Separate-and-conquer Regression

Technical Report TUD–KE–2010–01

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Knowledge Engineering

Abstract

In this paper a rule learning algorithm for the prediction of numerical target variables is presented. It is based on the separate-and-conquer strategy and the classification phase is done by a decision list. A new splitpoint generation method is introduced for the efficient handling of numerical attributes. It is shown that the algorithm performs comparable to other regression algorithms where some of them are based on rules and some are not. Additionally a novel heuristic for evaluating the trade-off between consistency and generality of regression rules is introduced. This heuristic features a parameter to directly trade off the rules consistency and its generality. We present an optimal setting for this parameter based on optimizing it on several data sets. The algorithm features two additional parameters that are also tuned on the same datasets as the heuristics parameter. The evaluation part of the paper gives insights on results obtained on tuning datasets that were split into two folds of equal size. The algorithm was tuned on the first set of these split databases and is evaluated on the hold-out folds and vice versa yielding two configurations of the rule learner. These are also evaluated on 9 testing datasets that were not used during the optimization.

Contents

1 Introduction 3

2 Related work 4

3 Separate-and-conquer rule learning and Regression 5
- 3.1 Separate-and-conquer for regression . . . 6
- 3.2 Error measures for regression . . . 7
- 3.3 Splitpoint processing . . . 8
- 3.4 Parameters of the algorithm . . . 9

4 Experimental setup 10
- 4.1 Evaluation methods . . . 11

5 Optimizing parameters 12
- 5.1 Optimization of the splitpoint and the left-out parameter . . . 12
- 5.2 Optimization of the heuristics parameter . . . 13

6 Results 16
- 6.1 Splitpoint processing . . . 16
- 6.2 Comparison with other systems on the two folds of the tuning datasets . . . 17
- 6.3 Comparison with other algorithms on the test sets . . . 17
- 6.4 Comparison of the size of the theories . . . 18
- 6.5 Comparison with *FRS* and *PCR* . . . 19
- 6.6 Comparison with *RegENDER* . . . 20
- 6.7 Discussion . . . 21

7 Conclusion and further work 22

1 Introduction

The accurate prediction of a numerical target variable is an important task in machine learning. There are several domains that can benefit from regression methods. For example, in the domain of financial data, it is a crucial issue to predict the volume of a credit. Here, classification algorithms can only provide a decision whether or not a credit should be given but they are not capable of predicting its size.

In the machine learning community the main task still is to predict a categorical outcome but through the last years the task of regression has gained more and more interest. Regression has its roots in the statistical community from where several algorithms were proposed over the years. The list includes the popular linear regression that is very efficient but still shows a good performance. The main advantage in using means of machine learning to solve the regression task lies in the comprehensibility of the models. Thus, a model that for example consists of simple IF-THEN rules is easy to understand and well to interpret. It enables a data miner to directly analyze the theories that are the output of a learning system. Furthermore, interpretable theories make it easier to find patterns in the data that were not obvious beforehand. Rules and trees are the two variants of interpretable models used in machine learning. As rules are typically more expressive because they are able to overlap, the goal of this work is the design of a rule learning system that on the one hand has a performance that is comparable to state-of-the-art algorithms and that on the other hand yields models that are still human-readable (assumed that the datasets are not too big).

There are several ways those rule may look like. The simplest type would be a rule that tests some attribute values and predicts a single value. More expressive models may use rules that are not predicting a single value but a linear model instead. Indeed, those are not as readable as the first alternative. In this paper we hence used the simpler model where each rule predicts a single value.

Furthermore there are several strategies to induce a set of rules. Some of them rely on the gradient-descent algorithm for finding a rule ensemble that optimizes some loss function. Others convert given trees into sets of rules. However, one of the most popular strategy in classification is the so called separate-and-conquer paradigm. Here, a rule is built by optimizing some heuristic criterion. Once an adequate rule is found it is added to the rule set, all examples that are covered by the rule are removed and the process continues as long as uncovered examples are left in the dataset. Due to its simplicity and its good performance in classification[1], we decided to use this strategy to design the algorithm.

The paper is started with a brief recapitulation of related work. Here, the basic concepts of regression are introduced and some existing mechanisms to deal with numerical target variables are described. It is followed by a short introduction of separate-and-conquer rule learning for classification. As next step the adaption that is necessary to extent separate-and-conquer rule learning to regression is specified. The error measures used in regression are introduced and the handling of numerical attributes is described. The Section is finished with a description of the parameters of the algorithm. In the next Section the experimental setup and the evaluation methods that are used during the experiments are specified. Then the method for optimizing the parameters of the algorithm is described. In the following Section the results are shown. Here, the splitpoint processing methods are evaluated. Then a comparison with existing methods on the two folds of the tuning datasets is given. In the following the experimental results on the test datasets are shown. The Section is finished with a comparison of the size of the theories and a brief discussion of the results. Then the paper is concluded.

[1] The famous RIPPER algorithm [4], one of the most accurate rule learners for classification also uses this strategy.

2 Related work

The separate-and-conquer strategy is not used frequently for learning regression rules. Exceptions include *predictive clustering rules* (PCR) [25], the FRS system [7], which is a reimplementation of the Fors system [16], and M5Rules [12, 21, 26] which generates the regression rules from model trees and uses linear models in the head of the rules. Predictive clustering rules are generated by modifying the search heuristic of CN2 [3, 2]. Instead of *accuracy* or *weighted relative accuracy* that was proposed in [17], it uses a heuristic that is based on the dispersion of the data. This algorithm also follows a different route by joining clustering approaches with predictive learning. The Fors system is able to handle numerical target attributes in Inductive Logic Programming (ILP). Therefore its application is distinct from classification.

The R^2 system [22] works to some extent analogously as other separate-and-conquer algorithms by selecting an uncovered region of the input data. But this selection differs from the mechanism used in regular separate-and-conquer learning. However, it also allows for rules to overlap and the rules predict linear models instead of a single target value.

Other mechanisms for learning regression rules are mainly based on ensemble techniques as used in the RuleFit learning algorithm [8] or in RegENDER [5]. The first algorithm performs a gradient descent optimization, allows the rules to overlap, and the final prediction is calculated by the sum of all predicted values of the covering rules instead of that of a single rule. The second one uses a forward stagewise additive modeling.

Another popular technique to deal with a continuous target attribute is to discretize the numeric values as a preprocessing step and afterwards employ regular machine learning methods for classification. Research following this path can be found in [23, 27]. The main problem hereby is that the number of bags for the discretization process is not known in advance. For this reason the performance of this technique strongly depends on the choice of the number of classes.

Algorithm 1 SeparateAndConquer(*Examples*)

```
PROCEDURE SeparateAndConquer(Examples)
  Theory ← ∅
  Rule ← InitializeRule()
  # loop until all but $n$ percent of the examples are covered
  while Covered(Examples) ≤ $n \cdot$ Size(Examples)
    # find the best rule
    Rule ← TopDownBeamSearch(Examples)
    # remember rule and remove covered examples
    Theory ← Theory ∪ Rule
    Examples ← Examples \ Covered(Rule,Examples)
  # add the default rule
  Theory ← Theory ∪ DefaultRule
  return Theory
```

3 Separate-and-conquer rule learning and Regression

Most inductive rule learning algorithms for classification employ a separate-and-conquer strategy for learning rules that allow to map the examples to their respective classes. The origin of this strategy is the AQ-Algorithm [18] but still most rule learning algorithms use this technique, most notably Ripper [4], arguably one of the most accurate rule learning algorithms for classification. The basic idea of the separate-and-conquer strategy [9] is to cover a part of the example space that is not explained by any rule yet (the conquer step) as shown in Algorithm 1. This region is covered by searching for a rule that fulfills some properties, i.e., has a low error on this partition of the input space (cf. Algorithm 2). After this rule is found, it is added to a set of rules, and all examples that are covered by the rule are removed from the data set (the separate step). Then, the next rule is searched on the remaining examples. This procedure lasts as long as (positive) examples are left. The two constraints that all examples have to be covered (also called *completeness*) and that no negative example has to be covered in the binary case (*consistency*) can be relaxed so that examples remain uncovered in the data or negative examples are covered by the set of rules. This relaxation mostly is driven from preventing overfitting. As depicted in Algorithm 1 the outer loop can be stopped before all examples are covered by setting the percentage of covered examples (parameter n in Algorithm 1).

In the end, the algorithm returns a set of subsequently learnt rules. For classification, each of the rules in the list is tested whether or not it covers the example. The first rule that covers the example (i.e., matches all the given attribute values) "fires" and predicts the value of the example by using the head of the rule. If no rule in the (decision) list covers the example the prediction is given by a special rule that usually predicts the majority class in the data. This default rule is always included as last rule when all examples are covered.

In the following we will have a closer look at the main step of the algorithm, namely how to navigate through the search space. Most of the algorithms build all possible candidate rules from the data by using all values for a given attribute and include these attribute-value pairs as conditions in a candidate rule or a refinement (the set of *Refinements* in Algorithm 2). For nominal attributes these values are given from the data itself but for numerical attributes usually all possible splitpoints are used. These splitpoints are calculated as the mean between two adjacent (previously sorted) values. For efficient computation of the new coverage statistics usually some properties of binary input data are used. Thus, the positive and negative coverage of the two partitions of the data is stored and the statistics of the next splitpoint can be computed easily by increasing the positive/negative coverage of the first fold and by decreasing the positive/negative coverage of the second one depending on the target value of the example that moves from the first to the second partition. The advantage here is that is is not necessary to iterate over all examples to compute an evaluation for a new candidate rule.

