Yale: Yet Another Learning Environment

Tutorial

Simon Fischer, Ralf Klinkenberg, Ingo Mierswa, Oliver Ritthoff

University of Dortmund,
Department of Computer Science,
Chair of Artificial Intelligence,
44221 Dortmund, Germany

{fischer,klinkenberg,mierswa,ritthoff}@ls8.cs.uni-dortmund.de
http://yale.cs.uni-dortmund.de/
http://www-ai.cs.uni-dortmund.de/

June 19, 2002

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In practical data mining applications data often has to be pre-processed to be usable by a chosen machine learning method and to achieve an acceptable level of performance in prediction. One central problem in this context is the representation of the examples by a good set of features, i. e. a set of features that allows the learning method to find a candidate hypothesis solving the learning task at hand within its hypothesis search space. Hence, finding a suitable set of features may be far more important for the overall learning success than the choice of a particular learning method. Therefore it is often necessary to use complex operator chains, combining different pre-processing and learning steps, rather than using a particular single learning scheme. While such methods can be combined manually and by writing 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.

A tool that meets all the described requirements is YALE - **Y**et **a**nother **l**earning **e**nvironment [9], which allows to easily specify and execute such learning chains for pre-processing, especially comprising feature generation and selection as well as multi-strategy learning. This modular, non-commercial environment supports nested operator chains and the exchange of individual operators by alternative operators and thereby the systematic evaluation and comparison of different operators and operator chains for the same (sub)task. To guarantee the re-usability and applicability to new tasks, this machine learning environment was designed to be easily extendable.

For an enhanced scalability and applicability, YALE was designed 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. Thus, the source and the way of handling the example is transparent to the other operators in the data mining chain. For efficiency reasons, YALE does not create copies of the data unless really necessary for the task. This a clear advantage compared to other existing data mining tools (see section 1.2).

1.2 Existing environments

There already exist several machine learning and data mining environments that provide a number of methods from machine learning, statistics, and pattern recognition. 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) [13], developed at University of Waikato, NZ, and MLC++[2] [7], 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 [2]. 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 [3], a wrapper approach [6], and a filter approach [5]. Additionaly WEKA provides meta classifiers like bagging [1] and boosting [11].

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 pre-processing, 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 own experiment and data management program. An additional shortcoming of WEKA is its

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

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.

1.3 Operators and operator chains

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. One central aspect is, that by enclosing other operators or operator chains, operators are arbitrarily *nestable*, so that even complex experimental setups can be built.

For example a nested cross-validation could be used to first optimize some parameters of a data pre-processing and learning chain (inner validation) and to then evaluate the performance of the whole experimental set-up (outer validation).

Operators that enclose other operators or operator chains are often referred to as *wrappers*. Typical examples of wrappers are cross-validation and feature selection wrappers [6]. Section 4 describes an example of such a nested cross-validation experiment for the learning task of selecting a good attribute set.

To explain some basic concepts behind operator chains in YALE, let us first consider a simple learning chain presented in figure 1.1 .

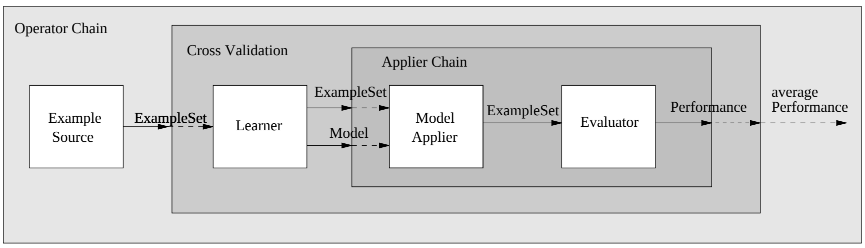

Figure 1.1: A simple example of a nested operator chain.

This figure shows an example of a simple nested operator chain. The outmost operator is an Operator Chain which sequentially applies its inner operators. The first of them is an instance of the operator ExampleSource which loads a set of labelled examples from a file. These examples are passed to a CrossValidation Chain which splits them up into a training and a test set. The Learner uses the training set for the generation of a model which then is used by the ModelApplier to label the test set. The Evaluator compares the generated and the actual labels and returns the performance value. Finally, the enclosing CrossValidation Chain outputs the average performance value of all cross-validation steps.

The operator chain in figure 1.1 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 [8], as a subclass.

Similarly, the things passed between operators can be organized in a hierarchy. As already explained in the previous section, among the objects 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 objects 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 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.

1.4 Data handling

The data is described in an *attribute set description file*[3] and the experiment is specified in a *configuration file*[4] containing a description of the employed

[3] For a more elaborate description of *attribute set description files* see section 3.3.2

[4] A detailed description of *configuration files* is provided in section 3.2

operator chain. 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*). 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.

For almost all operators in YALE it is transparent, whether the examples are read in from a text file, from main memory or from a database, and whether only one or all examples are kept in main memory at a time. The environment only uses an internal structure to iterate over the instances in an example set and hence does not need to distinguish between the different possible data sources and ways of managing the data. The source and way of handling the data only depend on the chosen example source operator and its parameterization.

1.5 Meta knowledge

While the use of meta knowledge is standard in database and software engineering systems, many learning systems do not consider meta knowledge. YALE supports the (optional) use of meta knowledge, especially for the task of *feature generation*.

Attributes in YALE are described by two data structures, the *value type* and the *block type*. While the *value type* of an attribute (or label) specifies the data type of the individual attribute (like e.g. *numerical* or *nominal*), the *block type* contains some meta-data about the attribute, e.g. if it is just an individual attribute or part of a more complex data structure like an interval boundary or a time series. For each data structure, there exists an ontology describing the hierarchical is-more-general-than-/is-superset-of-relation between the different types.

The information about the attribute types of an example is useful for feature generators, which can check their applicability to given attributes by verifying their attribute types. A generator extracting the maximum value of a time series can for example restrict its application to all value series 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 interval attributes.

Chapter 2

Installation notes

2.1 Download

The latest version of YALE is available on the YALE homepage:

`http://yale.cs.uni-dortmund.de/` .

The YALE homepage also contains this document, the YALE javadoc, example datasets, and example configuration files.

2.2 Installation

YALE is completely written in Java, which makes it run on almost every platform. Therefore it requires a Java Runtime Environment (jre) or a Java Development Kit (jdk) version 1.2 or above. Both are available at `http://java.sun.com/`. After unzipping the downloaded YALE version to a directory of your choice, it is necesssary that the `bin` directory is added to the `PATH` environment variable.

Congratulations: YALE is now installed. In order to check if YALE is working correctly, you can change to another directory and create the following file `installtest.xml`:

```
<operator name="Global" class="OperatorChain">
</operator>
```

Figure 2.1: Installation test

It shows the simplest experiment which can be done with YALE. More precisely, it isn't even an experiment, because nothing will be done. Nevertheless, start YALE typing `yale installtest.xml` and you should see the message "Experiment finished successfully" after a few moments if everything goes well. Otherwise the words "Experiment not successful" or another error message can be read. In this case something is wrong with the installation. Please check if you execute the correct file.

2.3 General settings

Certainly you have seen that a running installation of Java is essential for working with YALE. The possibility of using external programs such as machine learning methods is discussed in the operator reference (chapter 5). These programs must have been properly installed and must be executable without YALE, before they can be used in any YALE experiment. You have to specify the paths to this programs and some other general or platform dependant settings which do not need to be declared in every experiment file. Similarly one can specify different settings for each user.

In order to use the external operators, you need to download the software called by these operators, i.e. some of the following. Please make sure that the programs run properly without using YALE first and specify the paths to the programs as described below.

- mySVM is Stefan Rüping's implementation of a support vector machine [12].
  `http://www-ai.informatik.uni-dortmund.de/SOFTWARE/MYSVM/`
- SVM^{light} is Thorsten Joachim's implementation of a support vector machine.
  `http://svmlight.joachims.org/`
- The Weka package contains many learning and clustering algorithms. In order to use Weka, you do not need to specify a path in a config file, but simply put the file `weka.jar` from the downloaded Weka library in the `lib` directory of YALE. Please refer to the description of the Weka operators in section 5.3.16.
  `http://www.cs.waikato.ac.nz/ml/weka/`
- C4.5 is a widely used descision tree learner by Ross Quinlan [8].
  `http://www.cse.unsw.edu.au/~quinlan/`

You can specify the paths to external programs, which of course may be platform dependent in a network environment, or general settings, which should be applied for all users, in the YALE configuration files, which can be found in the `etc` directory. Additionaly, a user can declare options which exclusively hold for himself. In each line of these files a parameter can be specified by *key* = *value*. Comments start with #. Table 2.1 shows all possible keys. Please also consult the operator reference in chapter 5.

There are several ways to use these settings files. You can store them in different places or pass them to YALE via the command line. In all cases, local settings are more binding than global settings.

Key	**Description**
`yale.logfile.format`	declares if YALE should format the log file (`true` or `false` or `yes` or `no`)
`yale.mysvm.learncommand`	absolute path to the mySVM model learner
`yale.mysvm.applycommand`	absolute path to the mySVM model applier
`yale.svmlight.learncommand`	absolute path to the SVM^{light} model learner
`yale.svmlight.applycommand`	absolute path to the SVM^{light} model applier
`yale.c45.learncommand`	absolute path to the C4.5 decision tree learner
`yale.c45.rulecommand`	absolute path to the C4.5 tree to rule converter

Table 2.1: Possible global settings

global: In the `etc` subdirectory of the YALE directory with names `yalerc` for global settings or `yalerc.`*OS* for platform dependent global settings like absolute pathes to the learning methods. *OS* can be `SunOS`, `Linux` or other platforms supported by Java. You can pass the place where the global files are stored to YALE with the option `-Dyale.globalrc=filename` .

user: In his home directory, each user can declare settings that only apply to him in configuration files named `.yalerc` and/or `.yalerc.`*OS*.

directory: Files named like the files mentioned under global are also searched for in the current working directory from where YALE is invoked. All experiments executed in this directory use these settings.

commandline: After all you can start YALE with the option `-Dyale.rcfile = filename` , e.g.

```
yale -Dyale.rcfile=mySettings myExperiment.xml
```

YALE searches for the given file `mySettings` and for `mySettings.`*OS* with the current value of *OS*, which is the value of the Java property *os.name*. If you don't know the value of this porperty, scan the output of YALE for the configuration files it tries to read. If you want to use different settings for a certain experiment you can pass these settings to YALE in this way.

Chapter 3

First steps

3.1 First example

Let's start with a simple example similar to the one you know from the introduction (*simpleexample.xml*). The example at hand loads an example set from a file, generates a model using a support vector machine (SVM) and evaluates the performance of the SVM on this dataset by estimating the expected absolute and squared error by means of a ten-fold cross-validation. In the following we will describe what the parameters mean without going into detail to much. We will describe the used operators later on in this section.

But first of all let's start the experiment. We assume that your current folder contains the two files *simpleexample.xml* (see figure 3.1) and *attributes.xml*. Also make sure you have a `tmp` subdirectory to store temporary files in. Now start YALE typing `yale simpleexample.xml`. After a short while you should read the words "Experiment finished successfully". Congratulations, you just made your first YALE experiment. If you read "Experiment not successful" instead, something went wrong. In either case you should have a newly generated file named *simpleexample.log* in your working directory. In the latter case it should give you information about what went wrong. All kinds of debug as well as information messages and the measured absolute and squared errors are written to this file. Have a look at it now.

