Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-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 15. Oktober 2010

(Anne-Christine Karpf)

Bidirectional Rule Learning

Bidirektionales Regel-Lernen
Bachelor-Thesis von Anne-Christine Karpf
Oktober 2010

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Knowledge Engineering

Bidirectional Rule Learning
Bidirektionales Regel-Lernen

Vorgelegte Bachelor-Thesis von Anne-Christine Karpf

1. Gutachten: Johannes Fürnkranz
2. Gutachten:

Tag der Einreichung:

Contents

1 Introduction

In the field of Artificial Intelligence, machine learning algorithms play an important role. Providing a machine with the capability to not only apply a pre-defined strategy to a given task but to refine this strategy with experience greatly enhances its ability to solve real-world problems.

A program that can not only make decisions based on the information the programmer has specified a priori but also gains new information when provided with example data can solve new problems for which the ideal solution is not known at programming time. This is the case, for example, when the program will be confronted with unknown environments, or when the amount of data that has to be evaluated in order to find a good solution is too much for a human being to process. An example for the latter are applications in biology where a lot of data is available and the underlying concept is not yet known and not obvious for a human person that works with the data.

An application known from everyday life is the recognition of faces in pictures. If a photo camera is able to locate people's faces on a picture prior to taking it, it can automatically focus on the faces, since in most cases if people are on a picture, they are the main motive. For a human person, it is pretty easy to tell whether and where there are faces on a given picture. To specify what a computer or a photo camera must search for when looking for faces is more difficult. Since pictures are usually represented in the computer as a map of dots where each dot has a specified colour, one would have to define what a face is in terms of possible combinations and colours of these dots. Specifying all possible combinations of dots that might represent a face in a picture, both in existing and in future pictures, would be impossible for any human being because there are too many possibilities. When the program is able to learn, however, one can present it with pictures of faces and pictures without faces and the information which are which. Based on this data, the program itself can find the patterns that occur in the coloured dots when they form a face. When confronted with an unknown picture after the learning process, a picture for which the information whether it contains faces or not is not given, the computer can decide on the basis of its acquired knowledge whether the new picture contains a face or not. This is one of the examples where machine learning algorithms can be applied to solve a problem.

Rule learning algorithms are a special type of machine learning algorithms that formulate what they learn in terms of rules, which are ideally simple but accurate. Rules have an if-then-structure, which means that they express that if a set of given conditions holds true, another condition must also hold true. These rules are a representation of the knowledge the algorithm acquired during the learning phase and can afterwards be applied to new, unseen examples in order to classify them into categories. In the above example, possible categories would be "pictures with faces" and "pictures without faces".

A variant of rule learning is Separate-and-Conquer Rule Learning. In Separate-and-Conquer algorithms, the learning problem is tackled by first learning a rule that explains part of the training data and then recursively learning more rules for the rest of the training examples. As the term "separate and conquer" implies, the problem is first separated into smaller problems, namely the different parts of the example dataset, and then each of the smaller problems is solved (or conquered) by learning a rule for each of the parts of the training set. (cf. [Für99], p. 4)

Bidirectional Rule Learning is Separate-and-Conquer Rule Learning with a special search strategy, namely a bidirectional one. This means that it generates rules by taking a given rule and then refining it along two different search directions, either by specializing or by generalizing it. Specializing a rule means that conditons will be added to it, and generalizing means that conditions are deleted from it. The rules generated are evaluated to find out how good they are. If they fulfill a given quality criterion, they will be accepted as part of the resulting rule set, if not, they will be further refined.

In practice, usually rule learning algorithms that can only specialize a given rule are used, which is called a top down search strategy. However, the rules learned with these top down algorithms are often generalized after learning, which is called pruning of the rules and is performed using a set of examples that is independent from the set used for learning the rules.

The objective of this paper is to compare bidirectional and top down strategies and especially the bidirectional strategy without and the top down strategy with pruning, in order to answer the question if the rule quality can be improved by using a bidirectional strategy or if the advantage of the top down algorithm with pruning compared to a normal top down algorithm is due to the fact that the results are cross-checked on an independent pruning set.

To achieve this goal, an existing rule learning framework is enhanced by the implementation of a bidirectional search strategy. This bidirectional search strategy is then tested against existing strategies in terms of accuracy and complexity of the learned rule sets. Specifically, the algorithm using a bidirectional strategy is compared to another algorithm which uses a top down strategy to learn the rules and then prunes them on an independent test set afterwards.

