The SeCo-framework for rule learning

Technical Report TUD–KE–2010–02

Frederik Janssen, Johannes Fürnkranz
Knowledge Engineering Group, Technische Universität Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Knowledge Engineering

Abstract

This document describes a rule learning framework called *SeCo* (derived from Separate-and-Conquer). This framework uses building blocks to specify each component that is needed to build up a configuration for a rule learner. It is based on a general separate-and-conquer algorithm. The framework allows to configurate any given rule learner that fulfills some properties. These are mainly that it employs a separate-and-conquer strategy and therefor uses a top-down approach. All other building blocks are freely configurable within the framework. To evaluate the configured algorithm, a stand-alone evaluation framework was implemented. It allows to configurate all parameters necessary for the evaluation part.

Contents

Algorithm 1 AbstractSeCo*(Examples)*

```
Theory ← $\emptyset$
Examples ← SortExamplesByClassVal (Examples)
Growing = SplitData (Examples, Splitsize)
Pruning = SplitData (Examples, 1-Splitsize)
while Positive(Growing) $\neq \emptyset$
    Rule = FindBestRule(Growing,Pruning)
    Covered = Cover (Rule,Growing)
    if RuleStoppingCriterion (Theory,Rule,Growing)
        exit while
    Growing = WeightInstances (Covered)
    Theory = Theory $\cup$ Rule
Theory = PostProcess (Theory, Growing, Pruning)
return (Theory)
```

1 Separate-and-conquer rule learning in the SeCo-framework

1.1 Separate-and-conquer rule learning

The *Separate-and-conquer* or *covering* strategy is widely used among many different types of rule learning algorithms. It goes back to the *AQ*-algorithm [15]. The basic idea is to search the data for a single rule that fulfills some quality criteria. This rule is added to a list of rules and all examples that are covered by this rule are removed from the data. This process usually lasts as long as positive examples are left in the data. The result of such a rule learning system is a list of subsequently learnt rules.

To classify an unseen example this list could be used in two different ways that directly influence the learning process:

- Usage as *decision list* and
- usage as unordered list of rules.

When a decision list is learnt most algorithms at first order the data ascending by the frequency of n target classes. Then, rules are learnt for $n-1$ classes and for the most frequent class in the data no rule is induced. Conversely, when learning an unordered list of rules, rules are learnt for all classes in the data. In both cases exist a special rule that is inserted at the end of the list. This rule is called default rule and usually predicts the most frequent class in the data. Now, when a new example is classified the rules are checked whether or not they cover the example. If a decision list is used, the first rule that covers the example (i.e., matches all given attribute values) classifies it. In the latter case all rules that cover the examples are used to get the classification of the example. Conflicts here are solved by some kind of voting mechanism (i.e., simple voting, weighted voting or confidence scores).

To have a notion of positive and negative examples in the multi-class case, most often a class binarization is employed [8]. Therefor, each class is learnt against the union of all other classes. Hence, for the first (least frequent) class it is defined as the positive class and all remaining examples are referred to as the negative class. In the next step all examples of the first class are removed already. Then, the second least frequent class becomes the positive class and all remaining $n-2$ classes are defined as negative. Note that this type of binarization is ordered because all previous classes are removed. In an unordered fashion, no classes would be removed at all. Other mechanism are known to yield better results. Most importantly in [8] it was shown that the *pairwise* approach often is superior to the class binarization. It is currently planned to incorporate this approach in the framework. Right now, the framework only supports the ordered binarization and the usage of a decision list.

1.2 Unifying rule learners in the SeCo-framework

Despite of the great variability of the different separate-and-conquer algorithms, they all fit in a general framework. Algorithms of this kind can be characterized along three dimensions as described in [7]:

Algorithm 2 FindBestRule*(Growing, Pruning)*

```
InitRule = InitializeRule (Growing)
InitVal = EvaluateRule (InitRule)
BestRule = <InitVal, InitRule>
Rules = {BestRule}
while Rules $\neq \emptyset$
    Candidates = SelectCandidates (Rules, Growing)
    Rules = Rules $\setminus$ Candidates
    for Candidate $\in$ Candidates
        Refinements = RefineRule (Candidate, Growing)
        for Refinement $\in$ Refinements
            Evaluation = EvaluateRule (Refinement, Growing)
            unless StoppingCriterion (Refinement, Evaluation, Pruning)
                NewRule = <Evaluation, Refinement>
                Rules = InsertSort (NewRule, Rules)
                if NewRule > BestRule
                    BestRule = NewRule
    Rules = FilterRules (Rules, Growing)
return (BestRule)
```

- **Language Bias:** The hypothesis language of the algorithm, i.e., the type of rules that are used,
- **Search Bias:** The search method that is used to guide through the search space, and
- **Overfitting Avoidance Bias:** The mechanism used to avoid overfitting. This could be a pruning method or that rules are not added to the theory if they do not fulfill certain properties.

In [7] a general algorithm was given which consists of certain building blocks that enable to specify each of the dimensions specified above. In this work some basic building blocks were introduced and assigned to the three dimensions. Basically, most of the algorithms employ a top-down strategy where a rule is initialized as empty and then refined (here: specialized) until it met some criterion. The framework currently only supports this mode. Another approach is to do it the other way round by starting with a maximal specialized rule (usually initialized by taking a random example from the data) and by generalizing it as long as it fulfills a certain quality criterion. This would be the bottom-up approach.

The general framework is implemented in Java 1.6 and is currently under development. A first release is planned to be online soon. In the beginning of the development, the framework was implemented in the *weka*-environment [18]. By now, it is completely stand-alone, but still much of the code is directly adapted from *weka*. For this reason, some of the interfaces provided by *weka* are still present. For example, the core class of the SeCo-framework (AbstractSeCo) still extends the original class Classifier of the *weka* implementation. Due to this, the interfaces for classifying an instance are still available (classifyInstance and distributionForInstance). The method for inducing the classifier is also the same as in *weka*. Therefore, a classifier is built by using the buildClassifier method.

Algorithm 1 gives an overview of the general outer loop of the framework. Here, the data is split into a growing and a pruning set to implement algorithms which feature an additional pruning phase or an optimization procedure (e.g., Ripper [3]). Basically, this procedure calls the function FindBestRule that returns the best rule that can be found given the configuration of the framework and the dataset. It checks whether this rule fulfills some certain quality criteria (cf., the method RuleStoppingCriterion), removes the covered examples, adds the rule to the theory, and loops until all positive examples are covered[1]. When the theory is completed an additional post processing phase is employed. Here, for example, an optimization will take place.