Finally, when all candidates with one condition are generated a heuristic is used to determine the best one (the EvaluateRule method in Algorithm 2). Then, the best candidate rule is stored (*BestRule* in Algorithm 2) and refined by adding all possible remaining attribute-value pairs to yield all refinements with two conditions. For nominal attributes the used ones are stored (and not used any more) and for numerical ones the relations $<$ and $\geq$ are evaluated. This means that a numerical attribute may occur twice in one rule by using it for a test on $<$ and on $\geq$. This procedure usually proceeds as long as negative examples are covered. As mentioned before, mostly to prevent overfitting, the process can

Algorithm 2 TopDownBeamSearch*(Examples)*

```
Procedure TopDownBeamSearch(Examples)
  # remember the rule with the best evaluation
  BestRule ← MaxRule ← null
  BestEval ← EvaluateRule(BestRule,Examples)
  do
    # compute refinements of the best previous rule
    Refinements ← Refinements(MaxRule, Mode)
    # find the best refinement
    MaxEval ← $-\infty$
    for Rule ∈ Refinements
      Eval ← EvaluateRule(Rule,Examples)
      if Eval > MaxEval
        MaxRule ← Rule, MaxEval ← Eval

    # store the rule if we have a new best and if it covers at least minCov examples
    if MaxEval ≥ BestEval and Covered (Rule) ≥ minCov
      BestRule ← MaxRule, BestEval ← MaxEval

    # break loop when no more refinements
    until Refinements = ∅
  return BestRule
```

also be stopped before so that there are still some negative examples covered. In this step the algorithm also ensures that a minimum number of examples is covered. This is a user given value called *minCov* in Algorithm 2. For all experiments (cf. Section 5 and 6) we fixed the minimum coverage to 3 examples.

Note that missing attribute values are treated as they are never covered. When the class of an instance is missing it is removed from the dataset in a preprocessing step.

Another mechanism is to refine several candidate rules simultaneously. The idea of this strategy is to prevent the algorithm from getting stuck in local optima. There may be situations where the current selection of the best candidate rule may lead to a suboptimal path. Hence, the search returns a candidate rule that is not the best possible one. To guarantee that always the best possible candidate rule is returned, an exhaustive search has to be performed where all possible candidate rules are generated and the best one is selected. Usually this will be too costly. As a compromise the best b rules are refined simultaneously. In this work we left the beam at $b = 1$, yielding standard hill-climbing search.

The motivation behind the usage of simple hill-climbing search is twofold: On the one hand it is more efficient compared to a beam search. On the other hand there is some evidence that an increased beam size may lead to the problem of *over-searching* [19, 20]. Despite these drawbacks, in [15] it was discovered that the requirements of the heuristic drastically depend on the type of search method. Thus, a heuristic that works good in a hill-climbing scenario may be suboptimal when it is used inside a beam search. To avoid these problems we decided to use the simple hill-climbing search.

There are many different heuristics to navigate the search (for an overview see [10]) but all of them are trying to maximize the coverage of positive examples (p) and to minimize the negative coverage (n). To reach this objective different ways are employed but usually, in some way, there has to be a combination of consistency (i.e., the error or the negative coverage) of the rule and its generality (i.e., the number of examples that are covered). Most of the heuristics have a fixed trade-off but some of them feature a parameter to adjust it. In previous work the parameters of some of these heuristics were tuned, so that they achieve the most accurate trade-off between consistency and coverage [14]. In this work we follow the same path by defining a parametrized heuristic that trades-off error and generality of a rule and by tuning its parameter to yield the best fit between these two objectives.

When dealing with numeric target variables the algorithm has to be adapted in several ways as shown in the next Section.

3.1 Separate-and-conquer for regression

As noted above some of the features that come with categorical binary data do not apply for numerical target variables. Thus the algorithm had to be adapted in several ways. First of all the efficient computation of the coverage statistics for numeric input attributes can not be used any more. In regression there is no notion of positive and negative coverage.

Here, only the error of the rule on the covered data and the number of covered examples are available. For this reason a novel splitpoint method had to be developed that allows using only a subset of all possible splitpoints to prevent the algorithm from getting too inefficient (cf. Section 3.3). The heuristics that were introduced for the task of classification are also not suited for regression. Hence, an alternative error measure has to be defined. The default rule also has to be adapted because there is no majority class any more. A simple way to do this is to take the mean over all remaining examples as prediction. Another way would be to take the mean of all examples. We experimented with both settings (cf. Section 6). Finally, the methods for evaluating the final model had also to be adapted because using measures like *accuracy* (the percentage of correctly predicted examples) is not practicable any more. A detailed description of the evaluation measures is given in Section 4.1.

Finally the semantics of the rule itself had to be adapted. Thus, a regression rule now has as head a certain value instead of a (nominal) class. The predicted value is calculated as the mean over all examples covered by the rule. A rule consists of an antecedent where all attributes are tested and a consequent that predicts the value. An example for such a rule may be

$$\text{IF } att_1 = value_1 \text{ AND } att_2 < value_2 \text{ AND } att_2 \geq value_3 \text{ THEN } 3.125$$

3.2 Error measures for regression

There are several ways to compute the error of a rule or a model for regression tasks given in the literature. In this Section the most important ones are described.

In the following m denotes the total number of examples in the (current) dataset, y is the value of the current example, $\bar{y}$ is the value predicted by the rule, y' is the mean over all examples, r_{min} is the lowest value in the data and r_{max} is the highest value in the data (for the target attribute).

The *mean absolute error* is the mean of the sum of the absolute errors of all examples that are covered by the rule

$$L_{MAE} = \frac{1}{m}\sum_{i=1}^{m}|y_i - \bar{y}_i| \tag{3.1}$$

The *variance* or *mean squared error* is the mean of the sum of the squared error of all examples that are covered by the rule

$$L_{MSE} = \frac{1}{m}\sum_{i=1}^{m}(y_i - \bar{y}_i)^2 \tag{3.2}$$

To get the *root mean squared error* the root of the L_{MSE} is taken

$$L_{RMSE} = \sqrt{L_{MSE}} \tag{3.3}$$

A problem with all of the above measures is that they are domain-dependent. As the amplitude of the values in the domain is changing the amplitude in the error measures is changing as well. Thus, the errors are not comparable among different datasets. For using these measures to compute the heuristic value this may not be a problem because only candidate rules are compared to each other. But if a combination of the error and the coverage is taken this becomes crucial due to normalization issues. If the coverage is expressed as normalized value the error measure has to be also normalized.

Therefore, all values have to be normalized into the $[0,1]$ interval. For the mean absolute error this can be done by normalizing with the difference of the highest and lowest value in the data. Usually this should map the values in the $[0,1]$ interval but is is not guaranteed that this does always work. But for our purpose it seems to be sufficient to catch the majority of the values. The *normalized mean absolute error* is defined as

$$L_{nmae} = \frac{\sum_{i=1}^{m}|y_i - \bar{y}_i|}{m \cdot (r_{max} - r_{min})}. \tag{3.4}$$

For the normalization of the squared errors usually the normalization term is the deviation from the mean which is given by

$$L_{default} = \sum_{i=1}^{m}(y_i - y')^2. \tag{3.5}$$

Thus, the *relative root mean squared error*[1] becomes

$$L_{RRMSE} = \frac{L_{RMSE}}{\sqrt{\frac{1}{m} \cdot L_{default}}}. \tag{3.6}$$

All of the shown measures can be used for evaluating a single candidate rule but also for evaluating a whole theory (an ordered set of rules). A detailed description of how the algorithm is evaluated follows in Section 4.1.

So far there is no measure for the *coverage* of a rule. This could easily be derived by taking the number of covered examples. To map this in the interval $[0, 1]$ the term is normalized by the total number of examples yielding the *relative coverage*

$$relCov = \frac{1}{m} \cdot coverage(Rule). \tag{3.7}$$

We decided to combine the error and the generality of a rule by using the *relative root mean squared error* and the *relative coverage* by

$$h_{cm} = \alpha \cdot (1 - L_{RRMSE}) + (1 - \alpha) \cdot relCov. \tag{3.8}$$

Here, the parameter α enables a trade-off between the error and the generality of the rule. For $\alpha = 1$ the relative coverage is ignored and thus the rules are evaluated solely by inspecting their error. This setting would yield a model that consists only of rules that cover a single example in the data and thus clearly would lead to overfitting. The other extreme is to set $\alpha = 0$ which results in ignoring the error of the rule. A model built with this setting would only consist of the default rule, because its coverage is the highest that could be achieved by any rule. The optimal trade-off lies somewhere in between these two extremes. The method for finding an optimal setting is described in Section 5.

The heuristic h_{cm} is an adaption of a previously introduced heuristic called *relative cost measure* [10]. Its formula is given by

$$h_{cr} = c_r \cdot \frac{p}{P} - (1 - c_r) \cdot \frac{n}{N}. \tag{3.9}$$

with $p \equiv$ positive coverage of the rule, $n \equiv$ negative coverage of the rule, $P \equiv$ total number of positives and $N \equiv$ total number of negatives.