The log file starts with the experiment tree and contains a lot of warnings, because most of the parameters are not set. Don't panic, reasonable default values are used for all of them. At the end you find the experiment tree again. The number in squared brackets following each operator gives the number of times the operator was applied. It is one for the outer operators and ten within the ten-fold cross-validation. Every time an operator is applied a message is written to the log file indicating its input objects (like example sets and models). When the operator terminates its application it

```
<operator name="Global" class="OperatorChain">
  <parameter key="logfile"          value="simpleexample.log"/>
  <parameter key="logverbosity"     value="0"/>
  <parameter key="temp_dir"         value="./tmp"/>
  <operator name="ExampleSource" class="ExampleSource">
    <parameter key="attributes" value="attributes.xml"/>
  </operator>
  <operator name="XValid" class="XValidation">
    <parameter key="number_of_validations" value="10"/>

    <parameter key="type"           value="polynomial"/>
    <parameter key="degree"         value="3"/>

    <operator name="Learner" class="SVMLearner" parentlookup="1"/>
    <operator name="ApplierChain" class="OperatorChain">
      <operator name="Applier" class="SVMApplier" parentlookup="2"/>
      <operator name="Performator" class="PerformanceEvaluator">
        <parameter key="criteria_list" value="absolute squared"/>
      </operator>
    </operator>
  </operator>
</operator>
```

Figure 3.1: Simple example configuration file

writes the output to the log file again. You can find the average performance estimated by the cross-validation close to the end of the file.

Taking a look at the experiment tree in the log file once again, you will quickly understand how the configuration file is structured. There is one `operator` tag for each operator specifying its name and class. Names must be unique and have the only purpose of distinguishing between instances of the same class. Operator chains like the cross-validation chain may contain one or more inner operators. Parameters can be specified in the form of key-value pairs using a `parameter` tag.

We will now focus on the operators without going into detail too much. If you are interested in the the operator classes, their input and output objects, parameters, and possible inner operators you may consult the reference section of this tutorial (chapter 5).

The outermost operator called "Global" is an OperatorChain. An operator chain works in a very simple manner. It applies its inner operators successively passing their respective output to the next inner operator. The ouptut of the operator chain is the output of the last inner operator. While normal operator chains do not take any parameters, this particular operator chain (being the outermost operator) has two parameters: the name of the log file (`logfile`) and the name of the directory for temporary files (`temp_dir`).

The ExampleSource operator loads an example set from a file. An additional file containing the attribute descriptions is specified (*simpleexample.xml*). References to the actual data files are specified in this file as well (see section 3.3 for a description of the files). The resulting example set is then passed to the cross-validation chain.

The CrossValidation evaluates the learning method by splitting the input example set into ten subsets $S_1, \ldots, S_{10}$. The inner operators are applied ten times. In run number i the first inner operator, which is a SVMLearner, generates a model using the training set $\bigcup_{j \neq i} S_j$. The second inner operator, an evaluation chain, evaluates this model by applying it to the remaining test set S_i. The SVMApplier predicts labels for the test set and the PerformanceEvaluator compares them to the real labels. Afterwards the absolute and squared errors are calculated. Finally the cross-validation chain returns the average absolute and squared errors over the ten runs and their variances.

3.2 Configuration files

Configuration files are XML documents containing of only three types of tags. They define the operator tree and the parameters for the operators. Parameters can have a single value or a set of values. This feature must not be mistaken. If a parameter is defined as a set of values, then certain operators facilitate repetitive execution of operator chains with the different parameter values. Without those special operators only the first value for each parameter will be used.

operator

The `operator` tag represents one instance of an operator class. Three attributes can be set:

name: A unique name identifying this particular operator instance

class: The operator class. See the operator reference (chapter 5) for a list of operators.

parentlookup: If this parameter is set to the integer n, then every time a parameter for this operator is queried, but undefined for this operator, the n enclosing operator chains will be queried for the parameter as well. The default value is zero.

For instance, an operator tag for an operator that reads an example set from a file might look like this:

```
<operator name="MyExampleSource" class="ExampleSource">
```

If `class` is a subclass of `OperatorChain`, then nested operators may be contained within the opening and closing tag. Parameters for this operator are defined by means of `parameter` tags.

parameter

As discussed above, a parameter can have a single value or a set of values. In either case it must have a name which is unique for the operator. The attributes of the `parameter` tag are as follows:

key: A unique name for the parameter.

value: The value of the parameter. The character '$' has a special meaning. Parameter values are expanded as follows:

- **$n** becomes the name of the operator.
- **$c** becomes the class of the operator.
- **$a** becomes the number of times the operator was applied.
- **$t** becomes the system time.
- **$$** becomes $.

group: This attribute is optional and rarely needed. The respective operator description gives information about how grouped parameters are interpeted. The `group` attribute is used when the `key` does not have a fixed value.

In order to specify a filename for the above example one might use the following parameter:

```
<parameter key="attributes" value="myexamples.txt"/>
```

If the `value` attribute is not used, all text between the opening and closing tag is interpreted as the value. Similarly one can use the `set` tag to define a set of values.

set

This tag can only be used within a `parameter` tag. All enclosed text is interpreted as a comma separated list of parameter values. See section 5.5.9 for the purpose of this tag.

3.3 Input files

YALE knows four kinds of input files, which are discussed in this section.

3.3.1 Example data files

The data for an example set can be distributed over several files. Therefore it is possible to specify the way example files are parsed, there is a default meeting the usual requirements: examples are separated by newlines, the columns are separated by whitespace.

3.3.2 Attribute set description files

Attribute set description files are read by the ExampleSource operator. They are simple XML documents defining the properties of the attributes (like their name and range) and their source files.

The outer tag must be an `<attributeset>` tag. The only attribute of this tag may be `default_source=`*filename*. This file will be used as a default file if it is not specified with the feature.

The inner tags can be at most one `<label>` and at most one `<weight>` tag and an arbitrary number of `<attribute>` tags. Apart from their obvious meaning all of them have the same attributes:

`name`: The name of the attribute.

`sourcefile`: The name of the file containing the data. If this name is not specified, the default file is used.

`sourcecol`: The column within this file (numbering starts at 1). If this attribute is a label and your dataset is unlabelled you can omit the `sourcecol` attribute. Note that you cannot omit the whole `label` tag if you have unlabelled data and intend to label it! In that case you can specify this by setting `sourcecol` to `none`.

`sourcecol_end`: If this parameter is set, its value must be greater than the value of `sourcecol`. In that case, $sourcecol - sourcecol\_end$ attributes are generated with the same properties. Their names are generated by appending numbers to the value of `name`. If the `blocktype` is `value_series`, then `value_series_start` and `value_series_end` respectively are used for the first and last attribute bocktype in the series.

`unit`: The unit given as a list of m, s, kg, A, K, cd, mol followed by an exponent (only necessary if different from 1). Example : `kgms-2` ($= \frac{kg \cdot m}{s^2} = Newton$). The specification of of a unit is optional.

`valuetype`: One out of `nominal`, `numeric`, `integer`, `real`, and `ordered`.

`blocktype`: One out of `single_value`, `value_series`, `value_series_start`, and `value_series_end`

`blocknumber`: A block number.

`classes`: A whitespace-separated list of values for nominal attributes. For classification learners that can handle only binary classifications ("positive" and "negative") the first entry in this list is assumed to be the positive label.

See figure 3.2 for an example attribute description file.

3.3.3 Model files

Model files contain the models generated by learning operators in previous YALE runs. They can be read in and applied by the appropriate model applier, i.e. the appropriate subclass of ModelApplier.

3.3.4 Attribute generation files

An AttributeSetWriter can write an attribute set to a text file. This file can later be used by a FeatureGeneration operator to generate the same set of attributes in another experiment and/or for another set of data.

The attribute generation files can as well be generated by hand. Every line is of the form $f(attribute\_name_1, \ldots, attribute\_name_n)$. The functions can as well be nested. An example of a nested generation description might be: $f(g(a), h(b), c)$. See page 5.6.8 for a reference of the available functions.

```
<attributeset default_source="polynom.dat">
  <attribute name         = "att1"
             sourcecol    = "1"
             valuetype    = "real"
             blocktype    = "single_value"
             blocknumber  = "1"
             unit         = ""
  />
  <attribute name         = "att2"
             sourcecol    = "2"
             valuetype    = "real"
             blocktype    = "single_value"
             blocknumber  = "2"
             unit         = ""
  />
  <attribute name         = "att3"
             sourcecol    = "3"
             valuetype    = "real"
             blocktype    = "single_value"
             blocknumber  = "3"
             unit         = ""
  />
  <attribute name         = "att4"
             sourcecol    = "4"
             valuetype    = "real"
             blocktype    = "single_value"
             blocknumber  = "4"
             unit         = ""
  />
  <attribute name         = "att5"
             sourcecol    = "5"
             valuetype    = "real"
             blocktype    = "single_value"
             blocknumber  = "5"
             unit         = ""
  />
  <label name         = "label"
         sourcecol    = "6"
         valuetype    = "real"
         blocktype    = "single_value"
         blocknumber  = "6"
         unit         = ""
  />
</attributeset>
```

Figure 3.2: An example attribute set description file in XML syntax.

Chapter 4

Advanced experiments

At this point, we assume that you are familiar with the simple example from section 3.1. You should know how to read a dataset from a file, what a learner and an applier do, and how a cross-validation chain works. These operators will be used frequently and without further explanation in this chapter.

4.1 Feature selection

Let us assume that we have a dataset with numerous attributes. We would like to test, whether all of these attributes are really relevant, or whether we can get a better model by omitting some of the original attributes. This task is called *feature selection* and the *backward elimination* algorithm is an approach that can solve it for you.

Here is how backward elimination works within YALE: Enclose the cross-validation chain by a FeatureSelection operator. This operator repeatedly applies the cross-validation chain, which now is its inner operator, until the specified stopping criterion is complied with. The backward elimination approach iteratively removes the attribute whose removal yields the largest performance improvement. The stopping criterion may for example be that there has been no improvement for a certain number of steps. See page 67 for a detailed description of the algorithm. Figure 4.1 shows the configuration file.

You can try most of the algorithms in YALE with small modifications of this configuration file. You can try some of the following things:

- Use *forward selection* instead of backward elimination by changing the parameter value of *selection_direction* from `backward` to `forward`. This approach starts with an empty attribute set and iteratively adds the attribute whose inclusion improves the performance the most.

- Use the GeneticAlgorithm operator for feature selection instead of the

FeatureSelection operator (see page 69).

- Compare the results of the three approaches above to the BruteForce operator. The brute force approach tests all subsets of the original attributes, i.e. all combinations of attributes, to select an optimal subset. While this operator is prohibitively expensive for large attribute sets, it can be used to find an optimal solution on small attribute sets in order to estimate the quality of the results of other approaches.