In Algorithm 2 the inner loop for searching for the best rule is depicted. It basically is used to implement all methods necessary to search for a single rule. Here, the three different biases mentioned in Section 1.2 can be implemented by using the methods of this algorithm appropriately:

- **Language Bias:** Is implemented by the method RefineRule.
- **Search Bias:** Can be defined by using a search strategy via the InitializeRule and RefineRule methods, a search algorithm (SelectCandidates and FilterRules), and a search heuristic (EvaluateRule).

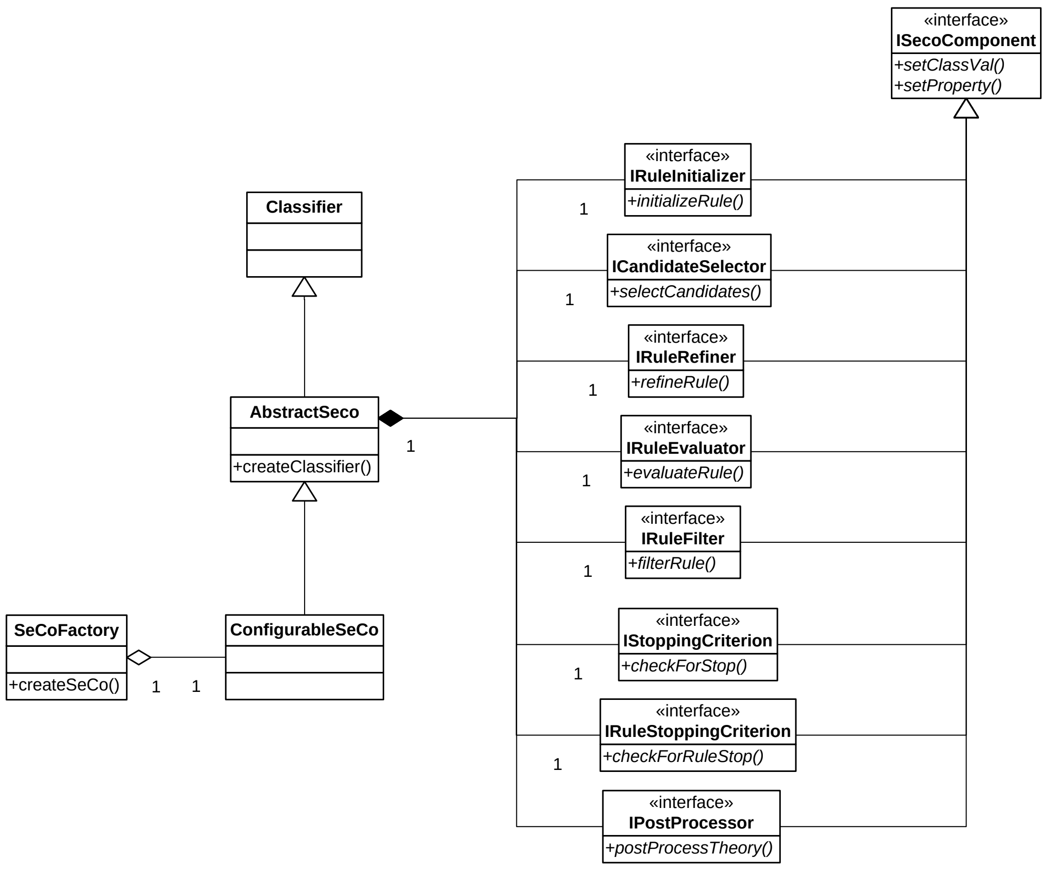
[1] Due to several Stopping Criteria there may be cases where not all positives are covered.

- **Overfitting Avoidance Bias:** Either is implemented by the two stopping criteria (RuleStoppingCriterion in the AbstractSeCo-algorithm or StoppingCriterion in the inner loop) or is defined via the PostProcess-method of the outer loop.

These two algorithms implement a general framework where most of the separate-and-conquer rule learners can be instantiated. In [7] an example for such an instantiation is given. In Section 3.2 some of the most popular algorithms are shown in their respective implementations using the SeCo framework. The framework is an enhanced version of a first implementation done in [16].

In the next Section an overview of the architecture of the framework is given.

Figure 2.1: *UML* diagram of the SeCo-structure



2 Architecture of the SeCo-framework

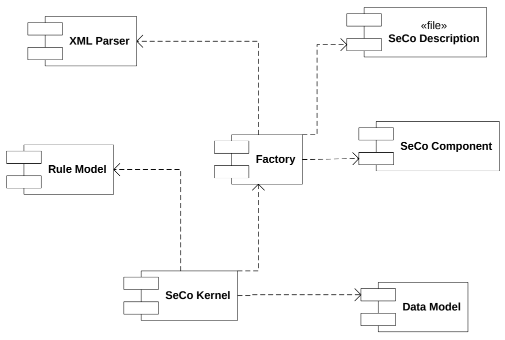
The framework consists of different general building blocks. These building blocks were derived from the pseudo-code given in Algorithms 1 and 2. As first step the framework is built up from the specification of each of these blocks given in a *xml*-file. If a building block is not specified, it will be initialized with the default class for this entity. These default classes implement the most general case of a building block which is best suited for the majority of separate-and-conquer algorithms. Each block is implemented via a Java interface and is described in detail in Section 3.

Usually the only information that is needed to start the framework is the path to a folder that contains dataset files. These files have to be in the .arff-format of *weka* [18]. Other database formats are not supported right now but there are many freely available converters that are able to convert different database formats. The information which datasets should be evaluated is the only mandatory information of the framework. If it is actually the only given information, the framework will initialize a default configuration and evaluate it using a 10-fold cross validation for each of the datasets.

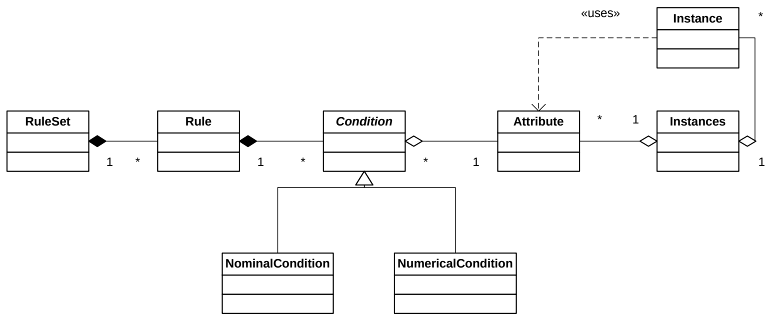
In fact, in most cases a second path is given by the user. This path points to a directory that contains xml-configurations for different algorithms. Both of these two informations are given in a property-file that is given to the *Evaluation Framework* which is a stand-alone component of the SeCo-framework. This evaluation framework currently can be configured in several ways to evaluate algorithms that are built by the SeCo-framework. A detailed description of the framework is given in Section 4.