It was designed for evaluating classification rules thus relying on the positive and negative coverage of a rule. In previous work [14] the optimal setting for the parameter of $h_{cr} = 0.342$ was found. It encodes a clear favor of the consistency (encoded by $\frac{p}{P}$, the true positive rate of a rule) over the coverage (denoted by $\frac{n}{N}$, the false positive rate of a rule). It achieved good performance among different classification heuristics as shown in [14] as it was the second best heuristic among all. Thus the motivation to modify exactly this heuristic was the good performance and that it is best suited to be adapted to regression. For this purpose, the consistency was defined by taking the *rrmse* of a rule and the coverage term was denoted by the *relative coverage* of the rule.

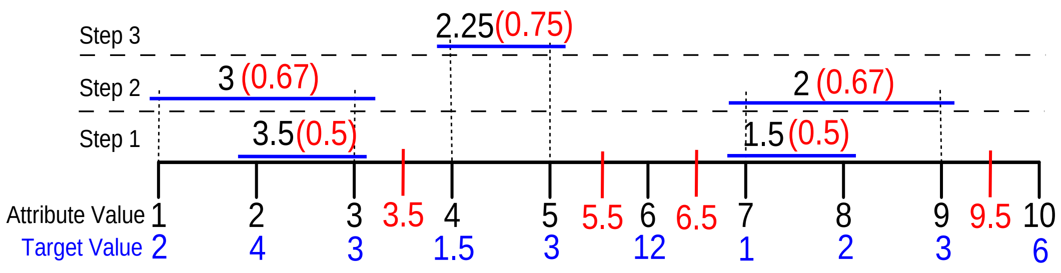
3.3 Splitpoint processing

As noted above, the generation of all possible splitpoints would be too costly. To avoid this a method for restricting the splitpoints for an attribute was developed. The basic idea comes from supervised clustering. Thus, we try to identify regions in the data of the current attribute that share a small (mean absolute) error computed on the target variable. The aim of the clustering is to yield partitions of the attribute that share a low error in the hope that the error of a rule that covers these regions will also be low. Clustering stems from the same motivation because it also guarantees that each cluster has the lowest possible error. The user has to define how many clusters and hence how many splitpoints are desired. We experimented with different settings but surprisingly a rather low number of splitpoints seemed to be sufficient (cf. Section 5.1).

Figure 3.1 displays how the cluster algorithm works. In the example in Figure 3.1 the attribute has 10 values moving equidistant from 1 to 10. The values depicted in blue are those of the target attribute of the respective example. In the first step the attribute values are ordered ascending and each value becomes a cluster containing exactly this value. Then two adjacent clusters are searched for which the error when using the mean of the two target values as prediction is the lowest. In the example these are the clusters 2 and 3, 7 and 8, and 8 and 9. Though the objective in the first step is to join two adjacent clusters both 2, 3 and 7, 8 are joined (its arbitrary whether to join 7 and 8 or 8 and 9). The mean of

[1] In the remainder of the paper the *relative root mean squared error* is abbreviated with *rrmse*.

Figure 3.1: Example of the splitpoint clustering method



Step 3
Step 2
Step 1
2.25(0.75)
3 (0.67)
2 (0.67)
3.5(0.5)
1.5(0.5)
Attribute Value 1 2 3 3.5 4 5 5.5 6 6.5 7 8 9 9.5 10
Target Value 2 4 3 1.5 3 12 1 2 3 6

the first cluster is $3.5 = \frac{4+3}{2}$ and the second one has a mean of 1.5 (depicted in black in Figure 3.1 above the number ray). If the mean absolute error is taken, both clusters have an error of 0.5, which is shown in brackets and in red in the corresponding figure. An error of $0.5 = \frac{|4-3.5|+|3-3.5|}{2}$ is also the lowest error that can be achieved given the example data.

In the second step the function is executed recursively and again those clusters are joined that have the lowest error among all possible clusters. So, in this step, cluster 1 is joined with the second cluster and the cluster with a value of 9 is joined with the third cluster. The error of both clusters grows to 0.67 because adding the respective example does yield a raise of the error (i.e., $L_{MAE} = \frac{|2-3|+|4-3|+|3-3|}{3} = 0.67$ for the first cluster). Joining any of the untouched clusters leads to a higher error which means that the cluster with next lowest error is built in step 3. After the second step two clusters containing at least 2 examples were built and therefore 5 splitpoints exist. In the example the user given number of splitpoints is set to 4. Hence another cluster has to be built until the algorithm is finished. This last cluster is derived by joining the clusters with the values 4 and 5 and it yields an error of 0.75.

After the third step 3 clusters are built and the splitpoints are simply derived by taking the mean between the values of two adjacent clusters or two values if the cluster contains only one example. The 4 splitpoints are 3.5, 5.5, 6.5, and 9.5 (depicted in red in Figure 3.1 in the number ray).

We have evaluated the effectiveness of the splitpoint method by comparing it to the usage of another splitpoint method where x splitpoints are selected equidistant. The results of this comparison are shown in Section 6.1.

For the computation of the error the *mean absolute error* was used. This choice is arbitrary but experiments with the *root mean squared error* did not yield any performance difference. Thus we decided to use this type of error measure.

3.4 Parameters of the algorithm

There are 3 parameters the user has to specify.

- The parameter of the heuristic,
- the number of splitpoints, and
- the percentage of examples that are left uncovered.

The parameter of the heuristic is optimized with a greedy procedure that narrows down the region of interest. This procedure is described in detail in Section 5.

The number of splitpoints is crucial for the runtime of the algorithm. Additionally, using all splitpoints may result in even worse performance. This is due to some effects that come with the interplay of the different components of the algorithm. Depending on the parameter of the heuristic, the number of splitpoints indirectly influences the partitioning (i.e., the coverage of the rule) of the input space. Clearly, if all splitpoints are used, there is always a possibility to select a single refinement that covers only one example (by using the test $<$ on the second lowest attribute value or by using $\geq$ on the highest value). This is impossible when the only refinement is given by a splitpoint right in the middle of all examples. For higher values of the heuristic's parameter and therefore a higher weight on the error, the choice of the number of splitpoints becomes more and more important. For a detailed description of this phenomenon see Section 6.7.

The last user-given parameter is the percentage of examples that are left uncovered by the outer loop of the algorithm. This parameter clearly depends on the dataset. During the experiments there was some evidence that we had included databases that basically encode randomness and for those learning anything results in worse performance (e.g. the dataset *quake*). On those datasets the best model will be given by one that simply predicts the mean over all examples like the default rule does (cf. Section 3.1).

Table 4.1: Overview of the databases used for optimization

name	# nominal attributes	# numeric attributes	missing class values	# instances
abalone	1	7	0	4177
auto-mpg	3	4	0	398
auto-price	1	14	0	159
breast-tumor	8	1	0	286
compressive	0	8	0	1030
concrete-slump	0	9	0	103
cpu	1	6	0	209
delta-ailerons	0	5	0	7129
echo-month	3	6	0	130
forest-fires	2	10	0	517
housing	1	12	0	506
machine	0	6	0	209
pbc	7	11	6	418
pyrim	0	27	0	74
quake	0	3	0	2178
sensory	11	0	0	576
servo	4	0	0	167
strike	1	5	0	625
triazines	0	60	0	186
winequality-white	0	11	0	4898

This Section describes the experimental setup that was used during the tuning phase and the evaluation of the algorithms. To optimize the 3 parameters of the algorithm all 20 datasets used for tuning were split into 2 folds of equal size. On the first fold, all steps of the optimization procedure were done and afterwards the best model found on these folds was evaluated on the second folds. This is also done vice versa. Hence, the experiments yield two configurations of the same algorithm that only differ in the parametrizations of the 3 parameters used in the algorithm. A test of the parametrizations on the hold-out folds of the tuning datasets is the first step of the evaluation. Additionally some insights are gained by evaluating the parametrizations also on those datasets that were used during the optimization. To complete the evaluation, the two resulting algorithms were also evaluated on 9 datasets that were not used for any optimization purposes.

The aim of the experiments was to optimize the parameters of the algorithm on a set of diverse datasets to capture characteristics of a wide variety of different datasets. Our hope was that by taking a set of datasets that are very different the parameters would be more stable. For this reason, we selected 29 databases in total from the UCI-Repository [1] and from Luis Torgos website[1]. After that the datasets are divided into 20 sets that were used during the tuning phase and 9 sets that were only used for evaluation purposes. Table 4.1 gives an overview of the tuning databases. As mentioned above the main motivation to select these datasets was to capture a lot of different learning problems. Thus, the number of nominal and numerical attributes should be different among the databases and the domains from which they origin should be as diverse as possible.

The 9 datasets that were used to evaluate the resulting configurations of the algorithm are shown in Table 4.2. The distribution among the 20 tuning databases in terms of nominal and numerical attributes as well as in terms of size should be approximately the same as in those datasets that were used for testing. Therefore both bags of data should contain some small, some medium and some big databases. The distribution in terms of attribute types should also be comparable for both the tuning and the testing databases.

[1] These databases can be downloaded at `http://www.liaad.up.pt/~ltorgo/Regression/DataSets.html`.

Table 4.2: Overview of the databases used for evaluation only

name	# nominal attributes	# numeric attributes	missing class values	# instances
auto93	6	16	0	93
auto-horse	8	17	2	205
cloud	2	4	0	108
delta-elevators	0	6	0	9517
meta	2	19	0	528
r_wpbc	0	32	0	194
stock	0	9	0	950
veteran	4	3	0	137
winequality-red	0	11	0	1599

4.1 Evaluation methods

The primary method to evaluate the algorithm was the *rrmse*. The advantages of this evaluation measure clearly lie in its domain-independency. As mentioned above it is mandatory to use a measure that does not differ among different domains because the goal was to get results that can be averaged over various datasets. For some of the experiments it would take too much space to include results on every single dataset. In those cases our means for evaluating the different algorithms was to average the results over all datasets. We are aware of the problems that come with averaging results over many different domains (i.e., some databases may be outliers with huge variance compared to the majority of the other datasets) and hence include a Friedman-Test with a post-hoc Nemenyi-Test as suggested in [6]. The resulting CD-charts give insights how good the algorithms perform by evaluating their ranking independently from using averages.

