YALE: Yet Another Learning Environment

Oliver Ritthoff*, Ralf Klinkenberg*, Simon Fischer*, Ingo Mierswa*, and Sven Felske**

* University of Dortmund, Department of Computer Science, Chair of Artificial Intelligence, 44221 Dortmund, Germany, E-Mail: {ritthoff,klinkenberg,fischer,mierswa}@ls8.cs.uni-dortmund.de

** University of Dortmund, Department of Chemical Engineering, Chair of Plant Engineering, 44221 Dortmund, Germany, E-Mail: felske@chemietechnik.uni-dortmund.de

Abstract. In many data mining and knowledge discovery applications, the problem at hand cannot be solved satisfactorily by just taking the raw data as it is and simply applying a single one-step machine learning method. Instead, some data pre-processing and maybe representation changes are necessary to provide the data in a form suitable for the learning task and the chosen learning method and to improve the performance of the learned model. One example is the task of predicting certain properties of a chemical given its chromatogram curve, i.e. a time series with concentrations of the chemical measured at different points of time at the end of a column the chemical is sent through. The sensor readings may be noisy and perhaps slightly shifted along the time axis between different measurements, so that chromatograms looking very similar to a human expert, may seem very different for a learning method depending on the chosen representation. The extraction or construction of characteristic, robust features may significantly improve the result obtained by a learning method. Hence in many data mining applications, like the one just mentioned, one rather considers chains of pre-processing and learning steps rather than just a single one-step method. This paper proposes YALE, **y**et **a**nother **l**earning **e**nvironment, which allows to easily specify and execute such data mining operator chains for pre-processing, especially feature generation and selection, and multistrategy learning. This modular, non-commercial environment supports nested operator chains and the exchange of individual operators and thereby the systematic evaluation and comparison of different operators and operator chains for the same (sub)task. This paper describes the basic concepts underlying YALE, demonstrates how to describe data mining operator chains in YALE, and provides an example application of YALE for feature generation and selection on chromatography time series data comparing different data pre-processing approaches.

Keywords. machine learning environment, data mining, knowledge discovery, data pre-processing, multistrategy learning, feature generation, feature selection

1 Motivation and Introduction

1.1 Data Pre-Processing and Data Mining

In many real-world data mining applications, the data has to be pre-processed to be usable by the chosen machine learning method and/or to achieve an acceptable level of performance in prediction. A central problem is the representation of the examples by a good set of attributes, i.e. a set of attributes that allows the chosen learner to find a candidate hypothesis solving the learning task at hand within its hypothesis search space. Without meaningful attributes that together convey sufficient information to make learning tractable, no machine learning technique will be successful. Hence finding a suitable set of attributes may be far more important for the overall success than the choice of a particular learning technique. Therefore *feature generation* and *feature selection* are helpful or even necessary for many data mining tasks.

The remainer of the first part of this paper identifies advantages and some desired properties of machine learning environments for our current research and shows, that two of the most popular freely available machine learning environments do not fully meet all of these requirements. In the second part of this paper, we propose, YALE, **y**et **a**nother **l**earning **e**nvironment, and describe its underlying modular, i.e. operator-based concept, the realization of operator chains, and how we try to meet the specified requirements. The third part demonstrates how to describe the data and an data mining operator chain for an experiment. The fourth part of this paper shows how the environment can be used to solve a practical learning task in the field of chemical engineering and provides a comparison of different operator chains, reaching from simple learning chains to complex operator chains combining different kinds of pre-processing methods including feature selection and generation. In the last parts we give a short summary of the current status and an outlook on the most important planned extensions.

1.2 Usefulness and Desired Properties of Machine Learning Environments

Many data mining tasks are solved best by not just taking the raw data as it is and simply applying a single one-step machine learning method, but by using a combination of data pre-processing and machine learning methods. While such methods can be combined manually and/or by writting special scripts whenever a new data mining task arises, much less effort is required, if a flexible machine learning and data mining environment can be used.

Such an environment should allow to easily specify and automatically execute data mining operator chains and to exchange operators in the chain by alternative operators for the same (sub)task and thereby support systematic comparisons and evaluations of individual methods as well as complex operator chains. For complex tasks and evalutions, operator chains should be nestable. Section 3 provides an example application that requires two nested cross-validations.