In Figure 2.1 a diagram in the *Unified Modeling Language (UML)* of the structure of the SeCo-framework is shown. It uses the standard components of a *UML* diagram such as arrows depicting generalization of classes, hollow diamond shapes standing for aggregation, and filled diamond shapes are encoding composition.

Figure 2.2: *UML* diagrams of the SeCo-components and the data model

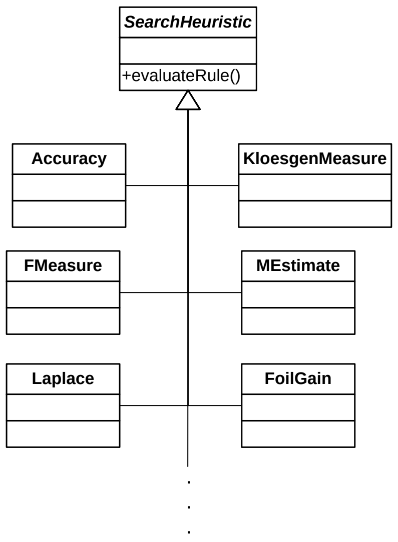


(a) SeCo-components



(b) SeCo data model

Figure 2.3: *UML* diagram of the SeCo heuristics



As mentioned above the framework is based on *weka*. For this reason, there is still an interface to a Classifier class. The SeCoFactory aggregates a ConfigurableSeCo which extends the actual classifier (or rule learner) called AbstractSeCo. This class holds all interfaces that are implemented either by a default setting or by implementations necessary for the current rule learner that should be configured by the SeCo-framework. Each of those components itself is a ISecoComponent and is implemented by the class that is used for this certain interface.

Additionally, the dataset file format of *weka* is still used in the SeCo-framework. Hence, the SeCo-framework also uses the .arff-file format.

The ISecoComponent has some basic methods. It is able to set the class value, i.e., which class in the dataset should be learned and it may set a property necessary for a certain interface. Thus, for example, the IPostProcessor has to know how many optimizations it should employ which is given by a property (in the *xml*-file). Each of the classes implementing one of the interfaces has to implement a certain method. Those could be a check for stopping to add rules to a rule set (checkForRuleStop) or a method that filters out rules (filterRule).

In Figure 2.2 a *UML*-diagram of the components and the data model of the SeCo-framework is given. The SeCo-framework consists of 7 components as depicted in Figure 2.2 (a). First, a description in *xml* is given to the factory which initializes the SeCo components by using the XML Parser. The kernel is the main entity of the framework and has access to the Rule Model as well as to the data model.

The data model depicted in Figure 2.2 (b) consists of everything that is necessary to build a set of rules on a set of instances. Both the Instances and the Instances classes are similar to those implemented in *weka*. The class Attribute handles the attributes of the .arff-file. The class Condition is able to build numeric or nominal condition on the attributes. A rule consists of a finite number of those conditions. The class RuleSet is used to store the rules.

A *UML* description of the heuristics is given in Figure 2.3. These heuristics are used to evaluate candidate rules. A detailed description of the heuristics that are currently implemented in the SeCo-framework is given in Table 3.2. The abstract class SearchHeuristic defines a heuristic by offering a method for evaluating rule (evaluateRule). Each heuristic extends this abstract class. Figure 2.3 only shows a subset of all heuristics (for a complete list see Table 3.2). Note that a parameter of parametrized heuristics can be configured via the setProperty method. If no parameter is given, all parametrized heuristics will be initialized with some optimal setting determined in [12].

The next Section is focused on the components of the framework and each of them is explained in detail.

Table 3.1: Elements and attributes of a SeCo description in *xml*

element	attribute	description
seco		the root element where the configuration is given
secomp		a SeCo component
	interface	An interface to the AbstractSeCo class. Possible values are selector, postprocessor, ruleevaluator, ruleinitializer, rulerefiner, rulestopcriterion, stopcriterion, and weightcriterion
	classname	name of the class that implements the component
	package	*optional* the Java-Package that contains the respective class (*default=seco.learners.components*)
jobject		an arbitrary Java-Object
	classname	the class name
	package	the package that contains the class
	setter	part of a string of the setter-method that aggregates the object with the one on the next level (if the method name is e.g., “setHeuristic”, the setter has to be “heuristic”)
property		sets the property of an object
	name	name of the property
	value	value of the property

First, all functions and interfaces of the AbstractSeCo procedure are shown. All of those are configurable via a *xml* configuration file (cf. Section 3.2). This file is used to specify the actual algorithm that should be implemented in the SeCo-framework. If no configuration is given the framework will use a default algorithm that is rather simple but nevertheless performs comparable to most of the more sophisticated algorithms (cf. [12]). In the following a brief description of the purpose of all functions is given.

SortExamplesByClassVal: This method is used for employing the class binarization method. It could either be an ordered class binarization (where the examples are sorted ascending by their class frequency and each one is tested vs. the rest) or a pairwise binarization (where a model is learnt for each pair of classes).

SplitData: This method is used to split the data into a *growing set* and a *pruning set*. Some algorithms use two different sets: One to build a rule (usually about 2/3 of the dataset) and one for pruning the rule (1/3 of the dataset). When it is used the size of the growing set could be specified via the *xml*-configuration file. If no split of the data is not used this size can be simply set to 100 % (which is also the default option).

FindBestRule: The FindBestRule-procedure is used to return the best rule that can be found given the configuration of the algorithm and the dataset.

RuleStoppingCriterion: This method is implemented by the RuleStopCriterion-interface in the SeCo-framework. Ways to use it can be for example to use a *MDL* based pruning. Here, a rule is only added to the theory if the minimum description length of this rule is beneficial compared to the length needed to encode the examples covered by the rule. An example where this kind of theory pruning is used is the Ripper algorithm [3].

WeightInstances: This method is implemented by the WeightModel-interface. It is used for performing an update of the example weights. In the case where regular covering is used the weights of the examples will be set to 0, which practically removes them (all other examples will have a weight of 1 and all weights are in {0,1}). In the case of *Weighted Covering* (e.g., [6], [4], [17], [14]) the weight of the example will only be reduced by a certain margin and during the evaluation of a rule these weights influence the quality of the rule depending on the weights of the examples that are covered by the rule.