The second mean for determining the quality was the *correlation coefficient* (also referred to as *Pearson's correlation coefficient*). It computes the correlation of all predicted values with the true outcome of the target variable which is equal to compute the covariance between two variables divided by the product of their variances. It is also domain-independent and widely used for evaluating regression algorithms (e.g., in [25]). The problem when using it with the kind of algorithm used in this work is that wide ranges of the examples are mapped onto the same value. In spite of algorithms like *M5Rules* that predicts a linear model which nearly maps each example to a different value, the separate-and-conquer based algorithm introduced in this paper only predicts one value for each rule. As rules typically cover more than one example[2] all examples covered by a rule are mapped to the same value. Furthermore, as a model that predicts a constant value for all examples would have a correlation coefficient of 0 this type of measure is not well suited for the kind of algorithm discussed here. Consequently, the values of the correlation coefficient for the separate-and-conquer based algorithm are lower by a certain margin. Therefore, this evaluation measure is not as expressive as the *rrmse* and only is included here for completeness.

[2] In our algorithm a rule has to cover at least 3 examples.

Table 5.1: Results for different parametrizations of the splitpoint computation (average RRMSE over the 5 parametrizations of the heuristic)

parameter	1	3	5	7	9	11	19
folds 1	1.0675	**0.9929**	1.0132	1.0067	0.9992	1.0126	1.0163
folds 2	1.0540	1.0256	1.0261	1.0245	**1.0209**	1.0427	1.0240

Table 5.2: Results for different parametrizations of the left-out-parameter (average RRMSE over the 5 parametrizations of the heuristic)

parameter	0	0.01	0.02	0.03	0.05	0.1	0.2
folds 1	0.9929	0.9787	0.9776	0.9759	0.9739	**0.9704**	0.9736
folds 2	1.0209	1.0221	1.0182	1.0156	0.9940	0.9835	**0.9701**

5 Optimizing parameters

This section describes how the three parameters of the algorithm were optimized. It is started with the rather simple identification of best values of the cluster-based splitpoint method and the percentage of examples that are left uncovered. Then, a more detailed description about the parameter optimization method for determining an optimal trade-off for the evaluation heuristic is given.

As mentioned in Section 4, for all parameter optimizations the datasets were split into 2 equal sized folds. Therefor, all datasets were randomized in advance using the unsupervised randomize function of *weka* [28]. All evaluation measures were computed using one run of a 10-fold cross validation. To make sure that the parameters are stable they were optimized on the first folds of all datasets and were then evaluated on the second folds of all datasets and vice versa which yields two configuration of the algorithm. If a single value becomes optimal on both folds, this would be great evidence that this parameter is stable among many different learning problems.

5.1 Optimization of the splitpoint and the left-out parameter

Though these two parameters are likely to have a small deviation among different databases, we decided to optimize them first and fix them until we start optimizing the parameter of the heuristic. We believe that the parameter of the heuristic has a stronger influence on the performance of the algorithm than the other two parameters. This was the reason to first focus on the less expressive parameters until starting to deal with the trade-off employed by the heuristic. Through the whole step of optimizing the algorithm's parameters an iterative optimization procedure that keeps one parameter fixed and searches for an optimal value for the other one was used. After the first parameter can be fixed, the search continues with the next one.

To start optimizing the splitpoint parameter the other two had to be fixed. In advance it is not known how to determine these values. Thus, the *left-out*-parameter was fixed to 0, therefore all examples have to be covered. In this case the default rule is built by using the mean of all examples. In other cases where not all examples have to be covered it is built by using the mean of all uncovered examples. In contrary, it is not obvious what parameter value can be used for the heuristic. For this reason, 5 different values were used during the optimization. To make a choice, the two extreme values were included (namely $\alpha = 0$, relying only on the coverage of the rule, and $\alpha = 1$, now practically ignoring the coverage and using only the relative root mean squared error of the rule, and some values in between, namely 0.4, 0.5, and 0.6). These values were used to include different properties of the heuristic. Clearly, using only two parameters would be suboptimal because there is some evidence that the optimal parameter rather would lie somewhere in the middle than at the beginning or the end of the parameter curve induced by outline the parameter values over the error [14]. We expected this parameter curve to be shaped like a U, where the two extreme values would yield a rather bad performance and the optimal value lies somewhere around 0.5 (cf. Section 5.2). Thus we decided to include 2 values around 0.5

and 0.5 itself. To have a combined error measurement for the optimization procedure the mean over the *rrmse* of these choices was chosen.

In the beginning, the *left-out*-parameter was fixed to a value of 0 yielding a starting point for the optimization of the *splitpoint*-parameter. To find the best value some intuitive values (1, 3, 5, 7, 9, 11, and 19) were used. All values bigger than 19 were skipped because a clear gain in runtime performance should be achieved. Using huge values would result in practically using all possible splitpoints and thus would not improve the algorithm's runtime[1].

Table 5.1 shows the results for the two optimization procedures (for the two folds of the divided datasets). As can be seen the best number of splitpoints was 3 on the first folds and 9 on the second folds (the error is depicted in bold in the figure). On the first folds, however, using 9 splitpoints yields the second best *rrmse* which lacks only 0.0063 behind the best performing number of splitpoints. On the second folds, using 9 splitpoints performed best followed by using 19 splitpoints. Interestingly, using 19 splitpoints could be seen as a very big number that was only tested to make sure that the algorithm cannot benefit from taking so many splitpoints into account. However, on these folds it seems that far more splitpoints are needed to yield an acceptable result. The best performing number of 3 from the first folds, lacks 0.0047 in terms of *rrmse* behind the best one and therefore is the third best method. Nevertheless, the gap between different parametrizations seem to be bigger on the first folds than on the second ones. Regarding the split of all tuning datasets into 2 folds of equal size, these results seem to reflect the randomness in splitting the datasets. An optimal result clearly would be that on both folds the same number of splitpoints performs best, but when taking the random split into account the results seem to be reasonable.

After the optimization of the splitpoint parameter the same procedure was employed to the *left-out*-parameter of the algorithm. Here, the splitpoints were already fixed to 3 for the algorithm tuned on the first folds and to 9 for the variant tuned on the second folds. To find the best value also some intuitive parameters were used. Thus, the values of 0, 0.01, 0.02, 0.03, 0.05, 0.1, and 0.2 were tested during this optimization. To cover all examples was included to make sure that is more effective to leave some examples uncovered. Clearly, this depends on the given dataset. But it also depends on the quality of the induced rules. For numerical target variables it can be useful to cover only those parts of the data that share some common characteristics. For the remainder of the data it could be beneficial to treat them independently from their characteristics, i.e., by assigning them the same target value. The basic idea here comes from separate-and-conquer algorithms for classification. Those algorithms also have a default rule that covers some remaining examples. Here, usually the biggest class in the data is taken which is possible because all other classes are covered by the rules that are found before. In regression this biggest class can be seen as those parts of the data that do not share attribute values that are correlated with the target value at all. Therefore, the usage of a default rule should be also beneficial when dealing with regression tasks.

As can be seen in Table 5.2 two different parameters performed best on the two folds. Practically this can be attributed to the same reasons that were already discussed during the optimization of the splitpoint parameter. Thus, on the one hand the randomized split of the data into 2 folds of equal size could be manipulated the characteristics of the datasets. On the other hand it could also be possible that there is no unique best value for leaving examples uncovered. This becomes even more obvious when the results on the two folds are compared. Here, it becomes also clear that leaving examples uncovered is mandatory for the performance of the algorithm.

On both folds covering all examples has the worst performance. On the other hand, it clearly is of interest that the best settings are not leaving a few examples uncovered, but are only covering 90 % on the first folds and even only 80 % on the second folds. The reasons of this are twofold: On the one hand, the setting of the *left-out*-parameter is influenced by the number of splitpoints that is used. On the other hand it seems that great parts of the data do not share certain characteristics and therefore it is beneficial to cover them with a constant target value. For example, on the dataset *quake*, the induction of a single rule is always worse than using only the default rule[2]. The reason for this could be that this dataset basically encodes randomness which implies that there are no patterns contained in the data. Thus, as an observation of these results, one might say that it is better to find rules that are able to generalize those parts of the data where a generalization is possible and to leave all other parts of the data untouched. The question of how the induced rules look like and how much of them are found on all datasets in average is addressed in Section 6.4 in more detail.