For the scalability and applicability of a learning environment, it should be able to read data from files, main memory, or a database, which ever seems to be most appropriate for the current task, without making changes in the data mining operators necessary when changing the data source or switching between keeping all or just one example at a time in main memory. The latter may be the preferable approach in case of very large data sets. Ideally the source and the way of handling the example set should be transparent to the other operators in the data mining chain. For efficiency reasons, a data mining enviroment should not create copies of the data unless really necessary for the task. To enhance the re-usability and applicability to new tasks, the machine learning environment should be easily extendable.

For our current research, a further requirement for a learning environment is the support of feature construction and selection methods, which often play a central role in data mining.

1.3 Existing Environments

There already exists several machine learning and data mining environments that provide a number of methods from machine learning, statistics, and pattern recognition. So one might wonder, why we should come up with YALE, **y**et **a**nother **l**earning **e**nvironment. This sections describes two of the most popular existing non-commercial learning environments and explains why they do not fully meet our requirements. These two freely available data mining enviroments are WEKA[1] (Waikato Environment for Knowledge Analysis) [8], developed at University of Waikato, NZ, and MLC++[2] [4], first developed at Stanford University, CA, USA, and then extended by Silicon Graphics, Inc. (SGI), CA, USA.

Weka is a collection of machine learning algorithms implemented in Java. WEKA supports a large number of learning schemes for classification and regression (numeric prediction) like decision tree inducers, rule learners, support vector machines, instance-based learners, naive Bayes, multi-layer perceptrons etc. and basic evaluation methods like cross-validation and bootstrapping [1]. WEKA has some pre-processing algorithms for the manipulation of attributes as well as three basic feature selection schemes, namely the feature correlation based approach [2], a wrapper approach [3] and a filter approach. Additionaly WEKA provides meta classifiers like bagging and boosting.

MLC++ is a library of C++ classes for supervised machine learning. It provides a number of learning schemes similar to those used in WEKA. Additionally wrappers around these basic inducers like a discretization filter, a bagging wrapper and a feature selection wrapper are provided.

Unfortunately neither of these two data mining environments meets all of our requirements, because for example both of them neither support the composition and analysis of complex operator chains consisting of different nested preprocessing, learning, and evaluation steps nor sophisticated feature generators for the introduction of new attributes. MLC++ supports operator chains in a rather restrictive way. One can only build wrappers around basic inducers (learning schemes), but not around nested operator chains. The same applies to WEKA, where nesting can only be realized by numerous comand line calls, by creating copies of (subsets of) the data set, and manual data file management or by writting your

[1] `http://www.cs.waikato.ac.nz/ ml/weka/`
[2] `http://www.sgi.com/tech/mlc/`

own experiment and data management program. An additional shortcoming of WEKA is its lacking scalability. It expects the example set to fit completely into main memory, which for many data mining tasks is not possible, and it is very slow on large data sets. For n-fold cross-validation WEKA creates n copies of the original data set, only one at a time, but still requiring the resources for the copying.

2 Basic Concepts of YALE

2.1 Operators and Operator Chains

YALE is machine learning environment that allows to describe and execute even complex data mining operator chains and experiments in a relatively simple way. Real-world data mining tasks are often solved by a sequence or combination of several data pre-processing and machine learning methods. In YALE, each such method is considered an *operator*.

A sequence of such operators is called an *operator chain*. An operator chain again is an operator, both in the sense of a definition as well as in the object-oriented programming sense. Operators may enclose other operators or operator chains and are then often refered to as *wrappers*. Typical examples of wrappers are cross-validation and feature selection wrappers (see e.g. [3]).

By enclosing other operators or operator chains, operators and operator chains are arbitrarily *nestable*, so that even complex experimental set-ups can be build. For example a nested cross-validation could be used to first optimize some parameters of a data pre-propessing and learning chain (inner validation) and to then evaluate the performance of the whole experimental set-up (outer validation). Section 3.3 describes an example of such a nested cross-validation experiment for the task of generating and selecting a good attribute set to solve a regression learning task.

But before turning to complex scenarios, let us first consider the simple learning chain shown in figure 1 to explain some basic concepts behind operator chains in YALE. This small and a little bit abstract operator chain uses a *learner* operator to construct a classification or regresion *model* from labeled *examples*. The examples are represented by attribute value vectors. The specification of the underlying *attributes* is the topic of section 3.3. The model could for example be a set of rules or a decision tree in the case of classification and a regression function in the case of regression.

