysis tools.

### Modular structure

Machines may contain submachines (equivalently: models may run other models—in sensible projects there is a 1-to-1 correspondence also between machine-to-machine and model-to-model dependencies) of any type and any level of abstraction. Also a single machine may have submachines of different types (for example few feature selection machines and few vectors selection machines plus one committee). The submachines can be seen as *slaves* of



**Figure 4.** Input and output interfaces. Circles on left and right sides of boxes represent inputs and outputs respectively.

the parent machine. The submachine does not need to be of a simple type—it may also be a more or less complex model (e.g. ensemble of complex methods, meta-learning, testing methods, etc.). Such solution is important in many cases, some typical applications are: testing machines (repeaters, monte carlo, cross-validation), ensembles, meta-level methods and others. The submachines can be called and used up to the needs of the parent machine—the parent machine may for example create 1000 machines and after that choose three of them and destroy the rest. The important view of submachines cover also the unification level for nontrivial machines as it was already presented in the previous section by the example of the SVM method which may contain a submachines devoted to the management of the kernels. Because of such definition, machines become clearer and much more effective. Such machine splits should be performed wherever the adaptive process consists of some naturally separable parts.

### Input–output interfaces

Machines may be connected using input and output interfaces which play the roles of plugs and sockets. And as in the world of plugs and sockets they must be compatible (in types and features). The connections are the way of information exchange between machines (and models as well). Output types define exact possibilities of the outputs. A single machine may have a few different outputs and a few inputs. Thanks to the inputs and outputs different types of learners may be connected to interact (for example clustering with data loader, classifier with transformer, tester with approximator and data, etc.). Figure 4 presents an example. Dependently on the type of connections, the first machine may understand the second one deeper or shallower (according to the needs which always are declared in the specification of inputs).

### Machine schemes

As presented in the introduction, any hierarchy of learning machines may be regarded as a single learner, because their overall sets of inputs and pa-

rameters may serve as one method's inputs and parameters, and the results altogether may be viewed as a single complex model. Therefore, it is sometimes very reasonable to enclose a hierarchy of methods (and models) in one complex structure with the functionality of a machine (and model). In our system such structure got the name of a *machine scheme* or just *scheme*, because its main task is to define a DAG (directed acyclic graph) of interconnected learners.

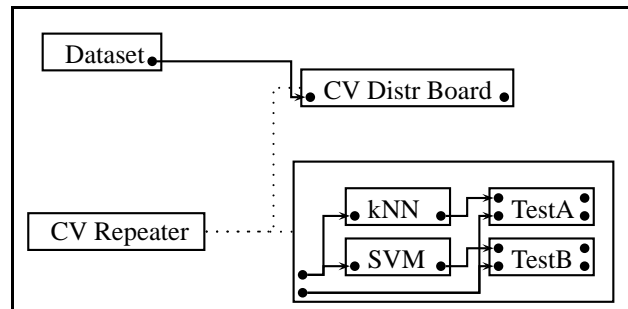
Schemes may also play the roles of submachines representing complex behavior. They may be equipped with inputs and outputs like any other learner. Scheme inputs may be connected with appropriate method inputs inside the scheme and analogously with the outputs. So, scheme inputs and outputs are sort of relays. The combination of the two concepts of schemes and submachines allows to build machines of any complexity with high efficiency (graphs of graph's).

A good example of how to use schemes is a machine of cross-validation of classifiers. In our system, it is a specialization of a general machine called *repeater*, which is responsible for multiple running of (possibly complex) scenarios. The repeater method is based on the concept of *distribution boards* and *distributors*. This means that each repeater uses an external *distribution board* to generate inputs for subsequent runs of the repeated procedure. A distribution board is allowed to generate a number of input collections, which are provided to other machines by a number of instances of a special machines called *distributor*. Each distribution board defines what distributors may be used with it, so that the repeater can do its job without compatibility clashes. The way, a repeater operates is the following:

- a defined number of times it produces an instance of the distribution board (according to its configuration),
- for each distribution board it generates a number of distributors (according to the information supplied by the board machine,
- for each distributor, it constructs a hierarchy of machines defined by a scheme with inputs collection compatible with the distributor.

In the case of repeated CV test, we define the *CV repeater* as a repeater with distribution board fixed to *CV distribution board*, which appropriately generates a given number of pairs of train-test datasets. Each pair of datasets is exhibited by a distributor, and is used to perform a single CV fold.

