Iterative Optimization of Rule Sets

Iterative Optimierung von Regel-Mengen

Master's Thesis von Jiawei Du

November 2010

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Knowledge Engineering Group

Iterative Optimization of Rule Sets
Iterative Optimierung von Regel-Mengen

vorgelegte Master's Thesis von Jiawei Du

Gutachten: Prof. Dr. Johannes Fürnkranz
Betreuer: Frederik Janssen

Tag der Einreichung:

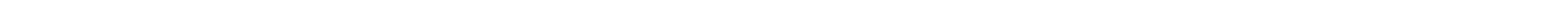
Erklärung zur Master's Thesis

Hiermit versichere ich die vorliegende Master's Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 16. November 2010

____

(Jiawei Du)



Danksagung

Ich möchte mich bei Herrn Prof. Dr. Johannes Fürnkranz für die Vergabe dieser interessanten Mastetarbeit sowie für seine Vorschläge bedanken, und mein Dank gilt Herrn Dipl.-Inf. Frederik Janssen der mir bei Fragen und Probleme geholfen haben. Ein besonderer Dank gilt meinen Eltern die mir durch ihr Vertrauen und ihre finanzielle Unterstützung mein Studium erst ermöglicht haben. Ich danke weiterhin meiner Frau Gujie Zou für ihre Liebe, ihre Geduld und ihre seelische Unterstüzung während meines Studiums und der Masterarbeit.

Abstract

Rule learning is one of the subfields in machine learning that is specialized in the generation of rules from the data. A rule set, which consists of a series of rules, can be seen as the "experience" that enables the system to do the same task more efficiently. Normally, the process of learning rule sets is called the building phase. It is suggested that a single rule in the building phase is optimized in most rule learning algorithms, while the improvement can also be gained for the entire rule set in a postprocessing phase. As a well-known algorithm, including the postprocessing phase, RIPPER is suitable for the benchmark. Moreover, two variations are also derived from the original RIPPER algorithm for comparison. The first one introduces a new pruning method and the second does a simplified selection criterion. At the end, all the algorithms mentioned in this thesis are implemented and validated in the simulation platform SeCo. In different parameter settings, some conclusions are made to optimize the rule set based on the simulation results.

Zusammenfassung

Regel-Lernen ist ein wesentlicher Bereich des Maschinellen Lernens. Unter dem Regel-Lernen versteht man die Extraktion der relevanten Regeln aus gegebenen Datenmengen. Mit der Regelmenge, die aus einer Reihe von Regeln besteht, können Maschinen die gleiche Aufgabe effizienter erledigen. Allgemein bezeichnet man den Lernprozess der Regelmengen als die Lernphase. Um eine gute Regelmenge zu erhalten, optimieren die meisten Algorithmen jede einzelne Regel in der Lernphase, während einige Algorithmen eine Optimierung der gesamten Regelmenge in einer Nacharbeitungsphase durchführen. In dieser Masterarbeit stellen wir eine Analyse für dem bekannten Algorithmus RIPPER (enthält beide Phasen) vor. Darüber hinaus wurden zwei Varianten basierend auf dem RIPPER Algorithmus entwickelt. Die erste führt eine neue Pruning-Methode ein und die zweite vereinfacht das originale Auswahlkriterium. Anschließend wurden der RIPPER Algorithmus und seine Varianten in der Simulationsplattform SeCo implementiert. Durch einen Vergleich der Algorithmen mit verschiedenen Parametern wurden eventuelle Vorteile und Nachteile untersucht.

Contents

1. Introduction 1
   1.1. Motivation . . . 1
   1.2. Outline of Thesis . . . 2

2. Background 3
   2.1. Learning Sets of Rules . . . 3
      2.1.1. Decision Tree Learning . . . 6
      2.1.2. Separate-and-Conquer . . . 7
   2.2. Preprocessing . . . 10
   2.3. Postprocessing . . . 11

3. SeCo Platform and the RIPPER Algorithm 14
   3.1. Introduction to SeCo . . . 14
      3.1.1. Framework of SeCo . . . 14
      3.1.2. Rules for One Class . . . 15
      3.1.3. Interfaces in SeCo . . . 15
      3.1.4. Default Components . . . 16
      3.1.5. XML Parser . . . 17
      3.1.6. Data Set Format . . . 18
   3.2. Algorithms . . . 18
      3.2.1. REP-based Algorithms . . . 18
      3.2.2. RIPPER Algorithm . . . 22
      3.2.3. Variant Abridgment . . . 25
      3.2.4. Simplified Selection Criterion . . . 27

4. Implementation in SeCo 28
   4.1. XML File . . . 28
   4.2. Generate an Initial Rule Set . . . 30
      4.2.1. Stratified Data . . . 31
      4.2.2. Growing Set and Pruning Set . . . 33
      4.2.3. Growing a Rule . . . 33
      4.2.4. Pruning a Rule . . . 36
      4.2.5. Condition Stop Criterion and Rule Stop Criterion . . . 38
   4.3. Iterative Optimization . . . 39
      4.3.1. Variants . . . 41
      4.3.2. Selection Criteria for Variants . . . 44
      4.3.3. Rule Reduction . . . 45
   4.4. Differences from JRIP . . . 46
      4.4.1. Order of Classes . . . 46
      4.4.2. Selection of Refinements . . . 47
      4.4.3. Minimal Number of Covered Examples . . . 47

5. Evaluation 48
   5.1. Data Sets . . . 48

5.2. Evaluation Methods . . . 48
5.3. Evaluation Dimensions . . . 50
5.3.1. Correctness . . . 50
5.3.2. Size of Rule Sets . . . 50
5.3.3. Number of Conditions in One Rule . . . 50
5.4. Performance Evaluation . . . 51
5.4.1. Results of SeCoRIP . . . 51
5.4.2. Comparison with JRIP . . . 53
5.4.3. Results of Variant Abridgment . . . 54
5.4.4. Results of Simplified Selection Criterion . . . 56
5.4.5. Convergence Properties of SeCoRIP . . . 58

6. Summary and Conclusions 62

Bibliography 64

A. Table 66

List of Figures

2.1. Confusion Matrix . . . 4
2.2. Learning Sets of Rules . . . 5
2.3. Decision Tree . . . 6
2.4. Decision Tree Learning versus Separate-and-Conquer . . . 8
2.5. Preprocessing and Postprocessing . . . 11

3.1. Package Hierarchy in SeCo . . . 14
3.2. Rules for One Class . . . 16
3.3. Structure of REP, I-REP and RIPPERk . . . 22
3.4. RIPPER / RIPPERk . . . 23
3.5. Pruning Methods . . . 26

5.1. SeCoRIP of 20 UCI Data Sets . . . 51
5.2. Comparison with JRIP . . . 53
5.3. Comparison of Average Correctness . . . 56
5.4. Comparison of Average Correctness 2 . . . 57
5.5. A Picture of the Definition of Convergence [10] . . . 58
5.6. Group A . . . 59
5.7. Group B . . . 60
5.8. Group C . . . 60
5.9. Group D . . . 61

List of Tables

2.1. A Simple Data Set . . . 3
2.2. Examples with Missing Attribute Values . . . 10
2.3. Intervals and Frequencies . . . 11
2.4. Covered Examples . . . 13
2.5. Discretization of Numeric Attributes . . . 13

3.1. Elements in SeCo XML Description [30] . . . 17
3.2. Structure of an ARFF File . . . 18
3.3. List of REP-based Algorithms . . . 19
3.4. Variants and Old Rule . . . 24

4.1. Original Training Set . . . 32
4.2. Randomized Bags . . . 32
4.3. Stratified Training Set . . . 33
4.4. weather.arff . . . 35
4.5. Searching for Critical Point of Numeric Attribute . . . 35
4.6. Variant Abridgment (1st Iteration) . . . 43
4.7. Variant Abridgment (2nd Iteration) . . . 43
4.8. Variant Abridgment (3rd Iteration) . . . 43

5.1. Legend of Properties in Data Sets . . . 48
5.2. 20 UCI Data Sets . . . 49
5.3. Win-Tie-Loss . . . 52
5.4. Profit . . . 52
5.5. The Average Number of Rules and Conditions (SeCoRIP) . . . 53
5.6. 9 Data Sets Marked in Gray . . . 55
5.7. Win-Tie-Loss 2 . . . 56
5.8. Win-Tie-Loss (Comparison of SeCoRIP and SeCoRIP') . . . 57
5.9. The Average Number of Rules and Conditions (SeCoRIP') . . . 58

A.1. SeCoRIP on 20 UCI Data Sets . . . 66
A.2. JRIP on 20 UCI Data Sets . . . 67
A.3. The Number of Rules and Conditions (SeCoRIP) . . . 68
A.4. Results of 1. Variant (*Abridgment*) . . . 69
A.5. Results of 2. Variant (*Accuracy*) . . . 70
A.6. The Number of Rules and Conditions (SeCoRIP') . . . 71
A.7. SeCoRIP on 20 UCI Data Sets (Part 2) . . . 72

List of Algorithms

2.1. A Generic Separate-and-Conquer Rule Learning Algorithm 9

4.1. XML for SeCoRIP_2 28
4.3. FindBestRule 34
4.4. PruneRule 38
4.5. PostProcessTheory 40
4.6. PruneOldRule 42
4.7. ReduceDL 46
4.8. Header Information of monk1.arff 46

1 Introduction

1.1 Motivation

With the development of computer science, the ability to collect and store data has greatly improved the accumulating of mass of data in the field of scientific research and daily life. It is necessary to analyze this data and turn it into valuable information and knowledge. These complicated tasks can be accomplished by machines. Machine learning is an important scientific discipline that is concerned with the design and development of the techniques for analyzing and inducing empirical knowledge from data. Empirical knowledge is what is collected through observation or experiments. By using that, machines are able to process the new data themselves. Due to their learning ability, these machines can be considered as "intelligent".

In machine learning, rule learning is a popular and well-researched method for extracting rules from the data. The extracted rules form a rule set, which is a kind of empirical knowledge that is suitable for the machines. It is to be mentioned that the data in rule learning systems is usually composed of a series of examples, which contain some attributes with values. The rule set can be used to determine which groups the examples belong to. This process is called "classification". A classifier builds a model that may be described by rules. This rule set is able to classify new, previously unseen examples. There are many different heuristics to evaluate the quality of rules. As a common heuristic, "accuracy" means how many examples in the data are correctly classified. In general, the more the examples are classified correctly, the better the rule is. In order to get a good rule set, most of the rule learning algorithms focus on how to extract a single rule from the data more effectively. In other words, they guarantee the quality of the rule set by adding a series of high-accurate rules. However, the learned rule set in the previous steps can be further processed as well [5]. In rule learning systems, the process of learning a rule set is called the building phase while the optimization of the learned rule set is called the postprocessing phase.

In addition, the real-world data that has to be processed are sometimes noisy and often inconsistent [2]. The term "noisy" means that the values of attributes in some examples are incorrect or even missing, and "inconsistent" means that some examples may belong to different groups, although they have the same values in their attributes. Due to the defective data, it is difficult to ensure that the learned rules are always of high accuracy. Thus, the quality of the rule set will be affected. In order to solve this problem, many different approaches are presented, such as pre-pruning and post-pruning of the learned rules, filtering the attributes, deleting the examples, etc. These approaches are usually applied in the building phase. With the help of them, the quality of the rule set will not suffer too much. However, optimization of the entire rule set can further solve this problem. In the postprocessing phase, the initial rule set was learned. Based on the existing rule set, it is easier and more effective to optimize the rules. Moreover, there is a chance that the rules derived from the noisy data can be found and corrected.

In this thesis, the methods for optimizing the rule sets are researched and analyzed. RIPPER [8] is a rule learning algorithm that includes a postprocessing phase. It was presented by William W. Cohen in 1995. The earliest prototype of RIPPER can be traced back to REP (Reduced Error Pruning). REP is a rule learning algorithm that takes a simplex approach (post-pruning) to optimize the rule set [26]. Normally, the rules in the rule set are so redundant that much time is needed to optimize them. Thus, the efficiency of REP is not satisfactory. The I-REP (Incremental Reduced Error Pruning) algorithm was developed by Johannes Fürnkranz and Gerhard Widmer in 1994. This algorithm changes the structure of

REP so that it can generate rules and form a rule set more easily and quickly. Moreover, the quality of rule sets generated by I-REP is better than REP [16]. It is to be mentioned that the I-REP algorithm does not employ a postprocessing phase. RIPPER was developed based on I-REP. It introduces the concept of postprocessing again to improve the rule set. In addition, this algorithm has a variant which is called RIPPER*k*. In the postprocessing phase in RIPPER*k*, the rule set learned in the building phase can be optimized iteratively. This means that the optimized rule set can be further processed. The parameter k refers to the number of optimization iterations. For example, RIPPER*2* means the rule set will be optimized twice. It is confirmed in [8] that the RIPPER algorithm is very competitive with other rule learning algorithms. Up to the present time, no method to improve RIPPER has been successfully developed. For this reason, RIPPER is set as a benchmark system in this thesis. Based on this system, the comparison of the methods for the optimizing of rule sets will be made.

1.2 Outline of Thesis

The thesis is structured as follows: Chapter 2 introduces the concept of rule learning and presents two common approaches for getting rules. The first approach, "decision tree learning", is an indirect method that cannot extract rules from the data directly. The rules are converted from a decision tree, which was constructed on the data in advance [26]. The second approach, "separate-and-conquer [22]", is a method that can directly extract rules from the data.

In Chapter 3, the framework of SeCo is introduced. SeCo is a simulation platform on which different rule learning algorithms can be simulated and evaluated. It is important to note that these algorithms should be implemented according to the separate-and-conquer approach. Secondly, the REP-based algorithms are introduced and the differences among them are compared. REP-based algorithms indicate the rule learning algorithms that are developed based on the original REP algorithm. Finally, two variants are presented based on the RIPPER algorithm. The first one introduces a new pruning method and the second does a simplified selection criterion.

Both the RIPPER algorithm and the variants mentioned above are implemented in SeCo. The implementation of the original RIPPER is called SeCoRIP and is explained in Chapter 4. In addition, as another implementation of the RIPPER algorithm in Weka [33], JRIP is compared with SeCoRIP.

In Chapter 5, the methods for evaluating SeCoRIP and the variants are explained. In addition, the whole evaluation process can be divided into three parts: Firstly, a conclusion is made by comparing the results of SeCoRIP with JRIP. Secondly, conclusions are made by comparing the results of the two variants with SeCoRIP. Finally, the convergence properties of SeCoRIP for the increasing of the number of optimization iterations is investigated.

In Chapter 6, the thesis is concluded with a summary of our findings.

2 Background

2.1 Learning Sets of Rules

The definition of learning is described in [21]: *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.* In practice, it can be considered as a process of extracting regularities from the given data to improve the performance.

However, the real-world data are often so difficult to process directly that they are usually converted into the machine-readable data in advance. In rule learning, the data are described as data sets, which contain a series of examples (also called instances). They are composed of attributes with values. A training set is part of the data set and is used to learn the rules. It should be noted that the classes of the examples (i.e. class values) in the training set are known. The term *class* indicates here which group the examples belong to. A simple training set including 9 examples is given in the following table:

Nr.	Outlook	Temperature	Humidity	Windy	Play Golf
1.	sunny	85	85	false	no
2.	rainy	85	85	true	no
3.	sunny	80	90	true	no
4.	overcast	83	86	false	yes
5.	rainy	70	96	false	yes
6.	rainy	68	80	false	yes
7.	overcast	64	65	true	yes
8.	sunny	72	95	false	no
9.	sunny	69	70	false	yes

Table 2.1.: A Simple Data Set

In table 2.1, {*Outlook, Temperature, Humidity, Windy*} are four attributes. It is noticeable that there are two kinds of attributes in this table; {*Outlook, Windy*} are nominal attributes and {*Temperature, Humidity*} are numeric attributes.

- **Nominal:** The values of nominal attributes are composed of several certain states, e.g. *Outlook = {sunny, rainy, overcast}*, *Windy = {true, false}*. The characteristics of them are discrete and not comparable. Moreover, the nominal values are case sensitive (i.e. $sunny \neq Sunny$).

- **Numeric:** The values of numeric attributes are expressed by numbers. The characteristics of them are continuous and comparable. The numeric attributes are usually used to describe the magnitude of certain items such as *Temperature, Humidity*, etc.

Each attribute in the examples has its individual value, so that the examples can be distinguished from each other. In addition, {*Play Golf*} is the class name and it has two class values, "yes" and "no". The class value of each example is dependent on the change in four attributes. This means, in this case, the process of rule learning can be interpreted as follows: "In which case the person would play golf". In rule learning, rules can be extracted from the data. Normally, a rule consists of two parts: the rule header, which encodes the target class (i.e. certain class name), and the rule body, which consists of attribute-value

pairs or conditions (i.e. attributes with possible values). According to table 2.1, an example of a concrete rule is given in the following equation:

$$\begin{array}{llll} \textbf{IF} & <-- \quad \textit{conditions} \quad --> & \textbf{THEN} & <\textit{target class}> \\ \textbf{IF} & < Outlook = rainy \ \& \ Temperature \leq 80 > & \textbf{THEN} & < yes > \end{array} \tag{2.1}$$

Moreover, it is possible to construct an empty rule that has no condition in its rule body. With the extracted rules, it is much easier to understand the training set. Furthermore, these rules will be used to further classify the new data, in which the class values of examples are unknown. For example, according to equation 2.1, all the examples that meet the conditions [Outlook = rainy & Temperature $\leq$ 70] will be classified as "yes".

The heuristic is an important component that helps in rule discovering, extracting and evaluating. In rule learning, there are two kinds of heuristics available, i.e. *evaluation heuristic* and *search heuristic*. An evaluation heuristic is a heuristic that is usually used to evaluate candidate rules. The term *candidate rules* indicate the rules that have a chance to be further processed. Furthermore, a search heuristic is a heuristic that is to guide the search algorithms in the right regions of the search space [15]. The search algorithms will be described in section 2.1.2. For this purpose, the essential information about the rules based on the training set is gathered. This information can be described in a confusion matrix. Such a matrix contains statistics of a single rule or statistics of a complete evaluation of a rule learning algorithm.

Actual Class		**Predicted Class**		
		Yes	*No*	
	Yes	TP	FN	**P**
	No	FP	TN	**N**
		Covered	Not Covered	

Figure 2.1.: Confusion Matrix

- **TP** (true positive) means the number of correct predictions with an example being positive.

- **FP** (false positive) means the number of incorrect predictions with an example being positive.

- **TN** (true negative) means the number of correct predictions with an example being negative.

- **FN** (false negative) means the number of incorrect predictions with an example being negative.

Note that the confusion matrix is only suitable for a two-class problem. For a multi-class problem (i.e. the number of classes is more than 2), it is possible to use the one-against-all method [12] to reduce the multi-class to binary one. According to the one-against-all method, all classes that are not the target class form a new class named the non-target class. In figure 2.1, **P** means the total number of positive examples in the data set (P=TP+FN), whereas **N** means the total number of negative ones (N=FP+TN). Furthermore, **Covered** means the covered examples by the rule (Covered = TP+FP), while **Not Covered**

means the rest of the examples in the data set (Not Covered = FN + TN). In general, heuristic methods are constructed based on this information. Here are some commonly used examples:

$$Accuracy = \frac{TP + TN}{\mathbf{P+N}} \tag{2.2}$$

$$Precision = \frac{TP}{Covered} \tag{2.3}$$

$$Laplace = \frac{TP + 1}{Covered + 2} \tag{2.4}$$

$$m - Estimate = \frac{TP + m\frac{\mathbf{P}}{\mathbf{P+N}}}{TP + FP + m} \tag{2.5}$$

$$Entropy = -\frac{TP}{TP + FP}\log\frac{TP}{TP + FP} - \frac{FP}{TP + FP}\log\frac{FP}{TP + FP} \tag{2.6}$$

In different rule learning algorithms, the heuristic methods used will be vary. It is also possible that several heuristic methods are combined into one scheme. Correspondingly, the global structure of the learning process can be summarized in figure 2.2. In the next section, two representative approaches for generating rules are introduced. The first approach, "separate-and-conquer", is a direct method that extracts rules from data. With the second one, "decision tree learning", the rules cannot be extracted directly but can be converted from a decision tree indirectly.

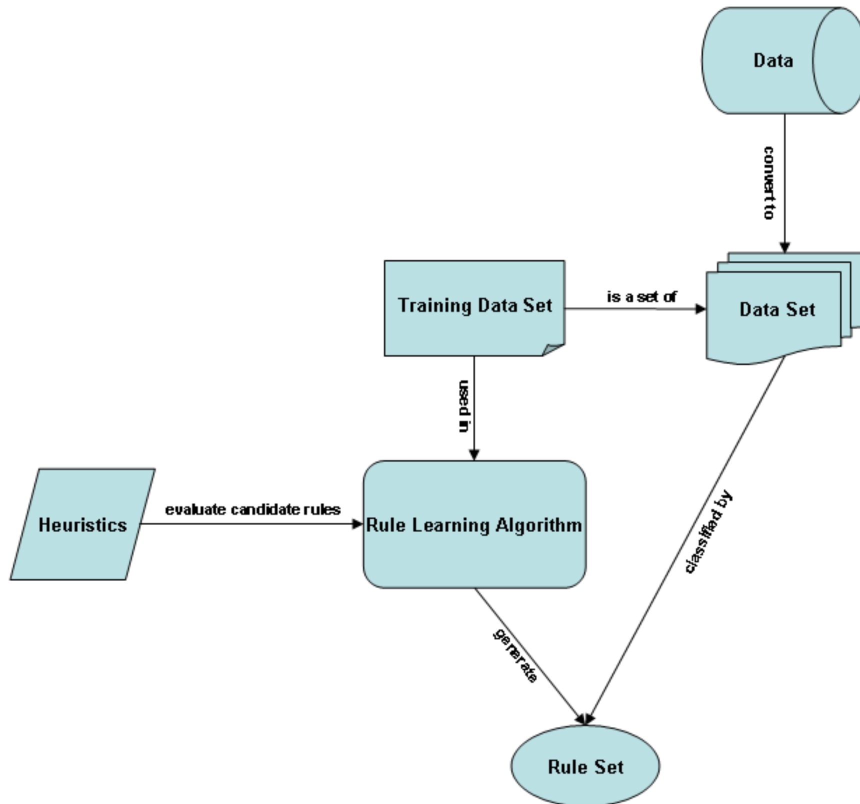


Figure 2.2.: Learning Sets of Rules

2.1.1 Decision Tree Learning

In Wikipedia, decision tree learning is defined as follows: *Using a decision tree as a predictive model which maps observations about an item to conclusions about the item's target value [34].* The earliest development of decision trees can be traced back to the research with concept learning systems [13] (CLS) by Hunt, Marin and Stone in 1966. As an essential method for classification and prediction, it is nowadays widely used in machine learning. Based on the decision tree, the learning systems can effectively deal with large data sets affected by noise and incompleteness [3, 25]. Moreover, it is possible to convert the decision tree to a series of rules which are more understandable for people. This means that the structure of the decision tree is especially important to ensure the quality of the extracted rules. Here, the process of getting a rule set is simply summarized into the following three steps:

1. Constructing a decision tree
2. Transforming it into a rule set
3. Simplifying the rules in the rule set

As mentioned before, each example in the data set contains a series of attributes. In order to construct a decision tree, some tests should be performed on these attributes. The whole process can be interpreted as successively "splitting" (divide-and-conquer strategy) the data set into subsets based on these tests. The distribution of the relevant data set is determined after each test. The original data set is inhomogeneous because the examples in it belong to different classes. The ideal subset should be homogeneous, which means the examples have the same class. In general, this "splitting" process can be stopped if all subsets are homogeneous. Figure 2.3 shows the corresponding decision tree with respect to the data set in table 2.1.