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

  <parameter key="logfile"       value="advanced1.log"/>
  <parameter key="logverbosity"  value="0"/>
  <parameter key="temp_dir"      value="./tmp"/>

  <operator name="Input" class="ExampleSource">
    <parameter key="attributes" value="./attributes.xml"/>
  </operator>

  <operator name="BackwardElimination" class="FeatureSelection">
    <parameter key="selection_direction" value="backward"/>

    <operator name="XVal" class="XValidation">
      <parameter key="number_of_validations" value="5"/>

      <parameter key="type"    value="polynomial"/>
      <parameter key="degree"  value="3"/>
      <operator name="Learner" class="SVMLearner" parentlookup="1"/>
      <operator name="ApplierChain" class="OperatorChain">
        <operator name="Applier" class="SVMApplier" parentlookup="2"/>
        <operator name="Evaluator" class="PerformanceEvaluator">
          <parameter key="criteria_list" value="absolute"/>
        </operator>
      </operator>
    </operator>
  </operator>
</operator>
```

Figure 4.1: A feature selection experiment

4.2 Splitting up experiments

If you are not a computer scientist but a data mining user, you are probably interested in a real-world application of YALE. May be, you have a small labelled dataset and would like to train a model with an optimal attribute set. Later you would like to apply this model to your huge unlabelled database. Actually you have two separate experiments.

4.2.1 Learning a model

This phase is basically the same as described in the preceeding section. We append two operators to the configuration file that write the results of the experiment to files. First, we write the attribute set to the file `selected_attributes.att` using an AttributeSetWriter. Second, we once again train a model, this time using the entire example set, and we write it to the file `model.mod`. For the configuration file see figure 4.2. Execute the experiment and take a look at the file `attributes.att`. It should contain the selected subset of the originally used attributes, one per line.

4.2.2 Applying the model

In order to apply this learned model to new unlabelled dataset, you first have to load this example set using an ExampleSource. You can now load the trained model using a ModelLoader. Unfortunately, your unlabelled data probably still uses the original attributes, which are incompatible with the model learned on the reduced attribute set. Hence, we have to transform the examples to a representation that only uses the selected attributes, which we saved to the file `attributes.att`. The FeatureGenerator loads this file and generates (or rather selects) the attributes accordingly. Now we can apply the model and finally write the labelled data to a file. See figure 4.3 for the corresponding configuration file.

As you can see, you can easily use different dataset source files even in different formats as long as you use consistent names for the attributes. You could also split the experiment into three parts:

1. Find an optimal attribute set and train the model.
2. Generate or select these attributes for the unlabelled data and write them to temporary files.
3. Apply the model from step one to the temporary files from step two and write the labelled data to a result file.

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

  <parameter key="logfile"        value="advanced2.log"/>
  <parameter key="logverbosity"   value="0"/>
  <parameter key="temp_dir"       value="./tmp"/>

  <parameter key="type"           value="polynomial"/>
  <parameter key="degree"         value="3"/>

  <operator name="Input" class="ExampleSource">
    <parameter key="attributes" value="./attributes.xml"/>
  </operator>

  <operator name="BackwardElimination" class="FeatureSelection">
    <parameter key="selection_direction" value="backward"/>

    <operator name="XVal" class="XValidation">
      <parameter key="number_of_validations" value="2"/>

      <operator name="Learner" class="SVMLearner" parentlookup="3"/>
      <operator name="ApplierChain" class="OperatorChain">
        <operator name="Applier" class="SVMApplier" parentlookup="4"/>
        <operator name="Evaluator" class="PerformanceEvaluator">
          <parameter key="criteria_list" value="absolute"/>
        </operator>
      </operator>
    </operator>
  </operator>

  <operator class="AttributeSetWriter" name="AttributeSetWriter">
    <parameter key="attribute_set_file" value="selected_attributes.txt"/>
  </operator>
  <operator name="ModelWriter" class="SVMLearner" parentlookup="1">
    <parameter key="model_file" value="model.mod"/>
  </operator>
</operator>
```

Figure 4.2: Training a model and writing it to a file

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

  <parameter key="logfile"         value="advanced3.log"/>
  <parameter key="logverbosity"    value="0"/>
  <parameter key="temp_dir"        value="./tmp"/>

  <operator name="Input" class="ExampleSource">
    <parameter key="attributes" value="./attributes.unlabelled.xml"/>
  </operator>

  <operator name="FeatureGenerator" class="FeatureGeneration">
    <parameter key="filename" value="selected_attributes.txt"/>
  </operator>

  <operator name="ModelLoader" class="ModelLoader">
    <parameter key="model_file" value="model.mod"/>
  </operator>

  <operator name="Applier" class="SVMApplier">
    <parameter key="type"          value="polynomial"/>
    <parameter key="degree"        value="3"/>
  </operator>

  <operator class="ExampleSetWriter" name="ExampleSetWriter">
    <parameter key="example_set_file" value="polynom.labelled.dat"/>
  </operator>
</operator>
```

Figure 4.3: Applying the model to unlabelled data

4.3 Parameter and performance analysis

In this section we show how one can easily record performance values of an operator or operator chain depending on varied parameter values. In order to achieve this, the YALE experiment chain described in this section makes use of two new YALE operators: ParameterOptimization and ExperimentLog.

We will see how to analyse the performance of a support vector machine (SVM) with a polynomial kernel depending on the two parameters (polynom) *degree* and ε.[1] We start with the usual experiment core: a validation chain containing a SVMLearner, a SVMApplier, and a PerformanceEvaluator. Now we would like to vary the parameters. In order to provide a set of parameters one can use the `set` tag. Figure 4.4 shows how this works: within the `parameter` tag there must be a `set` tag holding a comma-separated list of values. Since we have four different values for the parameter (polynom) *degree* and five different values for the parameter ε, there are 20 possible parameter combinations. The ParameterOptimization operator applies its inner operators 20 times using a different one of these combinations each time. Finally the ParameterOptimization operator returns an optimal parameter value combination. In our case, a polynom degree of 3 and the smallest tested ε, i.e. 0.01, should be selected, because this combination produces the optimal performance among the parameter value combinations tested, i.e. it minimizes the absolute regression error.

In order to create a chart showing the absolute error over the parameters *degree* and ε, we can create a datafile using the ExperimentLog operator. All parameters in the group `log` are evaluated. The key provides the name for the column and the value specifies where to retrieve the value or parameter from. See page 42 for details about the format. Figure 4.5 shows the resulting file. You can use tools like `gnuplot` to generate a fancy chart like that in figure 4.6.

Now you already know quite a lot about YALE. You can try to combine some of the experiments of this chapter:

- Analyse the performance of a feature selection algorithm over time (current number of generations).
- Try to optimize the parameters of a genetic algorithm (but be sure to have a good book to read meanwhile).

[1] The performance of a polynomial SVM also depends on other parameters like e.g. C, but this is not the focus of this experiment.

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

  <parameter key="logfile"        value="advanced4.log"/>
  <parameter key="logverbosity"   value="0"/>
  <parameter key="temp_dir"       value="./tmp"/>

  <operator name="Input" class="ExampleSource">
    <parameter key="attributes" value="./attributes.xml"/>
  </operator>

  <operator name="ParameterOptimization" class="ParameterOptimization">

    <operator name="Validation" class="RandomSplitValidationChain">
      <parameter key="split_ratio" value="0.5"/>

      <parameter key="type"  value="polynomial"/>
      <parameter key="epsilon"><set>0.01,0.03,0.05,0.075,0.1</set></parameter>
      <parameter key="degree"><set>1,2,3,4</set></parameter>

      <operator name="Learner" class="SVMLearner" parentlookup="1"/>
      <operator name="ApplierChain" class="OperatorChain">
        <operator name="Applier" class="SVMApplier" parentlookup="2"/>
        <operator name="Evaluator" class="PerformanceEvaluator">
          <parameter key="criteria_list" value="absolute"/>
        </operator>
      </operator>
    </operator>

    <operator name="ExpLog" class="ExperimentLog">
      <parameter key="filename" value="svm_degree_epsilon.txt"/>
      <parameter group="log"
                 key="degree"
                 value="operator.Learner.parameter.degree"/>
      <parameter group="log"
                 key="epsilon"
                 value="operator.Learner.parameter.epsilon"/>
      <parameter group="log"
                 key="absolute"
                 value="operator.Validation.value.performance"/>
    </operator>

  </operator>
</operator>
```

Figure 4.4: Parameter and performance analysis

```
# Generated by
# ExpLog[edu.udo.cs.yale.operator.ExperimentLogOperator]
# degree  epsilon   absolute
1         0.01      53.687639080124534
1         0.03      49.38624625677904
1         0.05      47.49028865368657
1         0.075     46.32938028661449
1         0.1       59.45448890118538
2         0.01      21.846579110313726
[...]
```

Figure 4.5: The performance of a SVM (`gnuplot` input data file automatically generated by YALE)

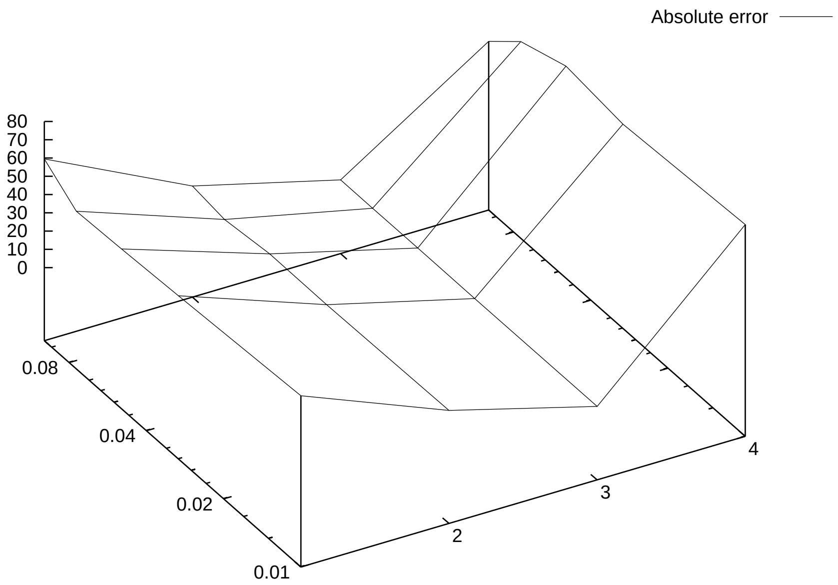

Figure 4.6: The performance of a SVM (plot generated by `gnuplot`)

Chapter 5

Operator reference

This chapter describes the built-in operators that come with YALE. Each operator section is subdivided into several parts.

1. An enumeration of the required input and the generated output objects. The input objects are usually consumed by the operator and are not part of the output. Operators may receive more input objects than required. In that case the unused input objects will be appended to the output and can be used by the next operator.

2. If the operator is an operator chain: A list of the required inner operators.

3. The parameters that can be used to configure the operators. Ranges and default values are specified. The values for boolean parameters can be `yes`, `true`, `on`, `no`, `false`, and `off`. Required parameters are indicated by bullets (•) and optional parameters are indicated by an open bullet (◦)

4. A list of values that can be logged using the ExperimentLog operator (see p. 42).

5. A textual description of the operator.

Notice that some operators extend the functionality of other operators. This is indicated by the word *extends*. In this case, the operator inherits its parent's input and output classes, parameters, and values. Actually all operators extend Operator or OperatorChain but this is not indicated.

Some of the operators are *abstract* and can not be used. They only serve as a superclass for other operators that have a common behaviour or purpose. Abstract classes are set *italic*.

5.1 Basic operators

5.1.1 *Operator* (abstract)

Values

- *applycount:* the number of times this operator was applied
- *time:* the time elapsed since the application of this operator started
- *looptime:* the time elapsed since the last application loop started (only relevant for some wrappers which call the `inApplyLoop()` method)

Description: This is the abstract superclass of all operators. It has no semantics apart from some basic values, that can be queried.

5.1.2 OperatorChain

Input

- the first inner operator's input

Output

- the last inner operator's output

Description: An operator chain consecutively calls its inner operators passing their respective output to the next inner operator.

5.2 Input/Ouput operators

In this section operators are described which provide functionalty to read or write data and results respectively.

5.2.1 ExampleSource

Input

- none

Output

- ExampleSet: the example set which is generated from the data in the given file

Parameters

- *attributes:* filename for the attribute XML file, see section 3.3
- ○ *separator_chars:* the characters given by this string are separators between the values. Default values are all whitespace characters and ','
- ○ *ignore_chars:* the characters given by this string are ignored by the operator
- ○ *comment_chars:* lines beginning with these characters will be ignored by the operator
- ○ *max_examples:* the maximum number of examples to read from the file, default: -1 (read all examples)

Description: This operator reads an example set from one or several files. You can specify several delimiter characters and characters that will be ignored totally. Additionally, comment characters can be given. The characters " and ' quote intermediate text. The default values of the ExampleSource operator can probably be used for most commonly used text file data formats.

5.2.2 DatabaseExampleSource

Input

- none

Output

- ExampleSet: the example set read from the database

Parameters

- *attributes:* filename for the attribute XML file, see section 3.3
- *query_file:* a file containing the SQL query
- ◦ *sample_size:* the maximum number of examples to read from the database

Description: This operator reads an example set from an SQL database.