At the configuration stage the CV machine may look like the one in figure 5. The dotted lines connecting "CV Repeater" with the "CV Distr Board"



**Figure 5.** Configuration of a CV Repeater for classification.

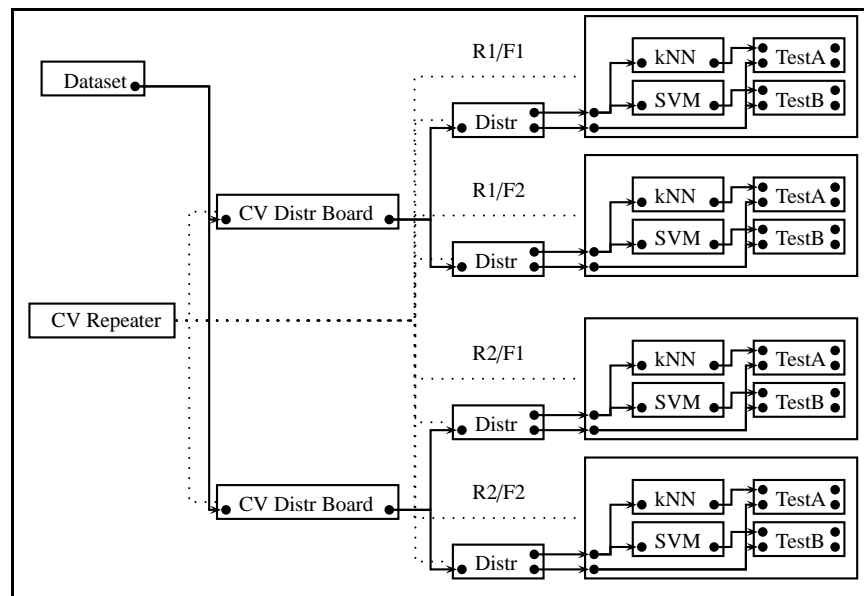
and the scheme, show the parent–child relation between the machines. The CV distribution board has a single input defining the dataset within which the CV is to be performed and a single output (the distribution model) providing information about the number of distributors needed and how to create their outputs. The scheme defines the scenario which is to be repeated. In this case it has two inputs (corresponding to the training and test datasets respectively) and allows the user to define its interior i.e. to put there required machines and bind their inputs and outputs. Any complex structure of machines can be generated (including data transformations, classifiers etc.). The example shown in the figure will train in parallel two classifiers (kNN and SVM) on the training data and test each of them in each fold of the CV.

At runtime the CV acts as standard repeater (described above). So, it creates given number of CV distribution boards, a number of distributors (equal to the product of the number of repetitions and the number of CV folds) and for each distributor, instantiates the scenario defined within the scheme. Full view of twice repeated 2-fold CV of the scenario defined in figure 5 is presented in figure 6. Again, the dotted lines show the parent–child relation between the “CV Repeater” and all of its submachines. Obviously the CV repeater machine may also control the results obtained with all of the children, calculate statistics etc.

There are no limits on the types of models that may occur within a scheme. We can place there different transformers, classifiers, approximators, ensembles, testers, help machines, data loaders, etc.

### Meta-schemes

Another application of schemes is to define template scenarios i.e. scenarios containing some templates (placeholders to be substituted by precise



**Figure 6.** CV Repeater of figure 5 at runtime.

methods). The template is an abstract machine scheme with specification of its inputs and outputs. Any model compatible with this specification may replace the template to obtain a precise, feasible scenario. Remember that, for example, a scheme with a classifier output may play the same role as other classifiers while having possibility to consist of more than one model, so it also fits the classifier template. The mechanism is especially useful in meta-learning.

Figure 7 presents an example of a meta-learning machine configuration with a template. The meta-learning here will search for a transformation (different transformations, which in particular may be complex structures of transformations, will be tried in the place of the transformation template “Trans. templ.”) maximizing some measure of quality of the collection of classifiers (the scheme output is a multi-output i.e. a collection of classifiers—in this case a collection of three classifiers: “C1”, “C2”, and “Decision Module” which combines decisions of the four classifiers). Please notice that transformations “T1” and the template one are shared in a very natural way, saving computational time and memory (in meta-learning taking care of as small computational complexity as possible is especially important).