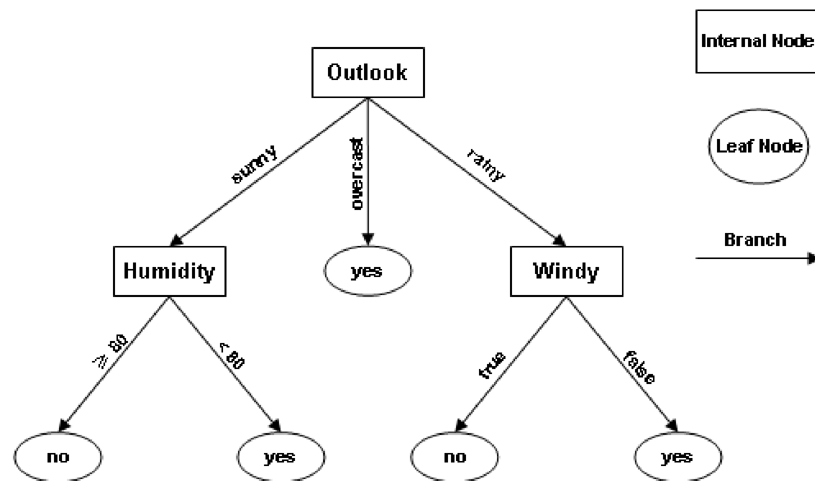


Figure 2.3.: Decision Tree

- **Internal Node** meaning a test on an attribute
- **Leaf Node** meaning the class or class distribution
- **Branch** meaning the outcome of the test

The data set at the root node {Outlook} contains the complete examples defined in table 2.1 and the subset at the internal node {Humidity} only has the examples that meet the condition [Outlook = sunny].

Because the subset with the condition [Outlook = overcast] is homogeneous, it is not necessary to split it any more.

In general, the most useful attribute for classifying examples should be selected at each internal node. The definition of "useful" depends on the heuristics used in the decision tree. Entropy and information gain calculation are two essential estimation criteria which can evaluate the quality of attributes in the decision tree. The entropy calculates the impurity of each subset related to the tested attributes. The basic rule is that the higher the entropy value is, the worse the purity of the subset is. The formula of the entropy can be found in equation 2.6 in section 2.1. Defined in the equation below, the information gain calculates the expected reduction in entropy caused by splitting the data set according to this attribute. Finally, the one with the highest information gain should be selected as the most useful attribute in this internal node.

$$Information\ Gain = Entropy(data\ set) - factor * \sum Entropy(subset) \tag{2.7}$$

The decision tree is able to classify an unknown example by traversing the tree from the root node to a leaf one that holds the result of the classification. However, large decision trees are difficult to understand, because the outcome of each node is determined by a series of relevant antecedent nodes. In order to understand the decision tree easily, we can convert it into a series of rules. The transformation is quite simple: In the decision tree, each path from the root node to the leaf node can be seen as an individual rule. For example, there are a total of five rules extractable according to the decision tree in figure 2.3. An example is given as follows:

$$R_1 : \textbf{IF} \quad < Outlook = sunny\ \&\ Humidity \geq 80 > \qquad \textbf{THEN} \quad < Play = no >$$
$$R_3 : \textbf{IF} \quad < Outlook = overcast > \qquad \textbf{THEN} \quad < Play = yes >$$

One advantage of these rules is that the order of the tested attributes in the rule is not important any more. However, extracted from the decision tree, the rules are merely the initial ones that may be subject to problems such as redundancy and repetition. Simplifying the initial rules as an essential idea is realized in most such learning systems. The basic concept applied here is to eliminate unnecessary rule antecedents in order to improve the performance of the rules. Related to the postprocessing, this process will be discussed in section 2.3.

2.1.2 Separate-and-Conquer

The concept of separate-and-conquer was presented by Ryszard S. Michalski in 1969 under the name *covering strategy* [20]. Different from decision tree learning, the separate-and-conquer strategy [22] is used instead of the divide-and-conquer strategy to process the data set (see figure 2.4). The advantage of this strategy is that rules can be extracted from the separated subsets (marked in red) of the data set directly. This means that there is no need to construct the complex decision tree.

In the separate-and-conquer strategy, the term *conquer* means the examples that are covered by the rules found previously should be removed from the data set. And the term *conquer* means the rules will continue to be searched in the rest of the examples. Algorithm 2.1 shows a generic separate-and-conquer rule learning algorithm that was presented in [15]. The given parameter *Data* means a data set which contains a series of examples. The learning process starts with an empty rule set. Normally, rules will be added to the rule set continuously until all the positive examples are covered. The procedure *FindBestRule* is responsible for learning a rule on the given data set. In the procedure *FindBestRule*, there can be many different methods for searching a rule. The search strategy and the search algorithm are two important components that can determine the method of getting a rule.

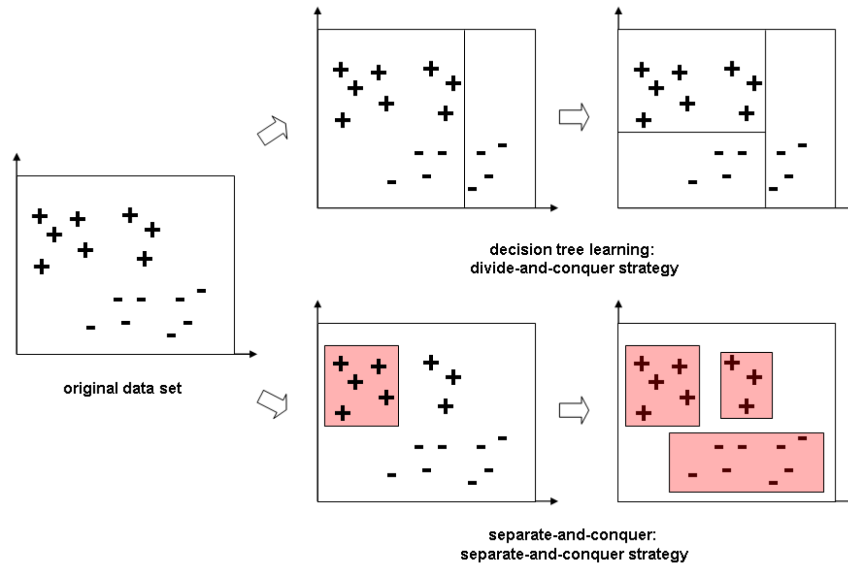


Figure 2.4.: Decision Tree Learning versus Separate-and-Conquer

- **Search Strategy**
  In the procedure *FindBestRule*, we can learn a rule in the direction of general-to-specific or specific-to-general. The terms *general* and *specific* are two basic concepts, which are usually used to describe the relations between rules. For example, if a rule R_1 covers more examples than a rule R_2 and all the examples that are covered by the rule R_2 are covered by the rule R_1 as well, we can say that the rule R_1 is more general than the rule R_2. Conversely, once can also say that the rule R_2 is more specific than the rule R_1. *Top-down* and *bottom-up* are two standard search strategies that are constructed on the above-mentioned relationship.
  - **Top-down** is a general-to-specific search. The specification starts with the most general rule (usually an empty rule) and specializes it successively until it only covers the positive examples.
  - **Bottom-up** is a specific-to-general search. The generalization starts with the most specific rule (randomly choosing a positive example from the data set) and generalizes it successively until it covers the negative examples.

  In general, the specification is processed by adding new conditions and the generalization is processed by deleting existing conditions respectively. This means that the number of conditions of a specialized rule must be larger than that of the original one. Moreover, for each original rule it is possible to generate several specialized or generalized rules. These rules are usually called the refinements of the original rule. For example, based on a given rule R, examples of the specialized rules $S_i(R)$ and the generalized rules $G_i(R)$ are given as follows:

$$
\begin{aligned}
R: &\ \mathbf{IF} < Outlook = sunny \ \& \ Windy = true > && \mathbf{THEN} < Play = no > \\
S_1(R): &\ \mathbf{IF} < Outlook = sunny \ \& \ Windy = true \ \& \ Humidity \geq 70 > && \mathbf{THEN} < Play = no > \\
S_2(R): &\ \mathbf{IF} < Outlook = sunny \ \& \ Windy = true \ \& \ Humidity < 50 > && \mathbf{THEN} < Play = no > \\
G_1(R): &\ \mathbf{IF} < Outlook = sunny > && \mathbf{THEN} < Play = no > \\
G_2(R): &\ \mathbf{IF} < Windy = true > && \mathbf{THEN} < Play = no >
\end{aligned}
$$

```
procedure SeparateAndConquer(Data)
{
    Theory = ∅
    while (Positive(Data) ≠ ∅)
    {
            Rule = FindBestRule(Data)
            Covered = Cover(Rule, Data)
        if RuleStopCriterion(Theory, Rule, Data)
            exit while
        Data = Data \ Covered
        Theory = Theory ∪ Rule
    }
    return (Theory)
}
```

Algorithm 2.1: A Generic Separate-and-Conquer Rule Learning Algorithm

- **Search Algorithm**
  The search algorithm can determine how to search for a rule. For example, in a top-down search strategy, we will start to learn a rule by successively specializing the most general rule. It is possible to generate several refinements if we separately add different conditions to the rule. The search algorithm has the right to decide which refinements could be a candidate for further specification. In rule learning systems, there are various search algorithms available, such as *hill-climbing*, *beam search* and *best-first search*.

  - **Hill-climbing** is the most commonly used search algorithm that tries to learn a rule with an optimal evaluation by continuously choosing the best refinement to be further processed and halting when no further improvement is possible [15]. This means that hill-climbing only keeps a single refinement at each step. However, it is difficult to guarantee that the chosen refinement is always really the best one.

  - **Beam search** is a search algorithm that could keep track of a fixed number (i.e. so-called beam size b) of refinements. It tries to learn a rule with an optimal evaluation by continuously choosing b best refinements to be further processed. Compared with hill-climbing, beam search can sometimes get better results, because it explores a larger search space of possible rules.

  - **Best-first search** is a search algorithm that tries to keep track of all refinements and may be seen as a beam search with an infinite beam size $b = \infty$. In the best-first search, a list is required to maintain the refinements constructed previously. Normally, best-first search continuously chooses the best refinement of the list and inserts all its refinements into the list. It is guaranteed that this search algorithm can find an optimal solution because the search space of possible rules is completely exhausted.

After learning a new rule, the examples covered by the rule will be removed from the original data set, if the rule passes the examination of the rule stop criterion. Otherwise the learned rule will be dropped and the while-loop will be stopped. In other words, the learning process is finished only when all the positive examples are covered or the rule stop criterion is met. The rule set that contains a series of rules will be returned as a result. The learned rule set is able to classify an unknown example by traversing those rules. As mentioned before, a rule is composed of two parts; the rule header encodes a target class and the rule body contains a series of conditions. In general, we can say that an example is covered by a rule if it meets all the conditions of the rule. This example will be then classified as the target class of the relevant rule.

According to the two approaches mentioned above, we can get a series of rules from a training set. A training set is a part of the data set that is derived from the real world. However, a collected training set is not directly suitable for generating rules; it usually suffers from problems such as noise, missing values, inconsistent data, and so on. Therefore, it is necessary to preprocess the training set to minimize the influence of these problems.

Nr.	Outlook	Temperature	Humidity	Windy	Play Golf
1.	sunny	85	85	false	no
2.	rainy	?	85	true	no
3.	sunny	80	90	?	no
4.	overcast	83	86	false	yes
5.	rainy	70	96	false	yes
6.	rainy	68	80	false	yes
7.	overcast	64	65	true	yes
8.	sunny	72	95	false	no
9.	sunny	69	70	false	yes

Table 2.2.: Examples with Missing Attribute Values

Table 2.2 shows a training set that contains some examples with missing attribute values. The missing value is usually described with the question mark "?". Because we don't know the values of those attributes exactly, it is difficult to use such a training set directly. In [4], some strategies are presented for resolving this problem.

- Routine *Ignore*: Ignore Missing Values
  The strategy used in this routine is quite simple; the examples with at least one missing attribute value are deleted from the training set before learning. For example, due to the missing values, the second and the third examples in the training set should be deleted. However, this routine would cause a shortage of examples when most of them contain missing values.

- Routine *Missing*: Missing Value as a Regular One
  The second strategy is only suitable for the nominal attribute. In this routine, a missing value is considered as an additional attribute value and the question mark "?" represents the missing value. In other words, the number of attribute values is increased by one for each nominal attribute that contains missing values. For example, in table 2.2, the attribute *Windy* should have three attribute values, namely *true*, *false* and "?".

- Routine *Common*: The Most Common Value
  In contrast to the two routines mentioned above, this one consults the raw data of the training set. For the nominal attribute, the frequencies of each attribute value are computed. A missing value of the nominal attribute is then substituted by the value with the maximal frequency. For example, in the attribute *Windy*, the attribute value *true* appears twice and the attribute *false* appears six times. Thus, the missing value in the third example is replaced by the attribute value *true*. For the numeric attribute, the entire numerical range is partitioned into a pre-specified number of equal-length intervals and their frequencies are computed. A missing value of the numerical attribute is then substituted by the mean value of the interval with the maximum frequency. For example, in the attribute *Temperature*, we divide the entire numerical range into five intervals and their frequencies

are listed in table 2.3. Because the second interval has the maximum frequency, the missing value in the second example is replaced by $\frac{68,4+72,8}{2} = 70,2$.

Interval	Frequency
64,0 $\sim$ 68,4	2
68,4 $\sim$ 72,8	3
72,8 $\sim$ 77,2	0
77,2 $\sim$ 81,6	1
81,6 $\sim$ 85,0	2

Table 2.3.: Intervals and Frequencies

In addition, the data of a training set are sometimes inconsistent; two or more examples belong to different classes, although they have the same attribute values. There are two general approaches to handling inconsistency in the data. The first approach is to randomly keep one of the examples and delete the others from the training set. This approach is quite simple, but it is possible that the deleted examples are more informative. Conversely, another approach is extremely costly, because the learning process will be executed more times for different training sets, in which one of the inconsistent examples remains each time.

2.3 Postprocessing

In general, the rule learning algorithm learns a rule set which tries to cover all of the positive and none of the negative examples. The learned rule set therefore works perfectly on the training set and makes no errors. However, such a rule set cannot be expected to have an absolutely high predictive accuracy on classifying unseen examples, because the learned rule set is sometimes more specific than the actual searched one. In other words, some conditions of rules or even some rules of the learned rule set are redundant. This problem is known as *overfitting*.

Moreover, in the previous section we mentioned that the training set used in the learning process is sometimes noisy. Noisy data is a problem for many rule learning algorithms, because it is difficult to distinguish between errorless and erroneous examples. Due to the noisy data, it is possible that the rule set attempts to add rules in order to cover negative examples that have erroneously been classified as positive and add conditions to rules in order to exclude positive examples that have a negative classification [15]. Thus, some rule learning algorithms employ a postprocessing procedure in which the entire rule set can be further processed after it was learned. In rule learning systems, the process of learning rule sets is called the learning phase (also called the building phase) and the process of optimizing learned rule sets is called the postprocessing phase. A simple sequence chart of the preprocessing and the postprocessing phases is presented in the following figure.

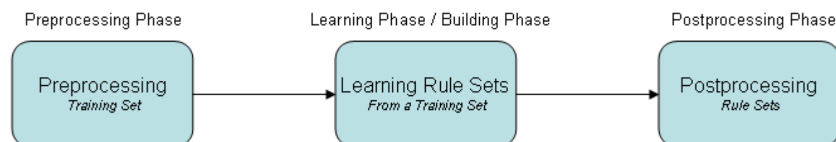


Figure 2.5.: Preprocessing and Postprocessing

As an optional component of rule learning algorithms, the postprocessing procedure consists of many various strategies and methods that can be categorized into the following groups.

- Rule Evaluation
  After a rule learning algorithm learns a rule set from the training set, the quality of the rule set is evaluated first. There are several widely used criteria for this purpose, such as the classification accuracy on unseen examples, the complexity of a rule set and its rules, the comprehensibility of a rule set, and so on. Based on these evaluation criteria, the optimized rule set in the postprocessing procedure will always be compared with the original one.

- Rule Pruning & Filtering
  The strategy used in this group aims at pruning redundant conditions from a rule and filtering unnecessary rules from the rule set. Its basic idea is to test whether the removal of a single condition or even an entire rule would lead to a decrease in the quality of the rule set.

  In rule learning algorithms, the used pruning methods usually employ various pruning heuristics. Based on the pruning heuristic, the learned rules can be pruned one by one. Each time, a newly constructed rule set that replaces the original rule with the pruned rule, is compared to the original rule set. If the quality of the new rule set is no worse than the original one, the relevant condition will be removed. In addition, if the pruned rule contains no conditions or covers no more positive examples, it is considered as an unnecessary rule and will be removed from the rule set.

  On the other hand, it is assumed that each rule will be removed from the rule set once. The newly constructed rule set is then compared with the original one. If the quality of the new rule set is no worse than the original one, the relevant rule is also considered as an unnecessary rule and will be removed. Thus, unnecessary rules can be filtered out.

- Rule Generation & Replacement
  In this group, methods are constructed to generate several new rules and substitute them for the rules of the learned rule set. Note that the search algorithms and search heuristics used here are usually different from that in the building phase. The basic idea is to test whether the replacement of a new rule could lead to an increase in the quality of the rule set. Normally, the generation of new rules can be summarized in two methods; the first one is to generate a completely new rule from the given training set while the second one is to construct a similar rule based on the original rule (e.g. change some conditions of the original rule).

- Rule Integration
  The size of a rule set is increased when more rules are added into it. Sometimes we will generate some rules that cover fewer examples so that all positive examples of a training set can be covered. Actually, these rules with a lower coverage are not suitable for a rule set, because they are usually more specific. Moreover, the size of the rule set will be huge if it contains too many these rules. This could cause the overfitting problem. Compared to the second group, the strategy used here does not filter rules but tries to integrate them with other rules. Its basic goal is to increase the coverage of rules and decrease the size of the entire rule set. In order to integrate two or more rules effectively, the attribute values of the covered examples should be analyzed first.

  For example, according to the decision tree in figure 2.3, there are a total of three rules with the

Nr.	Outlook	Temperature	Humidity	Windy	Play Golf
1.	sunny	85	85	false	no
2.	rainy	85	85	true	no
3.	sunny	80	90	true	no
4.	overcast	83	86	false	yes
5.	rainy	70	96	false	yes
6.	rainy	68	80	false	yes
7.	overcast	64	65	true	yes
8.	sunny	72	95	false	no
9.	sunny	69	70	false	yes

Table 2.4.: Covered Examples

Interval	Temperature Block
64 ∼ 67	T1
68 ∼ 71	T2
72 ∼ 75	T3
76 ∼ 79	T4
80 ∼ 83	T5
84 ∼ 87	T6

Table 2.5.: Discretization of Numeric Attributes

target class "yes" available. These rules are converted from the decision tree which is constructed on the training set in table 2.4.

$$R_1 : \textbf{IF} \quad < Outlook = sunny \ \& \ Humidity < 80 > \qquad \textbf{THEN} \quad < Play = yes >$$
$$R_2 : \textbf{IF} \quad < Outlook = rainy \ \& \ Windy = false > \qquad \textbf{THEN} \quad < Play = yes >$$
$$R_3 : \textbf{IF} \quad < Outlook = overcast > \qquad \textbf{THEN} \quad < Play = yes >$$

The rule set that contains these three rules covers all positive and no negative examples of the training set. The coverage of each rule is described as follows: The first rule covers one positive example and the other two cover two positive examples each.

Because the first rule has a lower coverage, we try to integrate this rule with the second one. In table 2.3, the covered examples of these two rules are marked in gray. By analyzing their attribute values we find that the values of the attribute *Temperature* among three examples are very close. In this case, we can use a method called "discretization of numeric attributes" to discrete the numeric attribute *Temperature*. Discretization is performed by dividing the values of a numeric attribute into a small number of intervals, where each interval is mapped to a discrete nominal symbol [18]. This method is usually used in the preprocessing phase when rule learning algorithms do not allow processing of numeric attributes. According to the result of the discretization described in table 2.5, the first two rules can be integrated into one rule R_* that covers all three examples and the size of the original rule set is reduced.

$$R_* : \textbf{IF} \quad < Temperature = T2 > \qquad \textbf{THEN} \quad < Play = yes >$$
$$R_3 : \textbf{IF} \quad < Outlook = overcast > \qquad \textbf{THEN} \quad < Play = yes >$$

3 SeCo Platform and the RIPPER Algorithm

3.1 Introduction to SeCo

SeCo is an implementation in the programming language Java that provides a platform for the simulating and testing of rule learning algorithms. Besides the simulation of an existing algorithm, it is also possible to develop a new learning algorithm within the platform. However, it needs to be mentioned that the relevant algorithm must correspond to the separate-and-conquer strategy. The characteristics of such a strategy are described in the previous chapter. The framework of the SeCo platform, in which the RIPPER algorithm is to be realized later, is the focus of this section.

3.1.1 Framework of SeCo

The outline of the framework in SeCo is oriented by [15], describing the global structure of a SeCo algorithm. Figure 3.1 shows the framework of SeCo according to the package hierarchy in Java. The package *heuristics* contains the prevalent evaluation criteria that are usually used in the rule learning algorithms, such as accuracy or correlation. The package *models* includes the essential models of rules and examples (e.g. comparator $=$, $>$, $\geq$ or the definition of rule body, rule head or example format). The package *evaluations* is in charge of the evaluation of the tested rule learning algorithms and its output format. The package *learners* is a larger component that consists of three parts, namely *components*, *core* and *factory*.

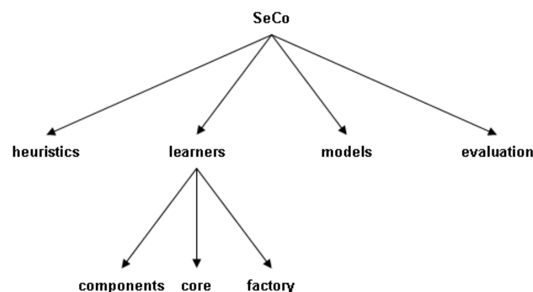


Figure 3.1.: Package Hierarchy in SeCo

Based on the separate-and-conquer strategy, the methods for learning single rules in the rule learning algorithms can be different (e.g. search heuristic, method of refining rules, rule stop criterion, etc.). Therefore, SeCo provides a series of interfaces that relate to these varieties so as to get its compatibility. These interfaces are defined in the package *core* and will be described in section 3.1.3. Moreover, SeCo provides some components that are implemented based on the predefined interfaces. These components can be found in the package *components*. On the one hand, different learning algorithms can be constructed by combining the appropriate components in an easy manner. On the other hand, one can also implement these interfaces to construct a new learning algorithm. In other words, the SeCo platform is not only configurable but also extendable. The construction of an integrated algorithm is achieved by the class SeCoFactory in the package *factory*. By parsing an XML configuration file (see section 3.1.5), the necessary information about the constructed algorithm is gathered to initialize the relevant components. Furthermore, in SeCo, the data set used in the learning algorithms is derived from an ARFF file [32], which is used in Weka

[33] as well. After initializing the learning algorithm and getting a data set, as the start point, the class AbstractSeCo in the package *core* is able to learn rules from the data set.

3.1.2 Rules for One Class

For a two-class problem (the examples in the training set have only two classes), we will usually define one of the classes as the target class and the other one as the non-target class. Then we need to learn a rule set for the target class. All the examples that are covered by the rule set will be classified as the target class and all the others as non-target class.

For a multi-class problem (the examples in the training set have many different classes), we must reduce the multi-class to binary one. Firstly, we will count the number of examples for each class and sort these classes in increasing order. In addition, the class that has the most examples is defined as the default class. Secondly, according to the sorted order of classes, we will learn a series of rule sets one by one. This means that we will start with the class that has the least examples. This class will be defined as the target class and all the other classes as the non-target class. Thirdly, a rule set for the target class will be learned and the covered examples will be removed from the training set. Then the learning process will continue with the next class in the sorted order likewise. It will be stopped if the rule sets for all the classes (except the default class) are learned. Finally, the examples that are not covered by any rule sets will be classified as the default class.

In SeCo, the learning of a rule set for one class is achieved by using the separate-and-conquer strategy. Figure 3.2 shows a global overview of SeCo's separate-and-conquer algorithm. The whole process can be divided into two phases: The building phase is responsible for learning an initial rule set and the learned rule set can be further optimized in the postprocessing phase. As mentioned before, a rule set is composed of rules. In the building phase, it is possible to generate a series of rules one by one. Each rule will be generated after two steps. Firstly, in the growing phase, a rule is usually grown from an empty rule (i.e. the rule has no condition in the rule body) and secondly, it should be pruned in the pruning phase. In addition, all the examples that are covered by the pruned rule should be removed from the training set. Then we can start to learn a new rule from the rest of the examples in the training set. This learning process will continue until the rule stop criterion (i.e. the implementation of the interface *IRuleStopCriterion*) is fulfilled and an initial rule set is constructed as a result. In the postprocessing phase, the learned rule set will be processed again to improve the performance of the rule set. Moreover, some procedures used in the building phase might be reused here. However, the concrete implementations of the interfaces that are relevant for these procedures would be different (e.g. take a different search heuristic or change the requirement for the rule stop criterion).