The example set and the learned model are passed to a *model applier* operator, that employes the model to predict labels for the examples. Finally, an *evaluator* operator takes the examples and compares their original labels with those predicted by the model to compute one or several *performance measures*, like e.g. classification error in the case of classification or like e.g. relative error or squarred absolute error in the case of regression. The performance results are then passed on in a vector.

Of course this learning chain is very simplified, because evaluating a learned model on the training data does not provide a good estimation of the true performance on previously unseen data, but this simple operator chain is only meant to illustrate different types of operators, how they are connected, and what types of things are passed between them.

2.2 Hierarchy and Exchangability of Operators

The simple operator chain in figure 1 showed rather abstract operator types like learner, model applier, and evaluator than concrete operators like e.g. a specific decision tree learner. The different types of operators can be organized in a hierarchy in the object-oriented programming sense with the class `Operator` at its top and for example `OperatorChain`, `Learner`, `ModelApplier`, and `Evaluator` as some of its descendants and with e.g. `DecisionTreeLearner` as a subclass of `Learner`, which again has `C45Learner`, a C4.5 decision tree learner [5], as a subclass.

Similarly, the things passed between operators can be organized in a hierarchy. As already explained in the previous section, among the things passed between operators in an operator chain are attribute sets, example sets, classification and regressions models, example sets with additional labels, and performance evaluation results. Each operator receives an `IOContainer` as input that may contain some of these things and delivers an `IOContainer` with such objects. During its execution, an operator may modify, remove, or add objects in the `IOContainer` before passing it to the next operator in the operator chain. Some operators may require certain objects to be present in their input and guarantee others to be in their output. For example a `Learner` requires a labeled set of examples as input and generates a model, a `ModelApplier` requires a model and a set of examples, for which it should predict the labels, and so on. Objects that are present in the input of an operator without being required are usually ignored and passed on to the next operator. YALE verifies that each operator receives its required inputs

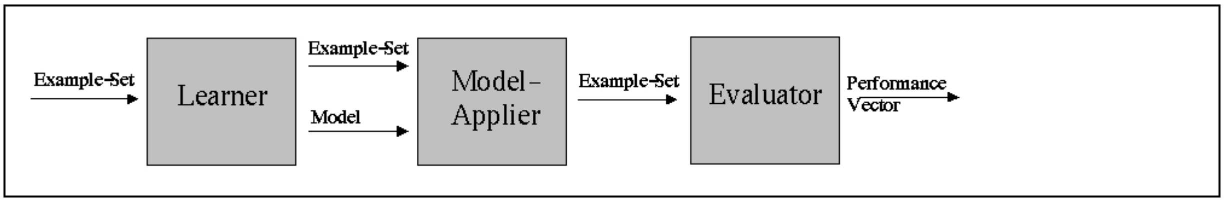

Figure 1 A small example operator chain for learning, applying, and evaluating a classification or regression model (all on the same example set, which is of course not advisable for any real evaluation).

before executing an operator chain.

In order to support the comparison of different operators and operator chains for the same (sub)tasks, operators are easily *exchangable* by other operators. The only premise is, that the subsequent operators in an operator chain, or more general all operators that share a common interface, have fitting input/output types.

2.3 Technical Issues

YALE has been implemented in Java for several reasons. First of all, Java is an *object-oriented* programming language, which makes it very well suited for handling operator chains as should have become obvious in the previous sections. Second, Java is available for all major computer platforms, which guarantees the *portability* of YALE. Another plus for Java is its JavaDoc tool that supports the documentation of the implemented classes by automatically generating a hyperlinked HTML documentation from the comments in the source code.

Some of the operators in YALE are simply wrappers for e.g. external learners and model appliers like e.g. the decision tree learner C4.5 [5] or the support vector machine implementation mySVM [6], which both are the original implementations by the authors in C and C++ respectively. Hence a re-implementation for these methods was not necessary and future updates can be incorpated without additional effort, as long as the data and command line formats of these externals programs remain unchanged. Since the learning step often is one of the most time consuming steps in a data mining chain, and fast execution speed is not the most distinct property of Java programs, external C and C++ programs like the ones mentioned may help to speed up the run times of experiments performed with YALE.

