

---

# Learning machines information distribution system with example applications

Norbert Jankowski and Krzysztof Grąbczewski

Department of Informatics  
Nicolaus Copernicus University  
Toruń, Poland

<http://www.is.umk.pl/> {norbert|kgrabcze}@is.umk.pl

**Summary.** When problem solving reduces to examination of a single or a few learning methods no sophisticated mechanisms of information exchange are necessary, but when we use meta-learning for extensive search through a huge space of hybrid models, the information exchange between subsequent models is crucial.

The information exchange between models must be universal, very flexible and as simple to define as possible. The design of an efficient system must include abstract methodology for transmission of amorphous information between different kinds of methods and optimizing different types of functions. It is highly important for meta-learning where different types of information must be collected and used by meta-processes at high level of abstraction. This article presents a universal information exchange system eligible for huge data mining tasks which was implemented in our meta-learning environment Intemi.

## 1 Introduction

Meta-learning [1, 2] techniques will more and more often supply successful models, which would be very difficult to find by human, because of their unusual structures. Some of our research have already born the fruits of very high accuracies of our classifiers solving tasks of the Feature Selection Challenge <sup>1</sup> [3] of NIPS 2003 and the Handwritten Digit Recognition Competition<sup>2</sup> organized with The Eighth International Conference on Artificial Intelligence and Soft Computing in 2006. The models we found were usually complex model structures consisting of some data transformations like standardization, feature selection, features construction based on principal components analysis and some committees of classifiers. We have also examined some aspects of member model competence in classification committees [4].

All such meta-learning machines (algorithms) require large amount of calculations (e.g. to validate the methods) before they point to most attractive solutions.

---

<sup>1</sup> <http://www.clopinet.com/isabelle/Projects/NIPS2003/>

<sup>2</sup> <http://www.icaisc.pcz.czest.pl/competition.htm>

Many candidates must be examined, numerous combinations validated often with different optimization criteria. To make it all possible we need a general data mining system, which efficiently manipulates such complex models. Such system must provide:

- uniform way of machines and model manipulation—the possibilities of adding, configuring, training and testing machines, exploiting and removing models in a standard way, implemented as a set of project management routines in such a way that does not burden the authors of particular machines with the administration efforts,
- **uniform access to results of learning and tests, so that meta-learning methods do not need knowledge about the specificity of particular models,**
- **uniform query system for gathering information from submachines, facilitating versatile and efficient functionality of analysis of gathered results.**

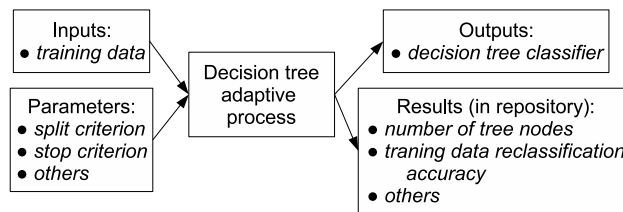
The second and third points define the main theme of this paper. The mechanisms must be uniform but not too restrictive, i.e. general enough to fit any kind of adaptive processes (also the results of machines which will be constructed in future). The abstraction of management routines facilitates communication between different machines and models within the project on appropriate, different levels of abstraction. Dependently on particular needs, general or detailed questions may be asked in a common language without the necessity to know the details of the methods being used. This provides an excellent source of knowledge (meta-knowledge) not only for basic analysis of datasets, but also for advanced meta-learning.

Section 2 sketches some ideas of the system with special emphasis on the topics of this article (more information is presented in [5]). Next, in section 3, we describe the abstraction of learning results representation. In section 4, the general methodology of exploring the results is presented and illustrated by examples in the last section 5.

## 2 Fundamental ideas of our data mining system

Our system is a general data mining tool eligible for any computational intelligence applications. There are many data mining systems available on the market, but we don't know any, providing so rich general functionality of the kernel and being so suitable for advanced meta-learning algorithms management. Abstraction of our system is based on generalized definitions of *method (machine)* and *model*, an abstract view of *inputs* and *outputs*, *parameters* and *results* and provides general tools for machine and model management, independent of the kind of the algorithm.