5.2.3 ExampleSetWriter

Input

- ExampleSet: the example set to be written to a file

Output

- ExampleSet: the example set which has been written to a file

Parameters

- *example_set_file:* name of the file the example set is to be written to

Description: This operator writes the values of all examples and labels of the example set into a file. Every line represents one example and is written in the following form:

```
[value1 value2...valueN] label1...labelN.
```

5.2.4 AttributeSetWriter

Input

- ExampleSet: the attributes of this example set will be written to the given file

Output

- ExampleSet: the input example set is returned

Parameters

- *attribute_set_file:* name of the file the attributes are to be written to

Description: This operator writes all attributes of an example set to a file. Each line holds the construction description of one attribute.

5.2.5 ModelLoader

Input

- none

Output

- Model: the loaded model

Parameters

- *model_file:* name of the file containing the model

Description: This operator loads a model from the given file, which has been written by a learning operator (see section 5.3.1). Afterwards, the model is passed on and can be used by subsequent operators.

5.2.6 ResultWriter

Input

- none

Output

- none

Description: This simple operator can be used at any position in an OperatorChain. It writes all results of the input objects into the result file and simply returns the objects.

5.2.7 ExperimentLog

Input

- none

Output

- none

Parameters

- *filename:* name of the output file
- *log:* in this group you specify the values of the parameters which should be looked up by the operator

Description: This operator saves almost arbitrary data to a given file and is therefore more powerful than the simple ResultWriter operator. All parameters in the group `log` are interpreted as follows: The key determines the column name in the log file. The value specifies where to get the value from. All values can be registered in the *log* parameter group by

`operator`.*opName*.{`parameter`|`value`}.*name*

and are described in the values list in each operator section.

Example: `operator.GA.value.generation` looks up the operator with the name "GA" and then searches for the current value of the field `generation`. Or `operator.SVMLearner.parameter.C` logs the value of the parameter `C` of the SVMLearner.

Hint: If you want to sort the output in some way, try the following trick: Specify the columns in decreasing order (the output will presumably respect this). Then use a Unix command like "sort" or any other tool to sort the output.

5.3 Machine learning algorithms

Acquiring knowledge is fundamental for the development of intelligent systems. The operators described in this section were designed to automatically discover hypotheses to be used for future decisions. They can learn models from the given data and apply them to new data to predict a label for each observation in an unpredicted example set.

5.3.1 *Learner* (abstract)

Input

- ExampleSet: the training example set

Output

- Model: the learned model

Parameters

- *model_file:* name of the file that the Learner writes the learned model into

Description: A Learner is an operator that encapsulates the learning step of a machine learning method. This can be an external program or realised within YALE.

5.3.2 *ModelApplier* (abstract)

Input

- ExampleSet: the examples to which the model is applied to
- Model: the model for predicting the labels of the examples

Output

- ExampleSet: the example set with predicted labels

Description: A ModelApplier predicts the labels of an example set by means of a previously learned model.

5.3.3 SVMLearner
***extends* Learner (p. 43)**

Input

- ExampleSet: the training example set

Output

- Model: the learned model

Parameters

- ◦ *model_file:* the Learner writes the learned model into this file
- *C:* the SVM complexity constant (*capacity constant*)
- *epsilon:* the SVM insensitivity constant
- *type:* the type of the SVM kernel; legal values are `dot` (linear kernel), `polynomial`, `radial` (RBF kernel), `neural` (sigmoidal kernel) or `anova`
- ◦ *degree:* only needed for `polynomial` or `anova` kernel
- ◦ *gamma:* only needed for `radial` or `anova` kernel
- ◦ *a:* only needed for `neural` kernel
- ◦ *b:* only needed for `neural` kernel
- ◦ *pattern:* use SVM for pattern recognition (classification); if this value is not given, the SVM will be used for regression, which is the default.
- ◦ *weighted_examples:* if the value of this parameter is true, a weight for each example will be used by the SVM. If this parameter is not specified, the SVM automatically uses weights if and only if they are given in the example set.

Description: The SVMLearner encapsulates an external implementation of Vladimir Vapnik's Support Vector Machine (SVM) [12], the mySVM learning algorithm by Stefan Rüping[1]. The path to the program should be given by the global parameter *yale.mysvm.learncommand* (see section 2.3 for download and parameter settings information). It supports pattern recognition, regression, and distribution estimation. Additionally, each instance can be weighted for learning. For a more extensive description, especially concerning the parameters, please consult [10] and [12].

---

[1] `http://www-ai.cs.uni-dortmund.de/SOFTWARE/MYSVM/`

5.3.4 SVMApplier
extends **ModelApplier (p. 43)**

Input

- ExampleSet: the examples to which the model is applied to
- Model: the model for predicting the labels of the examples

Output

- ExampleSet: the example set with predicted labels

Parameters

- *C:* the SVM complexity constant (*capacity constant*)
- *epsilon:* SVM insensitivity constant
- *type:* the type of the SVM kernel; legal values can be `dot` (linear kernel), `polynomial`, `radial` (RBF kernel), `neural` (sigmoidal kernel) or `anova`
- ◦ *degree:* only needed for `polynomial` or `anova` kernel
- ◦ *gamma:* only needed for `radial` or `anova` kernel
- ◦ *a:* only needed for `neural` kernel
- ◦ *b:* only needed for `neural` kernel
- ◦ *pattern:* use SVM for pattern recognition (classification); if this value is not given, the SVM will be used for regression, which is the default.
- ◦ *weighted_examples:* if the value of this parameter is true, a weight for each example will be used by the SVM. If this parameter is not specified, the SVM automatically uses weights if and only if they are given in the example set.

Description: The SVMApplier encapsulates an external implementation of the SVM, the mySVM applier algorithm by Stefan Rüping. The SVMApplier can use a model learned by a SVMLearner to predict the labels of examples. The path to the program should be given by the global parameter *yale.mysvm.learncommand* (see section 2.3 for download and parameter settings information). The model (i.e. the calculated hyperplane) is used to predict the labels of the example set. For a more extensive description, especially concerning the parameters, please consult [10] and [12].

5.3.5 SVMLightLearner
extends **Learner (p. 43)**

Input

- ExampleSet: the training example set

Output

- Model: the learned model

Parameters

- ◦ *model_file:* name of the file into which the Learner writes the learned model
- • *kernel_type:* type of the SVM^{light} kernel (`linear`, `polynomial`, `RBF`, or `sigmoid`). The default value for the kernel type is `linear`.
- ◦ *additional_parameters:* other parameters for the SVM^{light}, written like in the SVM^{light} command line, and passed directly like this to SVM^{light}.

Description: SVM^{light} is an implementation of Vladimir Vapnik's Support Vector Machine[12] for pattern recognition (classification) by Thorsten Joachims[2]. The algorithm has scalable memory requirements and can handle problems with many thousands of support vectors efficiently. This learner writes the training set into a file and calls the native version of the SVM^{light} which must be given by the global parameter *yale.svmlight.learncommand* (see section 2.3 for download and parameter settings information). It works fine for classification problems with two classes. If you need more than two classes, you should use the MultiClassLearner described in section 5.3.14. For a more extensive description, especially concerning the parameters, please consult [4].

[2]`http://svmlight.joachims.org`

5.3.6 SVMLightApplier
extends **ModelApplier (p. 43)**

Input

- ExampleSet: the examples to which the model is applied to
- Model: the model for predicting the labels of the examples

Output

- ExampleSet: the example set with predicted labels

Parameters

- *kernel_type:* type of the SVM^{light} kernel (`linear`, `polynomial`, `RBF`, or `sigmoid`). The default value is `linear`
- ◦ *additional_parameters:* other parameters for the SVM^{light}, written like in the SVM^{light} command line, and passed directly like this to SVM^{light}.

Description: This model applier calls the native version of the SVM^{light} model applier by Thorsten Joachims [4], whose path must be given by the global parameter *yale.svmlight.applycommand* (see section 2.3 for download and parameter settings information). This operator predicts the labels of the given ExampleSet.

5.3.7 C45Learner

extends **Learner (p. 43)**

Input

- ExampleSet: the training example set

Output

- Model: the learned model

Parameters

- *model_file:* name of the file the Learner writes the learned model into
- *s:* force subsetting of all tests based on discrete attributes with more than two values.
- *p:* probabilistic thresholds used for continuous attributes
- *g:* use the gain criterion to select tests. The default uses the gain ratio criterion.
- *m:* In all tests, at least two branches must contain a minimum number of objects (default 2). This option allows to alter the minimum number.
- *c:* sets the pruning confidence level (default 25%)

Description: C45Learner wraps the C4.5-algorithm by Quinlan [8]. It learns a decision tree, which it then transforms into a set of rules. It uses the external implementation whose path should be specified using the global parameter *yale.c45.learncommand* (see section 2.3 for download and parameter settings information) and delivers a RuleSet as model.

5.3.8 RuleSetApplier

extends **ModelApplier (p. 43)**

Input

- ExampleSet: the examples to which the model is applied to
- Model: the model for predicting the labels of the examples

Output

- ExampleSet: the example set with predicted labels

Description: The RuleSetApplier requires a RuleSet model as input (like the one learned by a C45Learner) and predicts the labels of the given example set.

5.3.9 ID3Learner
extends **Learner (p. 43)**

Input

- ExampleSet: the training example set

Output

- Model: the learned model

Parameters

- *model_file:* name of the file the Learner writes the learned model into

Description: ID3Learner is an internal learner based on Quinlan's [8] ID3 decision tree induction algorithm. It is a simple classification learner which can only handle nominal attribute values and delivers a decision tree as model.

The operator identifies the best attribute using the information gain criterion in each iteration step. Then it divides the example set according to the values of the specified attribute. This is done recursively until all examples of a created subset belong to the same class and this fact holds for all created example subsets.

5.3.10 DecisionTreeLearner
extends **ID3Learner (p. 49)**

Input

- ExampleSet: the training example set

Output

- Model: the learned model

Parameters

- *model_file:* name of the file the Learner writes the learned model into

Description: DecisionTreeLearner is also an internal learner, based on the ID3Learner. It can handle both nominal and numeric attributes, but can not process incomplete examples or prune the learned tree afterwards. The DecisionTreeLearner merges the subsets that are divided by one attribute and belong to the same class afterwards.

The operator identifies the best attribute using the information gain criterion in each iteration step. Then it divides the example set according to the values of the specified attribute. This is done recursively until all examples of a created subset belong to the same class and this fact holds for all created example subsets.

5.3.11 DecisionTreeApplier *extends* ModelApplier (p. 43)

Input

- ExampleSet: the examples to which the model is applied to
- Model: the model for predicting the labels of the examples

Output

- ExampleSet: the example set with predicted labels

Description: The DecisionTreeApplier is an internal model applier for decision trees of the ID3Learner described in section 5.3.9 or the DecisionTree-Learner in section 5.3.10. With a suitable model (i.e. decision tree) the operator can predict the labels of the given example set.

5.3.12 NeuralNetLearner *extends* Learner (p. 43)

Input

- ExampleSet: the training example set

Output

- Model: the learned model

Parameters

- *model_file:* name of the file the Learner writes the learned model into
- *hidden_layer:* number of neurons in the hidden layer. The default value is 10.
- *lambda:* approximation increment. The default value is 0.05.

Description: The operator learns a simple neural net which consists of an input layer, a hidden layer and an output layer. The output layer appends an additional 1 to the input vector. This operator is included in YALE and can be used directly like the other internal operators. The current version of this operator is limited to linear functions.

5.3.13 NeuralNetApplier
extends **ModelApplier (p. 43)**

Input

- ExampleSet: the examples to which the model is applied to
- Model: the model for predicting the labels of the examples

Output

- ExampleSet: the example set with predicted labels

Description: This operator applies a neural net learned by the NeuralNetLearner and delivers an example set with labeled examples.

5.3.14 MultiClassLearner

Input

- ExampleSet the learning example set

Output

- Model the learned model

Inner Operator(s)

1. The inner operator must take an ExampleSet as input and return a learned Model. Right now only the SVMlightLearner is allowed as inner operator.