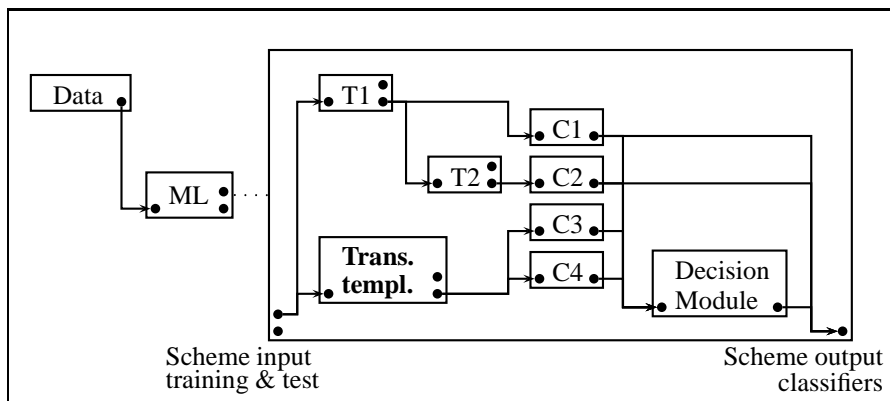


Figure 7. An example of meta-learning machine with transformation template.

## 5 Information exchange, results repository and query system

The main goal of results repository is to provide a common way for commenting on models which is especially useful for testers. There is no obligation for the models to use the results repository. It is rather an opportunity to present interesting information. Results repository collects information from every model in the project. In fact the repository is distributed according to the structure of the project and it can be read and analyzed in different ways by a special query system. The answers are special objects (with special output types) available as any other outputs, so they may be analyzed by other models (typically by testers, statistical significance analysis methods and especially by meta-learning methods). Results analyzed for example by one of meta-learning model may be again a source of information for another level of abstraction (may be after some pruning if necessary).

### 5.1 Results Repository

To provide uniform results management, we created external (to the model) results repository containing items in the form of *label-value* pairs. The string *label* lets queries recognize the values, which can be *objects* containing information of any type. The object may be a number, string, collection of other values, etc.

For an example, consider a classification test. It should have two inputs: the classifier and the data to classify, and at least two outputs: one exhibiting the labels calculated for the elements of the input dataset and one for the con-

fusion matrix. The natural destination for the calculated accuracy is the results repository. To add the value of accuracy variable, labeled as "*Accuracy*", we need to call:

```
machineBase.AddToResultsRepository("Accuracy", accuracy);
```

where the `machineBase` is the object of `MachineBase` class corresponding to our test model. Note that each model has its own results repository node which is accessible for models of higher levels (parent models). Exactly in the same way any other model can add anything to the results repository, for example SVM can provide the value of its margin, an ensemble can inform about its internals etc.

Another useful possibility (especially for complex models) is that the addition of label–value items can be done from outside of the model. For instance, the parent model can add some information about its submodel, which depends on the context in which the child model occurs, and which the child can not be aware of.

An example of such a parent is cross-validation (CV), which can label its submodels with information on *repetition* number and the CV *fold*:

```
submodelCaps.AddToResultsRepository("Repetition", repetition);  
submodelCaps.AddToResultsRepository("Fold", fold);
```

This labeling is performed in the context of model *capsule*, in which each model is placed because of some efficiency requirements, which are outside of the scope of this article. It may also be seen as labeling the connections between parent and children.

## Commentators

To extend the functionality of results repository, we came up with the idea of model *commentators*. It facilitates extending the information in results repository about particular models by external entities other than parent machines. The necessity of such solution comes from the fact that the author of the machine can not foresee all the needs of future users of the models and can not add all interesting information to the results repository. On the other hand it would not be advantageous to add much information to the repository, because of the danger of high memory consumption, which results repository is designed to minimize.

Commentators have access to machine's inputs and outputs, configuration and any *public* parts. In addition commentators may also calculate new values. Commentator may put to the repository a knowledge extracted from any part of the model, its neighborhood, or even from its submodels.

Commentator can be assigned to models by means of the `ConfigBase` class

in a simple way:

```
classifierTestConfig.DeclareCommentator(  
    new CorrectnessCommentator());
```

An example of useful commentator, defined for classification test, may be helpful in different statistical tests like McNemar's. To perform such tests, the information about the correctness of classification of all the instances of tested data is necessary (a vector of boolean values telling whether subsequent classifier decisions were right or wrong).