3 Describing Operator Chains and Experiments in YALE

Two types of text files are needed to set up experiments in YALE. The data is described in an *attribute descriptions file* (see section 3.1) and the experiment is specified in a *configuration file* containing a description of the employed data mining operator chain (see section 3.3). Both types of files use an XML format, because XML is well structured, easily human- as well as machine-readable.

Two additional types of files are needed to execute an experiment. One file type contains the examples (*attribute values file*) and the other contains the labels of the examples for a particular classification or regression task (*label file*). These two file types are also described in section 3.1.[3]

3.1 Describing the Data

YALE can process data sets that can be described in a single table, i.e. in an attribute-value vector format, in which each example is described by an attribute-value vector of equal fixed length.[4] Figure 2 shows a description of a simple attribute set in XML format, an *attribute descriptions file*. Examples based on the attribute set described here contain a value series of 500 numeric values, e.g. a time series, and a numeric label y, e.g. a function value for a regression task or a class label.

Each attribute is identified by its index, no, in the attribute vector and described by an <attribute ... /> block in the attribute descriptions file. The name of the attribute can be arbitrarily chosen. The examples in an examples set may have no, one, or several labels, each of which is described by a <label ... /> block in the attribute descriptions file.

While the valuetype of an attribute (or label) specifies the data type of the individual attribute, the blocktype contains some meta-data about the attribute, e.g. if it is just an individual attribute or an interval boundary or part of a time series. For the valuetype and the blocktype, there exist two ontologies describing the hierarchical is-more-general-than-/is-superset-of-relation relation between the differ-

[3] In a later version of YALE, these two latter file types are planned to be readable from a database, alternatively.

[4] YALEcannot handle multi-relational data, which MLC++ and WEKA cannot handle either.

```
<attributeset>
   <label     no="0"     name="y"       valuetype="numeric"/>
   <attribute no="0"     name="x_start" valuetype="numeric" blocktype="value_series_start" blocknumber="1"/>
   <attribute no="1-498" name="x"       valuetype="numeric" blocktype="value_series"       blocknumber="1"/>
   <attribute no="499"   name="x_end"   valuetype="numeric" blocktype="value_series_end"   blocknumber="1"/>
</attributeset>
```

Figure 2 Description of a simple attribute set in XML format. Examples based on the described attribute set contain a time series of numeric 501 values and a numeric label (e.g. function value).

ent types. The `valuetype` can for example be `nominal` or `ordered`, where `ordered` has the subtype `numeric`, which again has the subtypes `integer` and `real`. The `blocktype` can for example be `single_value` or `interval`. In the latter case, the attribute is the boundary of an interval and there has to be a second attribute with a neighboring index with the same `blocktype`, which describes the other boundary of the interval. If the block is a value series, e.g. a time series or a series of function values, there are three `blocktypes` to describe the first, the last, and any element in between: `value_series_start`, `value_series_end`, and `value_series`, respectively.

The `blocknumber` can for example be used to determine, which attributes belong to the same value series or which pair of attributes forms an interval. This way, an example may be described by several blocks of the same and/or different types without loosing track which attributes belong to the same block. The attribute set described in figure 2 contains only one value series (block) and hence all elements of that series have the same block number.

The information about the types and blocktypes of attributes is useful for feature generators, which can verify the types and blocktypes of attributes to check their applicability. A generator extracting the maximum value of a time series can for example restrict its application to all valueserieses in the attribute value vector of an example and generate the maximum for each such series separately while ignoring for example all individual attributes and intervall attributes.

To each attribute, a `construction description` can be added, which explains how a newly generated attribute has been generated. If for example a new attribute x has been generated from the already existing attributes a_i, a_j, and a_k using the definition $x := a_i - a_j/a_k$ for the computation of its values, this definition can be stored as construction description. So, if automatic feature generation and selection methods are used, their resulting optimized feature sets are human-readable and understandable.

3.2 Handling the Data

For almost all operators in YALE it is transparent, whether the examples are read in from a text file, from main memory or (as planned as an option later) from a database, and whether only one or all examples are kept in main memory at a time. They only use an internal structure to iterate over the instances in an example set and hence do not need to distinguish between the different possible data sources and ways of managing the data. The source and way of handling the data depends only on the chosen example source operator and its parametrization.