Parameters

- *model_file:* name of the file the Learner writes the learned model into

Description: A MultiClassLearner contains a classification learner which only differentiates between two classes. In many problems the data is classified into more than two classes and therefore every occuring class must be delimited against each other class.

The operator iterates over the classes which occur in the given data and the inner learner is used to learn a model which discriminates if the examples belong to the current class or not. Every learning step delivers a model and all models are combined into a MultiModel. This MultiModel can be used by the MultiModelApplier described in section 5.3.15.

5.3.15 MultiModelApplier
extends **ModelApplier (p. 43)**

Input

- ExampleSet: the examples to which the model is applied to
- Model: the model for predicting the labels of the examples

Output

- ExampleSet: the example set with predicted labels

Inner Operator(s)

1. The inner operator must take an ExampleSet and a Model as input and must return an ExampleSet as output. Up to now, only the SVMLightApplier is allowed as inner operator.

Description: A MultiModelApplier contains a classification applier which only differentiates between two classes. For predicting the labels of the given example set all models are iterated, for each example, and the classification with the greatest confidence is chosen.

5.3.16 WekaLearner
extends **Learner (p. 43)**

Input

- ExampleSet: the training example set

Output

- Model: the learned model

Parameters

- *weka_learner_name:* the fully qualified WEKA classname of the WEKA classifier/learner to be used, e.g. `weka.classifiers.Id3`
- ◦ All parameters from the group *weka_parameters* are assumed to be key-value pairs that are passed to the classifier/learner. The leading dash in front of the parameter name must not be part of the key!
- ◦ *model_file:* name of the file the Learner writes the learned model into

Description: This operator can wrap all classifiers from the Weka package. The classifier type can be selected by a parameter. See the Weka javadoc for descriptions (`http://www.cs.waikato.ac.nz/ml/weka/`) and consult section 2.3 for download and parameter settings information.

5.3.17 WekaApplier
extends **ModelApplier (p. 43)**

Input

- ExampleSet: the examples to which the model is applied to
- Model: the model for predicting the labels of the examples

Output

- ExampleSet: the example set with predicted labels

Description: This operator applies a Weka model. See the Weka javadoc (`http://www.cs.waikato.ac.nz/ml/weka/`) and consult section 2.3 for download and parameter settings information.

5.4 Cluster algorithms

In case that the example set has no label attribute, unsupervised learning methods can be applied. Clustering algorithms segment the example space according to a given distance measure.

5.4.1 *Clusterer* (abstract)

Input

- ExampleSet: the example set to be clustered

Output

- Model: the learned cluster model
- ExampleSet: the clustered example set

Parameters

- ◦ *cluster_file:* name of the file the Clusterer writes the learned cluster model into

Description: Clusterer is the abstract superclass of all clusterer operators. Values are assigned to the cluster attributes of the examples.

5.4.2 WekaClusterer
extends **Clusterer** **(p. 54)**

Input

- ExampleSet: the example set to be clustered

Output

- Model: the learned cluster model
- ExampleSet: the clustered example set

Parameters

- *weka_clusterer_name:* fully qualified WEKA classname of the WEKA clusterer to be used, e.g. `weka.clusterers.EM` for the expectation maximization approach
- ◦ All parameters from the group *weka_parameters* are assumed to be key-value pairs that are passed to the classifier. The leading dash in front of the parameter name must not be part of the key!
- ◦ *cluster_file:* name of the file the Clusterer writes the learned model into

Description: This operator can wrap all clusterers from the Weka clusterers package. The clusterer type can be selected by a parameter. See the Weka javadoc for descriptions of the clusteres types that are available (`http://www.cs.waikato.ac.nz/ml/weka/`).

5.4.3 ClusterWrapper
extends **OperatorChain (p. 38)**

Input

- ExampleSet a clustered example set

Output

- none

Description: This operator applies its inner operators once for each cluster.

5.5 Validation and performance evaluation

When applying a model to a real-world problem, one usually wants to rely on a statistically significant estimation of its performance. There are several ways to measure this performance by comparing predicted label and true label. This can of course only be done if the latter is known. The usual way to estimate performance is therefore, to split the labelled dataset into a training set and a test set, which can be used for performance estimation. The operators in this section realise different ways of evaluating the performance of a model and splitting the dataset into training and test set.

5.5.1 PerformanceEvaluator

Input

- ExampleSet: a labelled example set with true labels and predicted labels

Output

- PerformanceVector: a list of performance criteria and their values

Parameters

- *criteria_list:* possible performance criteria are `absolute error`, `scaled error`, `squared error`, `relative error`, `classification_error`, `accuracy`, `precision`, `recall`, and `fallout`.
- ◦ *skip_undefined_labels:* boolean value which indicates whether examples should be skipped if their label or predicted label is undefined

Values

- any of the performance criteria specified by the parameter *criteria_list* (may be specified)

Description: This operator evaluates a learning method by comparing the predicted labels of an example set with its true labels. Errors are counted, summed up, and finally the average of each criterion over all examples is computed. There are several commonly used criteria available as the list above shows. Some or all of them can be selected.

5.5.2 *ValidationChain* (abstract)

Inner Operator(s)

1. Training: The first inner operator expects a training ExampleSet as input. Usually this operator is a learner which returns a Model

2. Test: The second inner operator must be able to handle the output of the first inner operator plus a test ExampleSet and return a PerformanceVector. Usually this operator is a chain that contains a ModelApplier and a PerformanceEvaluator.

Input

- ExampleSet: the data set to be used to evaluate the learning algorithm

Output

- PerformanceVector: list of performance criteria for the evaluation

Values

- *performance:* value of the first performance criterion in the performance vector

Description: There are several ways of validating the performance of learning methods on a given data set. All validation chains in YALE inherit from this operator and have common inner operators, input and output. All of them split the ExampleSet into training and test set(s), train one or more Models and evaluate them. They differ in the way they split up the ExampleSet.

5.5.3 FixedSplitValidationChain
extends **ValidationChain (p. 57)**

Inner Operator(s)

1. Training: The first inner operator expects a training ExampleSet as input. Usually this operator is a learner which returns a Model
2. Test: The second inner operator must be able to handle the output of the first inner operator plus a test ExampleSet and return a PerformanceVector. Usually this operator is a chain that contains a ModelApplier and a PerformanceEvaluator.

Input

- ExampleSet: the data set to be used to evaluate the learning algorithm

Output

- PerformanceVector: list of performance criteria for the evaluation

Parameters

- *training_set_size:* the exact number of examples to be used for learning

Values

- *performance:* value of the first performance criterion in the performance vector

Description: A FixedSplitValidationChain splits up the example set at a fixed point, i.e. after a specific number of examples, into a training and a test set, and evaluates the model. The examples are not shuffled, i.e. their order is not changed.

5.5.4 RandomSplitValidationChain *extends* ValidationChain (p. 57)

Inner Operator(s)

1. Training: The first inner operator expects a training ExampleSet as input. Usually this operator is a learner which returns a Model
2. Test: The second inner operator must be able to handle the output of the first inner operator plus a test ExampleSet and return a PerformanceVector. Usually this operator is a chain that contains a ModelApplier and a PerformanceEvaluator.

Input

- ExampleSet: the data set to be used to evaluate the learning algorithm

Output

- PerformanceVector: list of performance criteria for the evaluation

Parameters

- *split_ratio:* relative size of the training set in comparison to the complete example set, i.e. the size of the fraction of the input example set to the use for training ($\in [0, 1]$, default: 0.7).

Values

- *performance:* value of the first performance criterion in the performance vector

Description: This operator evaluates the performance of an enclosed learning algorithm on a given data set L. This is done by randomly splitting up the example set into a training set (holding $split\_ratio \cdot |L|$ examples) and generating a model. The model is then evaluated using the remaining $(1 - split\_ratio) \cdot |L|$ examples.

5.5.5 XValidation
***extends* ValidationChain (p. 57)**

Inner Operator(s)

1. Training: The first inner operator expects a training ExampleSet as input. Usually this operator is a learner which returns a Model
2. Test: The second inner operator must be able to handle the output of the first inner operator plus a test ExampleSet and return a PerformanceVector. Usually this operator is a chain that contains a ModelApplier and a PerformanceEvaluator.

Input

- ExampleSet: the data set to be used to evaluate the learning algorithm

Output

- PerformanceVector: list of performance criteria for the evaluation

Parameters

- *number_of_validations:* the number of subsets into which the example set should be partioned ($\in [2, \infty]$, default value: 10)

Values

- *performance:* value of the first performance criterion in the performance vector
- *variance:* variance of this performance criterion
- *validation:* the number of the current validation

Description: This operator evaluates the performance of an enclosed learning algorithm on a given data set L. A k-fold cross-validation is done by splitting up the example set into $k = number\_of\_validations$ disjoint subsets L_i. Then k models M_j are generated using $\bigcup_{1 \leq i \leq k, i \neq j} L_i$ as training data and evaluating them on L_i. The average values of the performance criteria and their variances are calculated.

5.5.6 *MethodValidationChain* (abstract)

Input

- ExampleSet: the input example set to be used for the evaluation of a feature selection/generation method

Output

- PerformanceVector:
- AttributeVector:

Inner Operator(s)

1. The first inner operator should be a FeatureOperator (see p. 65) or a similar method which returns a modified ExampleSet.
2. The second inner operator, usually a learner, must be able to handle an ExampleSet.
3. The third inner operator must be able to handle an ExampleSet plus the output of the second inner operator, usually an applier plus PerformanceEvaluator, and return a PerformanceVector.

Values

- *performance:* value of the first performance criterion in the performance vector

Description: Similar to a ValidationChain (see p. 57), this operator evaluates the performance of an algorithm, but while the first evaluates a learning algorithm, the latter evaluates a feature selection method.

The input ExampleSet is split up into training and test set. The first inner operator (the feature selection method) operates on the training set. It may return an example set with modified, deselected or newly generated attributes. The second operator should be a learner which generates a model on another copy of the training set *using the new attributes.* The third operator must evaluate this model by using the remaining test set.

5.5.7 RandomSplitMethodValidationChain *extends* MethodValidationChain (p. 61)

Input

- ExampleSet: the input example set to be used for the evaluation of a feature selection/generation method

Output

- PerformanceVector:
- AttributeVector:

Inner Operator(s)

1. The first inner operator should be a FeatureOperator (see p. 65) or a similar method which returns a modified ExampleSet.
2. The second inner operator, usually a learner, must be able to handle an ExampleSet.
3. The third inner operator must be able to handle an ExampleSet plus the output of the second inner operator, usually an applier plus PerformanceEvaluator, and return a PerformanceVector.

Parameters

- *split_ratio:* relative size of the training set in comparison to the complete example set, i.e. the size of the fraction of the input example set to the use for training ($\in [0, 1]$, default: 0.7).

Values

- *performance:* value of the first performance criterion in the performance vector

Description: Splits the example set according to the description given for the normal RandomSplitValidationChain (see p. 59) and evaluates the inner method as described in section 5.5.6.

5.5.8 MethodXValidation
***extends* MethodValidationChain (p. 61)**

Input

- ExampleSet: the input example set to be used for the evaluation of a feature selection/generation method

Output

- PerformanceVector:
- AttributeVector:

Inner Operator(s)

1. The first inner operator should be a FeatureOperator (see p. 65) or a similar method which returns a modified ExampleSet.
2. The second inner operator, usually a learner, must be able to handle an ExampleSet.
3. The third inner operator must be able to handle an ExampleSet plus the output of the second inner operator, usually an applier plus PerformanceEvaluator, and return a PerformanceVector.

Parameters

- *number_of_validations:* ($\in [0, \infty]$, default: 10)

Values

- *performance:* value of the first performance criterion in the performance vector
- *variance:* variance of this performance criterion
- *validation:* the number of the current validation

Description: Splits the example set according to the description given for the normal XValidation (see p. 60) and evaluates the inner method as described in section 5.5.6.