3.1.3 Interfaces in SeCo

In SeCo, the interfaces are used to satisfy various requirements for the separate-and-conquer-based algorithms. Figure 3.2 shows the layout, and the responsibility of each interface is explained as follows:

- **IRuleInitializer** defines the method for generating the initial rule at the beginning of the growing phase.
- **ICandidateselector** defines the method for selecting part of the candidate rules for the further processing.
- **IRuleEvaluator** evaluates the learned rule according to certain evaluation heuristics.
- **IRuleRefine** defines the method for optimizing and improving a rule.

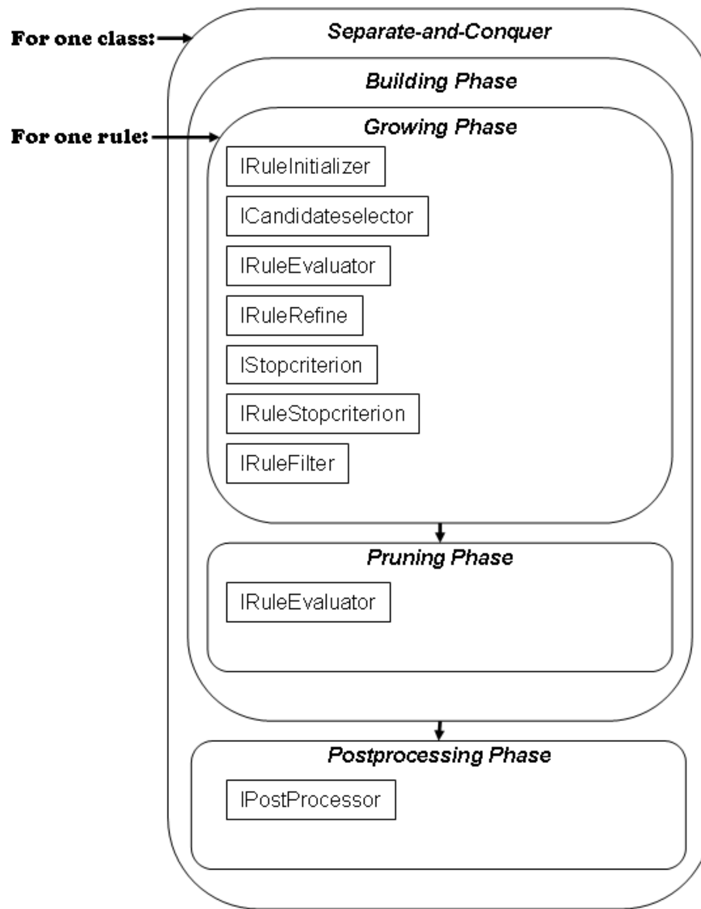


Figure 3.2.: Rules for One Class

- **IStopCriterion** defines the stop criterion for stopping the process of refining a single rule.
- **IRuleStopCriterion** defines the stop criterion for stopping the process of generating more rules.
- **IRuleFilter** defines the method for filtering the rules which are not to be further processed.
- **IPostProcessor** performs postprocessing on the learned rule set.

3.1.4 Default Components

In SeCo, if there are no extra relevant components explicitly declared in the XML file, several default components will be initialized automatically. In this section, their essential functions will be described.

- **TopDownRuleInitializer** -> IRuleInitializer
  This default rule initializer is able to select an empty rule as the initial rule; an empty rule only has the target class name in the rule header and has no condition in the rule body and covers all examples.
- **SelectAllCandidatesSelector** -> ICandidateselector
  The default candidate rule selector is able to take all the candidate rules from the rule list for further processing.

- **RuleEvaluator** -> IRuleEvaluator
  The default rule evaluator is able to evaluate a candidate rule on the given data set. Before calculating the evaluation value, the essential information will be gathered according to two classes (see the confusion matrix in figure 2.1).

- **NoNegativesCoveredStop** -> IStopCriterion
  The default condition stop criterion will stop adding conditions to a rule if the rule covers no more negative examples.

- **CoverageRuleStop** -> IRuleStopCriterion
  If a newly generated rule covers more negative than positive examples, there is no need to generate more rules, because it is impossible to construct a better one. The default rule stop criterion will stop generating more rules if the above-mentioned condition is met.

- **BeamWidthFilter** -> IRuleFilter
  The default filter is able to remove all the rules outside the "beam width". In other words, the rule set can keep a maximum number of "beam width" rules. The value of the beam width is determined by the parameter "BeamWidth". This parameter is changeable and "1" is set as its default value.

3.1.5 XML Parser

Implemented in SeCo, the components are used to realize various features of separate-and-conquer-based rule learning algorithms. By combining the relevant components, it is easy to rebuild the original learning algorithm again. In SeCo, the selection of the components is written in an XML (Extensible Markup Language) file. Table 3.1 describes the essential elements and attributes that might appear in this file. More details about the properties of this file can be found in [30] Chapter 4. The SeCoFactory class in the package *factory* is able to parse the XML file to get the essential information about the components as well as their setter methods and properties. In this way, all the involved components and undeclared default ones will be initialized in advance so as to construct the integrated learning algorithm.

Element	Attribute	Description
seco		the root element, which contains the configuration of a SeCo classifier
secomp		a SeCo component
	interface	the interface regarding class AbstractSeco, possible values are ruleinitializer, candidateselector, ruleevaluator, rulerefine, stopcriterion, rulestopcriterion, rulefilter and postprocessor
	class name	name of Java class that implements the component
	package	(optional) the Java package, which indicates the given Java class, seco.learners.components is accepted as a default value
jobject		an arbitrary Java object
	class name	see secomp
	package	see secomp
	setter	the name of the setter method without the prefix 'set', e.g. for the java method setHeuristic, the setter name must be 'heuristic'
property		define a specific property of an object
	name	the name of property
	value	the value of property

Table 3.1.: Elements in SeCo XML Description [30]

3.1.6 Data Set Format

Used in the learning algorithms implemented in SeCo, the data set is derived from the ARFF (Attribute-Relation File Format [32]) file. This is an ASCII text file which is used to describe a list of examples sharing a set of attributes. Table 3.2 shows the global structure of an ARFF file, which consists of two sections. The first one is the header information, which describes the name of the data set and a set of attributes that are used for describing the examples. Besides the name of each attribute, the corresponding type is declared as well. If the type of attribute is numeric, <attribute type> is easily written as "real" or "integer" ("real" means that the value of the numeric attribute should be real numbers and "integer" means integer numbers). However, if the type of attribute is nominal, then <attribute type> must declare its possible value in a set, e.g. {sunny, overcast, rainy}. Note that the values in the last attribute are taken as the class names. The second is the body part that lists all the examples one by one. The question mark in the example means that the value for the relevant attribute is missing. More details about the ARFF file are described in [32]. In SeCo, the class *Instances* in the package *mode* is responsible for parsing such a file. The indispensable information such as attributes and examples will be deposited in the relevant array lists.

Section	Element	Attribute
head	@relation @attribute	<data set name> <attribute name> <attribute type>
body	@data	<data set> e.g. sunny,85,85,FALSE,no rainy,70,?,TRUE,yes overcast,72,90,?,yes

Table 3.2.: Structure of an ARFF File

3.2 Algorithms

This section consists of three parts. Firstly, the history of REP-based algorithms is discussed, and their respective advantages described. Secondly, the methods used in the postprocessing phase in RIPPER, which is a popular rule learning algorithm that successfully optimizes the learned rule set, are analyzed. Finally, two variants are presented based on the RIPPER algorithm. These variants change the original methods used in the postprocessing phase.

3.2.1 REP-based Algorithms

As a method for simplifying decision trees, a prototype of **R**educed **E**rror **P**runing was proposed by J. Ross Quinlan in [27]. Its basic purpose is to search for the smallest decision tree from the initial tree with the highest accuracy on a separate test set (i.e. pruning set). Such a test set is used to examine the misclassification of the dummy tree. The dummy tree is constructed by replacing the internal node in the initial tree with the best possible leaf node related to the internal one. If it gets an equal number of errors or fewer on the test set, then the internal node is replaced by the leaf node. By repeatedly executing the same process on each internal node in the tree, the error rate as well as the size of the decision tree is continually decreased until no replacement is possible. The similar application of the REP method can be found in [22] as well. However, in this paper, a restricted type of decision tree named decision list is used to describe the concepts extracted from the data. Moreover, a top-down greedy approach (i.e. the prototype of separate-and-conquer strategy) is presented to form a decision list.

Year	Algorithm	Author
1987	REP Decision Tree	J. Ross Quinlan
1990	REP Decision Lists	Giulia Pagallo and David Haussler
1994	I-REP/ I-REP^2	Johannes Fürnkranz and Gerhard Widmer
1995	I-REP*/ RIPPER	William W. Cohen

Table 3.3.: List of REP-based Algorithms

In [6] it is mentioned that the application of REP could be adopted from propositional decision tree learning to relational concept learning. It generalizes the characteristics of REP algorithm in this way: Firstly, the training set is divided into two independent sets. The first one, the growing set, is used to learn a rule set (i.e. the growing phase) while the other one, the pruning set, is used to prune the learned rule set (i.e. the pruning phase). Secondly, in the growing phase, the generated rule set must fulfil the requirements as follows:

- **Completeness**: Each positive example in the growing set should be covered at least once
- **Consistency**: None of the negative examples should be covered

Thirdly, there are two operators available for pruning a rule set:

- **Delete-condition**: Deleting a condition from a rule
- **Delete-rule**: Deleting a rule from a rule set

In general, the process of pruning will continue until the further deletion might decrease the accuracy of the rule set. During the pruning phase, not only the size of the rule set but also the number of conditions of each rule will be reduced. Another advantage of REP is that the pruning phase is completely independent of the growing phase. REP is thus called a post-pruning algorithm. However, the disadvantage of such an algorithm is obvious. For a smaller data set, the division of the training set might directly lead to a shortage of examples for learning the initial rules. Moreover, in the growing phase, there is no stop criterion to restrict the improvement of rules, so the grown rules are sometimes more specific than the searched one. Hence, in the pruning phase, more resources are needed to prune these superfluous conditions and rules [7, 14]. Due to the limitations and insufficiencies of REP, there are many different approaches to extend and improve it. Table 3.3 gives a list of the REP-based algorithms that are sorted by the year of the development of the algorithm.

I-REP/ I-REP^2

In [16] some problems that exist in the original REP algorithm are pointed out and thereafter an improved algorithm, **I**ncremental **R**educed **E**rror **P**runing, is proposed as a solution. These problems cover four fields: efficiency, split of training set, separate-and-conquer strategy and bottom-up hill-climbing. Firstly, in REP, a rule set will only be pruned after it was learned. The total cost of REP is about $\Omega(n^4)$ (n is the number of examples). But the cost of the pruning is outweighing that of the learning by a huge margin. Secondly, due to the division of the training set, the distribution of examples in the growing and pruning set is predetermined and cannot be modified. In the case of non-uniform distribution of examples related to different classes, the behavior of both the learning and pruning algorithm will be influenced directly. Thirdly, in the separate-and-conquer strategy, the pruning of conditions of a rule will affect all subsequent rules. The rule is generalized if its conditions are pruned. Thus, this rule will cover more positive and negative examples. Those additional examples should be removed so that they cannot influence the learning of subsequent rules. Finally, REP applies a greedy bottom-up hill-climbing strategy for pruning the rule set. However, in noisy domains it can be excepted that the generated rule set is much too specific, because it should fulfil the requirements of *completeness* and *consistency*. Thus, REP has to do a lot of pruning

and this specific-to-general search can be expected to be slow and imprecise for noisy data.

Solving the above-mentioned problems is the goal of the I-REP algorithm. In contrast to REP, the pruning method used in I-REP does not aim at the entire rule set but at the individual rule. The main characteristic of this algorithm is the integration of the pre-pruning and post-pruning in the learning system. Post-pruning means that each rule will be pruned right after it was learned while pre-pruning will be realized by defining a rule stop criterion. The basic idea of the rule stop criterion is to stop the generation of rules even though some positive examples may still not be covered. The global process of I-REP can be summarized as follows:

1. Splitting the training set into growing set and pruning set
2. Learning a rule on the growing set
3. Pruning the learned rule on the pruning set
4. Testing the rule according to the rule stop criterion
   - True: Adding the rule to the rule set and update the training set by removing the positive and negative examples that are covered by the rule
   - False: It is not necessary to generate more rules, the process will be stopped
5. Repeat steps 1-4 until all positive examples in the growing set are covered by the current rule set

The pruning process in the 3rd step above is the same as the delete-condition operator defined in REP. It will continue until the further deletion would decrease the accuracy of the current rule. The search heuristic accuracy is defined in equation 3.1. In the 4th step, the accuracy of an empty rule is set as a threshold and is calculated as $\frac{\mathbf{N}}{\mathbf{P+N}}$. If the accuracy of the pruned rule is less than that of the empty rule, the pruned rule cannot be added to the rule set. Therefore, the delete-rule operator in REP is not needed in I-REP any more because the rule stop criterion prevents those low accuracy rules in advance. I-REP has a variation named I-REP^2 which keeps the same structure. However, I-REP^2 takes the precision/purity (see equation 3.3) as the search heuristic in the pruning process. The default threshold defined in I-REP^2 is a purity value of 0.5, which means that the number of positive examples covered by the rule must be more than that of negative examples.

$$\begin{aligned} Accuracy &= \frac{tp + \mathbf{N} - fp}{\mathbf{P+N}} && (3.1) \\ &\cong tp + \mathbf{N} - fp \\ &\cong tp - fp && (3.2) \\ Precision/Purity &= \frac{tp}{tp + fp} && (3.3) \end{aligned}$$

According to [16], the predicted total cost (running time) of I-REP is about $O(n \log^2 n)$ (n is the number of examples), which is obviously decreased in comparison to REP. Due to the removal of covered examples by the qualified pruned rule, the problem of generating redundant rules can be avoided as well. Furthermore, instead of deleting the conditions and rules from the complicated rule set in REP, I-REP is a top-down hill-climbing algorithm that recursively learns and adds new rules to the rule set. Therefore, the pruning of each individual rule is much simpler. For the problem of "split of training set" I-REP does not resolve it completely, but reduces its scope from learning and pruning of the entire rule set to each individual rule.

I-REP*

The efficiency of the rule learning algorithm I-REP is investigated well in [8]. It indicates that the runtime of I-REP is indeed fast, so it can deal with large data sets more efficiently. However, an existing problem

of the learned rule set is discovered by comparing it with another rule learning algorithm C4.5 [28]. In contrast to I-REP, C4.5 always needs much more time to generate a rule set, but the error rate of the learned rule set is often lower. Therefore, the construction of a new rule learning algorithm that combines the advantages of lower error rate and efficient running time is an interesting topic in [8]. Based on the original I-REP algorithm, three different modifications are presented. The first two modifications only improve the generalization performance while the third one reconstructs I-REP. In this section, we will only focus on the I-REP* algorithm (using the first two modifications) and the RIPPER algorithm (using all the modifications) will be described in section 3.2.2 in detail.

- A new pruning heuristic for guiding the pruning phase
- A new rule stop criterion
- The technique to optimize the learned rule set

Used in I-REP, *accuracy* is the original search heuristic in the pruning phase. It calculates the percentage of correctly classified examples. Equation 3.1 shows that the denominator in this heuristic is the sum of positive and negative examples. This value is equivalent to the number of examples in the pruning set and it is a constant. Therefore, the formula of this heuristic can be simplified just like in equation 3.2. Although the calculation of this formula is quite simple, there are some deficiencies. For example, it is assumed that a rule R_1 covers 100 positive examples and 1 negative example and the pruned rule R_2 based on R_1 covers 200 positive examples and 100 negative examples. Which one is better? According to equation 3.2, the pruned rule R_2 is preferred. Actually, the original rule R_1 is much more precise as well as meaningful. Therefore, an appropriate denominator is needed in the formula to avoid it. The new pruning heuristic is defined in equation 3.4 and integrates the two heuristics *accuracy* and *precision*. The goal of this new heuristic is not to calculate the percentage of correctly classified examples of the entire examples but to calculate the percentage of correctly classified examples of the covered examples.

$$new\ heuristic = \frac{tp - fp}{tp + fp} \tag{3.4}$$

However, in [11], it is argued that the functionality of this new heuristic is similar to the default pruning heuristic *precision* used in I-REP^2. It offers proof described in formula 3.5. Moreover, the experiment described in [8] has already confirmed that the pruning heuristic *accuracy* in I-REP performs better than the heuristic *precision* in I-REP^2. In order to further verify the efficiency of these heuristics, they will be tested in our implementations in SeCo later.

$$\frac{tp - fp}{tp + fp} = \frac{2tp - tp - fp}{tp + fp} = 2 * (\frac{tp}{tp + fp}) - 1 \propto \frac{tp}{tp + fp} \tag{3.5}$$

The original rule stop criterion in I-REP is relatively simple. For each pruned rule, if the accuracy of this rule is greater than that of its relevant empty rule, it will be added to the rule set. However, this criterion depends completely on the class distribution in the pruning set. If the number of examples related to the target class is less than that of examples related to other classes, this criterion has no significance. Another rule stop criterion used in I-REP^2 is much better. However, it often stops too soon given moderate-sized training sets, which is especially true when learning a rule set that contains many low-coverage rules [8]. Therefore, William W. Cohen proposes a new rule stop criterion which includes two aspects to solve this problem:

- The total description length of the rule set and the examples is d bits larger than the smallest description length of the previous rule set
- No more positive examples

In I-REP*, the new generated rule (after the growing and pruning phase) will be added into the rule set directly. Then the total description length of the new rule set and the examples will be computed. The term *description length* means the number of bits needed to encode both the rule set and the examples from which it was learned [23]. I-REP* will stop adding rules when this description length is more than d bits larger than the smallest description length obtained previously. Moreover, due to the increasing of the description length, the last added rule will be deleted from the rule set as well. Conversely, if this description length is smaller, I-REP* will continue to learn more rules and the old smallest description length will be replaced with the new one. Secondly, the learning process will also be stopped if there are no more remaining positive examples in the training set.

3.2.2 RIPPER Algorithm

RIPPER is a standard rule learning algorithm that contains both the building and postprocessing phase. In the building phase, an initial rule set for one class is learned and iteratively optimized in the postprocessing phase. It is notable that the postprocessing phase in RIPPER is only suitable for the two-class problem (target class or non-target class). In other words, for the multi-class training set, RIPPER will always start a new postprocessing process right after a rule set for one class is constructed. As mentioned before, as the initial prototype of REP-based algorithms, REP is only a post-pruning learning algorithm that always starts the pruning phase after the entire rule set has been learned. What is the relation between them and what is the advantage of the postprocessing phase in RIPPER? In this section, the structures of these REP-based algorithms will be analyzed, and then the methods used in the postprocessing in RIPPER will be presented and discussed.

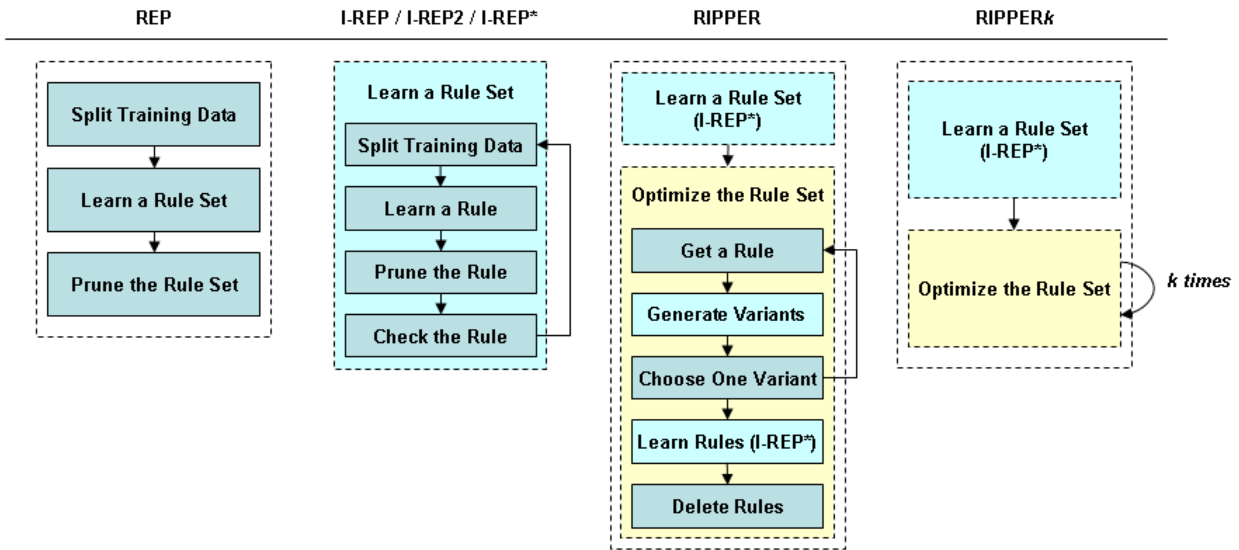


Figure 3.3.: Structure of REP, I-REP and RIPPERk

In figure 3.3, the global structures of the rule learning algorithms REP, I-REP and RIPPER are described. I-REP^2 and I-REP*, two variations based on I-REP, have the same structure, so they are placed in one group. The process of learning a rule set by REP can be easily summarized in three steps: The first step is to split the training set into the growing set and the pruning set, then an initial rule set is learned from the growing set in the second step and will be pruned on the pruning set at the last step. I-REP changes

the original structure where each rule is pruned right after it is learned. The learning process in I-REP is executed recursively until all of the positive examples in the growing set are covered or the predefined rule stop criterion is met. In this way, the complicated task of pruning the entire rule set is dispersed. I-REP* only redefines the pruning heuristic and rule stop criterion in I-REP to get a rule set that has a lower error rate. As a new rule learning algorithm, RIPPER is developed based on the original rule set constructed by I-REP*. It introduces the concept of postprocessing that optimizes the learned rule set again. However, the adopted optimization technique in RIPPER does not attempt to optimize each rule in its rule set directly, but to search for some possible variants based on the original rule, and selects a variant among them as the optimized rule. In the RIPPERk algorithm, this postprocessing phase can be executed iteratively to optimize the rule set for k times. The initial purpose of this is to further improve the quality of the rule set. It does achieve this goal, as confirmed in [8]. But is it always possible to get a better rule set, if the rule set is optimized more times? With the help of our implementations , the convergence properties of the RIPPER algorithm for the increasing of the number of optimization iterations will be analyzed and discussed later.

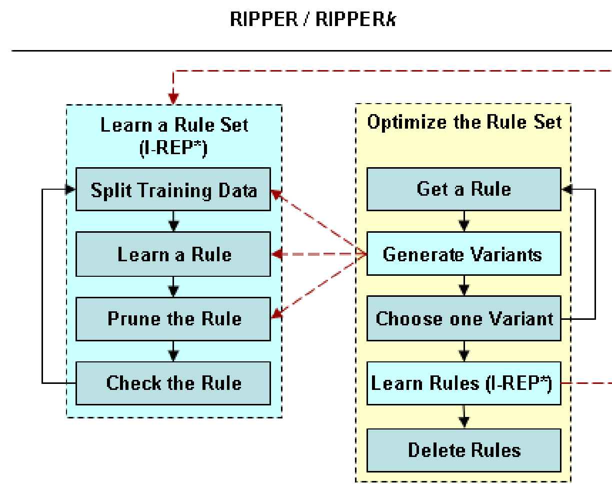


Figure 3.4.: RIPPER / RIPPERk

Figure 3.4 points out some relationships between the building and postprocessing phase in the RIPPER algorithm with the red lines. The structure of the learning phase is the same as that in the I-REP algorithm described in the last section. The postprocessing phase can be seen as an integrated component that consists of the following steps:

1. Getting an old rule from the learned rule set
2. Generating variants based on the old rule
3. Choosing a rule among the variants and the old rule according to the selection criterion
4. Repeating steps 1-3 until all of the rules in the rule set are optimized
5. Learning more rules to cover the remaining positive examples
6. Deleting some rules to reduce total description length of the entire rule set

For the first step, it is simple to take each rule from the rule set for optimization. The optimization of rules usually depends on the order they were learned. Assuming that there is a rule set $RS = \{R_1, R_2, ...R_k\}$, the first rule R_1 should be taken at first and two variants $R_1^{'}$ ($R_{1_Replacement}$ or $R_{1_Revision}$) will be constructed in the second step. Figure 3.4 shows that the process of generating variants is achieved by

executing similar steps defined in the building phase. However, the methods used for growing and pruning a variant is a bit different. Moreover, the learned variant will not be checked by the rule stop criterion. These differences are listed in table 3.4.