If the examples are read from a text file, i.e. an *attribute values file*, each line of the file correspondes to one example and the attribute values of one example are separated by one or several white-space characters (or some other user-defined separating character(s)) within the corresponding line of the file. The labels of the examples for a particular classification or regression task are stored in a separate file, a *labels file*. Depending on the learning task(s) at hand, there may be several labels files for the same data set. The label(s) of an example are stored in the same line in the labels file(s) as the attribute values of the example in the corresponding attribute values file.

3.3 Describing Operator Chains and Experiments

This section describes an example of a complex experimental set-up with nested cross-validations for the task of generating and selecting a good attribute set to solve an exemplary regression learning task (figure 4) and how it is represented in YALE in a configuration file in XML format (figure 5).

The learning task considered here is the regression task already mentioned in the abstract of this paper. Given the chromatogram curve of a chemical, i.e. a time series with concentrations of the chemical measured at different points of time at the end of a column the chemical is sent through, a prop-

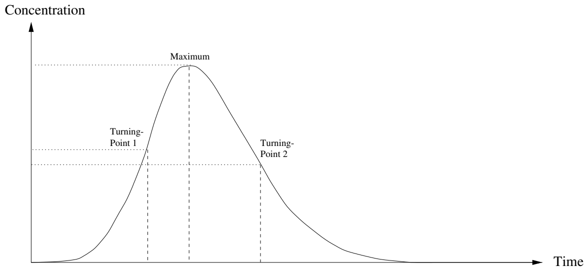

Figure 3 A chromatogram as a time series and some of its characteristic points.