In computational intelligence, the term *method (or learning machine)* is used to describe an *adaptive algorithm*. In our approach this term encompasses a broader (than usual) range of algorithms, because from the point of view of a general data analysis framework there is no reason to differentiate between the algorithms for classification, approximation or clustering and those for loading data, visualizing



**Fig. 1.** Decision tree learning machine structure

some aspects of data, testing classifiers etc. We define a *model* as a result of application of a *machine* with some particular *parameters* to objects given as *inputs*. A model is an information carrier—this information may be passed to other models by means of *outputs* and may be put into a special *results repository*.

The aspects of accessing inputs, output exhibition and management, access to submachines and their configurations, results repository navigation etc. have been implemented in *MachineBase* class. The class is common to all possible machines and allows the implementers to work only on the crucial code for particular machines. Similar idea lies behind the configuration of the machine. Hence, we have created a general *ConfigBase* class implementing the common functionality related to machine configuration (inputs and outputs definition, parameters, submachines configurations) and available to learning machines developers.

Machines within a project compose two hierarchies. One is defined by the *input–output relation*. The other is defined by the *parent–child (or submachine/submodel) relation*.

The distinction between outputs and results is technical and concerns the way they can be used by external methods. Both are the effects of the adaptive process, but outputs are to be bound with other machines inputs, and the results are deposited in a special repository, which makes them available even after the model itself is released (for example when a vast amount of model structures is tested, and together they would occupy too much memory).

An example of the scenario with inputs, parameters, outputs and results is shown in figure 1. It depicts a decision tree adaptive process with single input of training data and some parameters. The model exhibits classification routine as an output (for other machines use) and deposits some numbers in the results repository.

### 3 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 confusion 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 submachines, which depends on the context in which the child machine occurs, and which the child can not be aware of.

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

```
submachineCaps.AddToResultsRepository("Repetition", repetition);  
submachineCaps.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 submachines.

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).

## 4 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 Example applications of queries and series transformations

Consider the example of model structure presented in figure 2. 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. Each labeled box represents a single model. There are also four groups of classifiers and their tests enclosed by unnamed boxes—they are the scenarios run in each fold of the CV. The bullets in the left part of the boxes represent model inputs, and those at the right side—the outputs. The repeater model contains a sequence of runs resulting from the configuration of the distribution board. The two “Distr Board” boxes correspond to the CV distribution board, which splits its input dataset into 2 parts, preparing it for the CV. The “Distr” boxes are distributors—they use the distribution board output to exhibit proper training and test datasets as their outputs. The repeater created a scenario defined at the configuration stage for each of the distributors. The scenario assumed creation of kNN and SVM models with inputs bound to proper CV training data and one classification test for each of the two classifiers. The test machines’ inputs are bound to corresponding classifiers and CV test data respectively.

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

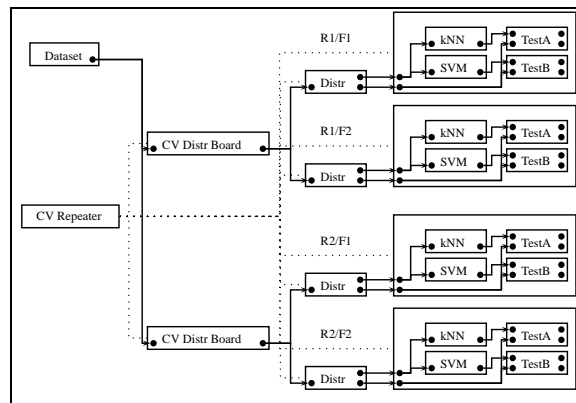


Fig. 2. A repeater machine performed twice a 2-fold cross-validation of two classifiers.

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 3 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):

```
s1 = s1.Unpack(); s2 = s2.Unpack();
```

Now series *s1* and *s2* contain the correctness for selected models and are ready to be used via McNemar test:

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

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. In this case the series of collections of correctness is also unpacked to one flat collection (result of `ll 4` test in a single series) and then put to the McNemar test:

```
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 *BasicStatistics* 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:

```
s1 = knnSeries.Group("Repetition");
```