5.5.9 ParameterOptimization

Inner Operator(s)

1. An arbitrary number of inner operators. The last of them must return a PerformanceVector, whose first value is used for the performance optimization.

Input

- any

Output

- ResultObject: An object that can only be used for logging purposes. A ResultWriter (see p. 5.2.6) will print the optimal parameter set

Parameters

- *operator:* name of an operator chain (default: this ParameterOptimizationOperator itself)

Description: This operator repeatedly executes its inner operators with changing parameter values. If the *operator* holds some inner operators with a total of n parameters, each of which having p_i parameter values. There are $\prod_{i=1}^{n} p_i$ ways of assigning these values to the corresponding parameters. The ParameterOptimizationOperator finds an optimal combination by trying all these combinations, applying its inner operators on the cloned objects, and selecting a combination that optimizes the value of the first performance criterion in the performance vector returned by the inner operator.

5.6 Feature selection and generation

Data preprocessing is an important means for improving the performance of a machine learning algorithm. Transforming the input data into a more suitable form can greatly enhance both efficiency and prediction accuracy. The following operators can generate new attributes from the original ones, e.g. by multiplying two numerical attributes. Even more important, it can find an optimal feature set automatically.

5.6.1 *FeatureOperator* (abstract)

Inner Operator(s)

1. Validation: This operator must evaluate an ExampleSet that it receives as input and return a PerformanceVector. Usually this operator is a ValidationChain (see p. 57).

Input

- ExampleSet: the input example set represented by the original feature set

Output

- ExampleSet: the example set represented by an optimal feature set
- PerformanceVector: the performance of the inner operators on the example set represented by the selected optimal feature set

Parameters

- *remove_unused:* Usually feature selection algorithms will produce an example set with irrelevant attributes switched off. If this parameter is set to true (default), the unused attributes will be entirely removed.

Values

- *generation:* number of the the current generation (in case of multi-generation feature transformation methods like EAs)
- *best:* the performance of the best individual over all generations
- *performance:* the performance of the best individual in the current generation

Description: This operator is the superclass of all feature selection and generation operators. Feature selection and generation algorithms try to find a set of input attributes that is optimal with respect to a given performance criterion, learning method, and dataset.

All feature selection and generation algorithms have one inner operator that evaluates an ExampleSet by creating a PerformanceVector. See section 4 for an example.

5.6.2 BruteForce

extends **FeatureOperator (p. 65)**

Inner Operator(s)

1. Validation: This operator must evaluate an ExampleSet that it receives as input and return a PerformanceVector. Usually this operator is a ValidationChain (see p. 57).

Input

- ExampleSet: the input example set represented by the original feature set

Output

- ExampleSet: the example set represented by an optimal feature set
- PerformanceVector: the performance of the inner operators on the example set represented by the selected optimal feature set

Parameters

- ◦ *remove_unused:* If this parameter is set to true (default), the unused attributes will be entirely removed when the algorithm terminates.

Values

- *generation:* number of the the current generation (in case of multi-generation feature transformation methods like EAs)
- *best:* the performance of the best individual over all generations
- *performance:* the performance of the best individual in the current generation

Description: This operator performs an exhaustive search over all combinations of attributes. Hence, it is very slow, but it is the only feature selection algorithm that is *guaranteed* to find the optimal feature set.

5.6.3 FeatureSelection
***extends* FeatureOperator (p. 65)**

Inner Operator(s)

1. Validation: This operator must evaluate an ExampleSet that it receives as input and return a PerformanceVector. Usually this operator is a ValidationChain (see p. 57).

Input

- ExampleSet: the input example set represented by the original feature set

Output

- ExampleSet: the example set represented by an optimal feature set
- PerformanceVector: the performance of the inner operators on the example set represented by the selected optimal feature set

Parameters

- *selection_direction:* `forward` or `backward`
- *keep_best:* use the n best individuals for initialising the next generation (default=1)
- *generations_without_improval:* terminate after n unsuccessful generations (default=1)
- *remove_unused:* If this parameter is set to true (default), the unused attributes will be entirely removed when the algorithm terminates.

Values

- *generation:* number of the the current generation (in case of multi-generation feature transformation methods like EAs)
- *best:* the performance of the best individual over all generations
- *performance:* the performance of the best individual in the current generation

Description: This operator covers the two well-known feature selection algorithms *forward selection* and *backward elimination*. Let us assume there are n attributes.

Forward selection: The initial generation has n attribute sets, each having one different attribute switched on. As long as the performance increases, a new generation is created by selecting the best individual and making as many copies as there are unused attributes. In each of the copies one additional attribute is switched on.

Backward elimination: The first generation has 1 attribute set having all n attributes switched on. As long as the performance increases, a new generation is created by selecting the best individual and making as many copies as there are attributes used in this individual. In each of this copy one of the used attributes is switched off.

5.6.4 GeneticAlgorithm
extends **FeatureOperator (p. 65)**

Inner Operator(s)

1. Validation: This operator must evaluate an ExampleSet that it receives as input and return a PerformanceVector. Usually this operator is a ValidationChain (see p. 57).

Input

- ExampleSet: the input example set represented by the original feature set

Output

- ExampleSet: the example set represented by an optimal feature set
- PerformanceVector: the performance of the inner operators on the example set represented by the selected optimal feature set

Parameters

- *population_size:* the fixed number of individuals in one generation.
- *maximum_number_of_generations:* performance independent stop criterion (terminate after n generations).
- ◦ *generations_without_improval:* performance dependent stop criterion (terminate after n generations without performance improvement).
- ◦ *keep_best_individual:* set to true for elitist selection (always keep the best individual). The default value is false.
- *p_initialize:* initial probability $\in [0..1]$ for a single feature of a first generation individual to be switched on.
- *p_mutation:* mutation probability $\in [0, 1]$. The default value is 0.05.
- *p_crossover:* probability for a single feature to be selected for crossover $\in [0..1]$.
- ◦ *crossover_type:* can be `one_point` or `uniform`.
- ◦ *remove_unused:* Usually feature selection algorithms produce an example set with irrelevant attributes switched off. If this parameter is set to true (default), the unused attributes will be entirely removed.

Values

- *generation:* number of the the current generation (in case of multi-generation feature transformation methods like EAs)
- *best:* the performance of the best individual over all generations
- *performance:* the performance of the best individual in the current generation

Description: If there are many attributes from which to select an optimal subset the search space quickly grows too large to exhaustively evaluate all attribute sets. A genetic algorithm can be used to represent attribute sets as single individuals and evolve them in a probabilistic approach to create good feature representations for the learning task at hand. In our case one may consider the individuals as a vector of flags that are set to true if the corresponding attribute is used and to false if it is unused.

A genetic algorithm operates in cycles, the so called generations. Each generation contains a fixed number of individuals. The first generation is initialised randomly. As long as no stopping criterion is complied with, certain operations are performed on the current population:

Mutation: With a given probability, the selection flag of a feature is flipped. This is done for all features of all individuals.

Crossover: With a given probability two individuals are selected and their information is combined. This can be done in two ways: In case of the so called "one-point-crossover", both feature vectors are cut at a fixed, randomly chosen split point; then the beginning of the first individual is joined with the end of the second and vice versa. In case of the so called "uniform-crossover", for each of the features the respective selection flags of both individuals are either swapped or not.

Evaluation: All individuals are evaluated by applying the inner operator to them. From the performance vector a fitness is calculated.

Selection: Finally the best n individuals are selected. This is done using the roulette wheel method. Imagine a roulette wheel with n partitions of a size proportional to the fitness of the corresponding individual. The ball is rolled n times selecting those n individuals in whose partition the ball stopped and copying them into the next generation.

5.6.5 GeneratingGeneticAlgorithm *extends* GeneticAlgorithm (p. 69)

Inner Operator(s)

1. Validation: This operator must evaluate an ExampleSet that it receives as input and return a PerformanceVector. Usually this operator is a ValidationChain (see p. 57).

Input

- ExampleSet: the input example set represented by the original feature set

Output

- ExampleSet: the example set represented by an optimal feature set
- PerformanceVector: the performance of the inner operators on the example set represented by the selected optimal feature set

Parameters

- *population_size:* the fixed number of individuals in one generation.
- *maximum_number_of_generations:* performance independent stop criterion (terminate after n generations).
- ◦ *generations_without_improval:* performance dependent stop criterion (terminate after n generations without performance improvement).
- ◦ *keep_best_individual:* set to true for elitist selection (always keep the best individual). The default value is false.
- *p_initialize:* initial probability $\in [0..1]$ for a single feature of a first generation individual to be switched on.
- *p_mutation:* mutation probability $\in [0,1]$. The default value is 0.05.
- *p_crossover:* probability for a single feature to be selected for crossover $\in [0..1]$.
- ◦ *crossover_type:* can be `one_point` or `uniform`.
- ◦ *remove_unused:* Usually feature selection algorithms produce an example set with irrelevant attributes switched off. If this parameter is set to true (default), the unused attributes will be entirely removed.
- *p_generate:* the probability for a single individual to be selected for generating new attributes.

- *max_number_of_new_attributes:* upper bound for the number of newly generated attributes for an individual in one generation.
- *reciprocal_value:* set to true in order to allow the generation of reciprocal values.
- *function_characteristica:* set to true in order to allow generation of function characteristics (local maximum, two turning points). To be applicable, a value series must be given that contains exactly one maximum and two turning points.
- *use_plus:* set to true in order to allow the generation of the sum of two attributes.
- *use_diff:* set to true in order to allow the generation of the difference of two attribute.
- *use_mult:* set to true in order to allow the generation of the product of two attributes.
- *use_div:* set to true in order to allow the generation of the quotient of two attributes.

Description: This operator is an extension of the described GeneticAlgorithm. In addition to the GA, this operator also generates new features in the mutation step. With a given probability for each individual, up to a given number of new features are appended to the feature vectors, e.g. by multiplying two of the numeric features. Hence, the length of the individuals (the length of the used/unused-vectors) can vary.

5.6.6 DirectedGeneratingGeneticAlgorithm *extends* GeneratingGeneticAlgorithm (p. 71)

Inner Operator(s)

1. Validation: This operator must evaluate an ExampleSet that it receives as input and return a PerformanceVector. Usually this operator is a ValidationChain (see p. 57).

Input

- ExampleSet: the input example set represented by the original feature set

Output

- ExampleSet: the example set represented by an optimal feature set
- PerformanceVector: the performance of the inner operators on the example set represented by the selected optimal feature set

Parameters

- *population_size:* the fixed number of individuals in one generation.
- *maximum_number_of_generations:* performance independent stop criterion (terminate after n generations).
- ∘ *generations_without_improval:* performance dependent stop criterion (terminate after n generations without performance improvement).
- ∘ *keep_best_individual:* set to true for elitist selection (always keep the best individual). The default value is false.
- *p_initialize:* initial probability $\in [0..1]$ for a single feature of a first generation individual to be switched on.
- *p_mutation:* mutation probability $\in [0, 1]$. The default value is 0.05.
- *p_crossover:* probability for a single feature to be selected for crossover $\in [0..1]$.
- ∘ *crossover_type:* can be `one_point` or `uniform`.
- ∘ *remove_unused:* Usually feature selection algorithms produce an example set with irrelevant attributes switched off. If this parameter is set to true (default), the unused attributes will be entirely removed.
- *p_generate:* the probability for a single individual to be selected for generating new attributes.