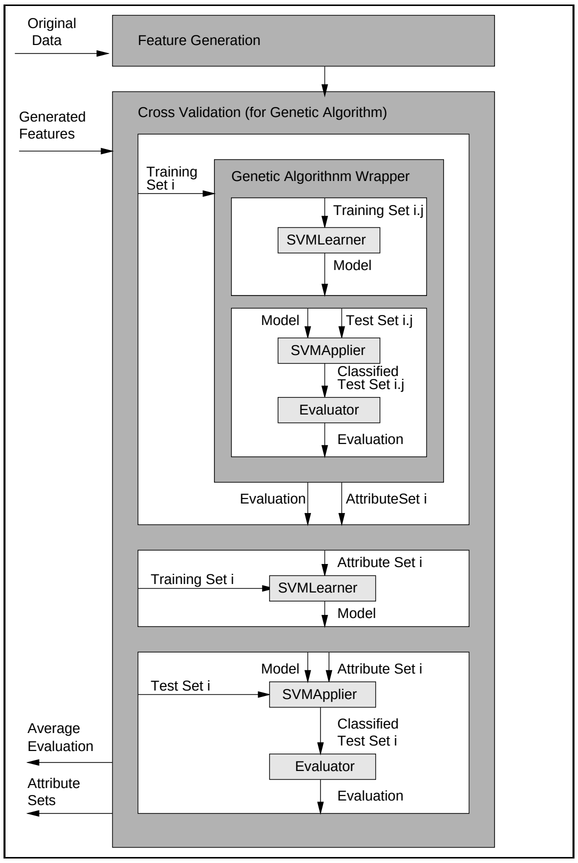

Figure 4 Nested operator chain for feature generation and selection.

```
<operator name="global" class="OperatorChain">

  <!-- Parameters for all SVM learners & appliers -->
  <parameter key="kernel_type" value="radial"/>
  <parameter key="gamma"       value="1"/>
  <parameter key="complexity"  value="1000"/>
  <parameter key="epsilon"     value="0.1"/>

  <operator name="DataSource"
            class="ExampleSource">
    <parameter key="attribute_descriptions"
               value="attribute_descriptions.xml"/>
    <parameter key="attribute_values"
               value="attribute_values.txt"/>
    <parameter key="labels" value="labels.txt"/>
  </operator>

  <operator name="InitialFeatureGenerator"
            class="FeatureGenerator">
    <parameter key="function_characteristica"
               value="true"/>
  </operator>

  <operator name="OuterValidation"
            class="CrossValidation">
    <parameter key="number_of_folds" value="4"/>

    <operator name="GA" class="GeneticAlgorithm">
      <parameter key="optimization_direction"
                 value="minimize"/>
      <parameter key="maximum_number_of_generations"
                 value="200"/>
      <parameter key="generations_without_improvement"
                 value="10"/>
      <parameter key="population_size" value="20"/>
      <parameter key="crossover_type"
                 value="uniform"/>
      <parameter key="p_initialize"    value="0.5"/>
      <parameter key="p_mutation"      value="0.01"/>
      <parameter key="p_crossover"     value="0.5"/>

      <operator name="InnerValidation"
                class="CrossValidation">
        <parameter key="number_of_folds" value="4"/>

        <operator name="GA_SVMLearner"
                  class="SVMLearner">

        <operator name="GA_ApplierChain"
                  class="OperatorChain">
          <operator name="GA_SVMApplier"
                    class="SVMApplier">
          <operator name="GA_Evaluator"
                    class="PerformanceEvaluator">
            <parameter key="performance_measures"
                       value="absolute_error"/>
          </operator>
        </operator>
      </operator>
    </operator>

    <operator name="Learner" class="SVMLearner">

    <operator name="ApplierChain"
              class="OperatorChain">
      <operator name="Applier" class="SVMApplier">
      <operator name="ExperimentEvaluator"
                class="PerformanceEvaluator">
        <parameter key="performance_measures"
                   value="absolute_error
                          squared_error"/>
      </operator>
    </operator>
  </operator>
</operator>
```

Figure 5 Nested operator chain for feature generation and selection described in XML for YALE.

erty of the chemical is to be predicted in the form of a specific numerical constant.

Since the sensor readings may be noisy and perhaps slightly shifted along the time axis between different measurements, the individual attribute in the time series of an example, i.e. the concentration measured at one individual point of time, is not very reliable. The construction of more robust features may significantly improve the result of the learning step. Therefore, in the operator chain depicted in figure 4 the original data is preprocessed by a feature generator adding more robust attributes to the example description. These new features include the position (time point) and the value (concentration) of the maximum of the chromatogram, the positions and values of the two turning points left and right of the maximum, and several distances and ratios computed from these function characteristics (see figure 3).

Since it is not obvious, which of the new and which of the original attributes are really helpful in solving the learning task, a feature selection step is performed to select an attribute set that is well-suited for the learning task. Here a feature selection wrapper based on a genetic algorithm, a probabilistically guided search heuristic, selects an appropriate attribute set, because a complete search of the space of all possible subsets of the attribute set would produce an infeasably large number of candidate attribute sets to be tested. Each candidate attribute (sub)set is evaluated by training a regression SVM on a subset of the examples available and testing its performance on the remainder of the available examples. To do this, the genetic algorithm operator includes two operator chains, a learning chain containing an SVM learner and an evaluation chain containing an SVM model applier and a performance evaluator, which computes for example the absolute error of the predictions of the SVM model on the evaluation examples.

If the genetic algorithm fulfills its termination criterion, e.g. a maximum number of iterations, it delivers its best attribute (sub)set found to an SVM learner that induces the final regression model, which is then evaluated by another evaluation chain containing an SVM applier and a preformance evaluator. This final evaluation determines, how well the combination of the genetic algorithm for the feature selection and the SVM for the regression task performs. In order to achieve a good estimation of this performance, a n-fold cross-validation wrapper repeats this feature selection and regression learning n times and averages the results of this n runs. This *outer cross-validation* is named "Cross Validation (for Genetic Algorithm)" in figure 4.

The outer cross-validation wrapper randomly splits the original data set into n equally sized parts. For each run, the i-th part is kept as a test set, while the remaining $n-1$ parts, "Training Set i", are passed to the genetic algorithm and the subsequent final SVM learner, whose learned model is then evaluated with the i-th part of the data set, "Test Set i". The performance is evaluated on the hold out test set to avoid an overly positive bias of the performance evaluation that would occur, if the regression model was evaluated with the same data that is was trained on.

Since the "Genetic Algorithm Wrapper" for feature selection estimates the performance of each candidate attribute set that it considers, it should also avoid such a bias. Hence this wrapper employes an *inner cross-validation* to estimate the performance of the SVM on the current candidate attribute set more robustly.[5] It splits the "Training Set i" into m folds and holds out the j-th part of this split in the j-th run of this inner cross-validation, so that the inner training is performed on the remainder of "Training Set i", i.e. "Training Set i.j", and the validation is performed using the j-th part, i.e. "Test Set i.j".