5.2 Optimization of the heuristics parameter

For the optimization of the heuristics parameter a framework similar to the one introduced in [14] was used. It employs a binary search to find the best parameter and was proven to yield stable parameters for classification heuristics as shown in [14]. The heuristic used in this work is a direct adaption of a previously introduced heuristic for classification. There, it was called *relative cost measure* and it directly trades off the true positive rate (*tpr*) and the false positive rate *fpr* of a

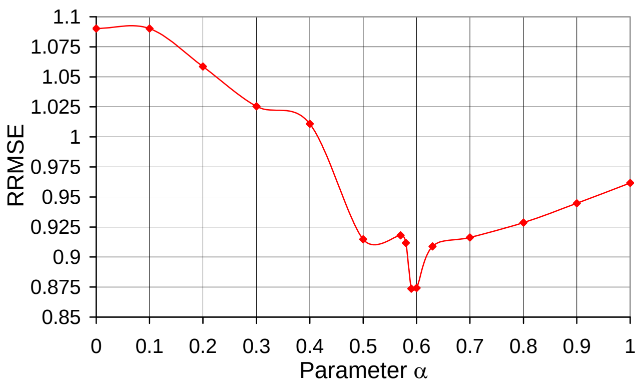
[1] Note that the number of disjunct values for an attribute in the data is rather small.

[2] The performance of a model that predicts the mean of all examples was superior to all other algorithms.

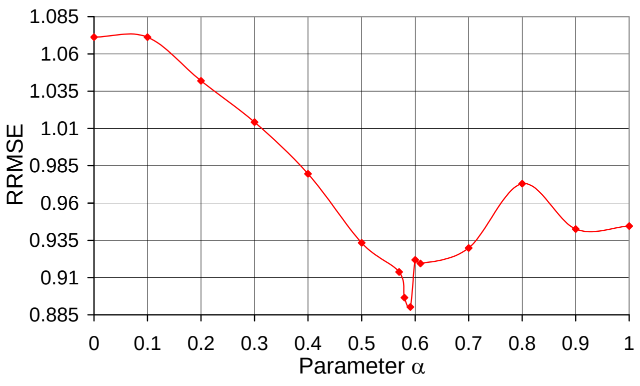
Table 5.3: A sample parameter search

Run	set which has to be searched	increment	best parameter	L_{RRMSE}
1	$\{0.1, ..., 1.0\}$	0.1	0.4	0.852
2	$\{0.37, ..., 0.43\}$	0.01	0.42	0.833
3	$\{0.417, ..., 0.423\}$	0.001	0.418	0.815
4	$\{0.4177, ..., 0.4183\}$	0.0001	0.4178	0.805
5	$\{0.41757, ..., 0.41763\}$	0.00001	0.4178	0.805

Figure 5.1: Parameters over RRMSE for both folds of the tuning datasets



(a) tuned on folds 1



(b) tuned on folds 2

rule. Due to the good results achieved with the abovementioned framework for this heuristic we decided to use the same framework to optimize the parameter of the h_{cm} metric.

The search is started with a range of intuitively appealing parameters. Thus, the two extremes of 0 and 1 are tested together with some values in between $(0.1, 0.2, ..., 0.9)$. All settings are evaluated by taking the average of the *rrmse* on the 20 datasets presented in Section 4. Then the best performing parameter is used for further inspection. Therefore, an area around this parameter is inspected in more detail. There are several choices to do this, but we decided to evaluate 6 parameters around the best one. Those are distributed equidistant around the best parameter with decreasing the step size from 0.1 to 0.01. This procedure is executed recursively, so in the next step the 6 parameters around the next best value are tested. The search stops if the *rrmse* improvement falls below a threshold of $t = 0.0005$. This choice was arbitrary but we believe that the effort that has to be made to narrow down the parameter for the next step of the search procedure is too high compared to the performance gain the next execution of the procedure may yield.

An example for a parameter search is given in Table 5.3. Here, the first 10 parameters lie between 0 and 1. After the first run 0.4 was the best parameter. Therefore the region around this parameter is refined further on. The procedure stops when the error does not improve any more, yielding 0.4178 as best parameter. Clearly this procedure is greedy and it is not guaranteed to find the best parameter. But for our purposes it was sufficient because we mainly wanted to find a set of parameters in which it is likely that the best one would be. For a detailed description of the parameter search see [14].

Figure 5.1 shows a graphical interpretation of the search for both of the split datasets. Our expectation was that the curves are shaped like a U. This is because very low and very high parameter settings should not perform good whereas some values in the middle should achieve the best performance. The pictures are similar to some extent. For both folds of the datasets very low parameter settings do not achieve good performance. In these cases the algorithm mostly does not learn anything on the datasets. It just returns a theory containing only the default rule which may be interpreted as the base line in terms of performance. This is because it simply predicts the mean over all target values in the dataset. Thus, for both settings this parameter setting lies clearly above a *rrmse* of 1. When the parameters are decreased the performance becomes better as long as the optimal setting is reached. After that the theories become worse again.

In the scenario at hand it seems that learning only the default rule results in the worst theories whereas learning a rule for each example (which corresponds to a parameter setting of 1) has at least a better *rrmse* than 1. For the parameters that are optimized on the first folds of the split datasets (Figure 5.1 (a)) the curve shows some fluctuations in the part located left of the best parameter. In spite of this behavior the curve depicted in (Figure 5.1 (b)) is monotonically decreasing in this area. For parameter settings that are bigger than the best parameter the curve in the left figure is now showing a monotone increase whereas it shows more fluctuations when the parameter is tuned on the second folds of the partitioned datasets.

Interestingly, the best parameters are very similar on both folds of the divided datasets. This means that the parameters are stable among different splits of the datasets. On the first folds of these datasets the best parameter lies at 0.59 and on the second folds it was 0.591. For the first folds the parameter 0.591 lacks only behind 0.007 in terms of *rrmse*. For the second folds the difference in performance between 0.591 and 0.59 was 0.001. Those results show that these parameters are stable among the two randomized splits of the datasets used for tuning.

Assumed that the best parameter lies somewhere in the region of 0.6, consistency should be preferred over coverage for regression rules. This also holds for classification rules where the preference of consistency is even stronger than in regression. Nevertheless it is an interesting result that the evaluation of a rules quality follows similar standards both in classification and in regression.

Table 6.1: Runtime of different splitpoint methods on the test set

method	runtime (in sec.)
3 equidistant splitpoints	2625.4
3 clustered splitpoints	1234.3

6 Results

In this Section the results are presented. First a comparison about different mechanisms of splitpoint computing is given. After that, a comparison of the algorithm in terms of *rrmse* and correlation coefficient with other rule-based and non rule-based regression algorithms is conducted. In the next Section information about the size of the learned theories is displayed. Since there was only one algorithm that is also based on rules (namely the *M5Rules*-system) that is freely available, this comparison is focused on the three rule-based algorithms. Nevertheless, we included some previously published results on *PCR* [24] and compare them to our implementation in terms of error and number of rules. The Section is finished with a discussion about the properties of the algorithm specific to the learning of regression rules with the separate-and-conquer strategy.

6.1 Splitpoint processing

Table 6.1 shows a comparison of the runtime of 2 different splitpoint methods. At first, 3 equidistant splitpoints per attribute were used. Then, 3 clustered splitpoints were employed. Evaluating all splitpoint was too costly. All runtimes depicted in Table 6.1 are the averages of 10 independent runs on a dual Pentium 4 2.8 GHz processor with 2 GB RAM on the 9 datasets used for testing (cf. Section 4).

As can be seen in Table 6.1 the clustered splitpoint computation is more efficient than the equidistant method. At first sight this may appear contrary to what could be expected. Due to the much more simpler computation of equidistant splitpoints this method should be faster than the clustering method. But note that this evaluation was done by letting the whole algorithm run on the 9 test datasets. Not surprisingly the quality of the equidistant splitpoints is worse compared to the clustered splitpoints. This results in a significant higher number of candidate rules that have to be evaluated during the search for the best rule. This is because a candidate rule has to be refined more often. Additionally, mostly for very small or huge datasets, the theories induced by the rule learning algorithm are bigger in terms of conditions per rule and the total number of rules for the splitpoints with lesser quality. On the dataset *delta-elevators*, which is a huge one (containing 9517 instances and 6 numeric attributes), the modification of the algorithm that uses clustered splitpoints found 356 rules that contain 3254 conditions. The algorithm with the equidistant splitpoint computation has found 755 rules with 8063 conditions instead. This is more than twice the size of the previous method in terms of number of rules and conditions. Clearly, this results in longer runtimes. The same situation unfolds for small datasets. The algorithm with clustered splitpoints needs 13 rules with 41 conditions to describe the dataset *veteran* that contains 137 instances and 3 numeric as well as 4 nominal attributes. In contrary, the method with the equidistant splitpoint computation has found a theory that contains of 17 rules with 55 conditions.

For the reasons mentioned above the splitpoint generation methods were additionally compared independently of their usage inside the algorithm. Here, the time that was needed to generate the splitpoints in a clustered fashion do not significantly differ from that which was needed when the equidistant method was used. It took 5.4 seconds to generate the splitpoints in a clustered mode and 2.7 seconds in equidistant mode. Thus, the most time consuming step of the algorithm is the evaluation of the candidate rules and not the generation of the splitpoints. Thus, the quality of the splitpoints is crucial and not the runtime needed to generate them.

Du to this, the algorithm clearly benefits from the usage of the clustering mechanism. On the one hand, the theories are smaller in terms of rules and conditions and on the other hand the algorithm can even benefit in terms of runtime. Another important observation is that the runtime also strongly correlates with the number of splitpoints. Assuming a dataset with 20 numerical attributes, 3 splitpoints and that only the first candidate rules are computed, this results in 60 candidate rules. Now if the splitpoints are increased to 10 we end up in 200 possible candidate rules that all have to be evaluated.