Art	Name	Growing Phase	Pruning Phase
Initial Rule	Old Rule R_i	growing a new rule from an empty rule	the pruning heuristic is guided to minimize the error of the single rule
Variant	Replacement $R_{i_Replacement}$	see Old Rule	the pruning heuristic is guided to minimize the error of the entire rule set
Variant	Revision $R_{i_Revision}$	further growing the given old Rule R_i	see Replacement

Table 3.4.: Variants and Old Rule

For the third step, the variant $R_{i_Replacement}$, $R_{i_Revision}$ and the old rule R_i are three candidate rules and one of them will be selected as the optimized rule. If the old rule outperforms the other two variants, it will be selected and the original rule set will remain unchanged. However, if one of the other two variants is better, it will be selected and replace the original old rule in the rule set. Due to the different coverage of the selected variant compared to the original rule, the residual examples that are available for the generating of rules will be changed. Thus, the old coverage of the rules in the original rule set may not be completely correct. As a candidate of the optimized rule, the variant *Replacement* grown from an empty rule is able to generate a completely new rule. This is a way to find a new best rule. In contrast to *Replacement*, *Revision* is grown by greedily adding conditions to the original old rule. In this way, in the growing phase, *Revision* should usually be more specific than the original rule. By pruning the *Revision* with a new pruning heuristic, there is also a chance of finding a better rule.

The initial rule set was learned in the building phase. The utilization of the known information about the rule set is an obvious advantage for RIPPER. In the postprocessing phase, the pruning heuristic defined is guided to minimize the error rate of the entire rule set. The rule set means that the old rule in the original rule set is replaced with the pruned variant, such as $\{R_1^{'}, R_2, ... R_k\}$. The evaluation of this rule set will be calculated according to equation 3.6 and be used as a reference value for the variant. Here tp_i means the number of positive examples that are covered by each rule in the rule set, and tn is the total number of negative examples that are not covered by this rule set. During the pruning phase, the existing conditions will be deleted successively from the variant to maximizes the evaluation value.

$$v(RuleSet) = \frac{\sum_{i=1}^{k}\{tp_i\} + tn}{\mathbf{P} + \mathbf{N}} \tag{3.6}$$

In RIPPER, the selection of the best rule is achieved by means of the heuristic minimum description length (MDL) among the variants *Replacement*, *Revision* and the old rule. As an important concept of information theory and learning theory, MDL was introduced in [29, 31]. Equation 3.7 shows a standard formula, where $I(H)$ is the amount of information needed to transmit the hypothesis H, and $I(E|H)$ means the amount of information needed to transmit a set of examples E from which this hypothesis H was derived. In other words, in an MDL calculation, each possible rule set derived from a training set can be characterized by its description length (i.e. the number of bits needed to encode both the rule set and the training set where it was learned). However, in fact, the former part is usually used to estimate the complexity of a rule set, but the latter one is able to measure the degree to which the rule set fails to account for the training set [23]. Therefore, the meaning of the MDL heuristic can be summarized in finding a tradeoff between the complexity and the accuracy of the rule set.

$$MDL(H) = I(H) + I(E|H) \tag{3.7}$$

In an MDL calculation, the rule sets used are some temporal rule sets, where the original rule is replaced with the variants (*Replacement*, *Revision*) respectively. By getting the MDL value on these temporal rule sets, the rule which constructs the temporal rule set with the smallest MDL will be considered the best one and be selected as the final optimized rule. Then the loop will return to the first step to optimize the second rule R_2 likewise. This process will continue until all of the rules $\{R_1, R_2, ...R_k\}$ in the rule set RS have been optimized once. In this way, a new rule set named RS' is constructed.

After optimizing all the rules in the rule set RS, some positive examples may not be covered by the new rule set RS' any more. Therefore, it is necessary to generate some new rules to cover them. In RIPPER, the methods for generating rules in the algorithm I-REP* will be used again. These new learned rules will be added to the rule set RS' successively. In this way, the size of the rule set RS' is increased. In RIPPER, another technique is used to limit the size of the learned rule set. For each rule R_i in the rule set RS', if the rule set without the rule R_i has a smaller description length than the original one, the rule R_i will be deleted. In other words, the RIPPER algorithm uses this approach to ensure that the final rule set always has the minimal total description length. Thus, there can be some positive examples that are not covered by the final rule set any more.

3.2.3 Variant Abridgment

In RIPPER, two variants *Replacement* and *Revision* are constructed to be compared with the original old rule. In this section, another variant named *Abridgment* is presented. This variant is generated by pruning the original old rule with a new pruning method.

$$R : Class = A : C_1, C_2, C_3, C_4$$

As mentioned before, a rule is composed of a series of conditions. In the postprocessing phase in RIPPER, a rule is always pruned by deleting the existing conditions according to their order in the rule. Normally, the pruning phase will be stopped if a further deletion will decrease the quality of the rule. For example, it is assumed that there is a rule R available. This rule has four conditions (i.e. C_1, C_2, C_3, C_4) in the rule body.

$$\begin{aligned}
&Replacement\ /\ Revision \\
&R_1 : Class = A : C_1, C_2, C_3 && (after\ 1.\ Iteration) \\
&R_2 : Class = A : C_1, C_2 && (after\ 2.\ Iteration) \\
&R_3 : Class = A : C_1 && (after\ 3.\ Iteration)
\end{aligned}$$

In the first iteration, C_4 will be deleted from the original rule R and the quality of the newly constructed rule R_1 will be evaluated based on the predefined pruning heuristic. If the rule R_1 outperforms the original one, the pruning method will continue to prune the other conditions likewise. However, by using this pruning method, some other candidate rules cannot be constructed. For example, it is impossible to get a rule R_* just like $[Class = A : C_1, C_2, C_4]$. It is possible that the rule R_* outperforms the other rules. In order to enlarge the search space in the pruning phase, a new pruning method is presented in this thesis. The basic idea of this pruning method is to prune the conditions of a rule regardless of their order in the rule. In other words, we can delete any conditions we want.

The rule learning algorithm RIPPER has two phases, namely the building phase and the postprocessing phase. In the building phase, an initial rule set was learned and will be optimized in the postprocessing phase. This new pruning method will be firstly used to prune the rules of the initial rule set, where each

pruned rule is called an "*Abridgment*". The process of the pruning method can be summarized as follows:

Assume that the first rule we take from the initial rule set is the rule R. Firstly, we will check the number of conditions that this rule has. Secondly, we can delete one of them in turn to construct some temporal rules. Because the rule R has four conditions, we can generate four temporal rules after the first iteration:

$$\begin{aligned}
&Abridgment\\
&R_{1'} : Class = A : C_2, C_3, C_4\\
&R_{2'} : Class = A : C_1, C_3, C_4\\
&R_{3'} : Class = A : C_1, C_2, C_4\\
&R_{4'} : Class = A : C_1, C_2, C_3 \qquad\qquad (after\ 1.\ Iteration)
\end{aligned}$$

Then we evaluate these temporal rules based on the measure "Error Rate" defined in the equation below:

$$Error\ Rate = \frac{fp + fn}{total} \tag{3.8}$$

where fp means the number of negative examples covered by the rule and fn is the number of positive examples that are not covered by the rule. *total* means the total number of examples in the training set. We compare the error rate of these temporal rules with that of the original rule. If a certain temporal rule has an error rate not higher than the original one and it is also the lowest value among these temporal rules, we take this temporal rule as the new candidate rule and try to delete the other conditions from it in the same way. Even if the temporal rule has the same error rate as that of the original one, we prefer to take the smaller one. And if there is no temporal rule meeting the above-mentioned conditions, we just stop the search. This action could be seen as a straightforward greedy search, but there is no guarantee that minimizing the error rate at each step can lead to a global minimum. Recalling the rule R, we simply name its conditions {1, 2, 3, 4}. Based on two different pruning methods, figure 3.5 shows the search spaces of possible candidate rules separately.

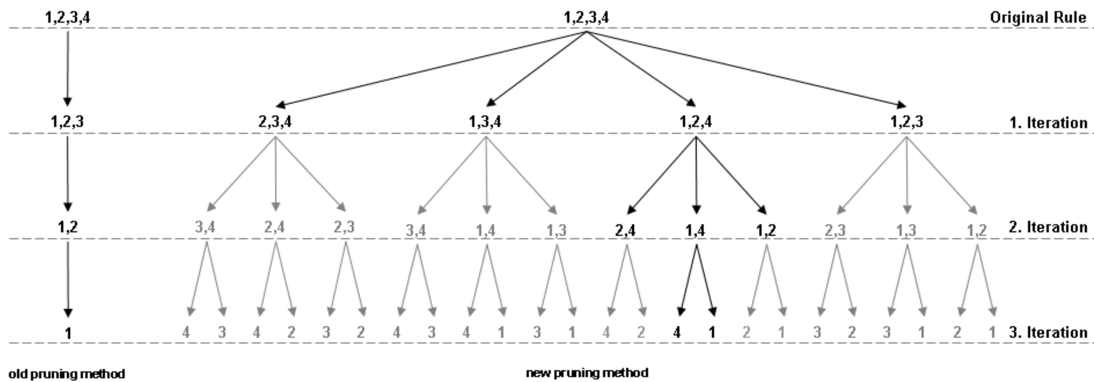


Figure 3.5.: Pruning Methods

The search space of the new pruning method is a tree structure that is obviously larger than that of the old pruning method. However, in the new pruning method, only the candidate rule with the lowest error rate will be selected to be further processed at each iteration. This means that some subtrees will not be routed (marked in gray). In order to try all the candidate rules in the tree, we can complete an exhaustive search when the number of conditions of the original rule is small.

3.2.4 Simplified Selection Criterion

In RIPPER, after we generate the variants based on the original rule, we always select one of them that can minimize the description length of the entire rule set and the examples. The MDL used here is a heuristic that is aimed at finding a trade-off between the complexity and the accuracy of a hypothesis [17]. In [8] it is confirmed that the RIPPER algorithm (with postprocessing phase) outperforms the I-REP* algorithm (no postprocessing phase). In other words, the optimized rules in the postprocessing phase in RIPPER should play a positive role. However, the calculation of the MDL is complicated so it increases the computational complexity of the whole program. Is it possible to use a simpler heuristic to get a similar result? In our implementation, we decide to try other heuristics instead of the predefined MDL in the selection criterion. In contrast to MDL, accuracy is a popular heuristic that is used in many rule learning algorithms. Moreover, it can often lead to a good result, even though it is easy to calculate. Therefore, besides the MDL heuristic, we will evaluate the variants according to the accuracy heuristic defined in the selection criterion as well as selecting the one that has the maximal value.

4 Implementation in SeCo

4.1 XML File

Both the RIPPERk algorithm and the two variants mentioned in the previous chapter are implemented in the SeCo platform. Note that the implementation of the original RIPPER in SeCo is simply called SeCoRIP. As described in section 3.1.5, the SeCo platform is configurable through an XML file, which contains all the essential information. By changing the class name and correlative properties in the file, many different rule learning algorithms can be constructed. In this section, we will explain the XML file that constructs our implementation SeCoRIP$_k$. For example, the following figure shows the XML file of the learning algorithm SeCoRIP$_2$.

```
<seco>
    <property name="growingSetSize" value="3"/>
    <property name="minNo" value="2"/>
    <property name="seed" value="1"/>

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

    <secomp interface="rulerefiner" classname="TopDownRefiner" package="seco.learners.
        topdown">
        <property name="nominal.cmpmode" value="equal"/>
    </secomp>

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

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

    <secomp interface="postprocessor" classname="PostProcessorRipper">
        <property name="optimizations" value="2"/>
        <property name="abrigment" value="0"/>
        <property name="selection" value="MDL"/>
    </secomp>
</seco>
```

Algorithm 4.1: XML for SeCoRIP$_2$

Interface

- **RuleEvaluator** -> IRuleEvaluator
  The class *RuleEvaluator* is a default component that is provided by SeCo. However, in the growing

phase, the RIPPER algorithm uses the heuristic foil's information gain to grow a new rule. Thus, we replace the default heuristic m-estimate with the new one. The heuristic foil's information gain is implemented in class seco.heuerstic.FoliGain.

- **TopDownRefiner** -> IRuleRefiner
  In class *TopDownRefiner*, the implementation of the interface *IRuleRefiner* is rewritten. Before we refine the given rule we prepare all the possible conditions that could appear in this rule. We then construct the refinements by adding each possible condition one by one.

- **MDLStoppingCriterion** -> IRuleStopCriterion
  The class *MDLStoppingCriterion* defines a new rule stop criterion in which we will continue learning rules for the target class until the rule stop criterion prevents it. In this class, the description length of the newly constructed rule set will be compared to the smallest description length obtained previously. The detail of the implementation of this class can be found in section 4.2.5.

- **PostProcessorRipper** -> IPostProcessor
  The class *PostProcessorRipper* is implemented according to the postprocessing phase in RIPPER. Normally, we prepare three variants (Replacement, Revision and Abridgment) for each of the rules in the initial rule set. Then we select one of them (including the old rule) based on different selection criteria.

Configurable Properties

- **GrowingSetSize** $g \in \mathbf{N}^+$. This parameter has two effects:
  1. $g = 1$: Generate a new rule without the pruning phase, i.e. grow a new rule by using the whole training set
  2. $g \neq 1$: g means the number of subsets splitting the training set into a growing and a pruning set, then a new rule is grown on the growing set and is pruned on the pruning set

  For simulating the RIPPERk algorithm, the pruning phase is needed, so "g=3" is set as the default value in the XML file. The distribution rate of the training set is defined in subsection 4.2.2

- **MinNo** $minNo \in \mathbf{N}^+$.
  The minimum number of examples a rule has to cover, default value is 2.

- **Seed** $seed \in \mathbf{Z}$.
  The seed to perform randomization for stratifying the training set, default value is 1.

- **Optimizations** $optimizations \in \mathbf{N}$.
  The number of optimization iterations in the postprocessing phase, default value is 2. If the value of *optimizations* is 0, the learned rule set constructed in the building phase will not be optimized.

- **Abridgment** $abridgment \in \{0, 1\}$.
  In the postprocessing phase, besides *Replacement* and *Revision* a new variant *Abridgment* will be constructed, if the value of *abridgment* is 1. Default value is 0.

- **Selection** $selection \in \{Heurstics\}$.
  Based on a selection criterion, one of the variants will be selected in the postprocessing phase. The evaluation heuristic defined in the selection criterion is used to evaluate these variants. The parameter *selection* indicates which heuristic is used. In SeCo, each heuristic is separately implemented in the package *seco.heuristics*. The possible values of *selection* refer to the class names of these heuristics (e.g. *Accuracy*, *Precision*, *Laplace* and so on).

The separate-and-conquer strategy is well described in [15] as follows: *Learn a rule that covers a part of the given training examples, remove the covered examples from the the training set (the separate part) and recursively learn another rule that covers some of the remaining examples (the conquer part) until no (positive) examples remain.* Algorithm 4.2 shows the pseudocode of the procedure *SeparateAndConquer*, which is implemented based on the above-mentioned separate-and-conquer strategy in the class *AbstractSeCo*. This procedure can generate a rule set for a given target class. For a multi-class problem (i.e. with many different classes), it is necessary to determine the order of processing classes first. In general, we will start with the class with the least number of examples to the class with the most.

```
procedure SeparateAndConquer(Data) {
    Theory = ∅
    newData = Data

    # the building phase
    while (Positive(newData) ≠ ∅)
    {
        if (GrowingSetSize ≠ 1)
        {
            newData = Stratify(newData, GrowingSetSize, Seed)
            SplitData(GrowingSetSize, newData, GrowData, PruneData)
            Rule = FindBestRule(GrowData)
            Rule = PruneRule(PruneData, Rule)
        }
        else
            Rule = FindBestRule(newData)
        if CheckForRuleStop(Theory, Rule, newData)
            exit while
        newData = newData \ Covered(Rule, newData)
        Theory = Theory ∪ Rule
    }

    # the postprocessing phase
    Theory = PostProcessTheory(Theory, Data)

    # remove the covered examples
    Data = Data \ Covered(Theory, Data)

    # return the rule set for one class
    return (Theory)
}
```

Algorithm 4.2: SeparateAndConquer

In the procedure *SeparateAndConquer*, the parameter *Data* means the given training set. The whole procedure can be divided into two phases, namely the building and the postprocessing phase. Because it is necessary to use the original training set in the postprocessing phase again, we set a new parameter *newData* that has the same examples as *Data*. Starting with an empty rule set (i.e. *Theory* in algorithm 4.2), new rules are generated and added to the rule set until all the positive examples are covered. In our implementation, a parameter *GrowingSetSize* is used to decide if a rule will be pruned after it is grown. If not, in the procedure *FindBestRule*, the whole training set can be used to grow a new rule directly. Oth-

erwise, the training set should be divided into two subsets in advance. The growing set (i.e. *GrowData*) is used to grow a rule and the pruning set (i.e. *PruneData*) is used to prune the learned rule. Moreover, in the procedure *Stratify*, the training set should be stratified before it is divided. The reason and methods for stratifying a training set will be explained in section 4.2.1. The pruning of a rule is achieved by running the procedure *PruneRule*. After a new rule is generated, it will be examined in the procedure *CheckForRuleStop*. Based on a predefined rule stop criterion, this procedure is used to determine whether it is useful to generate more rules or not. If the return value of *CheckForRuleStop* is true, the new learned rule will be dropped and the learning process will be stopped. If the return value of *CheckForRuleStop* is false, the new learned rule will be added to the rule set and the procedure will continue to generate more rules from the remaining examples. The implementation of the procedure *CheckForRuleStop* is defined in section 4.2.5.

If all the positive examples are covered or the rule stop criterion is met, the building phase is then finished and the initial rule set will be passed to the postprocessing phase. In the procedure *PostProcessTheory*, the learned rule set will be further optimized. The implementation of the procedure *PostProcessTheory* is described in section 4.3. After optimizing the rule set in the postprocessing phase, all the examples covered by the rule set should be removed. Finally, the generation of a rule set for one class is then finished and the current rule set will be returned.

4.2.1 Stratified Data

As mentioned in section 3.1.6, a training set will be collected by parsing an ARFF [32] file. In the training set, the order of the examples is kept the same as that in the original file. Moreover, in RIPPER, a rule will be generated in two phases, namely the growing and pruning phases. Thus, it is necessary to divide the training set into two subsets in advance. But if we divide the training set directly, it is possible that the distribution of classes in each subset is not reasonable, for example the growing set containing all the examples for class X and the pruning set only the examples for class Y. It is meaningless to prune rules on such a pruning set, because no examples in the pruning set will be covered. In order to avoid such a distribution, it is recommended that the training set is preprocessed. In the RIPPER algorithm, the training set will be stratified before the division. This process can be achieved by running the procedure *Stratify* implemented in class *RuleStats*.

$$\mathsf{Data} = \mathsf{Stratify}(\mathsf{Data}, \mathsf{GrowingSetSize}, \mathsf{Seed})$$

The entire process is summarized in the following four steps:

1. Generating k bags $\{bag_0, bag_1, ...bag_k\}$, each bag only contains the examples with regard to a certain class (k means the total number of classes in the training set), and sorting these bags in increasing order of the number of examples
2. Randomizing the examples in each bag (*Seed* is a parameter for the random function)
3. Generating g subsets $\{subset_0, subset_1, ...subset_g\}$ successively by taking a reasonable number of examples from each bag (the order of the bags is sorted previously and *GrowingSetSize* g means the number of subsets to split the training set)
4. Constructing the new training set by combining these subsets

In order to generate a reasonable subset and to avoid taking the same example for different subsets, we set up an initial offset for each subset ($\text{offset}(subset_i) = \text{i}$) in the 3rd step above. Starting with taking examples from the bag_0, which has the fewest examples, we only take the example whose sequence number in the bag is equal to the value of (offset+1), and update the offset by (offset = offset + g). Then we try to take the next example in this bag and handle the offset in the same way until the value of the offset is not

less than the number of examples in the bag (offset ≥ the number of examples in bag). In this case, we just stop taking examples from bag_0 and update the offset by (offset = offset - the number of examples in bag_0). After that, we try to get examples from the next bag (bag_1). After all the bags are explored, the generation of the first subset is done. Then next subsets are generated in the same way.

age	spectacle-prescrip	astigmatism	tear-prod-rate	contact-lenses
young	myope	no	reduced	none
young	myope	no	normal	soft
young	hypermetrope	yes	reduced	none
young	hypermetrope	yes	normal	hard
pre-presbyopic	myope	no	normal	soft
pre-presbyopic	myope	yes	reduced	none
pre-presbyopic	myope	yes	normal	hard
pre-presbyopic	hypermetrope	no	reduced	none
pre-presbyopic	hypermetrope	no	normal	soft
presbyopic	myope	yes	normal	hard
presbyopic	hypermetrope	no	reduced	none
presbyopic	hypermetrope	no	normal	soft

Table 4.1.: Original Training Set

age	spectacle-prescrip	astigmatism	tear-prod-rate	contact-lenses
presbyopic	myope	yes	normal	hard
young	hypermetrope	yes	normal	hard
pre-presbyopic	myope	yes	normal	hard
age	spectacle-prescrip	astigmatism	tear-prod-rate	contact-lenses
young	myope	no	normal	soft
pre-presbyopic	myope	no	normal	soft
pre-presbyopic	hypermetrope	no	normal	soft
presbyopic	hypermetrope	no	normal	soft
age	spectacle-prescrip	astigmatism	tear-prod-rate	contact-lenses
pre-presbyopic	hypermetrope	no	reduced	none
young	hypermetrope	yes	reduced	none
pre-presbyopic	myope	yes	reduced	none
presbyopic	hypermetrope	no	reduced	none
young	myope	no	reduced	none

Table 4.2.: Randomized Bags

An example is given and the original training set is defined in Table 4.1. It contains 12 examples that share 3 different class values. According to the 1st and 2nd step described above, we generate three bags and randomize each of them. Table 4.2 shows the results of three randomized bags, namely class(bag_0)=hard, class(bag_1)=soft and class(bag_2)=none. Assuming that the value of "g" is 3, the training set should be divided into three subsets. Here, we start with the $subset_0$ and the initial offset is 0. We take the example at the first position from bag_0 and update the value of the offset to 0+3=3. We find that the current offset is greater than the number of examples in bag_0 so we update the offset to 3-3=0 and start to get the examples from bag_1. Similarly we take the example at the first position from bag_1 and now the offset is 3 again, but the number of examples in bag_1 is 4, so we can take the example at the fourth position in this bag again and now the offset is updated to 3+3=6. Again, we stop the bag_1 and aim at the bag_2

by updating the offset to 6-4=2. This time we take the example at the third position in bag_2 and update the offset to 2+3=5, which is greater than the number of examples in bag_2, so the generation of $subset_0$ is done and it has 4 examples. We use the same method to get the $subset_1$ and $subset_2$ and to combine them to construct a new training set described in Table 4.3.

age	spectacle-prescrip	astigmatism	tear-prod-rate	contact-lenses
presbyopic	myope	yes	normal	hard
young	myope	no	normal	soft
presbyopic	hypermetrope	no	normal	soft
pre-presbyopic	myope	yes	reduced	none
age	spectacle-prescrip	astigmatism	tear-prod-rate	contact-lenses
young	hypermetrope	yes	normal	hard
pre-presbyopic	myope	no	normal	soft
pre-presbyopic	hypermetrope	no	reduced	none
presbyopic	hypermetrope	no	reduced	none
age	spectacle-prescrip	astigmatism	tear-prod-rate	contact-lenses
pre-presbyopic	myope	yes	normal	hard
pre-presbyopic	hypermetrope	no	normal	soft
young	hypermetrope	yes	reduced	none
young	myope	no	reduced	none

Table 4.3.: Stratified Training Set

4.2.2 Growing Set and Pruning Set

In class *SplittedInstances*, we implemented the procedure *SplitInstances*, which is used to split the stratified training set into a growing set and a pruning set.

$$\mathsf{SplitData}(\mathsf{GrowingSetSize}, \mathsf{newData}, \mathsf{GrowData}, \mathsf{PruneData})$$

In this procedure, a parameter *growingSetSize* "g" is needed to calculate the distribution rate *growPercent* for the growing set:

$$growPercent = \frac{g-1}{g} \qquad g \geq 2$$

According to the formula above, $\frac{g-1}{g}$ percent of the whole training set is used for the growing set and the rest $\frac{1}{g}$ percent of the training set is used for the pruning set. For example, if the value of "g" is 3, we will divide the stratified training set in table 4.3 as follows: The first 8 examples represent the growing set and the pruning set takes the remaining 4 examples.