The complete experiment now consists of two nested cross-validations. The inner cross-validation trains a model on the *training data* "Train Set i.j" and optimizes the choice of an attribute set using the disjunct *evaluation data* "Test Set i.j" to avoid a bias in the selection of the attribute set. The union of these two inner data sets, i.e. "Train Set i", becomes the training set for the final model in the outer cross-validation, which evaluates the final model on the disjunct *test data* "Test Set i". For a reliable performance estimation of the complete operator chain for feature selection and regression learning, the training, evaluation, and test data sets need to be disjunct, which is guaranteed here by the nested cross-validations.

After illustrating the experimental set-up with figure 4 let us now take a closer look figure 5 at how this experimental set-up can be speficied in an XML file in YALE. Each operator instance is enclosed in a `<operator ...>...</operator>` block specifying the operator `class` this instance belongs to, its individual `name`, and optionally some parameters. The outer most operator always

[5] This inner cross-validation is not explicitly shown here, but only implicitly by the indexing of the training and test sets.

is an `OperatorChain` named `global` enclosing the entire experiment.

The first operator inside the global chain is an data source, i.e. a `ExampleSource` operator, reading the attribute descriptions from the XML file `attribute_descriptions.xml`, the attribute value vectors of the examples from the file `attribute_values.txt`, and the labels of the examples from the file `labels.txt`. The second operator is the initial feature generator extending the examples with the function characteristica described above. The third operator is the outer cross-validation enclosing the genetic algorithm for the feature selection, the final SVM learner, and the final SVM applier and experiment performance evaluator, which computes the absolute and the squarred error results, that are then averaged by the outer cross-validation.

The genetic algorithm contains the inner cross-validation chain just as described above. In addition it contains some parameter definitions setting the optimization direction to the minimization of the result returned by the peformance evaluation in the inner cross-validation, here the average absolute error, limiting the maximum number of iterations without improvement and in total, specifying the population size and some genetic operator application probabilitie, etc..

Since any SVM model should be applied with the same parameters that it was learned with, the SVM parameters are specified only once here and moved to the top of the configuration file. If a parameter is not specified within an operator instance, YALE uses a *parent look-up* mechanism checking the enclosing operator instances for such a specification.

Obviously the transfer from the experiment design (figure 4) to the YALE configuration file (5) was quite straight forward and the resulting XML description is still quite readable considering the complexity of the experiment.

4 Example Application

The example application already shortly introduced in the abstract and in section 3, the regression learning task from chromatography time series data, was used for feature generation and selection experiments in this domain. Chromatography is used in chemical industry to separate temperature sensitive substances. A mixture of components is injected for a certain amount of time into a column filled with porous particles. Due to the different adsorption strength of the substances on the porous particles, the components have various velocities in the column and reach its end at different times, where they can be separated. The learning task considered here is to predict one of two characteristic constants of a substance given its chromatogram time series. These two constants are called *Henry* and *Langmuir*. The data sets contained 200 examples with 500 attributes each (500 equidistant time points of the chromatogram time series). The labels for parameter Henry were real numbers between 1 and 10 and the labels for parameter Langmuir were real numbers ranging from 1 to 100.

Based on the structure of the overall learning task described in the previous section, we compared the performance of a number of different learning chains. Table 1 shows the results of the accomplished experiments. The learner, that was used througout the following experiments was a regression SVM (see [7], [6]). We chose a radial-basis kernel with a gamma value of 1, a complexity value of 1000 and an espilon value of 0.1. The evaluation of the learning performance was done using absolute error and standard deviation comparing predicted and real parameter values (for Henry and Langmuir). In the first experiment we simply used the original (time series) features to learn and evaluate a SVM model without any preprocessing step - the corresponding learning chain (comprising learner, model applier and evaluator) was enclosed by a four-fold cross-validation. The second chain additionaly contained an, already mentioned, pre-processing (feature generation) operator, that generated numeric characteristics from the initial time series. This operator significantly reduced the given attribute/feature space from formely 500 to finally six features (see figure 3). For the next three chains we used different feature selection wrappers (namely forward selection, backward elimination and a genetic algorithm) that further reduced the set of possible attributes/features by selecting only the relevant features. The applied genetic algorithm uses a bitstring representation, i.e. an attribute set containing n attributes is represented by a single individual (a bitstring) of length n, whereas a single bit in the bitstring indicates that the corresponding attribute is selected (value 1) resp. deselected (value 0). The setting is that of a standard GA, using 200 generations, a population size of 20 individuals, a mutation probability of 0.01, a crossover probability of 0.5, fitness proportional selection and uniform crossover. The feature selection wrappers again are enclosed by an outer and an inner four-fold cross-validation.