- *max_number_of_new_attributes:* upper bound for the number of newly generated attributes for an individual in one generation.
- *reciprocal_value:* set to true in order to allow the generation of reciprocal values.
- *function_characteristica:* set to true in order to allow generation of function characteristics (local maximum, two turning points). To be applicable, a value series must be given that contains exactly one maximum and two turning points.
- *use_plus:* set to true in order to allow the generation of the sum of two attributes.
- *use_diff:* set to true in order to allow the generation of the difference of two attribute.
- *use_mult:* set to true in order to allow the generation of the product of two attributes.
- *use_div:* set to true in order to allow the generation of the quotient of two attributes.
- *type:* type of problem. The type may be *classification*, *regression* or *auto* (determines type automatically). *Auto* is the default value.
- *gain_ratio:* a boolean parameter which specifies, if the gain ratio criterion should be used. The default value is true.
- *epsilon:* the range of variation of the attribute values which is used for identifying the information gain for regression problems, the default value is 0.1.
- *use_predicted_label:* determines if the true value of the label or a predicted label should be used for regression information gain. The default value is false.
- *lower_mutation_bound:* lower bound for the mutation probability distribution which is generated from the information gain values.
- *upper_mutation_bound:* lower bound for the mutation probability distribution which is generated from the information gain values.
- *lower_generation_bound:* lower bound for the generation probability distribution which is generated from the information gain values.
- *upper_generation_bound:* lower bound for the generation probability distribution which is generated from the information gain values.

Description: By using a generating genetic algorithm, which generates new attributes and does not only select them, it can happen that many irrelevant attributes are generated. In addition, these individuals can be randomly generated and deleted several times.

It is probably a good idea to make the generating genetic algorithm smarter by using an information gain criterion to decide, which attribute should be selected or should be used to generate another one. With the new regression information gain criterion this is also possible for regression problems.

The information gain value is computed for each attribute. Then the more informative attributes will be preferably selected and used for generating new attributes. The underlying assumption is that it is better to generate new attributes from the informative ones.

5.6.7 YAGGA

***extends* GeneticAlgorithm (p. 69)**

Inner Operator(s)

1. Validation: This operator must evaluate an ExampleSet that it receives as input and return a PerformanceVector. Usually this operator is a ValidationChain (see p. 57).

Input

- ExampleSet: the input example set represented by the original feature set

Output

- ExampleSet: the example set represented by an optimal feature set
- PerformanceVector: the performance of the inner operators on the example set represented by the selected optimal feature set

Parameters

- *population_size:* the fixed number of individuals in one generation.
- *maximum_number_of_generations:* performance independent stop criterion (terminate after n generations).
- ∘ *generations_without_improval:* performance dependent stop criterion (terminate after n generations without performance improvement).
- ∘ *keep_best_individual:* set to true for elitist selection (always keep the best individual). The default value is false.
- *p_initialize:* initial probability $\in [0..1]$ for a single feature of a first generation individual to be switched on.
- *p_mutation:* mutation probability $\in [0, 1]$. The default value is 0.05.
- *p_crossover:* probability for a single feature to be selected for crossover $\in [0..1]$.
- ∘ *crossover_type:* can be `one_point` or `uniform`.
- ∘ *remove_unused:* Usually feature selection algorithms produce an example set with irrelevant attributes switched off. If this parameter is set to true (default), the unused attributes will be entirely removed.
- *reciprocal_value:* set to true in order to allow the generation of reciprocal values.

- *function_characteristica:* set to true in order to allow generation of function characteristics (local maximum, two turning points). To be applicable, a value series must be given that contains exactly one maximum and two turning points.
- *use_plus:* set to true in order to allow the generation of the sum of two attributes.
- *use_diff:* set to true in order to allow the generation of the difference of two attribute.
- *use_mult:* set to true in order to allow the generation of the product of two attributes.
- *use_div:* set to true in order to allow the generation of the quotient of two attributes.

Description: YAGGA is an acronym for Yet Another Generating Genetic Algorithm. Its approach to generating new attributes differs from the original one. The (generating) mutation can do one of the following things with different probabilities:

- Probability $p/4$: add a newly generated attribute to the feature vector.
- Probability $p/4$: add a randomly chosen original attribute to the feature vector.
- Probability $p/2$: remove a randomly chosen attribute from the feature vector.

Thus it is guaranteed that the length of the feature vector can both grow and shrink. On average it will keep its original length, unless longer or shorter individuals prove to have a better fitness. The used/unused-vector is no longer used.

5.6.8 FeatureGeneration

Inner Operator(s)

1. Validation: This operator must evaluate an ExampleSet that it receives as input and return a PerformanceVector. Usually this operator is a ValidationChain (see p. 57).

Input

- ExampleSet: the input example set represented by the original feature set

Output

- ExampleSet: the example set represented by an optimal feature set
- PerformanceVector: the performance of the inner operators on the example set represented by the selected optimal feature set

Parameters

- *filename:* generate all attributes listed in this file. You can use a file generated by a AttributeSetWriter (p. 41) for this purpose. See page 26 for details.
- *reciprocal_value:* generate all reciprocal values.
- *function_characteristica:* generate the function characteristics of all value series (maximum and two turning points).

Description: This operator can be used for data preprocessing. It generates a new set of attributes from the original data. It can either generate all possible attributes of a certain type or generate only attributes listed in a file.

5.6.9 MissingValueReplenishment

Parameters

- All parameters in the group `columns` are interpreted as follows: The key gives the number of the column, the value is one of "minimum", "maximum", and "average".

  Example: `<parameter group="columns" key="1" value="average"/>`

Input

- ExampleSet: the original data containing undefined attributes.

Output

- ExampleSet: the example set without undefined attributes.

Description: This operator replaces undefined attributes by the minimum, maximum, or average of the column.

Chapter 6

Extending YALE

Although we think that the described operators satisfy a great amount of usual data mining applications, it is quite simple to write your own operators. YALE delivers the mechanisms to manage your data and the provided operators can easily be nested to create complex experiments, but there are several learning methods not included and many other preprocessing operators are imaginable.

This chapter describes how you can write a YALE operator. Since YALE is entirely written in Java, every operator is written in Java too, and for nesting your operator into others, you have to follow some simple specifications.

6.1 Operator skeleton

As we have mentioned above, your operator has to be be written in Java. At least you should know the basic concepts of this language to understand what we are doing here. All needed information of the YALE classes can be found in the YALE API which should be available where you have downloaded YALE.

Every operator inherits its properties from the abstract class

edu.udo.cs.yale.operator.Operator

If you want to use inner operators, which increases the complexity of your operator, it has to extend

edu.udo.cs.yale.operator.OperatorChain

which itself extends Operator. But first look on the simple operator skeleton diagrammed in figure 6.1. It shows the very basic concept of every operator in YALE. The operator skeleton extends the class `Operator` as described above.[1] The two fields INPUT_CLASSES and OUTPUT_CLASSES contain all classes of `IOObjects` required as input by this operator and those

[1] Therefore the path to the `yale.jar` file, which you can find in the `lib` directory of YALE, is needed to be in the java `CLASSPATH` environment variable.

delivered by this operator. Classes which should act as in- or output of an operator must extend `edu.udo.cs.yale.operator.IOObject` (see section 6.2.2).

```
package my.new.operators;

public class TestOperator extends Operator {

    private static final Class[] INPUT_CLASSES = { };
    private static final Class[] OUTPUT_CLASSES = { };

    public void initApply() {}

    public IOObject[] apply() throws OperatorException {
        return new IOObject[0];
    }

    public Class[] getInputClasses() { return INPUT_CLASSES; }
    public Class[] getOutputClasses() { return OUTPUT_CLASSES; }
}
```

Figure 6.1: Operator skeleton

The method `initApply()` can be used to initialize some variables or to get parameters or everything else which should be done once the operator is created. See section 6.2.1 for how to retrieve the parameters of an operator. The method `apply()` is called by YALE every time the operator should perform its action. Here you have to specify what the operator does on the input objects and which result it should return.

Finally there are methods which return the fields specified above. These two methods and the `apply()`-method are abstract in `Operator` and you have to overwrite them in your own operator.

6.1.1 Writing simple operators

After these technical preliminary remarks we set an example which performs a very elementary task: It should write all examples of an ExampleSet into a file. First we consider that all we need as input for this operator is an example set. Since we will not manipulate it, we deliver the same example set as output. The fields for INPUT_CLASSES and OUPUT_CLASSES will therefore only contain one class: `edu.udo.cs.yale.example.ExampleSet`. If you use these fields, you can directly adopt the in- and output methods of the skeleton. Let us presume that your own operators are in the package my.new.operators. Now take a look at the operator in figure 6.2 and we describe the steps of the methods.

The first line in initApply() sets the name of the file to write the example set to. This method returns the value of the parameter "example_set_file", if it is declared in the operator section of the experiment configuration file. If this parameter is not given, the experiment ends immediately with an error message. For this, you can use the LogService, which we will discuss in section 6.2.4. The contents of this method will be performed once when the operator is initialized. At that moment, the in- and output of the operator will be checked to ensure that this operator can be used at this point of the surrounding operator chain. If ExampleSet is not contained in the output of the former operator, your operator can not work and YALE will terminate at the beginning of the experiment.

Now we know how the operator is initialized and how it is guaranteed, that you can nest it into an operator chain. But right now we have not seen how the operator writes the examples of the input example set into the specified file. This will be done in the apply() method. Later we will talk about the details of the implementation, e.g. how you can iterate over the examples of an example set. At this point we will only examine the general structure of our operator.

Primarily we need the input example set. The first line in the apply method delivers the first example set which is contained in the input of this operator. Then a stream to the specified file is opened and an iterator over the examples is created. With this ExampleReader you can pass through the examples of an example set like the way you do with a List iterator. For each example the values are written into the file and afterwards the stream to the file is closed. Each operator can throw an OperatorException to the calling operator, which would be done if any exception occured during writing the file. The last thing to do is creating a new array of IOObjects which contains no elements, since the input example set is only modified and no additional output was produced. This empty array is returned by the method.

6.2 Useful methods for operator design

We hope you have seen, how easy it is to extend YALE with self-written operators. Even though the needed information is contained in the YALE API documentation, we will discuss shortly the main functions you will use when writing your own operator.

6.2.1 Getting parameters

You know how to write parameters in an experiment configuration file and you have seen in the example of the ExampleSetWriter how we get a pa-

```
package my.new.operators;

import de.yale.example.ExampleSet;
import de.yale.example.ExampleReader;
import de.yale.tools.LogService;
import java.io.File;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;

public class ExampleSetWriter extends Operator {

  private static final Class[] INPUT_CLASSES =
    { ExampleSet.class };
  private static final Class[] OUTPUT_CLASSES =
    { ExampleSet.class };
  private String filename;

  public void initApply() {
    filename = getParameter("example_set_file");
    if (filename == null)
      LogService.logMessage("ExampleSetWriter '" + getName() +
                            "': Specify example set file " +
                            "in config file",LogService.FATAL);
  }
  public IOObject[] apply() throws OperatorException {
    ExampleSet eSet =
      (ExampleSet)getInput(ExampleSet.class, false);
    try {
      PrintWriter out =
        new PrintWriter(new FileWriter(new File(filename)));
      ExampleReader reader = eSet.getExampleReader();
      while (reader.hasNext()) {
        out.println(reader.next());
      }
      out.close();
    } catch (IOException e) {
      throw new OperatorException("Couldn't write to " +
                                  "example set file", e);
    }
    return new IOObject[0];
  }
  public Class[] getInputClasses() { return INPUT_CLASSES; }
  public Class[] getOutputClasses() { return OUTPUT_CLASSES; }
}
```

Figure 6.2: Implementation of an ExampleSetWriter operator

rameter just by calling a method

`getParameter(...)`

which is provided by the superclass Operator. Generally all methods to collect parameters from the experiment configuration file are inherited from Operator and all global settings specified in one of the global settings files (see section 2.3) can be gained by the method

`getProperty(String name)`

from the ParameterService[2]. Table 6.1 shows the methods of the class Operator in detail.