4.2.3 Growing a Rule

The original procedure *FindBestRule* supported by SeCo is implemented in class *AbstractSeCo*. The essential task is to grow a new rule by greedily adding conditions to an empty rule. However, the variant *Revision* in RIPPER is grown by adding conditions to the original rule directly. Therefore, the framework of the original procedure is extended by adding an IF-ELSE statement (see Algorithm 4.3). In the case of the given parameter *Rule* not being empty, this rule is set as the initial rule. Otherwise, the initial rule (only an empty rule with its class name) is generated by executing the procedure *InitializeRule* implemented in

class *TopDownRuleInitializer*.

In RIPPER, the foil's information gain (the heuristic is firstly used by the rule learning algorithm FOIL [24]) is defined as an evaluation heuristic to evaluate the quality of rules that are learned in the growing phase. This heuristic is used to calculate how much profit can be gained, if a rule is refined. In our implementation, this heuristic is implemented in class *FoilGain*. By executing the procedure *EvaluateRule*, this heuristic will be firstly used to evaluate the initial rule. Then the initial rule is set to the current *BestRule* and is added to the *RuleList*. In this procedure, the task of the *RuleList* is to keep those candidate rules that will be further processed. Moreover, all the rules are always sorted in decreasing order of evaluation value. Unless the *RuleList* is empty, we will successively take part of candidate rules from it and

```
procedure FindBestRule(Data, Rule) {
    if Rule = ∅
        InitRule = InitializeRule(Data)
    else
        InitRule = Rule
    InitVal = EvaluateRule(InitRule)
    BestRule = <InitRule, InitVal>
    RuleList = {BestRule}
    while RuleList ≠ ∅
    {
        Candidates = SelectCandidates(RuleList, Data)
        RuleList = RuleList \ Candidates
        for Candidate ∈ Candidates
        {
            Refinements = RefineRule(Candidate, Data)
            for Refinement ∈ Refinements
            {
                CurrVal = EvaluateRule(Refinement, Data)
                unless CheckForStop(Refinement, CurrVal, Data)
                {
                    RuleList = RuleList ∪ Refinement
                }
                if Refinement > BestRule
                    BestRule = Refinement
            }
        }
        RuleList = FilterRules(RuleList, Data)
    }
    return (BestRule)
}
```

Algorithm 4.3: FindBestRule

then refine each of them by executing the procedure *RefineRule* implemented in class *TopDownRefiner*. In each refining iteration, only one condition will be added to the original candidate rule. The search for an appropriate condition can be summarized as follows: First, all the attributes that do not appear in this candidate rule will be searched. Note that the nominal attribute can be used once only (e.g. the rule in (1) is not allowed, because the nominal attribute X appears twice), while the numeric attributes Y with different values are allowed to join in the same rule, such as the rule in (2).

- Class = A : X = Sunny, X = Overcast   X is a nominal attribute   (1)
- Class = A : Y > 0, Y < 90   Y is a numeric attribute   (2)

If the unused attribute is nominal, a new condition can be easily constructed by adding a possible nominal value to this nominal attribute. Then, this condition will be added to the candidate rule for constructing a new refinement. For the numeric attribute, it is more complicated to construct a new condition. Firstly, all the examples should be sorted in increasing order of the value of certain numeric attribute to generate a temporary example list. Secondly, the examples in this example list will be checked one by one. In two adjacent examples, if the class of one example is the target class and the other is not, a critical point will be set between them. It is calculated by averaging the values of the relevant numeric attribute. According to this critical point, two numeric conditions can be generated at the same time.

Assume that we have a candidate rule in equation 4.1 and we want to generate its possible refinements. The examples in table 4.4 are taken from the data set *weather.arff*.

$$play = yes : Outlook = rainy \tag{4.1}$$

Nr.	Outlook	Temperature	Humidity	Play
1.	rainy	70	96	yes
2.	rainy	68	80	yes
3.	rainy	65	70	no
4.	rainy	75	80	yes
5.	rainy	71	91	no

Table 4.4.: weather.arff

Nr.	Outlook	Temperature	Humidity	Play
1.	rainy	65	70	no
2.	rainy	68	80	yes
3.	rainy	70	96	yes
4.	rainy	71	91	no
5.	rainy	75	80	yes

Table 4.5.: Searching for Critical Point of Numeric Attribute

Based on the given candidate rule, the class value "yes" should be the target class and the other one "no" is the non-target class. Because the nominal attribute *Outlook* already exists in the candidate rule, only the other two numeric attributes, *Temperature* and *Humidity*, will be considered. Table 4.5 shows a temporary example list which is sorted by the numeric attribute *Temperature*. According to this table, we can get three critical points in total. Firstly, the class of the first example is "no" and the second one is "yes". Hence, the first critical point can be set to ($\frac{65+68}{2} = 66.5$) and two new refinements are generated as follows (the new added conditions are written in red):

- play = yes : Outlook = rainy, Temperature $\geq$ 66.5
- play = yes : Outlook = rainy, Temperature $<$ 66.5

Secondly, a critical point to ($\frac{70+71}{2} = 70.5$) is set between the third and fourth example in the list and the corresponding refinements are:

- play = yes : Outlook = rainy, Temperature $\geq$ 70.5
- play = yes : Outlook = rainy, Temperature $<$ 70.5

Finally, a critical point to $(\frac{71+75}{2} = 73)$ is set between the fourth and fifth example in the list and the corresponding refinements are:

- play = yes : Outlook = rainy, Temperature ≥ 73
- play = yes : Outlook = rainy, Temperature < 73

For the attribute *Humidity*, we can generate 6 refinements, so we get 12 refinements in total. After getting all of the possible refinements, we need to evaluate each of them by using the heuristic foil's information gain. This heuristic is used to calculate, how much profit can be gained, if the candidate rule is refined. The formula is defined as:

$$Gain(R_0, R_1) = tp * (\log_2(\frac{tp_1}{tp_1 + fp_1}) - log_2(\frac{tp_0}{tp_0 + fp_0}))$$

where R_0 is the original candidate rule and R_1 is one of the refinements based on R_0. tp_i and fp_i respectively mean the number of positive and negative examples covered by R_i. In addition, tp is the number of positive examples that are covered by both the R_0 and R_1. However, because R_1 is a refinement based on R_0, it is always more specific than R_0. Thus, the examples covered by R_1 must be covered by R_0 as well. This means that the value of tp should equal to that of tp_1. The profit can be calculated with this formula and its value is set as an evaluation value for the refinement.

The procedure *CheckForStop* will always examine the refinement after it is evaluated. This procedure is used to determine whether it is useful to further process the refinement or not. A refinement can only be added to the *RuleList* if it passes the examination. Otherwise, it will be dropped directly. The implementation of the procedure *CheckForStop* is described in section 4.2.5 in detail. In addition, if the evaluation value of a certain *Refinement* is better than that of the *BestRule* found previously, this *Refinement* will be set to *BestRule* immediately.

In order to control the number of candidate rules in the *RuleList*, a filter is used to delete some rules. The filter will be generated by executing the procedure *FilterRules*, which is implemented in class *BeamWidthFilter*. This class is a default component, in which a parameter "BeamWidth" is used (see section 3.1.4). In the XML file of SeCoRIP, the essential parameter "BeamWidth" is not declared expressly. This means that the value of *BeamWidth* should be the default value "1". In other words, *RuleList* can only keep one rule if it is filtered. Because the *RuleList* is always sorted in decreasing order of the foil's information gain, the rule with the maximal value will be left.

Normally, the candidate rules in the *RuleList* are refined recursively. In the case of an empty *RuleList*, this means that no more refinement is possible. Thus, the rule in the *BestRule* which is the best rule found so far will be returned.

4.2.4 Pruning a Rule

Based on the pruning method defined in [8], the procedure *PruneRule* is implemented in class *AbstractSeCo*. By using this method, some conditions will be deleted from the grown rule. Algorithm 4.4 shows the pseudocode of the procedure *PruneRule*. Because the pruning heuristics used in the building and postprocessing phase in RIPPER are different, the parameter *useWhole* is used to differentiate them in the procedure. Moreover, the parameter *Data* means the pruning set, and *Rule* means the given rule, which was grown in the growing phase.

$$DefaultMaxValue = \frac{defAccu + 1}{total + 2} \tag{4.2}$$

All the existing conditions in the given rule should be stored in the parameter *Conditions* in advance. In addition, the parameter *InitRule* is set as an empty rule that only has the rule header of the given rule. Assuming that the pruning set is classified by the empty *InitRule*, an evaluation value is calculated in the procedure *DefaultMaxValue*. The calculation of this value is based on equation 4.2, where "defAccu" means the number of examples that are covered by the empty rule and "total" means the total number of examples in the pruning set. This evaluation value is set to *MaxVal* and is used as a benchmark. Similarly, *InitRule* is set to *FinalRule*.

After that, the conditions stored in *Conditions* will be returned to *InitRule* successively. For each iteration, an IF-ELSE statement is used to determine which pruning heuristic will be used to evaluate the newly constructed *InitRule*. The pruning heuristic defined in the procedure *EvaluateRule* is usually used to evaluate the rule that was grown in the building phase. As mentioned in section 3.2.1, in the RIPPER algorithm a new pruning heuristic is used. It calculates the percentage of correctly classified examples in the covered examples. But this pruning heuristic can be converted to the common heuristic "precision" (see equation 4.3) [11]. Thus, in our implementation, the pruning heuristic used in the building phase is described as the formula in equation 4.4:

$$v(Rule, PrunePos, PruneNeg) = \frac{tp - fp}{tp + fp} = \frac{2tp - tp - fp}{tp + fp} = (2 * \frac{tp}{tp + fp} - 1) \propto \frac{tp}{tp + fp} \quad (4.3)$$

$$EvaluateRule(Rule, Data) = \frac{tp + 1}{tp + fp + 2} \quad (4.4)$$

where tp and fp mean the number of covered positive (negative) examples respectively, and the value of $tp + fp$ is the total number of covered examples. According to this heuristic, if a rule covers no examples, the denominator in the heuristic will not be zero. As mentioned in section 3.2.2, the pruning heuristic defined in the postprocessing phase in RIPPER is guided to minimize the error of the entire rule set. This means that each constructed *InitRule* will be added to the rule set (replacing the original *Rule*), and then the error of this rule set will be calculated. However, it costs too much time to calculate this value for each iteration. Thus, in our implementation, this value is calculated in a different way. Note that the replacement (i.e. replacing *Rule* with *InitRule*) will only affect the quality of the subsequent rules in the rule set. To be more exact, the coverage of the subsequent rules will be changed. Therefore, the examples covered by the subsequent rules will be removed from the pruning set in advance. In other words, in the postprocessing phase, the parameter *Data* contains the examples that are only covered by *Rule*. Then the procedure *PruneRule* will be used to prune *Rule* based on such a pruning heuristic, which only considers the error rate of the current pruned rule (i.e. *InitRule*). The pruning heuristic is described in equation 4.5, where tn means the number of negative examples that are not covered.

$$EvaluateRule^*(Rule, Data) = \frac{tp + tn}{\mathbf{P} + \mathbf{N}} \quad (4.5)$$

After calculating the evaluation value of each *InitRule*, this value will be compared with *MaxVal* found previously. If *InitRule* has a better evaluation value, the old *MaxVal* will be updated and the current *InitRule* will be set to the new *FinalRule*. The while-loop in the procedure will continue until all the conditions in *Conditions* are returned to *InitRule*. This means that the final *InitRule* should be the same as the original *Rule*. In this way, the pruning phase is done and *FinalRule* returns the rule, which has the best evaluation value.

```
procedure PruneRule(Data, Rule, useWhole) {
    Conditions = getBody(Rule)
    InitRule = getHead(Rule)
    FinalRule = InitRule
    MaxVal = DefaultMaxValue(Data)
    position = 0
    while (position < getSize(Rule))
    {
        Condition = getCondition(Conditions, position)
        InitRule = InitRule + Condition
        if (!useWhole)
            CurrVal = EvaluateRule(InitRule, Data)
        else
            CurrVal = EvaluateRule*(InitRule, Data)
        if (CurrVal > maxValue)
        {
            MaxVal = CurrVal
            FinalRule = InitRule
        }
        position++
    }
    return (FinalRule)
}
```

Algorithm 4.4: PruneRule

4.2.5 Condition Stop Criterion and Rule Stop Criterion

Condition Stop Criterion
In the growing phase, the essential task of the condition stop criterion is to examine a newly constructed *Refinement*, and then to decide whether it is useful to refine it further or not. This criterion is implemented in the procedure *CheckForStop* in class *NoNegativesCoveredStop* and can be summarized in two points:

$$\text{CheckForStop}(\text{Refinement}, \text{Val}, \text{Data})$$

1. None of the negative examples should be covered
2. More than *minNo* (default value is 2) positive examples must be covered

In pursuit of getting a rule with 100% accuracy, the refinements that cover negative examples will not be considered. Moreover, in order to ensure the coverage of each rule, a parameter *minNo* is set in our implementation. In this way, only the refinements with qualified coverage have the chance to be further refined. In addition, this parameter is changeable in the XML file.

Rule Stop Criterion
The essential task of the rule stop criterion is to examine a newly generated *Rule*, and then to decide whether it is useful to add it to the rule set or not. In addition, if the new *Rule* is not allowed to be added to the rule set, the process of searching rules is just stopped at the same time (see procedure *Separate-AndConquer*), because it is impossible to get a better rule from the remaining examples. The rule stop criterion is implemented in the procedure *CheckForRuleStop* in class *MDLStoppingCriterion* and can be summarized in three points:

$$\text{CheckForRuleStop}(\text{Theory}, \text{Rule}, \text{Data})$$

- DL > minDL + 64
- tp ≤ 0
- $\frac{fp}{covered} \geq 0.5$

The first point is implemented as described in section 3.2.1: After each *Rule* is generated, assuming that *Rule* is added to *Theory*, the total description length of the current rule set is calculated. This value is set as *DL* and *minDL* means the smallest total description length obtained so far. The value of *DL* will always be compared with that of *minDL*. If the value of *DL* is smaller, *minDL* will be updated. Conversely, if the value of *DL* is more than 64 bits [8] larger than that of $minDL$, the generated *Rule* cannot be added into *Theory* and the process of searching will be stopped as a result. The relevant calculation methods and formulas are described in section 4.3.2 in detail. According to the second and third points, if the generated *Rule* covers no positive examples or its error rate exceeds 50%, it will be considered as "unqualified" as well. The "unqualified" rule cannot be added to the rule set.

4.3 Iterative Optimization

Based on the prototype of RIPPERk, the procedure *postProcessTheory* is implemented in class *PostProcessorRipper*. This class is an implementation of the interface *IPostProcessor* predefined in SeCo. As mentioned before, k means that the rule set can be iteratively optimized for k times in the postprocessing phase. In our implementation, a parameter named *Optimizations* is used to represent k. The value of *Optimizations* is derived from the XML file, so it is easy to change it. Algorithm 4.5 shows the pseudocode of the procedure *PostProcessTheory*, where *Theory* means the initial rule set that was generated in the building phase and *Data* is the entire training set. As mentioned in section 3.2.2, in the postprocessing phase, each optimization iteration can be summarized in three parts:

1. **Optimizing each original rule in the initial rule set**
2. **Generating new rules for the remaining training set**
3. **Deleting some rules from the rule set**

In the postprocessing phase, the training set should be stratified and divided as well. The relevant procedures *Stratify* and *SplitData* were introduced in the previous section. Each rule in the initial rule set will be optimized in the order they were learned. At the beginning of each optimization iteration, the value of *position* is set to zero. The parameter *position* means which rule in the rule set is currently being handled. If its value is smaller than the size of *Theory*, the rule at this position will be removed from *Theory* and be set to *OldRule*. Because it is not necessary to optimize a rule that covers no example, the coverage of *OldRule* will be checked at first. If the coverage of *OldRule* is zero, it will be skipped directly. Then, the optimization will restart with the next rule in the rule set. (see * marked in Algorithm 4.5). By using different methods, some variants based on *OldRule* will be generated. According to the predefined selection criterion, a variant among the variants and *OldRule* will be selected as the final optimized rule and be set to *FinalRule*. In our implementation, there are three variants (i.e. *Replacement*, *Revision*, *Abridgment*) and two selection criteria (i.e. *MDL*, *Accuracy*) available. The generation of variants and the selection criteria will be described in sections 4.3.1 and 4.3.2. In addition, *FinalRule* will be added to *Theory* and all the examples covered by it should be removed from *Data*. At the same time, *position* will point to the next rule in the rule set. This means that the optimization of the current *OldRule* is finished.

Once each rule in the initial rule set is optimized likewise, a new rule set is generated. However, it is possible that some positive examples are not covered by the new rule set yet. Therefore, some new rules will be generated to cover them. If the value of *position* is not less than the size of *Theory*, some new rules

will be generated on the remaining training set successively. The process of generating new rules is the same as that in the building phase. The details of the involved procedures can be found in the previous section. The process of adding new rules will be stopped if there is no positive example remaining or the procedure *CheckForRuleStop* prevents the searching of more rules. Due to the above-mentioned process, the size of the rule set will usually get bigger.

```
procedure PostProcessTheory(Theory, Data) {
    for i < Optimizations
    {
        position = 0
        newData = Data
        while (Positive(newData) ≠ ∅) (*)
        {
            newData = Stratify(Data, GrowingSetSize, Seed)
            SplitData(GrowingSetSize, Data, GrowData, PruneData)
            if position < getSize(Theory)
            {
                OldRule = Theory(position)
                Theory = Theory \ OldRule
                if (!Cover(OldRule))
                {
                    position++
                    goto (*)
                }

                # generate some variants
                Replacement = getReplacement()
                Revision = getRevision()
                Abridgement = getAbridgement()

                # select one of them as the final rule
                FinalRule = getFinalRule(OldRule, Replacement, Revision, Abridgment)

                newData = newData \ Covered(FinalRule, newData)
                Theory = Theory ∪ FinalRule
                position++
            }
            else
            {
                newRule = FindBestRule(GrowData)
                newRule = PruneRule(PruneData, newRule)
                if CheckForRuleStop(Theory, newRule, newData)
                    exit while
                newData = newData \ Covered(newRule, newData)
                Theory = Theory ∪ newRule
            }
            Theroy = ReduceDL(Theory)
        }
        i++
    }
    return (Theory)
}
```

Algorithm 4.5: PostProcessTheory

After adding some new rules to the initial rule set, another procedure *ReduceDL* is used to delete some rules from the current rule set. The reason for this action is to ensure the quality of the final rule set. Normally, the rules that increase the total description length of the rule set will be deleted. The implementation of this procedure will be described in section 4.3.3 later. After that, an optimization iteration is then finished. The rule set stored in *Theory* can be further optimized in a new optimization iteration. In addition, the procedure *PostProcessTheory* will be stopped if the rule set is optimized for k times. Finally, the algorithm returns *Theory* as a result of the postprocessing phase.

4.3.1 Variants

In our implementation, based on each rule in the initial rule set, three variants will be generated. The first two variants *Replacement* and *Revision* are implemented according to the definition in the RIPPER algorithm (see section 3.2.2), and the variant *Abridgment* is generated by using a new pruning method (see section 3.2.3). This pruning method is guided to prune the conditions regardless of their order in the rule. In this section, the implementation of these variants will be described in pseudocode. In addition, the differences between them will be explained.

1. **Variant Replacement**

   $\text{Replacement} = \text{FindBestRule}(\text{GrowData}, \text{null})$
   $\text{PruneData} = \text{rmCoveredBySuccessives}(\text{PruneData}, \text{Theory}, \text{position})$
   $\text{Replacement} = \text{PruneRule}(\text{PruneData}, \text{Replacement}, \text{useWhole})$

   According to the pseudocode described above, the process of generating the variant *Replacement* can be summarized in the following three steps:

   a) Growing a new rule on the given growing set
   b) Updating the pruning set
   c) Pruning the grown rule on the newly constructed pruning set

   In the growing phase, the procedure *FindBestRule* will be used to grow the variant *Replacement*. The parameter *null* means that this variant should be grown from an empty rule directly. As mentioned in section 4.2.4, in the pruning phase, the pruning heuristic is guided to minimize the error of the entire rule set. However, it costs too much time to calculate this value directly. In order to get the value proportional to the error of the entire rule set, the given pruning set will be processed first. The parameter *position* means which rule in the rule set (i.e. *Theory*) is handled currently. According to the procedure *rmCoveredBySuccessives*, all the examples which are covered by the subsequent rules based on *position* should be removed from the pruning set. Finally, the grown *Replacement* will be pruned on the rest of the examples.

2. **Variant Revision**

   $\text{GrowData} = \text{CoveredByRule}(\text{GrowData}, \text{OldRule})$
   $\text{Revision} = \text{FindBestRule}(\text{GrowData}, \text{OldRule})$
   $\text{PruneData} = \text{rmCoveredBySuccessives}(\text{PruneData}, \text{Theory}, \text{position})$
   $\text{Revision} = \text{PruneRule}(\text{PruneData}, \text{Revision}, \text{useWhole})$

   According to the pseudocode described above, the process of generating the variant *Revision* can be summarized in the following four steps:

   a) Updating the growing set
   b) Growing a rule on the newly constructed growing set

c) Updating the pruning set

d) Pruning the grown rule on the newly constructed pruning set

In contrast to *Replacement*, the variant *Revision* is grown by greedily adding more conditions to the given old rule. Therefore, in the procedure *FindBestRule*, the parameter *OldRule*, which represents the old rule, is given. Moreover, in the growing phase, the grown rule should be more specific than the original one. In other words, the examples that are not covered by *OldRule* are redundant. Thus, according to the procedure *CoveredByRule*, they will be removed from the growing set in advance. The pruning phase of the variant *Revision* is the same as that of the variant *Replacement*.

3. **Variant Abridgment**

In section 3.2.3, the design and development of a new pruning method was introduced. The basic idea of this pruning method is to prune the conditions of a rule regardless of their order in the rule. By using this method, a new variant *Abridgment* can be generated on each rule of the initial rule set. In addition, because the generation of *Abridgment* does not contain the growing phase, it is not necessary to divide the training set. This means that the whole training set can be used as the pruning set. The pruning method is implemented in the procedure *PruneOldRule* in class *DefaultPostProcessor*. Algorithm 4.6 shows the pseudocode of this procedure. The parameter *Data* means the complete training set and *Rule* means the current old rule that should be pruned.

```
Abridgment = PruneOldRule(Data, Rule)

procedure PruneOldRule(Data, Rule) {
    minER = EvaluateRule(Rule, Data)
    while (getSize(Rule) > 1)
    {
        Conditions = getBody(Rule)
        cPosition = -1
        for Condition ∈ Conditions
        {
            position = getPosition(Condition)
            tempRule = deleteCondition(Rule, position)
            currER = EvaluateRule(tempRule, Data)
            if (currER ≤ minER)
            {
                cPosition = position
                minER = currER
            }
        }
        if (cPosition ≥ 0)
            Rule = deleteCondition(Rule, cPosition)
        else
            exit while
    }
    return (Rule)
}
```

Algorithm 4.6: PruneOldRule

In the pruning phase, the procedure *EvaluateRule* is used to evaluate the rule. The pruning heuristic is guided to minimize the error rate of the pruned rule. The formula of this pruning heuristic was defined in section 3.2.3. The given rule (i.e. *Rule*) will be evaluated first, and its evaluation value

will be set to *minER* as a default value. In general, the while-loop will be stopped if there is only one condition left in *Rule*. In a pruning iteration, each of the conditions in *Rule* are deleted separately, and a series of *tempRule* are generated. For example, k *tempRule* can be generated if *Rule* contains k conditions. Similarly, each *tempRule* should be evaluated in the procedure *EvaluateRule* and its evaluation value set to *currER*. This value will always be compared with *minER*. If the value of *currER* is equal or less than that of *minER*, *minER* will be updated. In addition, if no *tempRule* can improve the default error rate, the while-loop will be stopped as well. Otherwise, *Rule* will be replaced with a *tempRule*, which has the minimal error rate. This newly constructed *Rule* will be pruned in the next pruning iteration in the same way.