PostProcess: Implemented by the interface PostProcessor. Here, all additional phases that are employed after learning a theory are done. Examples include the optimization phase of Ripper [3] where each rule is inspected in more detail. Therefore, it is learned entirely new (yielding the *Replacement*), learned further on (resulting in the *Revision*), and left untouched. Then, a MDL-based mechanism is used to decide which of the three alternatives is used for the current rule.

In the following the methods of the TopDownBeamSearch procedure are inspected in more detail. As before, all functions are described rather briefly.

InitializeRule: Implemented by the interface RuleInitializer. This mainly is used for deciding which strategy is used. Hence, for a top-down strategy the rule is initialized as a rule with empty conditions (i.e., the body TRUE), thus covering all examples. If a bottom-up strategy is used the rule would be initialized from a random example. This rule would be encoding the example thus simply using conditions for each of the attribute with the respected value of this attribute of the example.

EvaluateRule: The interface RuleEvaluator implements the functionality of evaluating a rule's quality. It basically encodes the heuristic that is used to guide the search. Often, the performance of an algorithm depends strongly on the used heuristic. Therefore this building block is inspected in more detail in Section 3.1.

SelectCandidates: The CandidateSelector is responsible for deciding which Candidates are inspected during the search. Usually this is the part of the algorithm where duplicates are removed or where it is decided in which order the rules are refined. This is very important when more extensive types of search algorithms as an exhaustive search are used.

RefineRule: This method is implemented by the RuleRefiner interface. Here, the rule is refined by a refinement operator. Depending on the search strategy this refinement operator will be different. For top-down it will implement a specialization of a rule by adding a condition to it and for bottom-up it will generalize a rule by removing a condition.

StoppingCriterion: This method, implemented by the StopCriterion interface, is mostly used for pre-pruning issues. Thus, a rule could be learnt on the given growing set and right after it is finished it could be pruned on the pruning set.

FilterRules: The RuleFilter decides how many rules are refined at each cycle. Thus, in a hill-climbing scenario it will only return the best rule. If a beam search is employed it will return the b best rules (where b determines the size of the beam). When an exhaustive search is used it will return all rules.

An algorithm is specified via a *xml* configuration file. For this, each component has to be specified and all attributes have to be given in the *xml* configuration file. Such a configuration of an algorithm contains four different elements or attributes. Table 3.1 gives an overview of all possible elements and attributes of such a configuration file. In Section 3.2 some sample configurations are given. In the following, the specific properties of all elements are summarized.

A property for the RuleRefiner for example could be which kind of test is used inside a rule. Thus, here it is specified whether a condition of a rule should test an (nominal) attribute value on equality or inequality or both. Some properties are not connected to a certain interface but are directly for the AbstractSeCo class. They are encoded as special properties that are given at the end of the configuration file. In the following all possible properties are described. Note that some of the described properties are not implemented yet and are only listed for completeness.

In the following the functions of the two algorithms given in Algorithm 1 and 2 are inspected in detail. Therefore, the name of the function is used as identifier and it is shown which interface in the framework is connected to this component. Additionally all implementors of these interfaces are listed and their properties are shown.

TODO: *adapt the list below, elements that may change:*
- *object of TopDownRefiner (i.e., should be done in the code directly)*
- *include missing objects??? (currently they are in, but not all of them, i.e., see below)*
- *ruleinitializer for bottom up*
- *bottomuprefiner*

SortClasses *property* mode: orderedbinarization, pairwise (Note: Pairwise is not implemented yet)

SplitData *property* growingSetSize: number determining the size of the growing and pruning set (a value of 3, for example, means 2/3 for growing and 1/3 for pruning); Note: This is a general property given to the AbstractSeCo-class, see below

RuleStoppingCriterion *interface* IRuleStoppingCriterion, name: rulestopcriterion, classes implementing it:

- DefaultRuleStop: a rule may pass if it is better than the default rule and if it covers more positive than negative examples
- DefaultRuleStopWeighted: a rule may pass if it covers more than *positveHasToCover* weighted examples, *property* positveHasToCover: int (number of weighted examples a rule has to cover), *default=1*
- NoRuleStop: just returns *false*
- AqrRuleStop: like DefaultRuleStop but adapted to the metric used by AQR
- AqrRuleStopWeighted: like AqrRuleStop but with the addition that a rule has to cover *positveHasToCover* weighted examples, *property* positveHasToCover: int (number of weighted examples a rule has to cover), *default=1*
- CoverageRuleStop: a rule has to cover more positive than negative examples
- MDLStoppingCriterion: the minimum description length (MDL) of the theory with the new rule has to be lower than the MDL of the examples

WeightInstances *interface* IWeightModel, name: weightcriterion, classes implementing it:

- DefaultWeight: no weighting is done
- WeightNegativeToThePower: $weight(i) = 1 + e(i)^3$
- WeightPerIteration: if the example is covered, the new weight will be $example \cdot MultiValue$, *property* MultiValue: double, *default=0.5*

PostProcess *interface* IPostProcessor, name: postprocessor, classes implementing it:

- DefaultPostProcessor: Implementation of the Optimization Phase of Ripper, *property* optimizations: integer (number of optimizations), *default=2*

InitializeRule *interface* IRuleInitializer, name: ruleinitializer, classes implementing it:

- DefaultInitializer: initializes a rule as empty rule that covers all examples and predicts the majority class
- BottomUpInitializer (Note: bottomup is not implemented yet): initializes a rule with a given example, *properties* seedChoice: first, last, random (how to choose the example), *default=random*

EvaluateRule *interface* IRuleEvaluator, name: ruleevaluator, classes implementing it:

- DefaultRuleEvaluator, *object:* heuristic, *property* parameter1: double (parameter 1 of a parametrized heuristic), parameter2: double (parameter 2 of a parametrized heuristic)

SelectCandidates *interface* ICandidateSelector, name: selector, classes implementing it:

- DefaultSelector: selects all candidates

RefineRule *interface* IRuleRefiner, name: rulerefiner, classes implementing it:

- TopDownRefiner: a rule refiner used for most cases, *object:* heuristic (used for optimization purposes)
- AQRRefiner: the refiner of AQR, *properties* negativeSelection: integer (how to choose the negative examples, i.e., 1=random, 0=by their distance), *default=0*; seedChoice: first, last, random (how to choose the seed example), *default=random*
- BottomUpRefiner: refines a rule by deleting conditions

Properties for all classes: *property* nominal.cmpmode: int (which test should be used in a rule, i.e., 1=unequal ($\neq$), 2=equal ($=$), 3=both), *default=2*