After that, the *s1* series contains subseries with items representing subsequent CV tests. Now, the basic statistics transformer can be applied to the series:

```
s1 = s1.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. *MAP* may also be very useful with transformations like grouping, ungrouping, filtering and many others. Finally the *s1* series contains a sequence of subseries with basic statistics of CV repetitions 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. All these operations may be composed into a single transformation chain:

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

giving exactly the same result.

Now, the following code computes the statistics of inner CV test statistics:

```
s1 = s1.Transform(new BasicStatistics());
```

After that `s1["Mean"]` represents average accuracy and `s1["Standard deviation"]` represents standard deviation of the mean accuracies of the repetitions of CV tests.

#### *T-test and paired t-test*

To test the statistical significance of the differences between the results of kNN and SVM with paired t-test (it does not make much sense in the case of  $2 \times 2$ -fold CV, but the way to do it does not depend on the numbers of repetitions or CV folds), we need to collect the results of all the kNN models and SVM models separately in the same data distributions, as it was already presented at the beginning of this section with the main label set up to "Accuracy".

We can directly compute statistical significance between accuracies of kNN and SVM calling paired t-test as follows:

```
s = TTestPaired(knnSeries, svmSeries);
```

The `s` series can be indexed by "p-value" or "statistic" to get the most interesting values calculated by the t-test (`s["statistic"]` is the value of  $t$ ). In the case when `knnSeries` and `svmSeries` are not collected from the same CV test, and we are interested in calculating t-test, we just substitute `TTestPaired` by `TTest` in the call presented above.

If we want to compare mean accuracy and stability of two methods, in the analysis we can replace mean accuracies with the differences between means and standard deviations of accuracies:  $\overline{acc} - \alpha \sigma_{acc}$  where the  $\alpha$  is a parameter (typically equal to 1). If we are also interested in the t-test on differences between such measures, we may call it in the following way:

```
s = TTestPaired.Transform(s1 - s1.GetSeries("Standard deviation"),
    s2 - s2.GetSeries("Standard deviation"));
```

First, the series of differences are calculated (`s1 - s1.GetSeries("Standard deviation")`, and the same for `s2`). Note that the transformation `GetSeries` creates new series with "Standard deviation" as the main label and containing the same items as `s1` (the main label of `s1` is "Mean").

To call the Wilcoxon signed rank test or Mann-Whitney test all we need is the substitution of `TTestPaired` or `TTest` by `Wilcoxon` and `Mann-Whitney` respectively, in the above examples.

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 Summary

There is no meta-learning without meta-knowledge. The system of result repository and queries has been designed to facilitate advanced meta-learning. It can be suc-

cessfully used for miscellaneous, sophisticated applications in data mining including construction of different types of classification committees and other ensembles of models, feature selection and extraction, and many other fields.

Universal mechanisms for results repository services and powerful system of repository information retrieval and manipulation, offers incomparable possibilities never met in known meta-learning or other data mining systems. The results repository may contain heterogeneous information. Moreover the commentators may be used to extract additional information from already implemented machines to extend possibilities of further analysis. Using the system of results repository queries, series and series transformers, one can easily obtain answers for very broad range of questions and successfully mine for meta-knowledge.

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

## References

1. Pfahringer, B., Bensusan, H., Giraud-Carrier, C.: Meta-learning by landmarking various learning algorithms. In: Proceedings of the Seventeenth International Conference on Machine Learning, Morgan Kaufmann (June 2000) 743–750
2. Brazdil, P., Soares, C., da Costa, J.P.: Ranking learning algorithms: Using IBL and meta-learning on accuracy and time results. *Machine Learning* **50**(3) (2003) 251–277
3. Guyon, I., Gunn, S., Nikravesh, M., Zadeh, L.: Feature extraction, foundations and applications. Springer (2006)
4. Jankowski, N., Grąbczewski, K.: Heterogenous committees with competence analysis. In Nedjah, N., Mourelle, L., Vellasco, M., Abraham, A., Köppen, M., eds.: Fifth International conference on Hybrid Intelligent Systems, Brasil, Rio de Janeiro, IEEE, Computer Society (November 2005) 417–422
5. Grąbczewski, K., Jankowski, N.: Versatile and efficient meta-learning architecture: Knowledge representation and management in computational intelligence. In: IEEE Symposium Series on Computational Intelligence (SSCI 2007), IEEE (2007) 51–58