To illustrate what is going on, a simple example is given. Assuming that the training set (i.e. *Data*) contains 697 examples, the given *Rule* covers 647 examples correctly and 50 examples incorrectly. Thus, the error rate of *Rule* is ($\frac{50}{697} \approx 7.17\%$). This value will be set to *minER*.

$$\text{Rule}: \quad [\text{Class} = \text{c} : \text{surface_quality} = -, \text{carbon} < 0, \text{shape} = \text{SHEET}, \text{thick} \geq 1.201]$$

In the first pruning iteration, there are four conditions available. Each of them will be deleted to construct a *tempRule* respectively. The error rate of each *tempRule* is calculated and listed in the table below.

Condition Deleted	FP+FN	TP+TN	Error Rate
surface_quality = -	109	588	15.64%
carbon < 0	53	644	7.60%
shape = SHEET	46	651	6.60%
thick ≥ 1.201	25	672	3.59%

Table 4.6.: Variant Abridgment (1st Iteration)

If the condition [thick ≥ 1.201] is deleted, the error rate will decrease to 3.59%. This value is the minimal error rate in this iteration. Moreover, it is smaller than the value in *minER*. Thus, this condition is deleted and a new *Rule* $\text{Class} = \text{c} : \text{surface_quality} = -, \text{carbon} < 0, \text{shape} = \text{SHEET}$ is constructed.

Condition Deleted	FP+FN	TP+TN	Error Rate
surface_quality = -	319	378	45.77%
carbon < 0	20	677	2.87%
shape = SHEET	35	662	5.02%

Table 4.7.: Variant Abridgment (2nd Iteration)

In the second pruning iteration, the error rate can be improved further if the condition [carbon < 0] is deleted (see Table 4.7). The newly constructed *Rule* $\text{Class} = \text{c} : \text{surface_quality} = -, \text{shape} = \text{SHEET}$ is passed to the next iteration. The value in *minER* will be updated to 2.87%.

Condition Deleted	FP+FN	TP+TN	Error Rate
surface_quality = -	541	156	77.62%
shape = SHEET	80	617	11.48%

Table 4.8.: Variant Abridgment (3rd Iteration)

According to table 4.8, no improvement is possible in the third pruning iteration. The error rate

with regard to each *tempRule* is larger than that in *minER*. Thus, the procedure *PruneOldRule* will be stopped and *Rule* $\mathsf{Class} = \mathsf{c} : \mathsf{surface_quality} = -, \mathsf{shape} = \mathsf{SHEET}$ will be returned as a result.

4.3.2 Selection Criteria for Variants

After generating the variants, one of them will be selected as the optimized rule (i.e. *FinalRule* in the procedure *PostProcessTheory*). In our implementation, there are two selection criteria available; the first one prefers to select the variant which can generate a new rule set with the smallest minimum description length (MDL), while the second one simply selects the rule which has the maximal accuracy. The calculation of MDL and accuracy is implemented in the relevant procedures in class *RuleStats*.

1. **MDL Calculation**

   $\mathsf{oldDL} = \mathsf{calcMDL(Theory)}$
   $\mathsf{for}\ \mathsf{R_i} \in\ \mathsf{Variants}\ \mathbf{R}$
   $\quad \mathsf{Theory^*} = \mathsf{replaceRule(Theory, R_i, position)}$
   $\quad \mathsf{R_iDL} = \mathsf{calcMDL(Theory^*)}$
   $\mathsf{FinalRule} = \mathsf{min(oldDL, \{R_iDL\})}$

   The parameter *Theory* means the rule set which contains the original *OldRule*. The variants $\mathbf{R}$ are generated based on *OldRule* and the parameter R_i points to one of them. In addition, the parameter *position* means the position of *OldRule* in *Theory*. The procedure *replaceRule* is used to replace *OldRule* with the variant R_i and a newly constructed rule set is set to $Theory^*$. In the procedure *calcMDL*, each possible rule set (including the original *Theory*) will be evaluated based on the MDL. Finally, the rule whose generated rule set has the smallest MDL, will be considered as the best rule and be set to *FinalRule*. The essential formulas defined in the procedure *calcMDL* can be summarized as follows:

$$Potential(Theory, R_i) = DL(Theory) - DL(Theory \setminus R_i) + DL(R_i) \quad (4.6)$$

$$Potentials(Theory) = \sum Potential(Theory, R_i) \qquad R_i \in Theory \quad (4.7)$$

$$MDL(Theory) = DL(Theory) - Potentials(Theory) \quad (4.8)$$

   Normally, the minimum description length (MDL) of a certain rule set should be calculated in this way: Firstly, all the rules that increase the total description length (DL) will be removed from this rule set in advance, and the original rule set will be updated. Secondly, based on the result rule set the DL will be calculated again and this value is considered the so-called MDL. However, which rules will lead to an increase of the DL?

   In our implementation, this problem will be solved by getting the parameter *Potential*. In equation 4.6, *Potential* calculates the potential of decreasing the DL of *Theory* if the rule R_i is deleted. The equation is composed of two parts. The first part means the change of DL with regard to two rule sets, where the first one contains the rule R_i and the second one does not. The second part means the description length (DL) of the rule set (*Theory*) for a given rule (R_i). In equation 4.7, *Potential* will be summed up and be set to *Potentials*. In other words, each rule in *Theory* will be deleted once to check if the DL of *Theory* can be decreased. In this way, according to equation 4.8, the MDL of *Theory* can be calculated as well.

The formulas that are used to calculate the DL of a rule set (i.e. *DL(Theory)*) and the DL of a rule set for a given rule (i.e. *DL(R_i)*), are derived from [23].

$$DL(Theory) = \log_2(cover + uncover + 1) + S(cover, fp^*, FP/cover) + S(cover, fn^*, FN/cover) \quad (4.9)$$

$$S(n, k, p) = k * \log_2(\frac{1}{p}) + (n - k)\log_2(\frac{1}{1-p}) \quad (4.10)$$

Before calculating *DL(Theory)*, some essential information about the rule set *Theory* should be gathered. In equation 4.9, the parameter *cover* means the total number of examples covered by *Theory*, and *uncover* means the rest of them. In addition, fp^* and fn^* are two important parameters which are calculated by successively adding the number of false positive and false negative examples with regard to each rule in *Theory*. Function $S(n, k, p)$ is used to calculate the subset encoding length of k elements of a known set of n elements. In this function, p is an expected proportion which is calculated by $\frac{expected\ number\ of\ elements\ in\ the\ subset}{n}$. For example, in equation 4.9, FP and FN are two expected values. FP means the number of false positive examples, when all the positive examples are not covered by *Theory*. Similarly, FN means the number of false negative examples, when all negative examples are covered by *Theory*.

$$DL(R_i) = ||k|| + 0.5 * S(t, k, k/t) \quad (4.11)$$

$$||k|| \approx \log_2(\log_2(k)) \quad (4.12)$$

According to equation 4.11, the description length of a rule set for a given rule (DL(R_i)) is calculated. In this equation, k means the number of conditions in the rule and $||k||$ means the number of bits needed to send the integer k. According to equation 4.12, $||k||$ is only calculated as an approximation. Moreover, t means the total possible conditions that could appear in the rule. The factor 0.5 is used to adjust the possible redundancy in the attributes [28].

2. **Accuracy Calculation**

```
oldAcc = calcAcc(OldRule)
for Ri ∈ Variants R
   RiAcc = calcAcc(Ri)
FinalRule = max(oldDL, {RiAcc})
```

The basic process of accuracy calculation is the same as that in the MDL calculation. But the formula defined in the procedure *calcAcc* is much simpler (see equation 4.13).

$$Acc(R_i) = \frac{tp + tn}{\mathbf{P} + \mathbf{N}} \quad (4.13)$$

After evaluating the variants **R** and the old rule *OldRule*, the one of them, which has the maximal accuracy, will be considered as the best rule and be set to *FinalRule*.

4.3.3 Rule Reduction

As mentioned in section 3.2.2, after we optimized the rules in the initial rule set and added some new rules to cover the rest of positive examples, we will examine the new rule set again. Note that the MDL calculated in the last section is only an assuming value that a rule set can get when the "unqualified" rules

are removed. In other words, these rules haven't been removed from the rule set yet. In class *RuleStats*, a procedure *ReduceDL* is used to finish this task (see algorithm 4.7). In the MDL calculation, for each rule R_i, the parameter *Potential* means the potential that DL can be decreased if R_i is deleted. In the procedure *ReduceDL*, this value will be used as a benchmark. If the value of *Potential* is not less than 0, the relevant rule R_i will be deleted directly. In this way, all the rules that increase DL will be filtered. Finally, the filtered rule set will return as the result of an optimization iteration.

```
procedure ReduceDL(Theory) {
    for Rᵢ ∈ Theory
    {
        potential = potential(Theory, Rᵢ)
        if (potential ≥ 0)
            Theory = Theory \ Rᵢ
    }
    return (Theory)
}
```

Algorithm 4.7: ReduceDL

4.4 Differences from JRIP

The implementation of the RIPPER algorithm in the SeCo platform is called SeCoRIP. In the machine learning software Weka [33], JRIP is another version of RIPPER that was implemented by Xin Xu and Eibe Frank. In order to validate our implementation, JRIP is used to compare with SeCoRIP. In this section, some significant differences are listed.

4.4.1 Order of Classes

As mentioned in section 4.2, the procedure *SeparateAndConquer* is used to generate rules for one class. Therefore, the order of processing classes should be determined in advance. Normally, the procedure begins with the class with the least examples to the class with the most. However, the order in SeCoRIP and JRIP will be sometimes different, if several classes have the same number of examples.

```
@RELATION monk1
@ATTRIBUTE ...
@ATTRIBUTE class {0, 1}
```

Algorithm 4.8: Header Information of monk1.arff

Similar to SeCoRIP, the training set used in JRIP is derived from the ARFF file. Algorithm 4.8 shows the header information of an ARFF file *monk1*. In this file, there are two classes available. Moreover, class *0* contains the same number of examples as class *1*. In this situation, the order of these classes in SeCoRIP will always remain the same as in the ARFF file. This means that the rules for class *0* will be generated first. However, in JRIP it is possible that the order of processing classes is {1,0}, because a random function is used to determine the order of those classes that have the same number of examples. Due to the different order of processing classes, the generated rule sets in SeCoRIP might be a little different from that in JRIP.

4.4.2 Selection of Refinements

In section 4.2.3, the growing phase implemented in SeCoRIP was described. Based on a candidate rule, it is possible to generate a series of refinements. These refinements are added to *RuleList*. In SeCoRIP, the refinements in *RuleList* are always sorted in decreasing order of the foil's information gain. In other words, the refinement at the first position in *RuleList* should be the best one that has the maximal foil's information gain. Normally, this refinement will be selected for the next refining iteration. But if several refinements have the same maximal value, which one should be selected?

Normally, SeCoRIP and JRIP will select the first generated refinement that has the maximal value. However, it is difficult to guarantee that the order of refinements in SeCoRIP is the same as that in JRIP. Thus, this is the second reason that the rules in SeCoRIP are sometimes different from that in JRIP.

4.4.3 Minimal Number of Covered Examples

The parameter *minNo* is used to ensure the coverage of the generated rule. However, in SeCoRIP and JRIP, the method of using this parameter is different.

In the growing phase in SeCoRIP, the coverage of each possible refinement will be checked in the procedure *CheckForStop* (see section 4.2.5). Normally, the refinements with unqualified coverage will not be added to *RuleList* (see algorithm 4.3). As mentioned before, in *RuleList* a refinement will be selected for the next refining iteration. This refinement should not only have the maximal foil's information gain, but also a qualified coverage.

In JRIP, the coverage of each possible refinement will not be checked in advance. Normally, the refinement that has the maximal foil's information gain will be determined first. Secondly, the coverage of this refinement will be checked. Thirdly, this refinement will be refined further, if its coverage is qualified. Otherwise, it will be dropped and the searching process will be stopped directly (i.e no more rules for this class will be generated).

Based on the difference mentioned above, JRIP stops too soon if the coverage of the selected refinement is unqualified. This is the reason that SeCoRIP usually generates more rules than JRIP.

5 Evaluation

5.1 Data Sets

A series of data sets are used to validate SeCoRIP and its variants. These data sets are derived from the UCI Machine Learning Repository. *UCI is a collection of databases, domain theories and data generators that are used by the machine learning community for the empirical analysis of machine learning algorithms* [1]. Table 5.2 shows a global overview of these UCI data sets and the relevant elements are described in table 5.1. The reason we sort the data sets according to the types of attributes is to test whether RIPPER is good at handling numeric attributes as well as nominal ones. Moreover, not all the data sets we selected are complete. Some of them are affected by missing values.

Element	Description
Name	@relation (data set name) in the ARFF file
Examples	total number of examples in the data set
Attributes	total number of existing attributes
Nominal	number of nominal attributes
Numeric	number of numeric attributes
Classes	number of possible classes
Missing Value?	whether some values of the attributes are missing or not
Type	type of data set
Categorical	involved data sets have only nominal attributes
Numerical	involved data sets only have numeric attributes
Mixed	involved data sets have both nominal and numeric attributes

Table 5.1.: Legend of Properties in Data Sets

5.2 Evaluation Methods

Cross-Validation is a common evaluation method to evaluate the rule learning algorithms. In this section, the different types of cross-validation will be introduced.

- **K-fold cross-validation**
  In this method, the original data set, which contains n examples, will be divided into k subsets. Each subset is termed as a fold and the number of examples in each fold is almost the same. Then the rule learning algorithm will be executed k times. Each time, one of the k folds is used as the test set and the other *k-1* folds form the training set. During the execution, the rule set is learned from the training set and validated on the testing set. The result of the validation, i.e. the accuracy of the learned rule set, is calculated. By averaging the k results, a value of average accuracy is computed. This value can be seen as the single estimation for the algorithm with regard to the data set.

- **K-fold stratified cross-validation**
  According to k-fold cross-validation, the examples are distributed in k folds randomly. However, this method would cause an unreasonable distribution of classes. For example, there are two classes

Name	Examples	Attributes	Nominal	Numeric	Classes	Missing Value?	Type
kr-vs-kp	3196	36	36	-	2	no	Categorical
tic-tac-toe	958	9	9	-	2	no	
titanic	2201	3	3	-	2	no	
breast-cancer-data	286	9	9	-	2	yes	
congress-voting-1984-1	435	15	15	-	2	yes	
congress-voting-1984	435	16	16	-	2	yes	
mushroom	8124	22	22	-	2	yes	
soybean	683	35	35	-	19	yes	
audiology	226	69	69	-	24	yes	
glass2-database	163	9	-	9	2	no	Numerical
iris	150	4	-	4	3	no	
wine	178	13	-	13	3	no	
glass-database	214	9	-	7	7	no	
hepatitis	155	19	13	6	2	yes	Mixed
horse-colic-data	368	22	15	7	2	yes	
hypothyroid	3163	25	18	7	2	yes	
sick-euthyroid	3163	25	18	7	2	yes	
lymphography-data	148	18	15	3	4	no	
anneal	798	38	6	32	6	no	
cleveland-14-heart-disease	303	13	7	6	5	yes	

Table 5.2.: 20 UCI Data Sets

{A, B} available. All the examples with class *A* are distributed in the first *k-1* folds, which form a training set. The remaining examples with class *B* are in the last fold, which is used as a testing set. Because the testing set has no examples with class *A*, it is impossible to validate the learned rule set effectively. Thus, the estimation for the algorithm will be affected. In k-fold stratified cross-validation, the data set should be stratified before it is divided. This means that the examples with different classes will be distributed in the folds equally. In this way, the problem of the class distribution is resolved and the variance of the estimation can be decreased.

- **Leave-one-out cross-validation**
  This method is a variant of k-fold cross-validation where the parameter k is replaced with n (n indicates the number of examples in the data set). This means that each example can be considered as a fold. The rule set is learned from (n-1) examples. The last example is responsible for the validation of the rule set. Because the testing set has only one example, it is useless to stratify the data set. Therefore, the problem of the class distribution may exist and the results of validations may be incorrect. Moreover, due to the multi-division, the computational cost of this method is always very high and the computing time is much longer.

As mentioned in section 3.1.1, the package *evaluations* contains some evaluation methods for evaluating the rule learning algorithms implemented in SeCo. The evaluation method cross-validation is included. Based on the comparison mentioned above, k-fold stratified cross-validation is a more reasonable evaluation method. Furthermore, in [19] it is confirmed that "k=10" is an optimal value for cross-validation. Thus, 10-fold stratified cross-validation is used to evaluate our implementations.

5.3 Evaluation Dimensions

In SeCo, we implemented SeCoRIP and its two variants. In order to compare them with each other, they will be used to learn rules from a series of data sets. With the help of the cross-validation method, there are sufficient testing sets available to evaluate the learned rules. In this section, the evaluation dimensions with regard to the learned rules will be explained.

5.3.1 Correctness

Similar to the training set, in the testing set the class of each example is known. The correctness means what percentage of the examples in the testing set are classified correctly. As mentioned in section 5.2, in the evaluation method cross-validation, the value of average accuracy will be calculated. This value can be a single estimation of the correctness for a rule learning algorithm. Because we employ the 10-fold stratified cross-validation, the formula can be summarized in the following equation:

$$Correctness = \frac{\sum_{k=1}^{10} Average_k}{10} \tag{5.1}$$

5.3.2 Size of Rule Sets

The second evaluation dimension is the size of the constructed rule sets. This means the number of rules are needed to cover all the examples of the training set. According to the separate-and-conquer strategy, it is guaranteed that each example should be covered by at least one rule of a rule set. However, it is possible that a certain example will be covered by several rules if the rule set contains too many rules. Normally, we prefer to construct a rule set which uses fewer rules to cover the same number of examples correctly. Moreover, if the size of the rule set is smaller, it is easier to understand it. Note that some data sets could contain more than two classes. This means that several rule sets will be constructed for covering the examples with the relevant classes. Thus, the formula used in this evaluation dimension is defined as follows:

$$Size\ of\ Rule\ Sets = the\ sum\ of\ all\ rules\ in\ the\ constrcuted\ rule\ sets \tag{5.2}$$

5.3.3 Number of Conditions in One Rule

Improving a rule until it works perfectly on the training set and makes no errors would cause an overfitting problem. Normally, this rule would be more specific than the actual searched one. This means that the complicated rule could have a low coverage. In rule learning algorithms, many different methods are used to avoid the overfitting problem. Usually, we prefer to take a shorter rule which covers the same number of examples as the longer one. The number of conditions in a rule is defined as the third evaluation dimension. This value will be calculated according to the following equation:

$$Number\ of\ Conditions\ in\ One\ Rule = \frac{the\ sum\ of\ all\ conditions}{the\ size\ of\ the\ rule\ sets} \tag{5.3}$$

5.4.1 Results of SeCoRIP

As mentioned in section 4.3, SeCoRIP employs a postprocessing procedure for optimizing a rule set right after it was learned. In this section, we show the differences of SeCoRIP with and without the postprocessing on 20 UCI data sets (see table 5.2). The complete learning algorithm will be executed six times in total. Each time, the rule set learned in the building phase will be iteratively optimized in the postprocessing phase. The relevant number of optimization iterations is dependent on an external parameter *optimizations*. Here we call the algorithm SeCoRIP_0 if no optimization is processed in the postprocessing phase (i.e. $optimizations = 0$). Correspondingly, SeCoRIP_i means that the learned rule set will be iteratively optimized i times (i.e. $optimizations = i \in \{1, 2, 3, 4, 5\}$). The other parameters used for this experiment are described as follows: $growingSetSize = 3$, $minNo = 2$, $seed = 1$, $abridgment = 0$ and $selection = MDL$. The definition of these parameters and their values can be found in section 4.1.2. In this experiment, it is expected that SeCoRIP_i could always construct better rule sets than SeCoRIP_0.

According to the experimental results in table A.1, a simple line chart is given for all data sets. In figure 5.1, the x-axis represents the number of optimization iterations (i.e. the parameter *optimizations*) and the y-axis represents the *correctness* of the relevant rule sets. From the figure we can see that most of the lines present an upward trend. This means that the rule sets are really optimized in the postprocessing phase. In order to show the results more clearly, we construct a win-tie-loss list where those optimized rule sets (i.e. $optimizations \in \{1, 2, 3, 4, 5\}$) are separately compared to the rule sets with the parameter $optimizations = 0$.

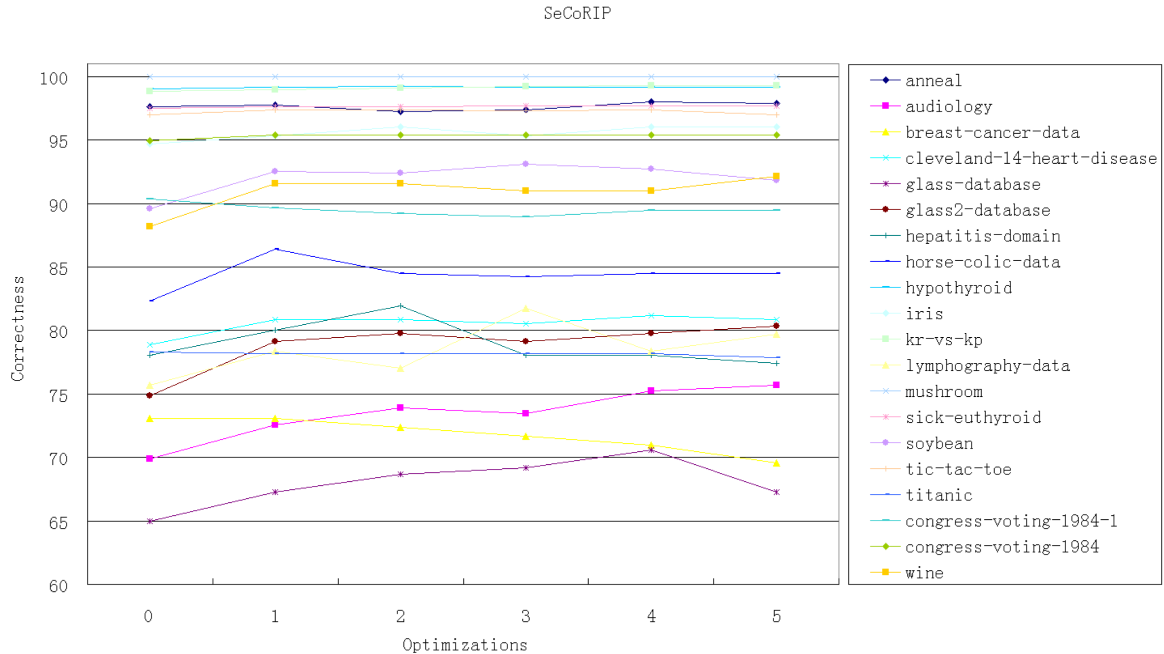


Figure 5.1.: SeCoRIP of 20 UCI Data Sets

For example, the second row in table 5.3 means that rule sets iteratively optimized twice in the postprocessing phase can get an increased correctness in 15 data sets, a worse correctness in four data sets and the correctness will not be changed in one data set.

Generally speaking from the entire table, the postprocessing procedure in SeCoRIP is able to increase the correctness of the learned rule sets in 16 data sets and the decrease of the correctness is mainly focused on the following three data sets: *breast-cancer-data*, *congress-voting-1984-1* and *titanic*. On the one hand, the type of these three data sets is *Categorical* (i.e. contain only nominal attributes). It can therefore be assumed that the postprocessing procedure is better at handling numeric attributes than nominal ones. On the other hand, in these data sets the rule sets constructed by SeCoRIP_0 always keep the best value of the correctness. Accordingly, we can give a conclusion here: If the correctness of the rule sets constructed by SeCoRIP_1 is worse than that of rule sets constructed by SeCoRIP_0, the postprocessing procedure is more likely to be useless for the entire rule learning algorithm. Moreover, in the data set *mushroom*, the correctness of the rule sets constructed by SeCoRIP_0 is already perfect, so there is no substantial room for improvement. Thus, it is also not necessary to employ a postprocessing procedure in such a condition.