## 5.2 Query system

The aim of the methodology of results repository and commentators is to ensure that all the models in the project are properly described and ready for further analysis. Gathering adequate results into appropriate collections is the task of the *query system*.

The features of a functional query systems include:

- efficient data acquisition from the hierarchy of models,
- efficient grouping and filtering of the collected items,
- a possibility to determine pairs of corresponding results (for paired t-test etc.),
- a possibility of performing different transformations of the result collections,
- rich set of navigation commands within the results visualization application, including easy model identification for a result from a collection, navigation from collections to models and back, easy data grouping and filtering, etc.

The main idea of the query system is that the results repository, which is distributed throughout the project, can be searched according to a query, resulting in a collection called *series*, which then can be transformed in a wide spectrum of ways (by special components, which can be added to the system at any time to extend the functionality) providing new results which can be further analyzed and visualized.

All the ideas of *series*, their *transformations* and *queries* are designed as abstract tools, adequate for all types of models, so that each new component of the system (a classifier, test etc.) can be analyzed in the same way as the others, without the necessity of writing any code implementing the analysis functions.

## Series

The collection of results obtained from results repository as a result of a query is called *series*. In the system, it is implemented as a general class `Series`, which can collect objects of any type. Typically it consists of a number of information items, each of which, contains a number of values. For example, each item of the series may describe a single model of the project with the value of its classification accuracy, the number of CV fold in which the model was created etc. Thus each item is a collection of label–value pairs in the same way as in the case of results repository. Such representation facilitates two main functions of the series: grouping and filtering.

## Series transformations

The series resulting from queries may not correspond right away to what we need. Thus, we have introduced the concept of *series transformations*. In general the aim of transformations is to convert a number of input series into a single output series. Some of the most useful transformations accessible in our system are:

- calculating properties (correlations, means, standard deviations, medians etc.) or statistical tests (t-test, McNemar test, Wilcoxon test, etc.),
- a concatenation of series into one series (e.g. for grouping together the results of two classifiers into a single collection),
- combining two (or more) series of equal length into a single series of items containing the union of label–value pairs from all the items at the same position in the input series,
- calculating different expressions on series.

## Queries

To obtain a series of results collected from results repository, we need to run a *query*. A query is defined by:

- the *root node*, i.e. the node of the project (in fact a model capsule), which will be treated as the root of the branch of the parent–child tree containing the candidate models for the query,

- the collection of *machine configurations* defining which models of the branch will actually be queried (the results are collected from models generated by machines using configurations from the collection),
- the *labels* to collect, which correspond to label-value pairs in results repository.

The result of running a query is a series: the collection of items corresponding to the models occurring in the branch rooted at the *root node* and being the results of running machines configured with one of the settings in *machine configurations* (see next section for illustrative examples). Each of the items is a collection of label–value pairs extracted for the collection of *labels*. For greater usefulness, the labels in the third parameter of the query, are searched not only within the part of results repository corresponding to the queried model, but also in the description of parent models (we will see the advantage of such a solution in the following examples).

### 5.3 Example applications of queries and series transformations

Consider the example of cross-validation model structure presented already in figure 6. It is a sketch of the hierarchy of models obtained with a repeater machine running twice 2-fold CV of two classifiers (in parallel) kNN and SVM.

After the structure of models is created and the adaptive processes finished, different queries may explore the results repository. The most desirable query in such case is certainly the query for the collection of CV test classification results of each of the classifiers. To achieve this we may define the query(-ies) in the following way:

```

Query q = new Query();
q.Root = repeaterCapsule;
q.AddMachineConfig(testA);
q.MainLabel = "Accuracy";
knnSeries = q.Series;

Query q = new Query();
q.Root = repeaterCapsule;
q.AddMachineConfig(testB);
q.MainLabel = "Accuracy";
svmSeries = q.Series;

```

the *root node* is the repeater node, the *machine configurations* include `testA` and `testB` configurations respectively for kNN and SVM (this ensures that the results will be collected only from the appropriate test models), and the main label is set to “Accuracy”. In some cases like in the below example of McNemar test, we may prefer to set the `MainLabel` to “Correctness” in place

of "Accuracy":

```
q.MainLabel = "Correctness";
```

Such queries return series (`knnSeries`, `svmSeries`) of four accuracies calculated by tests of KNN and SVM models respectively. Both the `Repetition` and the `CV-fold` labels are assigned to the connection leading to the box with classifiers (R1 or R2 and F1 or F2), so appropriate information must be included, but it is too technical to describe it here. As a result we obtain a series of four items consisting of the values of accuracy, repetition number and CV fold number.