StoppingCriterion *interface* IStoppingCriterion, name: stopcriterion, classes implementing it:

- DefaultStop: just returns *false*
- NoNegativesCovered: refines a rule as long as negative examples are covered

Table 3.2: Overview of all heuristics of the SeCo-framework

heuristic	fomula	type
MaxPosCover	p	Value
MinNegCover	$-n$	Value
Precision	$\frac{p}{p+n}$	Value
Laplace	$\frac{p+1}{p+n+2}$	Value
Accuracy	$p-n$	Value
Weighted Relative Accuracy	$\frac{p}{P}-\frac{n}{N}$	Value
Correlation	$\frac{pN-nP}{\sqrt{P\cdot N\cdot(p+n)\cdot(P-p+N-n)}}$	Value
Odds Ratio	$\frac{p\cdot(N-n)}{n\cdot(P-p)}$	Value
F-Measure	$\frac{(\beta^2+1)\cdot\frac{p}{p+n}\cdot\frac{p}{P}}{\beta^2\cdot\frac{p}{p+n}+\frac{p}{P}}$	Value
Kloesgen-Measure	$\left(\frac{p+n}{P+N}\right)^{\omega}\cdot\left(\frac{p}{p+n}\right)$	Value
m-estimate	$\frac{p+m\cdot\frac{P}{P+N}}{p+n+m}$	Value
generalized m-estimate	$\frac{p+m\cdot c}{p+n+m}$	Value
Linear Cost	$c\cdot p-(1-c)\cdot n$	Value
Relative Linear Cost	$c_r\cdot\frac{p}{P}-(1-c_r)\cdot\frac{n}{N}$	Value
Linear Regression	lin. combination	Value
Foil Gain	$p\cdot(log_2(\frac{p}{p+n}))-log_2(\frac{p'}{p'+n'})$	Gain

- LikelihoodRatio: implementing the likelihood ratio statistics to check whether a rule is significant, *property* threshold: {0.7, 0.75, 0.8, 0.85, 0.9, 0.95, .0.975, 0.99, 0.995}, *default=0.9*

FilterRules *interface* IStoppingCriterion, name: stopcriterion, classes implementing it:

- MultiRuleFilter: used to combine multiple filters
- ChiSquareFilter: performs a χ^2-test for significance of rules, *property* threshold: {0.7, 0.75, 0.8, 0.85, 0.9, 0.95, .0.975, 0.99, 0.995}, *default=0.9*
- BeamWidthFilter: used to do a beam search, *property* beamwidth: int (the size of the beam), *default=1*

General Properties these are properties that are interpreted by the AbstractSeCo-class directly

- ruleorder: unordered, ordered (determines what type of rule list should be used), *default=ordered*
- skipdefaultclass: true, false (determines whether a default class should be learned), *default=false*
- weighted: true, false (determines if weighted covering should be used), *default=false*
- growingSetSize: integer (used for setting the growing and pruning set size), *default=1*
- minNo: integer (used for guaranteeing a minimum coverage of examples for each rule), *default=1*

3.1 Configuration of the search heuristic

There are many different heuristics given in the literature (for an overview see [9]). One of the main differences between them is whether they evaluate a rule's quality on an absolute scale or on a relative one. The second type of heuristics measures the quality of a rule related to its predecessor whereas the first type of heuristics evaluate the performance independently of any previously learnt rule. Let us define the first type of heuristics as *Value Heuristics* and the second type as *Gain Heuristics*. Some algorithms use Value Heuristics (e.g., CN2 [2]), others use Gain Heuristics (as Ripper does). The SeCo-framework is able to deal with both types of heuristics.

Table 3.2 gives an overview of all heuristics that are currently implemented in the framework. The formula for each of them is given indeed, but the heuristics themselves are not explained in greater detail. The positive examples covered by a rule are denoted by p, the negative examples by n, the total number of positives in the dataset by P, and the total number of negative examples by N.

Figure 3.1: *xml* configuration of CN2

```
<seco>

  <secomp interface="ruleevaluator" classname="DefaultRuleEvaluator">
    <jobject package="seco.heuristics" classname="Laplace" setter="heuristic"/>
  </secomp>

  <secomp interface="rulerefiner" classname="TopDownRefiner" package="seco.learners.topdown">
  </secomp>

  <secomp interface="selector" classname="DefaultSelector">
  </secomp>

  <secomp interface="rulestopcriterion" classname="NoNegativesCoveredStop">
  </secomp>

  <secomp interface="stopcriterion" classname="LikelihoodRatio">
    <property name="threshold" value="0.9"/>
  </secomp>

  <secomp interface="rulefilter" classname="BeamWidthFilter">
    <property name="beamwidth" value="3"/>
  </secomp>

</seco>
```

3.2 Example configurations and additional options

In the following some well-known algorithms are shown as their respective implementations inside the SeCo-framework. Later on, some additional options of the framework are described. These include a *State Reporter* that informs the user about the current status of the framework and an evaluation framework that was implemented during the design of the SeCo-framework that is able to evaluate rule learning algorithms. This framework is described in detail in Section 4.

3.2.1 CN2

The famous CN2-algorithm[1] [1] uses a top-down approach with a beam size of 3. The conditions of a rule test for equality of a certain attribute value. The heuristic used to select the best refinement is *Laplace* (cf. Table 3.2). CN2 employs a test on significance for each rule. To check whether a rule is significant or not the likelihood ratio statistic is used. Figure 3.1 shows the corresponding configuration file.

As can be seen in Figure 3.1 the heuristic of the DefaultRuleEvaluator is set to *Laplace*. As RuleRefiner the TopDownRefiner is used. All rules are selected by the SelectCandidates method and as RuleStoppingCriterion the CoverageRuleStop is used. This means that a rule is only added if it covers more positive than negative examples. As StoppingCriterion the LikelihoodRatio is used and the threshold is set to 0.9. For filtering rules the BeamWidthFilter with a size of 3 is employed.

3.2.2 AQR

In Figure 3.2 a sample configuration of the AQ algorithm is given. Most importantly the RuleRefiner is different to the implementation of CN2. AQ refines rules by taking a positive seed example and converting it to a rule. Now, in contrary to other algorithms, the set of possible refinements contains only those of that rule. Then, a new rule is initialized as empty rule. The algorithm now chooses a negative example and tries to exclude it from the current rule by specializing the rule with the conditions generated by the positive seed example. This process runs as long as negative examples are covered by the rule. This kind of refinement procedure is implemented in the AQRefinerTopDown. In the default SeCo configuration of AQ no negative example has to be covered and the first positive example in the dataset is chosen as seed example. The beam size is set to 3, the heuristic is *Accuracy*, and a rule is only added if is better than the default rule.