Other bidirectional strategies exist already, but so far, they have not been extensively tested, especially not in comparison with top down algorithms using pruning afterwards. Implementing a bidirectional search strategy within the SeCo-framework for rule learning will on one hand enhance the framework by a new component and on the other hand provide a basis for a detailed evaluation of the bidirectional strategy in comparison to other algorithms. The SeCo-framework is a rule learning framework with a modulary structure in which different components can be combined to many different learning algorithms. Additionally, the framework provides the possibility to evaluate the performance of these algorithms on different datasets. Implementing the bidirectional strategy within the framework implies that the number of available components in the framework is increased and the comparison of bidirectional algorithms to other algorithms can be easily accomplished. The goal of this paper is to provide the SeCo-framework with a bidirectional search strategy and to obtain and discuss the results of the comparison between top down search with and without pruning and bidirectional search.

2 Rule Learning

The goal of a rule learning algorithm is to find a set of rules that, when applied to an unknown example, can classify this example as either a positive or a negative one with regard to some concept. Such a rule set is called a *classifier* because it can return a class assignment for an example whose class is unknown (cf. [Koh95], p. 1). (It is also possible to have more than just two classes, in which case the learning method has to be adapted a little.)

A separate-and-conquer algorithm will then try to find a rule that describes as many positive examples as possible but no negative examples. It then deletes the positive examples that are covered by the rule, and searches for another rule covering some of the remaining examples, and so on, until no positive examples remain. It then has a set of rules that, taken together, will classify all the positive examples from the training set as positive and all the negative ones as negative. It can now apply this newly learned rule set to examples yet unknown in order to classify these as either positive or negative ones.

A separate-and-conquer rule learning algorithm consists of several parts. First of all, the concept to be learned has to be represented in some way the algorithm can work with. This representation is called the hypothesis language, and the part it represents is the *language bias*. (cf. [Für99], p. 3 f.) What the algorithm also needs is a strategy to generate promising candidates for a new rule. It needs to know where to look for new rules, i.e. it needs a search algorithm, and a search strategy in order to know in which manner to proceed, for example bottom up or top down. Additionally, it has to evaluate the newly generated rules in order to chose the most appropriate one, which means it needs some search heuristics providing information on how "good" a given rule is. This part is what constitutes the *search bias*. (cf. [Für99], p. 4) Many algorithms also feature a third part, called the *overfitting avoidance bias*. The purpose of this part is to avoid that the rules learned are too close to the individual training examples instead of describing the general concept behind the examples. If the rules indeed are too much oriented towards the training examples rather than towards a general concept, one can apply heuristics that simplify the rules in order to make them more general so that they might be more accurate on new examples. Such a post-processing of the learned rule set is what is called *overfitting avoidance bias*. (cf. [Für99], p. 4)

In this paper, a special type of search strategy will be developed and evaluated: a bidirectional search strategy. The common choices for search strategies are either a bottom up or a top down approach. A top down search strategy means that the algorithm starts with an empty rule and searches for improvements of the current rule by adding new conditions to it, and a bottom up strategy starts with a most specific rule and then tries to improve it by removing conditions from it. The objective of a bidirectional strategy is to combine these two approaches: In every step, the algorithm has the choice to either add a new condition to the current rule or remove a condition, depending on which results in the best improvement of the current rule.

2.1 Basic Terms and Definitions

In order to express ideas clearly within this paper, some definitions are necessary. On the one hand, some terms specific to the thematic area of this paper may not be well-known to every reader. Even if they are well-known to some readers, it might still be good to specify what exactly they mean within this text in order to avoid misunderstandings, because some terms might have different meanings depending on their context. On the other hand, some terms for use within this text have to be defined to describe certain relations between rules or their status in the refinement process. Those are needed as a basis for describing certain concepts without repeating the explanation of the concept every time it is referred to, thus making the text less complicated.