### McNemar test of two classification models

To test statistical difference between correctness of two classification models, with McNemar test, we need the flags of answer correctness for each element of the test data. The collection of correctness flags can be obtained by adding to the configuration of classifiers tests the correctness commentator (`CorrectnessCommentator`) as it was presented in section 5.1 in the paragraph devoted to commentators. Additionally, the main label should be set to the "Correctness". Declarations:

```
s1 = knnSeries.Filter("Repetition", 1).Filter("Fold", 1);
```

```
s2 = svmSeries.Filter("Repetition", 1).Filter("Fold", 1);
```

select the results for particular models. By double filtering (superposition), models belonging to the first repetition and the first fold of CV are selected independently for kNN and SVM test series. Because the correctness label-value item contains a collection of correctness for each vector the `Unpack()` method must be used (it converts the collection of collections into one collection) on series `s1` and `s2`. Now we are ready to use McNemar test:

```
s = McNemar.Transform(s1.Unpack(), s2.Unpack());
```

The final series `s` contains test results: the p-value and McNemar statistic value, accessible via `s["p-value"]` and `s["statistic"]` respectively.

In even simpler way the McNemar test can be applied to all subsequent tests of the CV.

```
s = McNemar.Transform(knnSeries.Unpack(), svmSeries.Unpack());
```

### Basic statistics

When collecting `knnSeries` and `svmSeries` as it was presented at the beginning of this section with `MainLabel` set to "Accuracy", the basic statistics can be directly computed:

```
s = knnSeries.Transform(new BasicStatistics());
```

Indexing series *s* with "Minimum", "Mean", "Maximum", "Standard deviation" respective properties are captured (for example *s*["Mean"]). The Basic-Statistics transformer sets the "Mean" as the main label.

Using grouping, the independent statistics for each CV test can be computed. To do this first we have to group the series by the repetition index. After that, result series contains subseries with items representing subsequent CV tests. Then, the basic statistics transformer can be applied to the series through the MAP transformer:

```
s1 = knnSeries.Group("Repetition").MAP(new BasicStatistics());
```

The MAP transformer performs the transformation given as the parameter (here *BasicStatistics*) on each of the subseries and the results are collected into a new series. Note that the MAP in general can be nested if needed. Finally the *s1* series contains a sequence of subseries with basic statistics and by calling the ungroup transformer we obtain series of statistics of CV's (in the main series not in subseries):

```
s1 = s1.Ungroup();
```

Now *s1* contains items, one per single CV, and each one contains mean, minimum, maximum, variance and standard deviation.

The number of possible combinations of different commentators, queries and series grouping, filtering and transformations is huge even with a small set of basic commentators and transformations. The code that must be written is short and intuitive. Since the system is open in the sense that any SDK user can add new commentators and series transformations, the possibilities of results analysis are so rich, that we can claim, they are restricted only by user invention.

## 6 Easy development of new learning machines

Although the system is versatile, it is very important that the project management module does not burden SDK users (the method developers) with the necessity of deep familiarity with the engine mechanisms. To keep method development as simple as possible, the cycle of machine/model life must be very simple. To obtain this, we designed our system, so that each method is configured first, then its adaptive process started, and when the learning is finished, the model is fixed so that it will not change in the future—there is no need to implement the ways of reaction to the changes in other models. This is the point of view of a method developer. From the point of view of a user, each machine may be reconfigured and trained many times, but in fact, each time a new instance of the machine is constructed or reused. Thus, it is very impor-

tant to sensibly split complex machines into a set of smaller ones, because this will make submachines reuse more frequent, because small changes in method parameters may keep some part of the process unchanged, and in such cases the subprocesses not affected by the changes may be reused.

Another advantage of appropriate design of the SDK and basic methods available in the system is that they can “enforce” proper machines construction by SDK users. For example, in our system, the methods of feature selection based on rankings of features are defined in such a way, that the ranking is an output of a submachine. Adding new ranking based selection to the system consists in creating just the ranking submachine.