Table 6.2: Results in terms of L_{RRMSE} and correlation coefficient (CC) for different algorithms on the tuning datasets

	M5Rules		Linear Regression		MLP	
fold	L_{RRMSE}	CC	L_{RRMSE}	CC	L_{RRMSE}	CC
1	0.7425	0.6498	0.8145	0.6325	1.0154	0.6232
2	0.8058	0.6623	0.9116	0.6414	1.3890	0.6192

	SVMreg		SeCoReg tuned on folds 1		SeCoReg tuned on folds 2	
fold	L_{RRMSE}	CC	L_{RRMSE}	CC	L_{RRMSE}	CC
1	0.7917	0.6281	0.8736	0.5392	0.8976	0.533
2	0.85	0.6363	0.9291	0.5393	0.8903	0.5301

6.2 Comparison with other systems on the two folds of the tuning datasets

The main focus of the comparison is how good the algorithm performs against other regression algorithms. Table 6.2 gives an overview of the different algorithms compared to each other on the two folds using the relative root mean squared error (RRMSE) and the correlation coefficient (CC). The tuned algorithm is referred by the name *SeCoReg* in the remainder of the paper.

There are 4 other algorithms that are all implemented in *weka* [28] which were used to compare our system with. Clearly some of them are much more complex than our rather simple algorithm[1]. On the other hand most of them employ more complex models, i.e., hyperplanes like the *multilayer perceptron* or support vectors like the *SVMReg*. The *linear regression* is also a rather simple algorithm that nevertheless employs a quite trade-off between runtime and accuracy. The *M5Rules* algorithm uses rules to explain the data as well. But it does not have a single output value in the head of the rules like our algorithm. This rule learner predicts a linear model in each of the rules heads which makes it much more flexible because it is able to map the examples covered by one rule on many different outcome values. This is not possible when a single target value is predicted in the head of each rule. Therefore the goal of this algorithm is a bit different from our implementation. Here, the goal is to search the input space for regions that are good suited to fit a linear model, whereas our algorithm searches for partitions that share a target value with low variance.

All of the algorithms used for comparison are implemented in *weka* and all their parameters are left at default values. The reasons to select these 4 algorithms were that our implementation had to prove that it is comparable in terms of error to many other state-of-the-art systems. Another reason to select these particular algorithms for benchmark was the lack of freely available regression rule learning algorithms. The only free system we found was REGENDER and a comparison to this algorithm is given in Section 6.6.

We also had included a comparison with *PCR* [25, 24], but all displayed results are taken from the PhD-Thesis [24]. Therefore the comparison is only included to give insights how good the *SeCoReg*-algorithm may perform compared to other separate-and-conquer rule learning algorithms. The results obtained with the *FRS* system [7] are also included. Some of the datasets used in [24] were part of the tuning datasets of the experiments. Nevertheless, we think that it is worth to include this comparison (cf. Section 6.5), even it lacks some validity.

In Table 6.2 the results of all algorithm on the two folds of the tuning datasets are displayed. Since there were 20 datasets we decided to display only averages (both the *rrmse* and the correlation coefficient). Results of both derived *SeCoReg*-algorithms are shown together with their performance on the data sets on which they were tuned. First the results in terms of *rrmse* are evaluated. Not surprisingly both variants of the algorithm that were tuned on the respective folds are better than using them on the left-out folds. The ranking of the algorithms is similar on both experimental variants. The best one was the *M5Rules* algorithms followed by the *SVMreg*. The next best performance was achieved by the *linear regression*. The *SeCoReg* was ranked on the 4th place on both folds of the divided datasets, only slightly behind the *linear regression* (lacking 0.0831 behind on the first folds and 0.0175 on the second folds). The *Multilayer Perceptron* had the worst performance on both folds with a rather big gap between this algorithm and the next best one.

6.3 Comparison with other algorithms on the test sets

To validate the results on total different datasets the algorithm was also tested on 9 independent test sets (cf. Section 4) that were never used for tuning purposes. This step is necessary to make sure that the tuning datasets, even though they were split into two disjunct folds, were not overfitted during the parameter tuning phase. Table 6.3 displays the results

[1] Note that the algorithm neither has a pruning functionality nor an optimization phase.

Figure 6.1: Comparison of all algorithms against each other with the Nemenyi test the first folds of the tuning data. Groups of algorithms that are not significantly different (at $p = 0.05$) are connected.

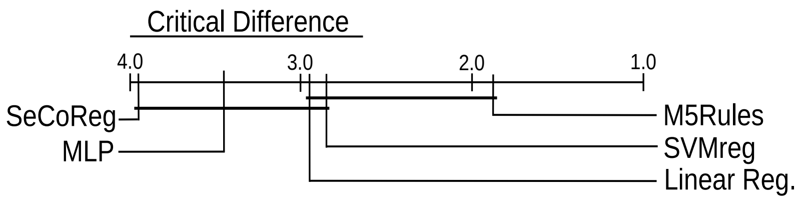
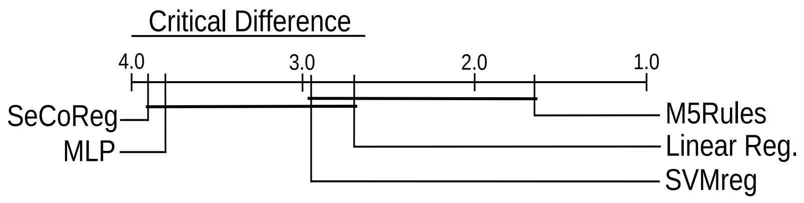


Figure 6.2: Comparison of all algorithms against each other with the Nemenyi test the second folds of the tuning data. Groups of algorithms that are not significantly different (at $p = 0.05$) are connected.



in terms of *rrmse* on the test databases for all of the 4 weka algorithms and the two configurations of the *SeCoReg*-learner. The ranking of the algorithms differs slightly compared to the results on the two folds of the partitioned tuning datasets. Hence, on the test set the *SVMreg* performs best followed by the *M5Rules*-system. On the third place the *SeCoReg*-learner that was tuned on the first folds of the tuning datasets appears. It was only slightly worse in performance compared to the *M5Rules*-learner. The next best algorithm is the second *SeCoReg*-learner which has achieved a marginal better *rrmse* than the *linear regression*. As in the previous experiments the *multilayer perceptron* was the worst algorithm among all of the different learners.

Thus, on the test sets the tuned *SeCoReg*-algorithm achieved better results than in the previous experiment. Here, the best configuration of the algorithm is ranked on the third place among the six algorithms. Note that the dataset *meta* shows huge standard deviations for some algorithms (*M5Rules*, *linear regression* and *MLP*). We attribute this to the separation of the data in the 10 folds of the cross validation.

But the results displayed in Table 6.3 are only averages. Therefore, a Friedman-Test was employed like in the previous Section (cf. Section 6.2). Contrary to the prior results, the Friedman-Test was not rejected at a p-value of 0.05 (the critical F-value was 2.196 but to reject the test it had to be bigger than 2.492). It would have be rejected at a p-level of 0.1 (where it was higher than 2.02 which was requested), but this was not significant enough to include these results in the paper. For this reason the Nemenyi-Test could also not be done on the test sets. Practically, this means that the *SeCoReg* algorithm does not differ significantly from the 4 weka algorithms at a significance level of 0.05. Thus, these algorithms cannot be distinguished in terms of performance.

Altogether, both tuned variants of the presented algorithm are not able to beat state-of-the-art systems. They are rather situated in between these algorithms (this holds at least for the test sets). Especially on these 9 sets it became clear that the *SeCoReg* rule learners are able to achieve a performance comparable to the results of the 4 *weka* algorithms. Due to the rather simple design of the *SeCoReg*-algorithm these results seem to be promising. In spite of the quality of the algorithm in terms of error it clearly lacks behind in terms of theory size and runtime. A more detailed analyzation of the model size is given in the next section (cf. Section 6.4).

6.4 Comparison of the size of the theories

In this Section an overview of the size of the learned theories is given. Note that the comparison of the *SeCoReg*-algorithm and the *M5Rules*-system is not fair in advance. This is because the first algorithm predicts a single value in the head of each rule whereas the latter predicts a linear model. The objective of the partitioning of the input space with each rule follows different paths. The first algorithm tries to find regions in the input space that share more or less identical target values. The latter one searches the data for regions in which a linear model could be fit best. For this reason the latter algorithm should achieve a similar performance with smaller rules and also less of them at all. To incorporate the size of

Table 6.3: Results in terms of L_{RRMSE} for different algorithms on the test set

dataset	SVMreg	M5Rules	Linear Regression	MLP	SeCoReg tuned on folds 1	SeCoReg tuned on folds 2
auto-horse	0.32± 0.08	0.37± 0.14	0.32± 0.11	0.34± 0.10	0.52± 0.18	0.61± 0.11
auto93	0.66± 0.12	0.58± 0.19	0.67± 0.20	0.57± 0.19	0.65± 0.17	0.85± 0.29
cloud	0.39± 0.12	0.42± 0.16	0.40± 0.13	0.62± 0.33	0.61± 0.19	0.67± 0.15
delta-elevators	0.61± 0.01	0.60± 0.01	0.61± 0.01	0.63± 0.01	0.78± 0.03	0.77± 0.03
meta	0.92± 0.08	1.86± 1.58	2.33± 1.72	1.40± 0.90	1.00± 0.02	1.01± 0.03
r_wpbc	1.03± 0.16	1.14± 0.19	1.04± 0.13	2.20± 0.56	1.35± 0.20	1.27± 0.18
stock	0.37± 0.05	0.14± 0.03	0.36± 0.04	0.20± 0.04	0.25± 0.03	0.26± 0.04
veteran	0.93± 0.15	1.23± 0.61	1.07± 0.36	3.01± 1.78	1.09± 0.22	1.21± 0.33
winequality-red	0.82± 0.03	0.81± 0.03	0.81± 0.03	0.95± 0.08	0.98± 0.09	0.95± 0.04
averages	0.6739	0.7942	0.8456	1.1017	0.8040	0.8438

Table 6.4: Number of rules and conditions on the test sets

	M5Rules		SeCoReg from fold 1		SeCoReg from fold 2	
dataset	# rules	# conditions	# rules	# conditions	# rules	# conditions
auto-horse	1	0	19	37	9	18
auto93	1	0	11	27	12	19
cloud	4	3	5	9	6	13
delta-elevators	4	5	356	3254	426	4705
meta	6	9	30	48	20	37
r_wpbc	3	2	36	193	8	32
stock	19	59	81	447	54	209
veteran	1	0	13	41	28	89
winequality-red	1	0	192	783	243	713
averages	4.44	8.67	82.56	537.67	89.56	648.33

the linear model in the comparison is infeasible in a straight-forward way. Hence, the size of the linear models is ignored and the comparison centers only on the rules themselves.