One of the most important terms for this paper is the *rule* itself. A rule consists of a head and a body. The head contains one condition, whereas the body can contain several conditions, or none at all. If the body contains no conditions, the rule is called an *empty rule*. The meaning of a rule is the following: If all the conditions in the rule's body hold true, the condition in its head must hold true as well. In the case of rule learning, the head of the rule denotes the target concept that is to be learned. For example, if one wants to specify the relationship between the outside temperature and whether the sun shines or not, one could introduce attributes for these concepts. These could be, for example, "sunshine" with possible values "yes" and "no" to express whether the sun is shining or not, "rain" with possible values "yes" and "no" to express whether it is raining or not, and "temperature" with possible values "cold", "cool", "warm" and "hot" to express how the temperature is outside. One can now build conditions with these attributes, for example "sunshine = no", which means the sun does not shine, "temperature = warm" (It is warm outside), or "rain != no" (It does not not rain, which means it rains). If one wants to express now that whenever the sun does not shine and it rains, it is cold outside, one can form the following rule: "temperature = cold :- sun = no, rain = yes.". The first part of the rule, i.e. the part before the ":-", is the head of the rule, and the part after the ":-" is the body. The different conditions within the body are separated by "," and the rule ends with a ".".

If one now wants the computer to learn by itself which conditions hold true when it is cold outside, one can provide it with a set of *examples* and let a learning algorithm find patterns in them. An example would in this case be the weather that has been observed on a given day. This can be specified as a set of conditions. For example, there might be a day where the sun was not shining, it rained and it was cold, then a day where the sun was not shining but it did not rain either and it was cool, and then a day where the sun was shining, it was not raining and it was hot. This would be three examples we could provide our algorithm with. As a set of conditions, they would look like that: "sun = no, rain = yes, temperature = cold", "sun = no, rain = no, temperature = cool" and "sun = yes, rain = no, temperature = hot". One would need more examples to actually let the learning algorithm come up with anything useful, but for understanding the idea behind it, these three examples will suffice.

If one now wanted the learning algorithm to find out what conditions hold true when it is cold outside, one would provide it with these and other examples and the *target concept* it has to learn. The target concept is in this case whether it is cold or not. Therefore, "temperature = cold" would always be the head of the rules the algorithm learns. The bodies of the rules would be created by the algorithm as it examined the examples. In the end, it would come up with one or more rules, for example the rule "temperature = cold :- sun = no, rain = yes." from above. Provided with new examples where it is not specified whether it is cold or not, it could now make a prediction whether it is cold or not in these examples, based on the examples it has seen before and the rules it has extracted from them. "rain = yes, sun = no" would be classified as "cold", whereas "rain = no, sun = no" and "rain = no, sun = yes" would be classified as not cold.

If one wants to know more than whether it is cold or not, one can add intermediate values as well: Temperature could be cold, cool, warm, or hot. In this case, there are more than two classes between which the learning algorithm must differ. This kind of problem is called a *multi-class problem*. To adress multi-class problems, the learning algorithm must make sure that in the end, the rule set learned will propose just one class for an unknown example and not several different ones because more than one rule holds true. A possible way to do this is to learn a rule set with a certain order and always predict the class of the first rule that holds true for the example. This kind of ordered rule set is called *decision list*. Another possibility is to view the multi-class problem as a problem of just two classes, learning a rule set for each class separately by treating the examples of this class as positive examples and the examples of all the other classes as negative examples. In order to prevent the rules from predicting multiple classes in this case, each rule is assigned a weight and the prediction of the rule with the highest weight is taken to classify the example. (cf. [Für99], p. 11-12)

That way, one can specify via one or more rules which conditions a given example must fulfill so that it belongs to a certain class of examples, whether there are two or more classes. An *example* is a set of conditions that hold true in the specific situation the example describes. For example, when one wants to describe the weather on a given day, one could do so by specifying whether the sun is shining or not, and whether it is cold, hot, or something in between.

The examples the algorithm uses to find a rule set are called the *training examples* or *training set*. Within these examples, a value for the target concept is always specified. Sometimes the algorithm gets another set that is independent of the training set, called the *pruning set*. The pruning set is used to check how good the rules learned on the training set work on different, unknown examples. The value for the target concept must be specified for the pruning set as well, otherwise the algorithm will not know if it classifies the examples from the pruning set correctly. If the rules it has learned before work well on the pruning set, they are left as they are. If, on the other hand, they are found to be too closely oriented towards the specific examples of the training set, they are often *pruned*(hence the name pruning set), i.e. conditions are deleted from them, in order to make them more general and hopefully more apt to new, unknown examples. For the new, unknown examples usually the value for the target concept is not specified and has to be guessed by the algorithm with the help of the rules it has learned earlier.