[1] Note that we implemented the improved version of Clark and Boswell.

Figure 3.2: *xml* configuration of AQ

```
<seco>

  <secomp interface="ruleevaluator" classname="DefaultRuleEvaluator">
    <jobject package="seco.heuristics" classname="Accuracy" setter="heuristic"/>
  </secomp>

  <secomp interface="rulerefiner" classname="AqrRefinerTopDown" package="seco.learners.aqr">
    <property name="negativeSelection" value="0.0"/>
    <property name="seedChoice" value="first"/>
  </secomp>

  <secomp interface="selector" classname="DefaultSelector">
  </secomp>

  <secomp interface="stopcriterion" classname="NoNegativesCoveredStop">
  </secomp>

  <secomp interface="rulestopcriterion" classname="AqrRuleStop">
  </secomp>

  <secomp interface="rulefilter" classname="BeamWidthFilter">
    <property name="beamwidth" value="3"/>
  </secomp>

  <property name="skipdefaultclass" value="true"/>

</seco>
```

TODO: *What's with the skipdefaultclass thing?*

3.2.3 Ripper

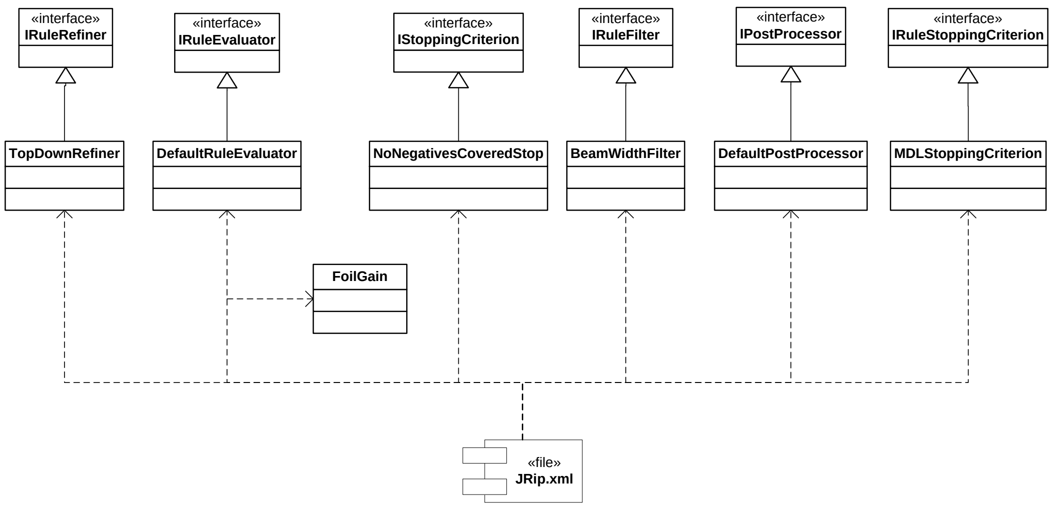
In Figure 3.4 the configuration of the Ripper algorithm is shown[2]. Here, the heuristic *FoilGain* is used to select the best refinement. A rule is refined until it does not cover any more negatives. The main difference to the other two algorithms is that in Ripper a split of the examples into a growing and a pruning set is done and that a post processing is employed. The size of the split is given by the *numFolds* property. Usually 2/3 of the examples are used to build the growing set and 1/3 of the examples are taken for the pruning set. This is done to learn a rule on the growing set and prune it directly afterward on the pruning set. This strategy is known as Incremental Reduced Error Pruning (IREP) [10]. But the Stopping Criterion for adding a rule to the rule set is different here. Hence, a MDL-based stop is performed. It basically states that the minimum description length (MDL) of the rule set containing the new rule has to be lower than the description length computed by simply encoding the examples themselves.

Ripper also works in a top-down fashion. Therefore, the a new rule is initialized as one without any condition. Despite the two other algorithms, this one uses a simple hill-climbing search (referred to as a beam search with size 1). Whenever the learning of a rule set is finished, Ripper applies a optimization phase as post-processing step. This optimization works incrementally by learning each rule completely new, by adding more conditions to it, and by leaving it untouched. From the resulting 3 rules the best one is used to replace the original rule in the rule set. Ripper applies 2 optimizations of the rule set. Thus, the PostProcessor also uses 2 optimizations.

In Figure 3.3 an *UML* diagram of the Ripper algorithm as implementation in the SeCo-framework is shown. Each of the interfaces is implemented by the classes specified in the *xml*-file given in Figure 3.4. Only the TopDownRefiner and the RuleEvaluator have access to the heuristic which is a class on its own. All other components are initialized straight-forward as specified in the *xml* file.

[2] Note that we implemented the *weka* version of Ripper named JRip.

Figure 3.3: *UML* diagram of the implementation of Ripper in the SeCo-framework



TODO: *Change xml to a version where the refiner has no access to the heuristic, this will be implemented soon (similar as depicted in the uml figure)*

3.2.4 SeCo

The SeCo-framework was used in previous work to instantiate a simple algorithm that was required to optimize some of the parametrized heuristics [12]. The results of the work reported in [12] were twofold: On the one hand some default parameters for 5 of the parametrized heuristics shown in Table 3.2 were determined[3] and on the other hand a heuristic called Linear Regression was derived by using a meta-learning setting.

As the configuration given in Figure 3.5 unfolds, the algorithm is similar to CN2 with three exceptions. At first, the heuristic is different. The given configuration was used with 5 of the parametrized heuristics and with some of the simpler ones that were used to compare the influence of the heuristics. Secondly, there is no significance test here, a rule is added if it covers more positive than negative examples. At last, the beam size is reduced to 1 yielding standard hill-climbing search.

In spite of its simplicity, the algorithm nevertheless yields acceptable results, even not statistically significant worse than those of [3].

[3] These 5 heuristics were the m-estimate, the Kloesgen-Measure, the F-Measure, the Linear Cost and the Relative Linear Cost.