Table 6.4 shows the number of rules and conditions for the two *SeCoReg*-algorithms and the *M5Rules*-system. As expected the *M5Rules*-system always has a much lower number of rules and conditions. On 4 out of 9 datasets this algorithm has not found any rule but just uses a linear model to fit the data. For all other datasets around 5 rules seem to be sufficient to divide the data in regions where the linear models could be fit best with the only exception of the dataset *stock* where 19 rules with 59 conditions are needed. But still, this algorithm needs a significant lower number of rules than the *SeCoReg*-algorithm. As mentioned above the comparison is not fair due to the different objectives of the algorithms and their different expressiveness of the single rules.

When comparing the two variants of the *SeCoReg*-algorithm it becomes obvious that the one that was tuned on the second folds needs less rules in 4 of 9 cases. It is superior in terms of number of conditions in 6 of 9 cases. Hence, it seems that using more splitpoints results in approximately the same number of rules but in a lower number of conditions. Due to the wider range of possible splits for a candidate rule when using more splitpoints this makes sense. But still the version tuned on the first folds of the tuning datasets is superior in terms of error. Thus, on the one hand it seems that using more conditions in each rule results in a better performance. But on the other hand, for some datasets, it turned out that the quality of each rule is also of great importance. Therefore, more rules are better for the datasets *auto-horse*, *meta*, and *stock* whereas the sole quality of the rules is crucial for the datasets *cloud* and *veteran* where smaller theories yield better performance.

6.5 Comparison with *FRS* and *PCR*

Table 6.5 gives an overview of the results taken from the PhD-Thesis [24] for *FRS*, *PCR-ordered*, *PCR-unordered* and the two configurations of the *SeCoReg* rule learner. The two datasets from the test set are depicted in italics. All others were taken from the tuning sets. Hence, the errors are given by using the model tuned on the first folds on the second folds and vice versa. The Table gives an overview of the different algorithms in terms of *rrmse* (the lowest error is depicted in

Table 6.5: Comparison with *FRS* and *PCR*

	FRS-ordered		PCR-ordered		PCR-unorderd		SeCoReg from fold 1		SeCoReg from fold 2	
dataset	RRMSE	# rules	RRMSE	# rules	RRMSE	# rules	RRMSE	# rules	RRMSE	# rules
auto-price	0.91± 0.01	19	**0.52**± 0.13	3	**0.52**± 0.15	3	0.71± 0.28	7	**0.52**± 0.17	7
cloud	0.94±0.03	1	0.72± 0.24	3	0.81± 0.25	3	**0.61**± 0.19	5	0.67± 0.15	6
cpu	0.88± 0.01	18	0.52± 0.29	3	0.65± 0.42	3	0.98± 0.56	4	**0.48**± 0.15	2
housing	0.80± 0.00	126	0.67± 0.12	7	0.67± 0.11	7	**0.58**± 0.18	30	0.66± 0.15	20
quake	**1.00**± 0.00	4	1.16± 0.05	129	1.02± 0.04	8	1.16± 0.04	15	1.13± 0.05	29
sensory	**1.00**± 0.00	1	1.07± 0.10	34	**1.00**± 0.10	10	1.06± 0.06	15	1.04± 0.05	0
servo	1.01± 0.02	1	0.61± 0.32	5	**0.49**± 0.20	3	0.77± 0.20	7	0.75± 0.94	6
strike	0.99± 0.01	19	1.26± 0.48	14	1.10± 0.41	13	**0.84**± 0.16	31	0.98± 0.09	50
veteran	**1.02**± 0.04	1	1.22± 0.48	8	1.09± 0.27	5	1.09± 0.22	13	1.21± 0.33	28
averages	0.95	21.11	0.86	22.89	0.82	6.11	0.86	14.11	0.83	16.44
average ranks	3.39		3.56		2.61		3.00		2.44	

Table 6.6: Comparison with *RegENDER*

	RegENDER-10	RegENDER-100	RegENDER-rules-folds1		RegENDER-rules-folds2		SeCoReg from fold 1	SeCoReg from fold 2
dataset	RRMSE	RRMSE	RRMSE	# rules	RRMSE	# rules	RRMSE	RRMSE
auto93	1.024 ± 0.49	1.012 ± 0.49	1.018 ± 0.52	11	1.011 ± 0.5	12	0.652 ± 0.17	0.854 ± 0.29
cloud	0.751 ± 0.25	0.779 ± 0.28	0.667 ± 0.21	5	0.676 ± 0.21	6	0.608 ± 0.19	0.670 ± 0.15
delta_elevators	0.660 ± 0.01	0.654 ± 0.01	0.716 ± 0.02	356	0.726 ± 0.02	426	0.782 ± 0.03	0.772 ± 0.03
meta	4.764 ± 4.74	4.853 ± 4.32	4.728 ± 4.2	30	4.738 ± 4.15	20	1.004 ± 0.02	1.008 ± 0.03
r_wpbc	1.321 ± 0.18	1.410 ± 0.21	1.414 ± 0.21	36	1.297 ± 0.19	8	1.348 ± 0.2	1.266 ± 0.18
stock	0.295 ± 0.03	0.240 ± 0.04	0.241 ± 0.04	81	0.241 ± 0.04	54	0.249 ± 0.03	0.263 ± 0.04
veteran	1.404 ± 0.77	1.509 ± 0.95	1.496 ± 0.82	13	1.472 ± 0.95	28	1.086 ± 0.22	1.207 ± 0.33
winequality_red	0.850 ± 0.03	0.899 ± 0.03	0.903 ± 0.04	192	0.901 ± 0.05	243	0.983 ± 0.09	0.946 ± 0.04
averages	1.38	1.42	1.4	90.5	1.38	99.63	0.84	0.87
average ranks	3.88	3.88	3.75	-	3.38	-	3.0	3.13
avg. without *meta*	0.9	0.93	0.92	99.14	0.9	111.0	0.82	0.85
avg. ranks without *meta*	3.71	3.57	3.86	-	3.29	-	3.29	3.29

bold) and the number of rules. The average ranks of the algorithms are also given. A Friedman-Test was employed on these ranks, but it was not rejected which means that the algorithms are not significantly different.

As mentioned before this comparison is actually not suitable for several reasons: On the one hand the errors of *FRS* and the two alternatives of *PCR* (one using an ordered set of rules and the other employing an unordered rule set) are taken from the PhD-thesis of B. Ženko [24]. On the other hand most of the datasets were used during tuning of the parameters. Indeed, the depicted errors were never computed on parts of the data where the parameter tuning took place, but the characteristics of the datasets were changed anyhow. This also may result in smaller numbers of rules.

As can be seen in Table 6.5 *PCR-unordered* was the best algorithm closely followed by the *SeCoReg* tuned on the second folds. But the first algorithm needs about 10 rules less in average to achieve this performance. This mostly could be attributed to the different classification mechanisms (unordered list of rules vs. decision list). It seems that *PCR-ordered* and *SeCoReg* from folds 1 are similar in terms of performance. The latter has a significantly lower number of rules. The *FRS* system was inferior both in terms of error and in terms of number of rules.

6.6 Comparison with *RegENDER*

Table 6.6 shows a comparison to *RegENDER* [5]. The dataset *auto-horse* contains missing class values which cannot be handled by *RegENDER*. Therefore, this dataset was left out. In addition the results on the dataset *meta* showed strong fluctuations as mentioned before. For this reason results without this dataset are also included. *RegENDER* has a parameter to specify the number of rules in the ensemble. To make a choice the algorithm was tested with 10 and 100 rules and with the same number of rules the two *SeCoReg* variants had found on the test sets. Clearly, using more rules will result in a lower error as experiments in [5] show, but we think it is fair to run the algorithm with the same number of rules as used in the *SeCoReg*-learner to get a good impression of the performance.

The *SeCoReg*-algorithm was slightly better in average *rrmse* and the average rank was also better for the experiments including the dataset *meta* and for those where it was left out. Nevertheless, a Friedman Test was rejected ($p = 0.05$) for the experiments inlcuding *meta* but the Nemenyi Test showed that all algorithms were in the same equivalence class (the critical distance extends over all algorithms) and therefore do not differ statistically significant. For the experiments without *meta* the Friedman Test was even not rejected.