	SeCoRIP_0
SeCoRIP_1	16-2-2
SeCoRIP_2	15-1-4
SeCoRIP_3	14-2-4
SeCoRIP_4	15-2-3
SeCoRIP_5	14-2-4

Table 5.3.: Win-Tie-Loss

Secondly, we try to check the profit of the correctness that the postprocessing procedure gained. In table 5.4, the third column shows the mean value of the correction on the 20 data sets and the fourth column gives the profit according to the following equation:

$$Profit_{(i+1)} = \frac{Avgc_{(i+1)} - Avgc_i}{Avgc_i} \qquad i \in \{0, 1, 2, 3, 4\} \tag{5.4}$$

This equation calculates the profit that the postprocessing procedure gained when it optimizes the learned rule sets more times. In this table, SeCoRIP_1 gets the best improvement and its *Profit* is 1.59%. This means that the correctness of the learned rule sets can be much improved even if they have been optimized in the postprocessing phase only once. However, it's interesting that not all the values of *Profit* are positive numbers. For example, the relevant *Profit* in the fourth and sixth rows is -0.08% and -0.21% respectively. Moreover, the maximum mean value of the correctness appears in the fifth rows and the corresponding *Profit* is 0.12%. According to the above, it can be assumed that the postprocessing procedure with a higher number of optimization iterations could further improve the correctness of the relevant rule sets, but it cannot be guarantied that the correctness would be increased monotonically. In order to verify this assumption, we will investigate the convergence properties of the SeCoRIP algorithm in section 5.4.5.

Algorithm	Opt.	AvgCorr.	Profit
SeCoRIP_0	0	86.19	-
SeCoRIP_1	1	87.56	1.59%
SeCoRIP_2	2	87.61	0.06%
SeCoRIP_3	3	87.53	-0.08%
SeCoRIP_4	4	87.64	0.12%
SeCoRIP_5	5	87.45	-0.21%

Table 5.4.: Profit

Finally, we compare the number of rules and conditions between the original and the optimized rule sets. The relevant results are given in table A.3. In order to show these results more clearly, the average number

of rules and the average number of conditions in one rule are calculated based on the 20 data sets. In table 5.5, it is shown that $SeCoRIP_0$ constructs about 9 rules for covering all the examples of the training set, while $SeCoRIP_1$ only need about 7 rules. Moreover, the number of conditions in one rule is also decreased by 0.29. However, no obvious changes of these values could be found with the increasing of the number of optimization iterations. According to the results in this table, it can be confirmed that the number of rules and conditions can be decreased when the SeCoRIP algorithm employs the postprocessing procedure.

Algorithm	Opt.	AvgRules.	AvgCond. in one Rule
$SeCoRIP_0$	0	8.75	1.94
$SeCoRIP_1$	1	7.35	1.65
$SeCoRIP_2$	2	7.25	1.69
$SeCoRIP_3$	3	7.40	1.73
$SeCoRIP_4$	4	7.55	1.73
$SeCoRIP_5$	5	7.50	1.73

Table 5.5.: The Average Number of Rules and Conditions (SeCoRIP)

5.4.2 Comparison with JRIP

JRIP is another version of the RIPPER algorithm that is implemented in the Weka software [33]. In order to validate the correctness of our implementation, we try to compare our results with JRIP. Similarly, we execute JRIP in Weka six times and the used parameters are the same as in SeCoRIP.

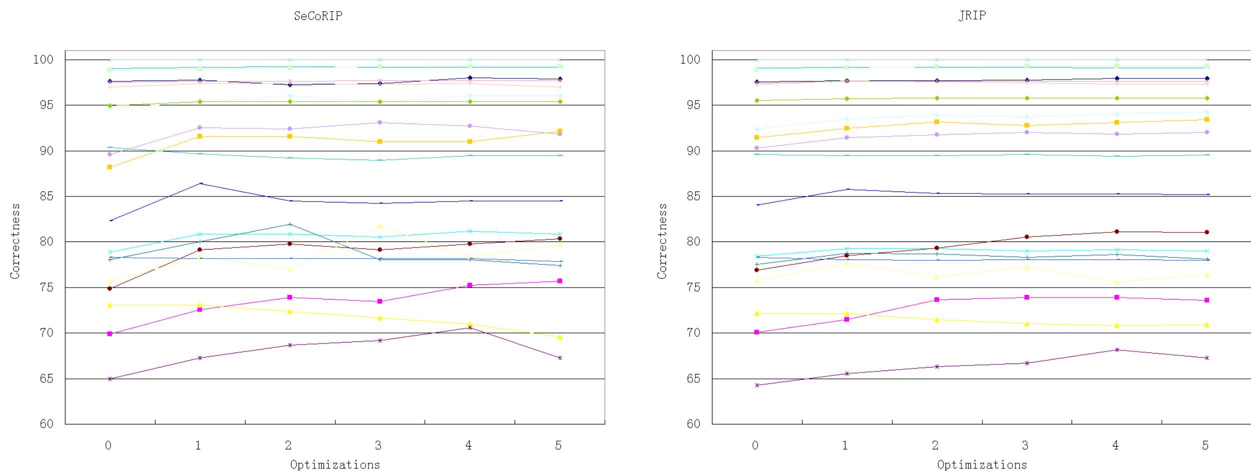


Figure 5.2.: Comparison with JRIP

Table A.2 gives the relevant experimental results of JRIP and the right line chart in figure 5.2 is constructed based on it. Because there are some implementation differences between JRIP and SeCoRIP (see section 4.4), the concrete trend of lines in two line charts could not be exactly the same. However, we find that the lines constructed by JRIP seem to be more smooth than the ones constructed by SeCoRIP. For example, the variation of the correctness in the data sets *horse-colic-data* (the dark blue one) and *glass-database* (the purple one) is more significant in the left line chart.

By comparing two line charts, we find that JRIP and SeCoRIP have three similarities: Firstly, most of the rule sets can be optimized well in the postprocessing phase and there exists an observable increase

of the correctness at the x-axis *optimizations* $\in \{1, 2\}$. Secondly, both JRIP and SeCoRIP cannot work well with the data sets: *breast-cancer-data*, *congress-voting-1984-1* and *titanic*. According to their concrete experimental results in table A.2, we can confirm our conclusion mentioned in the previous section: The postprocessing procedure is more likely to be useless for the entire rule learning algorithm if the correctness of the rule sets constructed by SeCoRIP_1 is worse than that of rule sets constructed by SeCoRIP_0. Thirdly, based on the number of optimization iterations, the correctness of the optimized rule sets is not increased monotonically. For example, the line of the data set *wine* (the dark yellow one) has a lower point at the x-axis *optimizations = 3*. Moreover, the line of the data set *lymphography-data* (the straw yellow one) is even similar to a wave curve.

In addition, we focus on the experimental results of the following seven data sets: *cleveland-14-heart-disease*, *hepatitis-domain*, *horse-colic-data*, *hypothyroid*, *lymphography-data*, *tic-tac-toe* and *congress-voting-1984*. In JRIP, the rule sets learned from these data sets can get a maximum value of the correctness, so long as they have been optimized at most two times (i.e. *optimizations = 1 or 2*). In other words, the correctness of these rule sets cannot be further increased, though they could be processed in the postprocessing phase more times. Similarly, this situation occurs in the five of the above-mentioned data sets in SeCoRIP. However, the rule sets learned from the data sets *lymphography-data* and *cleveland-14-heart-disease* get their maximum values at the third and fourth optimization iteration respectively. Referring to table 5.2, we find that five of the seven date sets has the type *Mixed* (i.e. contain both nominal and numeric attributes) and the type of the remaining two is *Categorical* (i.e. contain only nominal attributes). Furthermore, the number of nominal attributes is obviously more than that of numeric attributes in these *Mixed* data sets. In the previous section we assumed that the postprocessing procedure could be better at handling numeric attributes than nominal ones, because the decrease of the correctness only occurs in the data sets whose type are *Categorical*. Here this assumption can be further confirmed according to the distribution of nominal and numeric attributes.

5.4.3 Results of Variant Abridgment

In this section we show the results of our first variant. In this variant a new pruning method (see section 3.2.3) is introduced to generate a rule called *abridgment* in the postprocessing phase. In this pruning method, the conditions of a rule can be pruned regardless of their order in the rule. According to the feature mentioned above, this pruning method should have more effect on the rules which contain more conditions. This is because such rules have more possibilities to be pruned.

Compared to the default parameter setting in SeCoRIP, the different parameter used here is *abridgment = 1*. In this experiment, we also execute the learning algorithm on the 20 UCI data sets six times and the relevant experimental results are given in table A.4. Based on this table, we can see that the values of the correctness are only changed in nine of the given data sets. In the following table, these data sets are marked in gray.

In table 5.6, the fifth and sixth columns respectively present the total number of rules and conditions of the rule sets that are constructed in the original algorithm SeCoRIP_0. According to these two columns, the last column in the table calculates the number of conditions in one rule. Note that the calculated value is a mean value; the number of conditions in a concrete single rule can therefore be fewer or more. The data sets in this table are sorted in increasing order of the values in the last column. This table shows that the data sets marked in gray usually have a relatively high value. As mentioned in section 4.3.1, the variant *abridgment* is constructed by pruning each old rule that was learned in the building phase. On the one hand, it is difficult to prune an old rule if it contains too few conditions. The result of the pruned rule is more likely just the original one. Thus, the new pruning method will take no effect in such a case. On the other hand, in the postprocessing phase the variant *revision* is grown from the old rule and then pruned by

Name	Examples	Attributes	Rules ($SeCoRIP_0$)	Conditions ($SeCoRIP_0$)	Number of Conditions in One Rule
horse-colic-data	368	22	2	2	1.00
iris	150	4	3	3	1.00
wine	178	13	3	3	1.00
breast-cancer-data	286	9	4	5	1.25
mushroom	8124	22	9	12	1.33
anneal	798	38	9	13	1.44
glass2-database	163	9	6	9	1.50
titanic	2201	3	4	6	1.50
hepatitis	155	19	3	5	1.66
lymphography-data	148	18	9	15	1.67
congress-voting-1984	435	16	4	7	1.75
soybean	683	35	29	54	1.86
audiology	226	69	21	40	1.90
hypothyroid	3163	25	4	8	2.00
glass-database	214	9	16	39	2.44
cleveland-14-heart-disease	303	13	8	21	2.63
tic-tac-toe	958	9	10	28	2.80
kr-vs-kp	3196	36	17	56	3.29
congress-voting-1984-1	435	15	6	20	3.33
sick-euthyroid	3163	25	8	28	3.50

Table 5.6.: 9 Data Sets Marked in Gray

using the old pruning method. It is also possible that the constructed *abridgment* is the same as *revision*. In this case, the experimental results will not be changed, though the new pruning method is used.
Secondly, we compare the experimental results of the first variant with that of the original SeCoRIP algorithm. In figure 5.3, the x-axis represents the number of optimization iterations and the y-axis represents the *average correctness*, which is the mean value of the *correctness* of the nine data sets. Based on this figure, we can see that the first variation cannot work as well as the original SeCoRIP, because its relevant values are obviously worse. For example, at the x-axis *optimizations = 1*, the value for the original SeCoRIP is 85.95, while the first variant gets only 85.48. This means that there is a correctness difference of about 0.47. Moreover, the largest correctness difference between these two algorithms is even 0.65 at the x-axis *optimizations = 3*.

Referring to table A.4 and table 5.6, we find that the one and only highlight of this variation is focused on the two data sets *kr-vs-kp* and *sick-euthyroid*. The correctness of the optimized rule sets learned from them has noticeably increased. Note that the number of conditions in one rule corresponding to these two data sets is extremely high; the first one reaches 3.29 and the second gets even the largest number 3.50. This means that each rule of rule sets should have at least three or four conditions. This provides substantial search space for the new pruning method. Based on the results mentioned above, it can be assumed that the new pruning method could have a positive impact on rule sets whose rules normally contain more than three conditions. However, the correctness in the data *congress-voting-1984-1*, which has the second largest value of 3.33, is not increased yet. We find that the higher value here is only a special case, because it will be reduced substantially, if we disrupt the order of examples in the relevant data set randomly. In other words, this value should not be so high in normal cases.

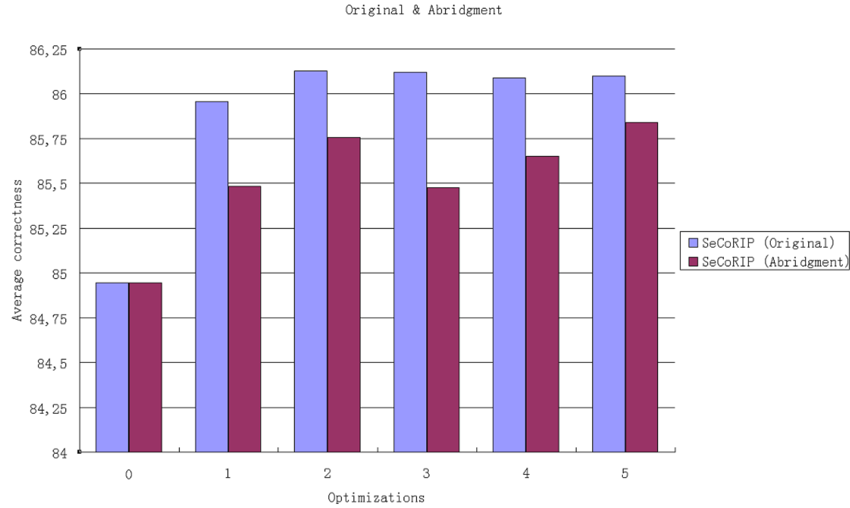


Figure 5.3.: Comparison of Average Correctness

5.4.4 Results of Simplified Selection Criterion

In this section we show the results of our second variant. In this variant the heuristic *accuracy* is used instead of *MDL* in the selection criterion. This means that in the postprocessing phase the variant (i.e. among *replacement*, *revision* and the original *old rule*) with the maximal accuracy will be selected as the best optimized rule at the end. Because the calculation of *MDL* is relatively complex, the goal of this experiment is to check whether the second variant with a simplified selection criterion can get similar results compared with the original SeCoRIP. In this experiment, the second variant will be executed six times on the 20 UCI data sets and the different parameter declared in the XML file is *selection = Accuracy*.

Here we simply call SeCoRIP'$_i$ the second variant with *optimizations = i*. Moreover, SeCoRIP'$_0$ is the same as SeCoRIP$_0$, because both of them contain no postprocessing phase. The concrete experimental results of SeCoRIP' are presented in table A.5. Based on this table, we construct a win-tie-loss list where those optimized rule sets (i.e. *optimizations* $\in \{1, 2, 3, 4, 5\}$) are separately compared to the rule sets with the parameter *optimizations = 0*.

	SeCoRIP'$_0$
SeCoRIP'$_1$	12-1-7
SeCoRIP'$_2$	12-3-5
SeCoRIP'$_3$	14-2-4
SeCoRIP'$_4$	12-2-6
SeCoRIP'$_5$	13-3-4

Table 5.7.: Win-Tie-Loss 2

In table 5.7 we can see that the postprocessing procedure of SeCoRIP', which takes the simplified selection criterion, can regularly improve the correctness of rule sets in about 13 data sets. Compared to the results of SeCoRIP in table 5.3, this modified postprocessing procedure is not satisfactory. The original SeCoRIP can get the improvement of the correctness in about 16 data sets and the correctness deterioration is mainly focused on the data sets *breast-cancer-data*, *congress-voting-1984-1* and *titanic*. However, in SeCoRIP', the rule sets learned from the data sets *anneal* and *lymphography-data* can no longer be improved. Moreover, the correctness of the rule sets learned from the data set *kr-vs-kp* decreases noticeably.

According to the experimental results in table A.1 and A.5, we construct another win-tie-loss list which shows an intuitive comparison of these two algorithms. Note that SeCoRIP' will only be compared to SeCoRIP with the same number of optimization iterations. For example, the first row in the following table means that SeCoRIP'$_1$ outperforms SeCoRIP$_1$ in four data sets, gets a worse result in 12 data sets and has a tie in four data sets.

Variant	Win-Tie-Loss	Original
SeCoRIP'$_1$	4-4-12	SeCoRIP$_1$
SeCoRIP'$_2$	7-3-10	SeCoRIP$_2$
SeCoRIP'$_3$	11-2-7	SeCoRIP$_3$
SeCoRIP'$_4$	10-1-9	SeCoRIP$_4$
SeCoRIP'$_5$	8-3-9	SeCoRIP$_5$

Table 5.8.: Win-Tie-Loss (Comparison of SeCoRIP and SeCoRIP')

In table 5.8 we can see that most of the rule sets constructed by SeCoRIP' are worse than those constructed by SeCoRIP in the case of the postprocessing procedure with a low number of optimization iterations. This situation will be changed when the number of optimization iterations is increased. The third and the fourth rows in the table show that SeCoRIP' can extract better rule sets from at least 10 data sets. This change is also observable in figure 5.4. In this figure, the x-axis represents the number of optimization iterations and the y-axis represents the *average correctness*, which is the mean value of the *correctness* on the 20 given data sets. At the x-axis *optimizations = 1*, the *accuracy correctness* for the original SeCoRIP is 87.56 while the second variant SeCoRIP' gets only 87.00. The correctness difference between them is about 0.56. With the increasing of the number of optimization iterations, this difference is reduced gradually. The correctness difference is 0.48 at the x-axis *optimizations = 2*. Furthermore, at the x-axis *optimizations = 3*, SeCoRIP' gets its best *accuracy correctness* 87.45 and the correctness difference at this point is only 0.09. This is a very good result for SeCoRIP'. However, the results of SeCoRIP' are not consistent, because the correctness difference is extended at the x-axis *optimizations = 4* once again.

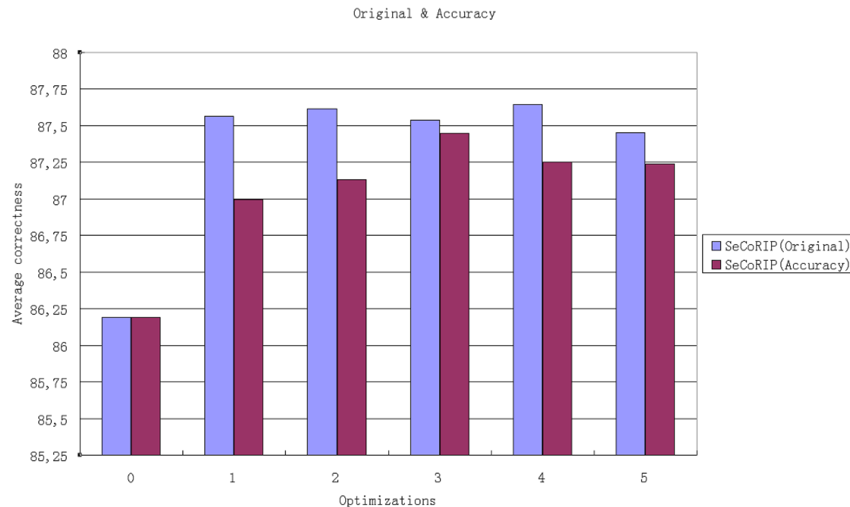


Figure 5.4.: Comparison of Average Correctness 2

On the other hand, the number of rules and conditions is also decreased when the relevant rule sets have been optimized in the postprocessing phase of the SeCoRIP' algorithm. The relevant experimental results are given in table A.6. Based on this table, the average number of rules and the average number of condi-

tions in one rule are calculated. In table 5.9, it is shown that those optimized rule sets usually take 2 rules less than the original rule sets. Moreover, the number of conditions in one rule is decreased by about 0.2. These results are similar to that of the original SeCoRIP algorithm.

Algorithm	Opt.	AvgRules.	AvgCond. in one Rule
SeCoRIP'$_0$	0	8.75	1.94
SeCoRIP'$_1$	1	7,05	1.70
SeCoRIP'$_2$	2	7.00	1.72
SeCoRIP'$_3$	3	7.25	1.74
SeCoRIP'$_4$	4	7.05	1.74
SeCoRIP'$_5$	5	7.25	1.77

Table 5.9.: The Average Number of Rules and Conditions (SeCoRIP')

Based on the comparison results mentioned above, the following conclusions can be made for the second variant: Firstly, the postprocessing procedure with a simplified selection criterion still has the ability to improve the correctness of the given rule sets. Moreover, the optimized rule sets usually contain fewer rules and the number of conditions in each rule will be decreased as well. Secondly, not all the data sets that can be processed well in SeCoRIP are also suitable for SeCoRIP' (e.g. the data sets *anneal*, *lymphography-data* and *kr-vs-kp*). Thirdly, SeCoRIP' can not work as well as SeCoRIP, because the rule sets constructed by SeCoRIP' are often worse than those constructed by SeCoRIP. However, this difference can be reduced with the increasing of the number of optimization iterations.

5.4.5 Convergence Properties of SeCoRIP

In this section, we discuss the convergence properties of the SeCoRIP algorithm for the increasing of the number of optimization iterations. In order to collect more information, we execute SeCoRIP$_i$ on the 20 UCI data sets an extra five times (i.e. *optimizations* = i ∈ {6, 7, 8, 9, 10}) and the relevant experimental results are presented in table A.7.

In Wikipedia, the definition of convergence is described as follows: "Convergence is the approach toward a definite value, a definite point, a common view or opinion, or toward a fixed or equilibrium state" [9]. In order to get a better understanding, we give a picture that shows a simple explanation of the definition of convergence. In this figure, it is observed that those red points converge to a constant point in the positive direction of the x-axis.

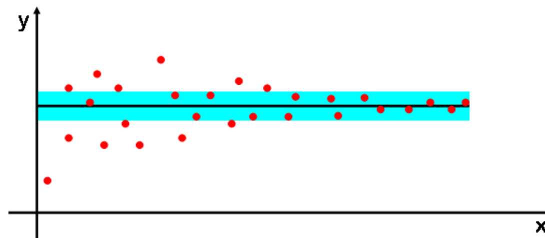


Figure 5.5.: A Picture of the Definition of Convergence [10]

According to the experimental results in table A.1 and A.7, we find that the changing trend of the correctness for the 20 UCI data sets are diverse. In order to make it easy for analyzing, the given data sets are divided into several subgroups, in which the relevant data sets have a similar changing trend of the correctness.

- **Group A**

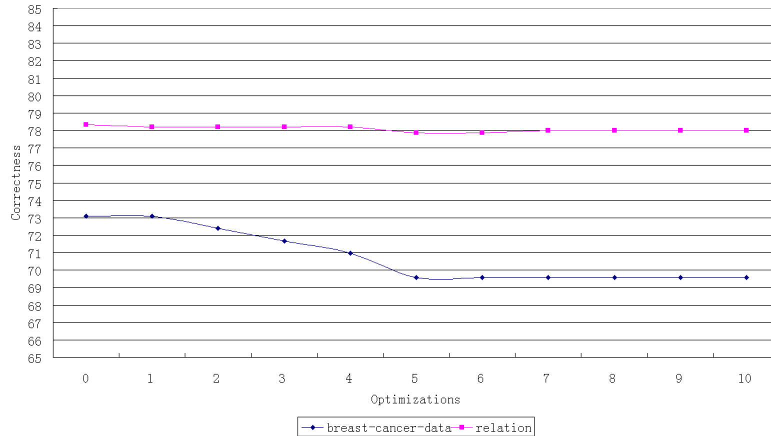


Figure 5.6.: Group A

The first group contains the following data sets: *breast-cancer-data*, *titanic* and *congress-voting-1984-1*. The feature of these data sets is that they only contain nominal attributes and the number of nominal attributes is relatively small. It is interesting that the rule sets learned from these data sets can never be improved in this experiment. Moreover, with the increasing of the number of optimization iterations, the correctness of the relevant rule sets is gradually decreased and approaches a constant value at the end. Actually, we can say that the postprocessing procedure of SeCoRIP has a negative effect in such a case.

Figure 5.6 shows a representative sample of two interpolated lines that are constructed based on the relevant data sets. In this figure, the maximal value of the line always appears at the start point of the x-axis (i.e. *optimizations = 0*). On the other hand, it is shown that the points of the red line converge to a constant point with the value 78.01 and the points of the blue line converge to a constant point with the value 69.58. Note that the value of the constant point is actually not the minimal value.