Figure 3.4: *xml* configuration of RIPPER

```
<seco>

  <secomp interface="ruleevaluator" classname="DefaultRuleEvaluator">
    <jobject package="seco.heuristics" classname="FoilGain" setter="heuristic"/>
  </secomp>

  <secomp interface="rulerefiner" classname="TopDownRefiner" package="seco.learners.topdown">
    <jobject package="seco.heuristics" classname="FoilGain" setter="heuristic"/>
  </secomp>

  <secomp interface="stopcriterion" classname="NoNegativesCoveredStop">
  </secomp>

  <secomp interface="rulestopcriterion" classname="MDLStoppingCriterion">
  </secomp>

  <secomp interface="ruleinitializer" classname="DefaultRuleInitializer">
  </secomp>

  <secomp interface="rulefilter" classname="BeamWidthFilter">
    <property name="beamwidth" value="1"/>
  </secomp>

  <secomp interface="postprocessor" classname="DefaultPostProcessor">
    <property name="optimizations" value="2"/>
  </secomp>

  <property name="growingSetSize" value="3"/>

  <property name="minNo" value="2"/>

</seco>
```

Figure 3.5: *xml* configuration of SECO

```
<seco>

  <secomp interface="ruleevaluator" classname="DefaultRuleEvaluator">
    <jobject package="seco.heuristics" classname="MEstimate" setter="heuristic"/>
  </secomp>

  <secomp interface="rulerefiner" classname="TopDownRefiner" package="seco.learners.topdown">
  </secomp>

  <secomp interface="selector" classname="DefaultSelector">
  </secomp>

  <secomp interface="stopcriterium" classname="NoNegativesCoveredStop">
  </secomp>

  <secomp interface="rulestopcriterion" classname="CoverageRuleStop">
  </secomp>

  <secomp interface="rulefilter" classname="BeamWidthFilter">
    <property name="beamwidth" value="1"/>
  </secomp>

</seco>
```

4 Evaluation package

In this Section the *Evaluation framework* is described in detail. It is used to evaluate different configurations of the SeCo-framework. Therefor, a folder containing the datasets (called "dataFolder" in a *xml* file) and a folder containing the configurations (named "classifierFolder") should be given. Only the dataset folder is mandatory, if no configuration is given the SeCo-framework will initialize an algorithm with default components. This configuration will be the SeCo algorithm as discussed in Section 3.2.4. It is also possible to evaluate algorithms on one particular dataset. To do so, the file can be specified in the data folder instead of using a whole folder.

Additionally to the two folders several properties can be described in a property-file which is given to an instance of the *Evaluation framework*. In Figure 4.1 a sample configuration of such a property-file is displayed. The most basic options for the evaluation are the parameters for the cross-validation. Here, the number of folds and passes can be specified. For the partitioning into different folds the initial seed can also be set by the user. This mainly is for guaranteeing that one configuration yields the same results when it is initialized with the same seed for the random number generator. Additionally, it could be specified if a stratified cross-validation should be done or not. If a stratified one is used, it is guaranteed that the a priori distribution of the dataset carries over to each fold. The values for these parameters depicted in Figure 4.1 are also the default values when no parameters are given.

In the configuration file some direct properties of the evaluation are also handled. These include the logging of the evaluation and are used to determine whether the state of the evaluation should be reported or not. If the state should be reported a windows called StateReporter opens and informs the user about the current state of the evaluation. If the parameter "interact" is true the user gets information about possible errors that may happen during the evaluation. These could be that a configuration is not capable to learn the current dataset due to some restrictions, i.e., that the algorithm is not able to handle numeric attributes, the data folder contains datasets that are not in the .arff-format or even completely different file types, and so forth. The most important parameter is the mode. There are three different modes the evaluation can handle (the values of those parameters are shown below also):

cv (default): The evaluation uses a cross-validation to evaluate the given configuration on the given set of databases.

traintest: The evaluation builds models on the training databases specified in the first data folder and uses these models to evaluate the algorithms on the test databases given in the second data folder.

existing: The evaluation loads existing models from the classifier folder and uses them to classify the instances given in the data folder.[1]

The last parameter that has to be specified in the *xml* file of the evaluation is the output mode. Here, the user can define if the output of the classifiers (i.e., the rule set, the number of rules/conditions, the macro/micro accuracy and so on) should be displayed in the java console or if it is desired to output these statistics into a file. This could either be a standard (unformatted) text file (similar to the console output) or a comma-separated file (csv) to import it into programs that are able to interpret those file formats (e.g., Open Office).

The evaluation also offers a summary at the end of a complete run. Here, the macro- and micro-average accuracy of all configuration on all datasets are shown. Additionally a ranking is computed by averaging the ranks that the configurations achieved on each dataset (ranked by their macro-average accuracy). If two configuration achieved the same macro-averaged accuracy they divide the virtual ranks among each other[2]. Mostly this is used to compute statistical measures, e.g., to perform a significance test for multiple comparisons as suggested in [5]. In this work at first a Friedman-Test ensures that the algorithms differ significantly in their performance. If so, a post-hoc Nemenyi-Test is used to determine a so-called critical distance. The results of this kind of comparison is a figure where the average ranks of each algorithms are plotted.n-Test ensures that the algorithms differ significantly in their performance. If so, a post-hoc Nemenyi-Test is used to determine a so-called critical distance. The results of this kind of comparison is a figure where the average ranks of each algorithms are plotted.

[1] Note that the models have to be created on the same .arff structure as used in the datasets, e.g., the attributes have to be the same.

[2] If there are 5 algorithms $a_1, \ldots, a_5$ for example and they are ranked as followed: 1. a_1, 2. a_2, 3. and 4. and 5. a_3, a_4, and a_5, the ranks of $a_3, \ldots, a_5$ would be $\frac{3+4+5}{3} = 4$.

Figure 4.1: property-file of an evaluation configuration

```
<validationProperty
  interact="false"
  reportState="true"
  log="true"
  logFile="d:\logs\evaluation_log.txt"
  mode="cv">

  <runtimeProperty name="seed" value="1">
  </runtimeProperty>

  <runtimeProperty name="pass" value="1">
  </runtimeProperty>

  <runtimeProperty name="folds" value="10">
  </runtimeProperty>

  <runtimeProperty name="stratified" value="true">
  </runtimeProperty>

  <output format="csv" path="d:\results\results.csv">
  </output>

  <output format="screen"
  </output>

  <dataFolder>
    d:\data\arff\test-databases
  </dataFolder>

  <classifiersFolder>
    d:\classifiers
  <classifiersFolder>

</validationProperty>
```

5 Conclusions and further work

In this paper the rule learning framework SeCo was introduced. This framework is based on the separate-and-conquer strategy. It was shown that some well-known separate-and-conquer rule learning algorithms could be instantiated in the SeCo-framework by simply specifying some general building blocks via a *xml*-file. These building blocks are the main components of the framework. Thus, each rule learning algorithm that can be fit in these general blocks can be implemented in the framework. All of these building blocks were described in detail. Additionally, the structure of the framework was visualized by *UML*-diagrams. These show that the framework can easily be extended, just by adding new classes that implement an interface of the building blocks. For example, if a new stopping criterion is desired one can just add a class with a different functionality of the stopping condition.