## 6.1 1NN as a simple model example

Despite the multitude of different contexts in which each learner must be functional, its implementation should be very simple. One of the most important assumptions about our system design was that the particular learners must be in some sense isolated from the complexity of the system engine, so that the developers of new methods may create them with as little effort as possible. An example to confirm this, is the full source code of a simple 1NN classifier presented in figure 8.

From the available object-oriented programming languages (or rather environments) we selected C# and .NET for implementation of our system, because we estimated it was the most complete and best suitable for our meta-learning design.

To implement a new (learning) machine one needs to code two classes: one for configuration of the learner and one for the learner itself. The configuration class for the example 1NN classifier is `OneNNConfig`. The minimum requirement for a configuration class is to implement the `IConfiguration` interface, which means that two methods must be defined: `Configure` and `Clone`. The `Configure` method is parameterized by an object of class `ConfigBuilder`, which provides control over the inputs, outputs and some other configuration routines. The 1NN classifier configuration defines a single input (training data) and a single output (the classifier).

The learner class is `OneNN`. The `Machine` attribute preceding the definition of the class informs the system about the name of the learning method and corresponding configuration class. The class implements two interfaces `IMachine` and `IClassifier`. To implement the former interface two methods are defined: `SetMachineBase`, which is called by the system at the start of the machine life to supply the method with necessary information about its context, and `Run`, which is the function to implement the adaptive process of

```

1 public class OneNNConfig : IConfiguration
2 {
3     public void Configure(ConfigBuilder confBuilder)
4     {
5         confBuilder.DeclareInput("Input dataset",
6             new Type[] {typeof(IDataTable), typeof(ITargets)}, false);
7         confBuilder.DeclareOutput("Classifier",
8             new Type[] {typeof(IClassifier)}, false, null, null);
9     }
10    public object Clone() {return new OneNNConfig();}
11 }
12 [Machine("1NN", typeof(OneNNConfig))]
13 public class OneNN : IMachine, IClassifier
14 {
15     IMachineBase machineBase;
16     public void SetMachineBase(IMachineBase mb) {machineBase = mb;}
17     IDataTable trnD, targets;
18     public void Run(ref bool shouldTerminate)
19     {
20         trnD = machineBase.OpenInput("Input dataset") as IDataTable;
21         targets = (trnD as ITargets).Targets;
22         machineBase.CloseInput("Input dataset");
23     }
24     public IDataTable Classify(IDataSet data)
25     {
26         IDataTableBuilder t = targets.CopyStructure(data.VectorsCount);
27         IVectorEnumerator ed = (IVectorEnumerator)data.VectorEnumerator();
28         IVectorEnumerator et = (IVectorEnumerator)trnD.VectorEnumerator();
29         SquareEuclideanMetric m = new SquareEuclideanMetric();
30         for (int i = 0; i < data.VectorsCount; i++)
31         {
32             ed.GoToVector(i);
33             double d, minDist = System.Double.PositiveInfinity;
34             int minId = -1;
35             for (int j = 0; j < trnD.VectorsCount; j++)
36             {
37                 et.GoToVector(j);
38                 d = m.Distance(ed, et, minDist);
39                 if (d < minDist) {minId = j; minDist = d; }
40             }
41             t.Values[i,0] = targets[minId,0];
42         }
43         return t.Build();
44     }
45     public bool VerifyTestData(IDataSet data)
46     {
47         int featureCount = trnD != null? trnD.FeaturesInfo.Count : 0;
48         IDataTable dt = data as IDataTable;
49         return dt != null && dt.FeaturesInfo.Count == featureCount;
50     }
51 }

```

**Figure 8.** Full source code of a 1NN classifier.

the machine. In the case of 1NN classification, no learning is necessary—it is enough to ensure access to the training data set for finding nearest neighbors, which is achieved by storing the reference to the training data in the `trnD` field. To get access to an input we need to open it (line 20 of the source code) and after we are finished, we close it (line 22).

The `IClassifier` interface has two methods: `Classify` and `VerifyTestData`.