`Method`	**Description**
`getParameter(String key)`	Returns the parameter value of the given key and `null` if it was not set.
`getParameterAsString (String key, String default)`	Returns the parameter value of the given key and the default value if it was not set.
`getParameterAsBoolean (String key, boolean default)`	Returns true if value is "true", "yes", "y", or "on".
`getParameterAsDouble (String key, double lowerBound, double upperBound, double default)`	Returns the value of the parameter key. If the parameter is not set or not a number, the method returns the default value. The value is restricted to the range specified by lowerBound and upperBound.
`getParameterAsInt (String key, int lowerBound, int upperBound, int default)`	Returns the value of the parameter key. If the parameter is not set or not a number, the method returns the default value. The value is restricted to the range specified by lowerBound and upperBound.
`getParameterCategory (String key, String[] categories, int default)`	Returns the index of the parameter key in the given categories. If the parameter is not set or none of the categories, the method returns the default index.
`getParameterGroup(String group)`	Returns the parameters specified in the given group as a `List`.

Table 6.1: Methods for obtaining parameters from `Operator`

[2] `edu.udo.cs.yale.tools.ParameterService`, the described method is static.

6.2.2 Input and output

In our example we have only modified the example set by writing the examples into a file. Other operators can also consume their input and don't deliver it back at the end of the `apply()` method. A learning operator consumes an example set to produce a model and so does a cross validation to produce a performance value of the learning method.

Therefore two ways exist to get the input of an operator: One way is to consume the input by using the operator and the other is to leave the `IOObject` in the input for the next operator. In our example, we leave the ExampleSet in the input because we only modify it like a filter. You can say that the operator will not delete the example set from the input and hence we use a `getInput()` method with a `false` boolean flag. The signature of the method is

```
IOObject getInput(Class class, boolean deleteFromInput)
```

It returns the first IOObject of the desired class which is found in the input of the operator and it deletes the object from the input of the next operator if the boolean value is `true`.

Generally speaking, you use the method with a `true` flag if you want to consume the input and with `false` if you want to leave it as input for the next operator.

6.2.3 Iterating over an ExampleSet

YALE is about data mining and one of the most frequent applications of data mining operators is to iterate over a set of examples. This can be done for preprocessing purposes, for learning, for applying a model to predict labels of examples, and for many other tasks. We have seen the mechanism in our example and we will describe it shortly.

The way you iterate over an example set is very similar to the concept of iterators, e.g. in terms of `Lists`. The methods which are provided have the same signature as the methods of a Java `Iterator`. The first thing you have to do is to create such an iterator, called ExampleReader. The following code snippet will show you how:

```
ExampleReader reader = exampleSet.getExampleReader();
while (reader.hasNext()) {
  Example example = reader.next();
  //...do something with the example...
}
```

Figure 6.3: Creating and using an ExampleReader

Assume `exampleSet` is a set of examples which we get from the input of the operator. First of all, a reader is created which can be used like an iterator, traversing through the examples in a loop. The classes ExampleSet, ExampleReader, and Example are provided within the YALE package `edu.udo.cs.yale.example`. Please check the YALE API documentation to see what you can do with an example.

6.2.4 Log messages

If you write your operator, you should make some logging messages so that users understand what your operator is doing. It is especially reasonable to log error messages as soon as possible. YALE provides some methods to log the messages of an operator. We distinguish between *log messages* and *results*. Of course you can write your results into the normal log file specified in the experiment configuration file. But the intended way to announce results of the experiment is to use a ResultWriter (see section 5.2.6) which writes each currently available result residing in his input. For this purpose two classes exist, a class `LogService` and a class `ResultService`. The latter can be used by invoking the static method

```
logResult(String result)
```

or by simply using a ResultWriter as described above.

LogService[3] provides some useful methods to log all messages which are described in table 6.2. All messages which have fatal log verbosity level (or above) will terminate the program! Possible levels are MINIMUM, TASK, STATUS, OPERATOR, WARNING, INIT, EXCEPTION, ERROR, FATAL, and MAXIMUM.

`Method`	**Description**
`logMessage(String message, int verbosityLevel)`	Writes the message into the log file, if the verbosity level is high enough.
`logException(String message, java.lang.Throwable exception)`	Writes the message and the stacktrace of the exception into the log file.
`logFatalException (String message, java.lang.Throwable exception)`	Writes the message and the stacktrace of the exception into the log file and terminates YALE.

Table 6.2: Methods for logging purposes

[3] LogService can be found at `edu.udo.cs.yale.tools.LogService`

6.3 Building operator chains

Now you can extend YALE by writing operators which perform tasks on a given input and deliver the input or additional output to a surrounding operator. We have discussed the specifications to create the operator in such a way that it can be nested into other operators. But what we have not seen is the possibility to write your own operator chain, i.e. operators which contain inner operators to which input will be given and whose output is used by your operator. What makes an operator to an operator chain is the possibility to contain other inner operators.

The way you create an operator chain is straightforward: First your operator does not extend Operator directly any longer, but OperatorChain instead. Now you can simply use inner operators via the method

```
getOperator(int index)
```

which delivers the inner operator with the given index. You can call the `apply()` method of this operator, YALE takes care of the correct execution.

The second thing you have to do is to declare how many inner operators your operator can cope with. Therefore every operator chain has to overwrite two abstract methods from OperatorChain:

`getMinInnerOps()`
and
`getMaxInnerOps()`

which returns the minimum and maximum number of inner operators. If these numbers are equal, your operator chain must include exactly this number of inner operators or YALE will terminate at the beginning of an experiment.

6.3.1 Additional input

But what if you want to add additional `IOObjects` to the input of an inner operator? A cross-validation operator for example, divides an example set into subsets and adds certain subsets to the input of a learning operator and others to the input of an operator chain which includes a model applier and a performance evaluator. In this case your operator has to consume the original `IOObject` and adds others to the input of the inner operators.

In section 6.2.2 we have seen how an operator can get the input and how it is declared if the `IOObject` is consumed or not. If your operator should add a certain `IOObject` to the input of an inner operator it simply has to call the `apply()` method of the inner operator in a way like

```
apply(getInput().append(new IOObject[] { additionalIO }))
```

The method `getInput()` delivers the container of YALE which provides the input and output objects of the operators[4]. To this container the `append()` method adds the additional `IOObjects` as an array.

You can call this method also, if you want to use the same `IOObject` as input for an inner operator several times, e.g. in a loop, or if you want to add more than one `IOObject` to the input of an inner operator.

6.3.2 Using output

Inner operators can produce output which your surrounding operator must handle. The call of the `apply()` method of an inner operator delivers a container like the one described above. You can get the `IOObjects` out of this container in the same way like getting the `IOObject` of a certain class by the operator. Figure 6.4 shows the methods to append additional input for the inner operator and getting specified output from the result of the `apply()` method. The example set is split before into training and test set.

```
[...]
Learner learner = getOperator(0);
IOContainer container =
  learner.apply(getInput().append(new IOObject[] {trainingSet}));

ModelApplier applier = getOperator(1);
applier.apply(getInput().append(new IOObject[] {testSet}));
[...]
```

Figure 6.4: In- and output of an inner operator

Mostly you do not need to do anything about adding additional input or getting the output and YALE will manage the in- and ouput for your operator. Pay attention to the fact that you do not need to care about the learned model: YALE copes with the learned model for your model applier.

6.4 Adding your operator

At this point you know all the tricks to write your own operator and the tool kit which is provied by YALE for this purpose. The last thing you have to do is to declare your operator to YALE. Every operator must comply with the following terms:

1. The fully classified classname of your operator must be in your java `CLASSPATH` variable.

[4] `edu.udo.cs.yale.operator.IOContainer`

2. A short name for displaying you operator in a graphical user interface must be specified and

3. A long and meaningful name to specify the operator in an experiment configuration file is required.

To link these conditions to one another you have to specify them in a operator description file which is located in the `etc` directory of the Yale home directly(see section 2.2). In the file `etc/operators.xml` all operators of Yale are specified with their fully specified classname and both a short and a meaningful name. Every entry holds for one operator and they are written like the ones in figure 6.5. As you can see, you simply add the entries for your own operators at the end of this file.

```
<operators>

  <!-- YALE Operators -->
  <operator
     name="ExampleSetWriter"
     short="MyESWriter"
     path="de.yale.operator.ExampleSetWriter"/>

  <operator
     name="MySVMLearner"
     short="mySVM"
     path="de.yale.operator.learner.SVMLearner"/>

  <operator
     name="XValidation"
     short="XVal"
     path="de.yale.operator.XValidation"/>

  [...]


  <!-- Your Own Operators -->
  <operator
     name="MyExampleSetWriter"
     short="MyESWriter"
     path="my.new.operators.ExampleSetWriter"/>

  <operator
     name="MyPreprocessing"
     short="MyPreP"
     path="my.new.operators.Preprocessing"/>
</operators>
```

Figure 6.5: Declaring operators to YALE

Chapter 7

Acknowledgements

This work was supported by the Deutsche Forschungsgemeinschaft (DFG[1]), Collaborative Research Center on Computational Intelligence (SFB 531[2]) at University of Dortmund.

We would like to thank Stefan Haustein for providing his marvellous XML parser kxml[3], which we use for parsing almost all configuration files.

We are grateful to Stefan Rüping for providing his implementation of a support vector machine[4]. We thank Thorsten Joachims not only for providing the SVM^{light} [5], but also for giving useful advice. We highly appreciate the operators written by Timm Euler and like to thank him for many good questions and even better answers.

We thank the Weka[6] developers for providing an open source Java archive with lots of machine learning operators.

We thank Martin Scholz for continuously encouraging us to implement a fancy GUI, which is hopefully coming soon.

[1] `http://www.dfg.de/`
[2] `http://sfbci.uni-dortmund.de/`
[3] `http://www.kxml.org/`
[4] `http://www-ai.informatik.uni-dortmund.de/SOFTWARE/MYSVM/`
[5] `http://svmlight.joachims.org/`
[6] `http://www.cs.waikato.ac.nz/ml/weka/`

Bibliography

[1] L. Breiman. Bagging predictors. *Machine Learning*, 13(2):30–37, 1996.

[2] B. Efron and R. Tibshirani. *An introduction to the bootstrap*. Chapman & Hall, New York, USA, 1993.

[3] M. A. Hall. *Correlation-based feature selection for machine learning*. Dissertation, Department of Computer Science, University of Waikato, Hamilton, New Zealand, 1999.

[4] T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*, chapter 11. MIT Press, Cambridge, MA, 1999.

[5] K. Kira and L. Rendell. The feature selection problem: Traditional methods and a new algorithm. In *The Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 129–134. AAAI Press, 1992.

[6] R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial Intelligence Journal, Special Issue on Relevance*, 97(1–2):273–324, 1997.

[7] 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/`.

[8] John Ross Quinlan. *C4.5: Programs for Machine Learning*. Machine Learning. Morgan Kaufmann, San Mateo, CA, USA, 1993.

[9] O. Ritthoff, R. Klinkenberg, S. Fischer, I. Mierswa, and S. Felske. YALE: Yet Another Machine Learning Environment. In R. Klinkenberg, S. Rüping, A. Fick, N. Henze, C. Herzog, R. Molitor, and O. Schröder, editors, *LLWA 01 – Tagungsband der GI-Workshop-Woche* Lernen – Lehren – Wissen – Adaptivität, pages 84–92, Germany, October 2001. Technical Report No. 763, Department of Computer Science, University of Dortmund.

[10] Stefan Rüping. *mySVM Manual.* Universität Dortmund, Lehrstuhl Informatik VIII, 2000.
`http://www-ai.cs.uni-dortmund.de/SOFTWARE/MYSVM/`.

[11] Robert E. Schapire. A brief introduction to boosting. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1401–1406, Stockholm, Sweden, 1999. Morgan Kaufmann.

[12] Vladimir N. Vapnik. *Statistical Learning Theory.* Wiley, Chichester, UK, 1998.

[13] I. H. Witten and E. Frank. *Data mining: Practical machine learning tools and techniques with Java implementations.* Morgan Kaufmann, San Francisco, CA, USA, 2000. `http://www.cs.waikato.ac.nz/ml/weka/`.

Index