The framework comes with an own tool for evaluation purposes. Currently, not all desired functionalities are available in this framework. But we plan to include most of them before the SeCo-framework is released.

5.1 Ongoing work within the evaluation package

Currently, the evaluation package is mainly used to evaluate rule learning algorithms. In the future we plan to enrich this package so that it becomes our main means for evaluation purposes in all areas covered by our group. These mainly are multilabel classification, label/object ranking, ordinal classification, hierarchical classification, and preference learning. Some of those can also be combined, e.g., one can perform a label ranking for instances that have multiple classes assigned (multilabel). The key step here is to define proper interfaces so that an extension of the framework could be easily done without changing the whole structure of the framework. Then, all necessary features cam be implemented one by one.

Another planned key feature of the evaluation framework is a statistical package. Here, all state-of-the-art methods to guarantee statistical significance will be implemented. These include rather simple ones like a paired t-test or a sign test but are not limited to those. Hence, more sophisticated methods like those proposed in [5] are also planned to be included. The aim is to compare different classifiers on multiple datasets and to output figures where the average rank of each classifier is plotted together with a bar that denotes the critical distance. Then, one can easily see which classifiers are statistically significant better than others.

The last feature would be a wrapper to the *weka*-environment. Thus, each classifier implemented in *weka* can also be evaluated using the evaluation framework. Due to some direct adaption from *weka*, some interfaces are still used in the SeCo-framework (cf. Section 1.2). Therefore the implementation of a *weka*-wrapper should be rather simple.

5.2 Further work within the SeCo-framework

Certainly, the framework currently does not cover all separate-and-conquer algorithms. For example, a bottom-up search is not supported right now. But, due to the modular design of the software, this kind of search should be realizable in a simple way. Actually, the RuleInitializer interface has to be modified by using a rule that is build from a given example[1] and the RuleRefiner interface has to be modified to delete a condition as refinement step (which actually generalizes the rule). The StoppingCriterion has not to be changed and all other interfaces can also be used to build a configuration that searches the search space in a bottom-up fashion.

In previous work, we experimented with an efficient exhaustive search and some of the heuristics listed in Table 3.2. In this work it was found that some heuristics tend to yield much better performance when used in an exhaustive search scenario[11]. On the other hand, mostly the heuristics that were tuned in [12], did not perform very well when used with exhaustive search. Nevertheless, it is planned to include an efficient exhaustive search in the SeCo-framework.

There are also some algorithms that do not use a decision-list for classifying unseen example. Algorithm of this kind use all rules in the list that cover the example and then employ some voting mechanism or ordering of the list. The original version of Ripper also allows an unordered mode. Both alternatives have their advantages: A decision list, on the one hand, allows very efficient classification because the first rule in the ordered list is used to classify the example, an unordered list, on the other hand, may yield a more accurate prediction because it does not rely on a single rule.

Another interesting approach to enrich the framework is to extent it to scenarios different than classification. In Regression, for example, the target variable is not nominal any more but a numerical one. Thus, a rule has to predict a

[1] A rule can be build by using the attribute-value pairs of the example as conditions for the rule.

numerical value instead of a nominal class. There is a first version of a regression rule learner that uses the separate-and-conquer strategy[13]. In the future, it is planned to include this regression rule learner into the SeCo-framework.

Bibliography

[1] P. Clark and R. Boswell. Rule induction with CN2: Some recent improvements. In *Proceedings of the 5th European Working Session on Learning (EWSL-91)*, pages 151–163, Porto, Portugal, 1991. Springer-Verlag. 13

[2] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3(4):261–283, 1989. 12

[3] W. W. Cohen. Fast effective rule induction. In *ICML*, pages 115–123, 1995. 4, 9, 10, 15

[4] W. W. Cohen and Y. Singer. A simple, fast, and effective rule learner. In *AAAI '99/IAAI '99*, pages 335–342, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence. 9

[5] J. Demsar. Statistical comparisons of classifiers over multiple data sets. *Machine Learning Research*, Jan(7):1–30, 2006. 17, 19

[6] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997. 9

[7] J. Fürnkranz. Separate-and-conquer rule learning. *Artificial Intelligence Review*, 13(1):3–54, February 1999. 3, 4, 5

[8] J. Fürnkranz. Round robin classification. *Journal of Machine Learning Research*, 2:721–747, 2002. 3

[9] J. Fürnkranz and P. A. Flach. Roc 'n' rule learning—towards a better understanding of covering algorithms. *Mach. Learn.*, 58(1):39–77, January 2005. 12

[10] J. Fürnkranz and G. Widmer. Incremental Reduced Error Pruning. In W. Cohen and H. Hirsh, editors, *Proceedings of the 11th International Conference on Machine Learning (ML-94)*, pages 70–77, New Brunswick, NJ, 1994. Morgan Kaufmann. 14

[11] F. Janssen and J. Fürnkranz. A re-evaluation of the over-searching phenomenon in inductive rule learning. In *Proceedings of the SIAM International Conference on Data Mining (SDM-09)*, pages 329–340, Sparks, Nevada, 2009. 19

[12] F. Janssen and J. Fürnkranz. On the quest for optimal rule learning heuristics. *Machine Learning*, 78(3):343–379, March 2010. DOI 10.1007/s10994-009-5162-2. 8, 9, 14, 19

[13] F. Janssen and J. Fürnkranz. Separate-and-conquer regression. Technical Report TUD-KE-2010-01, TU Darmstadt, Knowledge Engineering Group, 2010. 20

[14] N. Lavrač, B. Kavšek, P. Flach, and L. Todorovski. Subgroup discovery with CN2-SD. *Journal of Machine Learning Research*, 5:153–188, 2004. 9

[15] R. S. Michalski. On the quasi-minimal solution of the covering problem. In *Proceedings of the 5th International Symposium on Information Processing (FCIP-69)*, volume A3 (Switching Circuits), pages 125–128, Bled, Yugoslavia, 1969. 3

[16] M. Thiel. Separate and Conquer Framework und disjunktive Regeln. Master's thesis, TU Darmstadt, 2005. In German (English title: *Separate and Conquer Framework and Disjunctive Rules*). 5

[17] S. M. Weiss and N. Indurkhya. Lightweight rule induction. In *ICML '00: Proceedings of the 17th International Conference on Machine Learning*, pages 1135–1142, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. 9

[18] I. H. Witten and E. Frank. *Data Mining — Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann Publishers, 2nd edition, 2005. 4, 6