The `Classify` method gets a data set with elements to be classified and returns the vector of class labels (in fact a matrix, because in general a single classifier may determine several vectors of classes in parallel, but here, for maximum simplicity, we assume only one classification task). To return a matrix of class labels (a data table) we need a data table builder (defined in line 26). The vector enumerators (lines 27–28) give an efficient and safe access to the training and classified data. We need them to calculate distances and determine the nearest neighbors (lines 29–40) to make the decisions (in line 41). Distance calculation is performed by a simple class `SquareEuclideanMetric`, which can be easily replaced by other metric. In particular we could define a metric dealing with other kind of data (e.g. texts, some descriptions of 3D structure of molecules etc.) and it will make the 1NN classifier eligible for classification of such data (not necessarily tabular, numerical data as in the example).

The method `VerifyTestData` is designed to check the compatibility between the training and test data sets—in the case of our Euclidean metric, we need two data tables containing vectors from feature spaces of the same dimensionality.

## 7 Summary

We have presented some theoretical background of a general data mining system and parts of its implementation. Thanks to the comprehensive abstraction of learning machines and models, we have created a very general but still highly effective system. Modular architecture causes that nothing must be reimplemented or recalculated. Models of any complexity may be composed by any combinations of available methods.

Thanks to the general and flexible engine, new models (also the complex ones) can be implemented effectively with the SDK and may be instantly used with full functionality, without any need for changes in the engine. Moreover, by means of SDK any type of models can be implemented: classifiers, approximators, testers, measures and even models of completely new types, unknown

yet.

Models in the project are connected using input and output interfaces in a natural way giving the opportunity to efficiently exchange information. The results repository collects interesting data (comments) about the models and can be easily explored and analyzed with the help of our query and series system.

The project may contain any number of data sources, any number of simple or complex models of any kinds, which can coexist and cooperate in a number of ways. Such models may easily exchange information on different levels of abstraction.

The versatility of the system predestines it to a broad range of applications including the most sophisticated ones like advanced meta-learning approaches. The riches of different models and their types opens the gates to powerful exploration and explanation of data and can not be compared to any other existing system.

## Acknowledgement

The research is supported by the Polish Ministry of Science with a grant for years 2005–2007.

## References

- [1] H. Bensusan, C. Giraud-Carrier, and C. J. Kennedy. A higher-order approach to meta-learning. In J. Cussens and A. Frisch, editors, *Proceedings of the Work-in-Progress Track at the 10th International Conference on Inductive Logic Programming*, pages 33–42, 2000.
- [2] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.
- [3] Pavel Brazdil, C. Soares, and J. Pinto da Costa. Ranking learning algorithms: Using IBL and meta-learning on accuracy and time results. *Machine Learning*, 50(3):251–277, 2003.
- [4] W. Duch. Software and datasets. <http://www.phys.uni.torun.pl/~duch/software.html>, 2006.
- [5] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley, 2 edition, 2001.

- [6] I. Guyon, S. Gunn, M. Nikravesh, and L. Zadeh. *Feature extraction, foundations and applications*. Springer, 2006.
- [7] Isabelle Guyon. Nips 2003 workshop on feature extraction. <http://www-clopinet.com/isabelle/Projects/NIPS2003/>, December 2003.
- [8] Isabelle Guyon. Performance prediction challenge. <http://www.modelselect.inf.ethz.ch/>, July 2006.
- [9] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, 2001.
- [10] N. Jankowski and K. Grąbczewski. Learning machines. In I. Guyon, S. Gunn, M. Nikravesh, and L. Zadeh, editors, *Feature extraction, foundations and applications*, pages 29–64. Springer, 2006.
- [11] KDnuggets. Software suites for Data Mining and Knowledge Discovery. <http://www.kdnuggets.com/software/suites.html>.
- [12] Richard Lowry. Concepts and applications of inferential statistics. <http://faculty.vassar.edu/lowry/webtext.html>, 2005.
- [13] T. Mitchell. *Machine learning*. McGraw Hill, 1997.
- [14] Bernhard Pfahringer, Hilan Bensusan, and Christophe Giraud-Carrier. Meta-learning by landmarking various learning algorithms. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 743–750. Morgan Kaufmann, June 2000.
- [15] B. D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, Cambridge, 1996.
- [16] B. Schölkopf and A. Smola. *Learning with Kernels*. MIT Press, Cambridge, MA, 2002.
- [17] Y.H., Peng, P.A. Falch, C. Soares, and P. Brazdil. Improved dataset characterisation for meta-learning. In *The 5th International Conference on Discovery Science*, pages 141–152, Luebeck, Germany, January 2002. Springer-Verlag.