6.7 Discussion

In Section 3.3 the generation of the splitpoints was discussed. Here, a problem was mentioned that the quality of the rules directly depends on the number of splitpoints. This is because the number of splitpoints governs the possibilities when refining a candidate rule. If all splitpoints are used, there will always be a candidate rule that uses the very first or last value of the attribute and therefore covers only one example. The less splitpoints are used the higher is the probability that the candidate rules are only able to select values situated in the middle of the attribute's value range.

In general, the quality of the candidate rules depends on how many examples they have covered. Usually a rule that covers only one example will have an error of 0 (independently from the measure used) but also very bad coverage statistics. During the design of the algorithm it turned out that the error has a higher influence than the coverage (that was also confirmed by the best value of the heuristic, cf. Section 5.2). Thus, our choice to constraint a rule by preventing it from covering less than 3 examples somehow is a mean to prevent the rules from covering only single examples. The design of the heuristic also can be seen as an attempt to find rules that guarantee a higher coverage. Additionally, as mentioned above, the number of splitpoints is indirectly used to circumvent this problem. The observation that leaving rather big parts of the data uncovered results in good performance (cf. Section 5.1) contributes also to receive rules of higher quality. Thereby, the algorithm finds rules for those parts of the data where certain patterns are contained and leaves the remainder of the examples untouched. If those parts of the data should be also explained by rules, these would certainly be low coverage rules that only encode exceptions.

The abovementioned observations show that all parameters of the algorithm somehow contribute to the quality of the rules and also that they are all interwove with each other. To modify one parameter could have great impact on all other parameters. Due to the lack of knowledge of how they are interwoven the employed iterative optimization procedure seems to be the most promising attempt to get stable parameters.

To inspect the abovementioned problems in more detail, the rules were studied more precisely. At first, the 7 rather small datasets were inspected, followed by the 2 huge ones (*delta-elevators* and *winequality-red*). For the first configuration of the algorithm the ratio of low coverage rules to the total number of rules was 0.44 for the small datasets and 0.49 for the huge ones. This means that nearly half of all found rules were low coverage rules. This number is even slightly bigger for the huge datasets.

For the second configuration that was tuned on the second folds of the partitioned tuning datasets the ratio was 0.44. This is exactly the same ratio as with the first configuration. For the two huge datasets the number was also bigger. Here, 54 % of all rules were covering 3 examples. Unfortunately, we have no clues about related values for classification rules. Nevertheless there is some work about the so-called small disjuncts[2] [13]. There, the authors claim that in spite of the low quality of those rules they are nonetheless quite important for classification accuracy.

Inducing a set of rules that does not contains any low coverage rule is practically impossible. There are always regions in the data that do not follow certain patterns. Perhaps those parts that encode randomness are more frequent in regression databases than in classification datasets. However, the percentages of the small coverage rules are still acceptable. For this reason we believe that the configuration for the *SeCoReg* rule learner found in this paper yields interpretable theories that are comparable to state-of-the-art systems even if the introduced algorithm is rather simple.

[2] Small Disjuncts are defined as rules that cover only one positive and no negative examples.

7 Conclusion and further work

In this paper a new rule learning algorithm for the task of regression was presented. It is based on the separate-and-conquer strategy and uses a decision-list for classification of unseen examples. It was shown that the algorithm performs comparable to different state-of-the-art algorithms implemented in *weka* and *RegENDER*, a rather new algorithm. Nevertheless, the new rule learner could not compete in terms of theory size. But here, the comparison is not fair due to the more expressive language of the rules induced by *M5Rules*. For huge databases the rule sets lack interpretability but they are of acceptable size for small to medium sized datasets.

A new splitpoint generation method was introduced based on a supervised clustering approach. This method proved to support the quality of candidate rules and even results in lower runtime compared to naive methods like the generation of equidistant splitpoints. Nevertheless, the number of generated candidate rules directly depends on the number of splitpoints. But as shown in the experiments at least for one configuration of the algorithm a number of 3 splitpoints per numerical attribute was enough.

At last, a novel rule learning heuristic was introduced that clearly improves the algorithms performance due to its flexibility in weighting the error of a rule with its coverage. An optimal setting for this regression rule heuristic was presented and it proved to be stable since the parameter values are nearly the same. An interesting observation is that, as known from classification, in regression the rules consistency also should be preferred over its coverage.

As future work the runtime of the algorithm still needs a lot of improvement. For huge databases it clearly lacks behind state-of-the-art algorithms. One way to do this could be to introduce error bounds that make a so-called forward pruning possible. This method makes it possible to stop the refinement of a rule when it is clear that it could not beat the virtual best refinement. In classification this could be easily computed by assuming that the best virtual rule covers the same number of positive examples but no negative example. In regression however it is not clear how to define this best possible refinement.

Another promising path to optimize the algorithm would be to adapt the advantages of algorithms like *M5Rules* which predicts linear models in the head of each rule. On the one hand the performance of the algorithm should be drastically improve when using linear models instead of single target values. On the other hand, much of the interpretability of the rule set would be lost when doing so.

The last simple improvements would be to use some kind of pruning functionality to keep the theory size small. Strategies like iterative reduced error pruning (cf. [11]) that is used in the famous Ripper-algorithm [4] seem to be promising possibilities to do so. The method for classification could also be altered by using an unordered mode of the algorithm where rules are learned for all classes. For classification all rules that cover the example are used. Hereby, one of the main challenges is the mechanism to solve conflicts in the predictions.

Bibliography

[1] A. Asuncion and D. Newman. UCI machine learning repository, 2007. 10

[2] 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. 4

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

[4] W. W. Cohen. Fast effective rule induction. In *ICML*, pages 115–123, 1995. 3, 5, 22

[5] K. Dembczyński, W. Kotłowski, and R. Słowiński. Solving regression by learning an ensemble of decision rules. In *ICAISC '08*, pages 533–544, Berlin, Heidelberg, 2008. Springer-Verlag. 4, 20

[6] J. Demsar. Statistical comparisons of classifiers over multiple datasets. *Machine Learning Research*, 7:1–30, 2006. 11

[7] D. Demšar. Obravnavanje numericnih problemov z induktivnim logicnim programiranjem. Master's thesis, Faculty of Computer and Information Science, University of Ljubljana, Slovenia, 1999. In Slovene. 4, 17

[8] J. H. Friedman and B. E. Popescu. Predictive learning via rule ensembles. *Annals Of Applied Statistics*, 2:916, 2008. 4

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

[10] J. Fürnkranz and P. Flach. ROC 'n' rule learning – Towards a better understanding of covering algorithms. *Machine Learning*, 58(1):39–77, 2005. 6, 8

[11] 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. 22

[12] G. Holmes, M. Hall, and E. Frank. Generating rule sets from model trees. In *Twelfth Australian Joint Conference on Artificial Intelligence*, pages 1–12. Springer, 1999. 4

[13] R. Holte, L. Acker, and B. Porter. Concept learning and the problem of small disjuncts. In *IJCAI-89*, pages 813–818, IJCAI-89-address, 1989. Morgan Kaufmann. 21

[14] F. Janssen and J. Fürnkranz. An empirical investigation of the trade-off between consistency and coverage in rule learning heuristics. In T. Horvath, J.-F. Boulicaut, and M. Berthold, editors, *Proceedings of the 11th International Conference on Discovery Science (DS-08)*, pages 40–51, Budapest, Hungary, 2008. Springer Verlag. 6, 8, 12, 13, 14

[15] 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. 6

[16] A. Karalič and I. Bratko. First order regression. *Machine Learning*, 26(2-3):147–176, 1997. 4

[17] N. Lavrač, B. Kavšek, P. Flach, and L. Todorovski. Subgroup discovery with CN2-SD. *Journal of Machine Learning Research*, 5:153–188, 2004. 4

[18] 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. 5

[19] S. K. Murthy and S. Salzberg. Lookahead and pathology in decision tree induction. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1025–1031, Montreal, Canada, 1995. Morgan Kaufmann. 6

[20] J. R. Quinlan and R. M. Cameron-Jones. Oversearching and layered search in empirical learning. In *IJCAI*, pages 1019–1024, 1995. 6

[21] R. J. Quinlan. Learning with continuous classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, Singapore, 1992. World Scientific. 4

[22] L. Torgo. Data fitting with rule-based regression. In *In Proceedings of the 2nd international workshop on Artificial Intelligence Techniques (AIT'95*, 1995. 4

[23] L. Torgo and J. Gama. Regression by classification. In *In Proceedings of SBIAŠ96, Borges*, pages 51–60. Springer-Verlag, 1996. 4

[24] B. Ženko. *Learning Predictive Clustering Rules*. Phd-thesis, University of Lubljana, Slovenia, 2007. 16, 17, 19, 20

[25] B. Ženko, S. Džeroski, and J. Struyf. Learning predictive clustering rules. In *In 4th IntŠl Workshop on Knowledge Discovery in Inductive Databases: Revised Selected and Invited Papers, volume 3933 of LNCS*, pages 234–250. Springer, 2005. 4, 11, 17

[26] Y. Wang and I. H. Witten. Induction of model trees for predicting continuous classes. In *Poster papers of the 9th European Conference on Machine Learning*. Springer, 1997. 4

[27] S. M. Weiss and N. Indurkhya. Rule-based machine learning methods for functional prediction. *Journal of Artificial Intelligence Research*, 3:383–403, 1995. 4

[28] I. H. Witten and E. Frank. *Data Mining — Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann Publishers, 2nd edition, 2005. 12, 17