- **Group B**

The second group contains the following data sets: *tic-tac-toe*, *cleveland-14-heart-disease*, *horse-colic-data*, *hypothyroid*, *hepatitis-domain* and *congress-voting-1984*. In this group, the relevant data sets usually contain more nominal attributes than numeric ones. With the help of the postprocessing procedure, it is confirmed that the rule sets learned from these data sets can be much improved. The changing trend of the correctness in this group can be concluded as follows: With the increasing of the number of optimization iterations, the correctness of the relevant rule sets is strongly increased and then gradually decreased until it approaches a constant value. In such a case, it is suggested that the number of optimization iterations should be limited to a smaller value.

Figure 5.7 shows a representative sample of two interpolated lines that are constructed based on

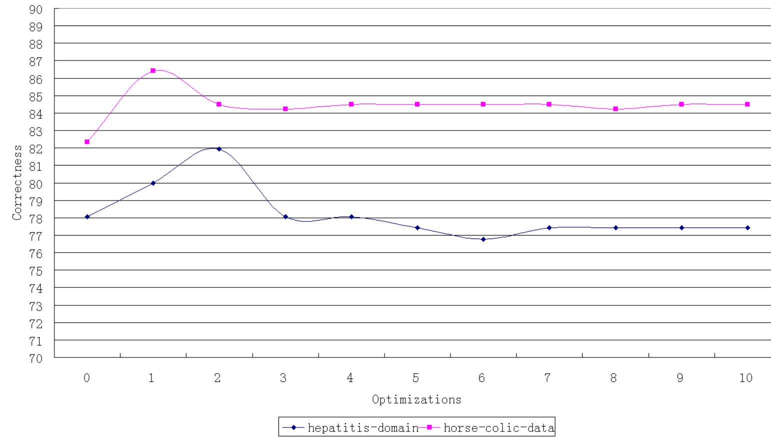


Figure 5.7.: Group B

the relevant data sets. In this figure, the maximal value of the line mainly appears at the x-axis *optimizations* $\in \{1, 2\}$. On the other hand, it is shown that the points of the red line converge to a constant point with the value 84.51 and the points of the blue line converge to a constant point with the value 77.42.

- **Group C**

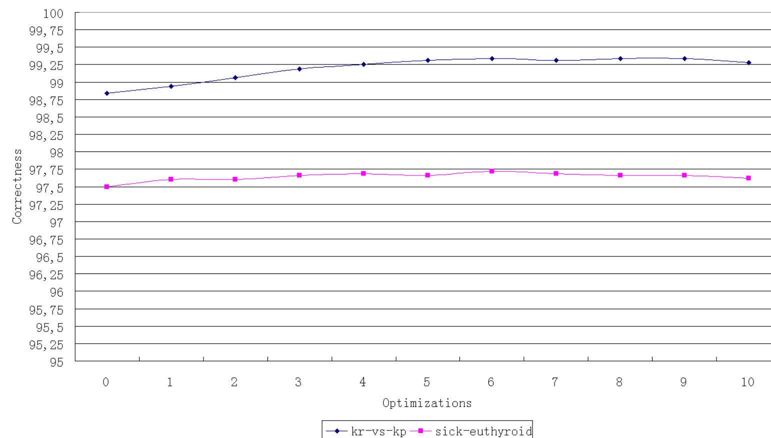


Figure 5.8.: Group C

The third group contains the following data sets: *audiology*, *kr-vs-kp* and *sick-euthyroid*. In this group, the growth of the correctness is relatively stable and regular, which means that the rule sets can be further improved when they have been optimized in the postprocessing procedure more times. The changing trend of the correctness can be concluded as follows: With the increasing of the number of optimization iterations, the correctness of the relevant rule sets is gradually increased and approaches a constant value at the end. In such a case, it is worthwhile increasing the number of optimization iterations to get a better result.

Figure 5.8 shows a representative sample of two interpolated lines that are constructed based on the relevant data sets. In this figure, the maximal value of the line mainly appears at the x-axis

optimizations $\in \{5, 6, 7\}$. On the other hand, it is shown that the points of the red line converge to a constant point with the value 97.66 and the points of the blue line converge to a constant point with the value 99.34.

- **Group D**

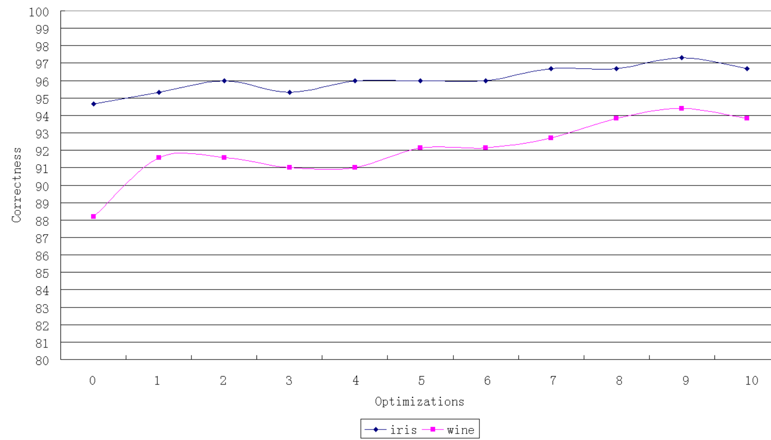


Figure 5.9.: Group D

The last group contains the remaining data sets: *anneal*, *glass2-database*, *iris*, *wine* and *soybean*. The relevant data sets mentioned above usually contain more numeric attributes than nominal ones. Moreover, the data sets that contain only numeric attributes are included as well. In this group, the rule sets learned from these data sets can also be further improved when they have been optimized in the postprocessing procedure more times. However, compared to the third group, the correctness of the relevant rule sets is gradually increased without approaching a constant value. This means that there is no signal of convergence to be found in such a case. Figure 5.9 shows a representative sample of two interpolated lines that are constructed based on the relevant data sets. In this figure, we cannot give the interval in which the maximal value mainly appears, because these two lines show a upward trend in the interval [8, 10] on the axis.

6 Summary and Conclusions

In rule learning, rule sets learned from the data are used to classify the previously unseen examples of the new data. In order to get a good rule set, most of the rule learning algorithms focus on how to learn a rule from the data more effectively, while some other algorithms attempt to optimize the rule set after it was learned. RIPPER is a popular algorithm which employs a special postprocessing procedure. The main feature of this postprocessing procedure is that it enables the iterative optimization of rule sets. In this thesis, we implemented this algorithm for analyzing the efficiency of the postprocessing procedure.

Based on our experimental results, it was confirmed that the postprocessing procedure of the RIPPER algorithm can effectively improve the given rule set in most cases. Compared to the original rule set, the optimized one can classify the examples of the testing set more accurately. The features of the optimized rule set can be concluded as follows: Firstly, the size of the optimized rule set is always smaller than that of the original one. Secondly, the number of conditions in each rule of the optimized rule set is decreased. Accordingly, the coverage of the relevant rule is increased. Thirdly, for the entire optimized rule set, the increment of the covered positive examples is usually more than the negative ones. According to the first two points it is shown that the overfitting problem can be resolved well in the postprocessing procedure of the RIPPER algorithm. However, the RIPPER algorithm also has a weakness. In our experiments it was shown that the rule set learned from the data set containing only the nominal attributes cannot always be processed well in the postprocessing procedure. Sometimes it is also possible that the optimized rule set has a lower accuracy than the original one.

Secondly, two variants based on the original RIPPER algorithm are presented in this thesis. In the first variant a new pruning method is used to construct a rule called *abridgment* in the postprocessing procedure. In this pruning method, the conditions of a rule can be pruned regardless of their order in the rule. Compared to the original pruning method, this new one enlarges the search space of the given rule. However, it was confirmed that this method has no significant effect on the rule, which contains too few conditions. Conversely, the postprocessing procedure with the new pruning method can often get a better result than the original one, if the rules of the given rule set contain sufficient conditions. In other words, the optimized rule set of the first variant can achieve higher accuracy in such a case.

In the second variant, the postprocessing procedure employs a simplified selection criterion in which the original heuristic *minimum description length* is replaced with the heuristic *accuracy*. By comparing the experimental results of these two algorithms, we find that the second variant can not work as well as the original algorithm, especially when the postprocessing procedure takes a low number of optimization iterations. The rule set constructed by this variant usually has lower accuracy. However, this difference can be slightly reduced when the number of optimization iterations is increased. The another problem is that not all the data sets that can be processed well in the original algorithm are also suitable for this variant. Based on these comparison results, we confirmed that the heuristic *minimum description length* is not easily replaceable, although its computing formula is relatively complex.

The last important result in this thesis is the convergence properties of the RIPPER algorithm. On the one hand, with the increasing of the number of optimization iterations, the accuracy of the optimized rule sets often converge to a definite value when the relevant rule sets are learned from the data sets that contain more nominal attributes than numeric ones. Note that the definite value here is usually not the maximum value obtained so far. On the other hand, the signal of convergence cannot obviously be

detected when the relevant rule sets are learned from the data sets that contain more numeric attributes than nominal ones.

Bibliography

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

[2] P. B. Brazdil, K. Konolige, Boston Kluwer, Pavel Brazdil Peter, and Peter Clark. Learning from imperfect data. In *in Machine Learning, Meta-Reasoning and Logics, P. Brazdil and K.Konolige (eds*, pages 207–232. Kluwer Academic Publishers, 1990.

[3] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[4] I. Bruha and F. Franek. Comparison of various routines for unknown attribute value processing: Covering paradigm. *Journal of Artificial Intelligence Research*, 10:939–955, 1996.

[5] Ivan Bruha and A. Famili. Postprocessing in machine learning and data mining. *SIGKDD Explor. Newsl.*, 2(2):110–114, 2000.

[6] Clifford Brunk and Michael J. Pazzani. An investigation of noise-tolerant relational concept learning algorithms. In *ML*, pages 389–393, 1991.

[7] William W. Cohen. Efficient pruning methods for separate-and-conquer rule. In *In Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 988–994. Morgan Kaufmann, 1993.

[8] William W. Cohen. Fast effective rule induction. In *In Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.

[9] Convergence. Definition of convergence. `http://en.wikipedia.org/wiki/Convergence`. [Online; accessed 1-October-2010].

[10] Convergence. A picture of the definition of convergence. `http://www.maths.abdn.ac.uk/~igc/tch/ma2001/notes/node18.html`. [Online; accessed 1-October-2010].

[11] Oliver Dain, Robert Cunningham, and Stephen Boyer. Irep++, a faster rule learning algorithm. In Michael W. Berry, Umeshwar Dayal, Chandrika Kamath, and David B. Skillicorn, editors, *SDM*. SIAM, 2004.

[12] Thomas G. Dietterich and Ghulum Bakiri. Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2:263–286, 1995.

[13] Philip J. Stone Earl B. Hunt, Janet Marin. *Experiments in induction*. Academic Press, New York, 1966.

[14] Johannes Fürnkranz. A tight integration of pruning and learning (extended abstract). In *ECML '95: Proceedings of the 8th European Conference on Machine Learning*, pages 291–294, London, UK, 1995. Springer-Verlag.

[15] Johannes Fürnkranz. Separate-and-conquer rule learning. *Artificial Intelligence Review*, 13:3–54, 1999.

[16] Johannes Fürnkranz and Gerhard Widmer. Incremental reduced error pruning. In *ICML*, pages 70–77, 1994.

[17] Michael P. Georgeff and Chris S. Wallace. A general selection criterion for inductive inference. In *ECAI*, pages 219–228, 1984.

[18] Randy Kerber. Chimerge: Discretization of numeric attributes. In *AAAI*, pages 123–128, 1992.

[19] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, pages 1137–1145, 1995.

[20] Ryszard S. Michalski. On the quasi-minimal solution of the general covering problem. 1969.

[21] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[22] Giulia Pagallo and David Haussler. Boolean feature discovery in empirical learning. *Mach. Learn.*, 5(1):71–99, 1990.

[23] J. R. Quinlan. Mdl and categorical theories (continued). In *In Machine Learning: Proceedings of the Twelfth International Conference, Lake Taho*, pages 464–470. Morgan Kaufmann, 1995.

[24] J. R. Quinlan and Jack Mostow. Learning logical definitions from relations. In *Machine Learning*, pages 239–266, 1990.

[25] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[26] J. Ross Quinlan. Generating production rules from decision trees. In *IJCAI'87: Proceedings of the 10th international joint conference on Artificial intelligence*, pages 304–307, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[27] J. Ross Quinlan. Simplifying decision trees. *Int. J. Man-Mach. Stud.*, 27(3):221–234, 1987.

[28] J. Ross Quinlan. *C4.5: Programs for Machine Learning (Morgan Kaufmann Series in Machine Learning)*. Morgan Kaufmann, 1 edition, January 1993.

[29] J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465 – 471, 1978.

[30] Matthias Thiel. Separate and conquer framework und disjunktive regeln. Master's thesis, TU Darmstadt, May 2005. Diplom.

[31] C. S. Wallace and D. M. Boulton. An information measure for classification. *The Computer Journal*, 11(2):185–194, 1968.

[32] Weka. Attribute-relation file format. `http://weka.wikispaces.com/ARFF+%28book+version%29`. [Online; accessed 18-Mai-2010].

[33] Weka. Weka: Data mining software in java. `http://www.cs.waikato.ac.nz/ml/weka/`. [Online; accessed 17-Juni-2010].

[34] Wikipedia. Decision tree learning. `http://en.wikipedia.org/wiki/Decision_tree_learning`. [Online; accessed 18-Mai-2010].

A Table

Name	Optimization Iterations					
	0	1	2	3	4	5
anneal	97.62	97.74	97.24	97.37	97.99	97.87
audiology	69.91	72.57	73.89	73.45	75.22	75.66
breast-cancer-data	73.08	73.08	72.38	71.68	70.98	69.58
cleveland-14-heart-disease	78.88	80.86	80.86	80.53	81.19	80.86
glass-database	64.95	67.29	68.69	69.16	70.56	67.29
glass2-database	74.85	79.14	79.75	79.14	79.75	80.37
hepatitis-domain	78.06	80	81.94	78.06	78.06	77.42
horse-colic-data	82.34	86.41	84.51	84.24	84.51	84.51
hypothyroid	99.05	99.18	99.21	99.18	99.18	99.18
iris	94.67	95.33	96	95.33	96	96
kr-vs-kp	98.84	98.84	99.06	99.19	99.25	99.31
lymphography-data	75.68	78.38	77.03	81.76	78.38	79.73
mushroom	100	100	100	100	100	100
sick-euthyroid	97.5	97.6	97.6	97.66	97.69	97.66
soybean	89.6	92.53	92.39	93.12	92.68	91.8
tic-tac-toe	96.97	97.39	97.39	97.29	97.39	96.97
titanic	78.33	78.19	78.19	78.19	78.19	77.87
congress-voting-1984-1	90.34	89.66	88.2	88.97	89.43	89.43
congress-voting-1984	94.94	95.4	95.4	95.4	95.4	95.4
wine	88.2	91.57	91.57	91.01	91.01	92.13

Table A.1.: SeCoRIP on 20 UCI Data Sets

Name	Optimization Iterations					
	0	1	2	3	4	5
anneal	97.55	97.7	97.67	97.72	97.95	97.92
audiology	70.1	71.47	73.62	73.92	73.93	73.56
breast-cancer-data	72.18	72.13	71.45	71.04	70.86	70.89
cleveland-14-heart-disease	78.41	79.27	79.26	79	79.1	78.97
glass-database	64.25	65.57	66.29	66.72	68.18	67.24
glass2-database	76.88	78.5	79.35	80.55	81.09	81.04
hepatitis-domain	77.55	78.77	78.71	78.29	78.65	78.12
horse-colic-data	84.02	85.73	85.32	85.26	85.26	85.21
hypothyroid	99.08	99.17	99.15	99.14	99.11	99.1
iris	92.4	93.47	93.93	93.73	94.07	94.27
kr-vs-kp	98.9	99.11	99.21	99.28	99.31	99.31
lymphography-data	75.75	77.64	76.17	77.19	75.63	76.4
mushroom	99.99	100	100	100	100	100
sick-euthyroid	97.4	97.59	97.6	97.64	97.65	97.65
soybean	90.31	91.43	91.77	92.02	91.81	92.04
tic-tac-toe	97.23	97.66	97.55	97.46	97.33	97.29
titanic	78.33	78.06	78.01	78.02	78.04	78.01
congress-voting-1984-1	89.61	89.47	89.47	89.58	89.42	89.54
congress-voting-1984	95.54	95.72	95.75	95.75	95.75	95.75
wine	91.46	92.44	93.14	92.75	93.1	93.42

Table A.2.: JRIP on 20 UCI Data Sets

Name	Optimization Iterations											
	0		1		2		3		4		5	
	Rules	Cond.	Rules	Cond.	Rules	Cond.	Rules	Cond.	Rules	Cond.	Rules	Cond.
anneal	9	13	9	14	10	16	9	14	7	9	7	8
audiology	21	40	17	25	18	32	18	27	20	33	21	38
breast-cancer-data	4	5	2	1	3	4	3	4	3	4	3	4
cleveland-14-heart-disease	8	21	3	5	3	4	3	4	3	4	3	4
glass-database	16	39	10	23	8	17	8	15	8	17	8	16
glass2-database	6	9	4	5	4	5	4	5	4	5	4	5
hepatitis-domain	3	5	2	2	2	2	2	2	2	2	2	2
horse-colic-data	2	2	3	4	3	4	3	4	3	4	4	6
hypothyroid	4	8	2	2	2	2	3	6	3	6	3	6
iris	3	3	3	3	3	2	3	3	3	3	3	3
kr-vs-kp	17	56	17	51	17	51	18	53	18	54	18	54
lymphography-data	9	15	8	11	7	12	7	10	8	12	6	9
mushroom	9	12	9	12	9	12	9	12	9	12	9	12
sick-euthyroid	8	28	7	21	5	14	6	19	6	18	6	17
soybean	29	54	26	47	26	48	27	55	29	55	28	54
tic-tac-toe	10	28	10	28	9	24	9	24	9	24	9	24
titanic	4	6	4	6	4	6	4	6	4	6	4	6
congress-voting-1984-1	6	20	4	9	5	13	5	13	5	13	5	13
congress-voting-1984	4	7	4	6	4	6	4	6	4	6	4	6
wine	3	3	3	4	3	4	3	4	3	4	3	4

Table A.3.: The Number of Rules and Conditions (SeCoRIP)

Name	Optimization Iterations					
	0	1	2	3	4	5
anneal	97.62	97.74	97.24	97.37	97.99	97.87
audiology	69.91	72.57	74.34	73.45	73.89	75.66
breast-cancer-data	73.08	73.08	72.38	71.68	70.98	69.58
cleveland-14-heart-disease	78.88	79.21	79.87	79.54	80.53	80.2
glass-database	64.95	67.29	68.69	69.16	70.56	67.29
glass2-database	74.85	79.14	79.75	79.14	79.75	80.37
hepatitis-domain	78.06	80	80	76.77	78.06	77.42
horse-colic-data	82.34	86.41	84.51	84.24	84.51	84.51
hypothyroid	99.05	99.18	99.21	99.18	99.18	99.18
iris	94.67	95.33	96	95.33	96	96
kr-vs-kp	98.84	99.06	99.19	99.31	99.37	99.41
lymphography-data	75.68	77.03	77.03	79.05	77.7	79.05
mushroom	100	100	100	100	100	100
sick-euthyroid	97.5	97.69	97.69	97.69	97.72	97.57
soybean	89.6	92.53	92.39	93.12	92.68	91.8
tic-tac-toe	96.97	97.18	97.08	96.87	96.97	96.66
titanic	78.33	77.87	77.87	77.87	77.87	77.87
congress-voting-1984-1	90.34	88.74	88.74	88.74	88.74	88.74
congress-voting-1984	94.94	95.4	95.4	95.4	95.4	95.4
wine	88.2	91.57	91.57	91.01	91.01	92.13

Table A.4.: Results of 1. Variant (*Abridgment*)

Name	Optimization Iterations					
	0	1	2	3	4	5
anneal	97.62	97.24	97.49	97.49	97.12	97.74
audiology	69.91	70.35	72.57	72.57	71.24	74.34
breast-cancer-data	73.08	74.48	73.08	73.08	73.08	72.38
cleveland-14-heart-disease	78.88	79.54	79.21	79.21	78.22	78.22
glass-database	64.95	64.02	70.56	67.29	69.16	67.29
glass2-database	74.85	79.14	79.75	82.21	80.37	80.98
hepatitis-domain	78.06	80	78.06	78.71	78.71	78.06
horse-colic-data	82.34	85.05	84.24	84.51	83.97	83.7
hypothyroid	99.05	99.11	99.18	99.21	99.21	99.21
iris	94.67	95.33	96	95.33	96.67	95.33
kr-vs-kp	98.84	98.81	98.62	98.44	98.12	98.19
lymphography-data	75.68	75	71.62	77.03	75	75.68
mushroom	100	100	100	100	100	100
sick-euthyroid	97.5	97.41	97.66	97.72	97.72	97.66
soybean	89.6	91.95	91.8	92.53	92.09	91.65
tic-tac-toe	96.97	97.91	97.49	97.49	97.6	97.49
titanic	78.33	78.06	77.56	77.56	77.6	77.28
congress-voting-1984-1	90.34	89.89	88.28	89.66	89.66	90.11
congress-voting-1984	94.94	95.63	95.63	95.63	95.63	95.63
wine	88.2	91.01	93.82	93.26	93.82	93.82

Table A.5.: Results of 2. Variant (*Accuracy*)

Name	Optimization Iterations											
	0		1		2		3		4		5	
	Rules	Cond.	Rules	Cond.	Rules	Cond.	Rules	Cond.	Rules	Cond.	Rules	Cond.
anneal	9	13	7	8	9	14	8	12	7	9	8	11
audiology	21	40	17	25	20	37	20	36	21	39	20	38
breast-cancer-data	4	5	3	6	2	2	4	6	3	4	3	4
cleveland-14-heart-diseas	8	21	3	5	4	8	4	8	4	8	4	8
glass-database	16	39	9	21	9	26	9	22	7	14	9	21
glass2-database	6	9	4	5	4	6	4	6	4	6	4	6
hepatitis-domain	3	5	3	5	3	5	3	6	3	6	3	6
horse-colic-data	2	2	3	4	3	4	3	4	3	4	4	6
hypothyroid	4	8	2	2	2	2	3	6	3	6	3	6
iris	3	3	3	3	3	3	3	3	3	3	3	3
kr-vs-kp	17	56	15	41	14	38	14	39	14	38	14	37
lymphography-data	9	15	7	10	6	10	8	14	7	12	7	12
mushroom	9	12	9	12	9	12	9	12	9	12	9	12
sick-euthyroid	8	28	7	25	5	15	4	10	4	11	4	11
soybean	29	54	26	48	25	48	29	62	27	52	29	61
tic-tac-toe	10	28	10	28	9	24	9	24	9	24	9	24
titanic	4	6	4	6	4	5	2	1	3	3	3	3
congress-voting-1984-1	6	20	3	5	4	9	4	9	5	13	4	10
congress-voting-1984	4	7	3	3	2	1	2	1	2	1	2	1
wine	3	3	3	4	3	4	3	4	3	4	3	4

Table A.6.: The Number of Rules and Conditions (SeCoRIP')

Name	Optimization Iterations				
	6	7	8	9	10
anneal	97.87	97.49	97.37	97.62	97.87
audiology	76.99	74.78	75.22	73.89	74.78
breast-cancer-data	69.58	69.58	69.58	69.58	69.58
cleveland-14-heart-disease	79.87	80.2	80.2	80.2	80.2
glass-database	64.95	66.82	63.55	67.76	69.16
glass2-database	79.14	79.14	79.14	80.98	80.98
hepatitis-domain	76.77	77.42	77.42	77.42	77.42
horse-colic-data	84.51	84.51	84.24	84.51	84.51
hypothyroid	99.18	99.18	99.18	99.18	99.15
iris	96	96.67	96.67	97.33	96.67
kr-vs-kp	99.34	99.31	99.34	99.34	99.28
lymphography-data	75.68	73.65	75.68	80.41	79.73
mushroom	100	100	100	100	100
sick-euthyroid	97.72	97.69	97.66	97.66	97.62
soybean	92.68	92.97	93.12	93.41	92.68
tic-tac-toe	96.76	96.76	96.97	96.97	97.18
titanic	77.87	78.01	78.01	78.01	78.01
congress-voting-1984-1	88.74	88.74	88.74	88.74	88.74
congress-voting-1984	95.4	95.4	94.71	95.4	95.4
wine	92.13	92.7	93.82	94.38	93.82

Table A.7.: SeCoRIP on 20 UCI Data Sets (Part 2)