The first experiment acted as a baseline for the

Operator chain	Absolute error (*std. dev.*) for Henry constant	Absolute error (*std. dev.*) for Langmuir constant
Original data	2.067 (*0.295*)	24.125 (*2.970*)
Feature generation	0.167 (*0.044*)	3.078 (*0.9204*)
Feature generation & forward selection	0.057 (*0.0016*)	1.348 (*0.3312*)
Feature generation & backward elimination	0.056 (*0.0557*)	0.997 (*0.1459*)
Feature generation & genetic algorithm	0.054 (*0.0021*)	1.037 (*0.0331*)

Table 1 Absolute error and standard deviation for the target values Henry constant and Langmuir constant for operator chains with and without feature generation and selection.

evaluation of more sophisticated operator chains and showed, as expected, that using only the original attributes results in a poor performance. Generating characteric and more robust (time series) features significantly improved the overall performance. In case of parameter Henry, the usage of the generated feature set lead to a reduction in the absolute error of 92%. In case of parameter Langmuir, the reduction amounted to 87%. A further improvement in the learning performance could be achieved by applying an operator chain that included an additional feature selection wrapper past the feature generation operator. Compared to the result using only a feature generator the error amounted to about 30% of the former error for parameter Henry and to about 40% for parameter Langmuir.

5 Conclusions and Outlook

We proposed YALE as machine learning environment that allows to easily describe even complex nested data mining operator chains and demonstrated its applicapility to an exemplary feature generation and selection application. The realization of the complex experimental set-up was straight forward and the exchange and comparison of different operator chains were not much of an additional effort. The experimental results showed significant improvements through the generation and selection of characteristic, robust features.

For the future, we plan to add a database interface to YALEto alternatively read the examples from a database. Furthermore we intent to implement an interface to WEKA to make use of its large number of learning methods and preprocessing operators, which seems more sensible than re-implementing them. We also plan to implement further data pre-processing operators.

6 Acknowledgments

This work was supported by the Deutsche Forschungsgemeinschaft (DFG), Collaborative Research Center on Computational Intelligence (SFB 531) at University of Dortmund.

References

1. B. Efron and R. Tibshirani. *An introduction to the bootstrap.* Chapman & Hall, New York, USA, 1993.
2. M.A. Hall. *Correlation-based feature selection for machine learning.* Dissertation, Department of Computer Science, University of Waikato, Hamilton, New Zealand, 1999.
3. Ron Kohavi and G.H. John. Wrappers for feature subset selection. *Artificial Intelligence Journal, Special Issue on Relevance*, 97(1–2):273–324, 1997.
4. Ron Kohavi, Dan Sommerfield, and James Dougherty. Data mining using MLC++: A machine learning library in C++. In *Tools with Artificial Intelligence*, pages 234–245, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press. `http://www.sgi.com/tech/mlc/`.
5. John Ross Quinlan. *C4.5: Programs for Machine Learning.* Machine Learning. Morgan Kaufmann, San Mateo, CA, USA, 1993.
6. Stefan Rüping. *mySVM Manual.* Universität Dortmund, Lehrstuhl Informatik VIII, 2000. http://www-ai.cs.uni-dortmund.de/SOFTWARE/MYSVM/.
7. Vladimir N. Vapnik. *Statistical Learning Theory.* Wiley, Chichester, UK, 1998.
8. Ian H. Witten and Eibe Frank. *Data mining: Practical machine learning tools and techniques with Java implementations.* Morgan Kaufman, San Francisco, CA, USA, 2000. `http://www.cs.waikato.ac.nz/ ml/weka/`.