A rule is said to *cover* an example if all conditions in the rule's body hold true for the example. If the target concept specified in the head of the rule, like "temperature = cold", also holds true for the example, the rule correctly classifies the example. If it does not, it classifies it incorrectly. If the example is actually a negative one but classified as positive, it is said to be a *false positive*. If it is actually positive and classified as negative, it is called *false negative*.

For the learning process, each rule needs to have a value describing the rule's quality. This is necessary for the algorithm in order to compare rules to each other and to have a way to decide whether a rule it has created is good enough or whether it has to further improve it. Usually, the value of a rule depends in some way on the number of examples it classifies correctly and the number of examples it classifies incorrectly.

In the following, some new terms are introduced to describe concepts important for this paper. During the process of finding good rules for a particular classification problem, a given rule can be changed in order to increase its quality, which is called *refining the rule*. The rule that is currently being refined (i.e. in the current iteration of findBestRule) is called the *candidate*. The rules resulting from the refining of one rule will be called *refinements* of that rule, and the rule they are refinements of is called their *predecessor*. Refinements of a rule can be either generalizations or specializations of their predecessor. The *generalization* of a rule is a refinement of this rule obtained by deleting one condition from the rule, which makes the refinement more general than its predecessor, which means that the refinement potentially covers more examples than its predecessor did. The *specialization* of a rule is a refinement of this rule obtained by adding one condition to the rule, which results in a rule that is more specific than its predecessor, i.e. potentially covers less examples.

Refinements resulting at the end of one iteration in findBestRule will be passed to the next iteration as new candidates for refinement.

2.2 The Rule Learning Task

The objective of Separate-and-Conquer Rule Learning algorithms is to solve the inductive concept learning problem. An algorithm is provided with positive and negative examples for some target concept, the information what the target concept is, and maybe some background knowledge. The algorithm is then expected to generate from this information a description of the target concept in the form of rules that can be used to classify new examples. The examples are represented as a fixed number of attributes and their values in the given example. The rules constitute a test for certain values of attributes in the examples. (cf. [Für99], p. 6)

The concept of examples and rules consisting of attributes and their values also defines the set of possible solutions.

A rule consists of a head and a body. The head is one condition that must follow if the set of conditions in the body holds true. A rule is said to cover an example if all the conditions in its body hold true in the given example.

2.3 Bidirectional Rule Learning

Characteristic for a bidirectional search strategy is that it searches the space of possible solutions not only in one but in two directions. One point in the search space is a rule, and from there you can get to other rules by either adding conditions to the current rule or by deleting conditions from it. These are the two directions in which the rule can be refined using a bidirectional search strategy. If the rule is refined by adding a condition to it, it is specialized, because afterwards it will usually cover less examples. If a condition is deleted, the rule is said to be generalized, because the new rule will potentially cover more examples than before. Other search strategies that search in only one direction are top down search, which only specializes the rules, and bottom up search, which only generalizes the rules. The logical consequence is that top down search must start with an empty rule, meaning a rule that contains no conditions in its body, because it is the most general rule possible and only from this rule, all other possible rules in the search space can be reached by specialization. Likewise, a bottom up strategy must start with the most specific rule possible, which is a rule that contains all possible conditions, because only from this rule, all other rules can be reached by generalization. The bidirectional strategy on the other hand can start at a random point in the search space because it can go in both directions and therefore still reach all other rules if it starts at some random point. (cf. [FW93], p. 3)

2.4 Related Work

Several approaches exist that apply a bidirectional strategy to the rule learning task. In the following, these are shortly described to give an overview of what possibilities exist and to provide a basis for the development of an own approach, which will be described in chapter 3. The approaches presented here will be characterised by the representation of the rules they use, i.e. the hypothesis language and by the type of steps they allow from one rule to another.

2.4.1 The Version Space algorithm

The earliest reference to a bidirectional approach found in the literature is the Version Space algorithm (cf. [MIT82], p.212-215). This algorithm uses a concept similar to rules as hypothesis language, which are the so-called *generalizations* that check via a matching predicate whether a given instance belongs to the set of instances corresponding to this generalization. The goal is to describe positive and negative instances using a set of generalizations, i.e. a more general description of the data that shows patterns within the data. (cf. [MIT82], p. 204) The Version space algorithm goes through the instances one by one. It has two sets of possible generalization, one with the most-specific generalizations that fit the instances processed so far, denoted by S, and another one with the most-general generalizations for the instances seen so far, denoted by G. When processing a new negative instance, the generalizations that match it are deleted from S, while the generalizations in G are made more specific so that they do not match the example anymore. What the algorithm does is approximately the following: When processing a new positive instance, the generaliaztions that do not match it are deleted from G, and the generalizations in S are generalized so that they match the instance. In the end, a set of generalizations will remain that matches all positive instances in the data set and no negative instance. (cf. [MIT82], p. 212-213)

2.4.2 The Swap-1 covering procedure

The goal of this algorithm is to induce rules with a low complexity, because they may be more adequate for unknown data although more complex rules might fit the training data better. Part of its strategy to achieve this goal consists of not only adding conditions to the rule being refined, but also allowing to replace or delete conditions from the rule. The Swap-1 algorithm refines a rule by first trying to "swap" one of its components, which means it replaces the component by another one or deletes it. If no such swap is found, a new component is added to the rule instead. Both for the swap and the addition of a component, the algorithm looks for the single best choice to improve the rule quality. The hypothesis language for this rule learning task consists of solutions in disjunctive normal form classifying examples by several disjunctive terms, which consist of a conjunction of conditions that are checked on a given example. In order to avoid cases where the rule set predicts more than one class for a given example, a priority can be assigned to the classes or the rules in order to decide which prediction will be chosen when more than one exists. (cf. [WI91], p. 678-679)

2.4.3 The IBL-Smart learning program

This approach uses a combination of knowledge-based symbolic learning and instance-based numeric learning. Its goal is to learn a classifier that predicts a numerical value for each example. In order to learn this classifier, it can use its background knowledge and the training data. The program consists of two parts in order to handle both numerical and symbolic attributes efficiently. One part is a symbolic learning component taking care of the symbolic attributes, the other one is an instance-based learning component predicting the numeric values by interpolation. Within the symbolic learning component, rules can be refined by adding conditions to them. Additionally, predicates in the rules body can be replaced in the following cases: If there exists a rule in its domain theory containing this predicate as head - the predicate would then be replaced by the conditions in its body. If the rule's body contains an attribute the values of which are proportionally related to another attribute, the first attribute can be replaced by the second. (cf. [Wid93], p. 371-377)

2.4.4 k-Opt - Bidirectional Search with Stochastic and Deterministic Local Optimization

Another bidirectional strategy is proposed by Mladenić. The paper is about learning algorithms using stochastic and deterministic local optimization and provides an empirical analysis of different versions of these algorithms compared to a decision tree learning algorithm. (cf. [Mla93], p. 1)

The rules and examples are expressed in terms of a fixed set of attributes of which each has a fixed set of values. They are represented as a binary vector with as many values as possible conditions exist, and each position in the vector corresponds to a certain condition. If the value in the vector is one, it means the corresponding condition is part of the rule, if it is zero, the condition is not part of the rule. A rule expresses that if a set of conditions holds true, the example belongs to a certain class. To get from one of these rules to another, one can change one bit in the binary vector. This means that in each transformation step, one condition can be either added or deleted. If the value in the vector was one and is changed to zero, the condition is deleted and the rule is generalized. If it was zero and is changed to one, the condition is added and the rule is specialized. In all algorithms the paper describes, the refinement process is started with a random rule. (cf. [Mla93], p. 2)

The first algorithm is a stochastic local search algorithm based on simulated annealing. In each step, it randomly chooses one value in the current rule and modifies it. If this modification improves the rule quality, the new rule is saved as the current rule. If not, it may still be saved as the current rule with some probability that depends on a parameter T, the temperature from simulated annealing, which is decreased in every iteration. This process continues iteratively until T is smaller than some fixed end temperature. A variant of this algorithm uses Markovian Neural Networks to randomly select more than one value for modification and then probabilistically chooses one of them to actually be modified, with better modifications in terms of rule quality being more likely to be selected. (cf. [Mla93], p. 3)

The second group of algorithms are deterministic local search algorithms. These algorithms are called k-Opt, with k denoting a positive integer. In each step, they search for up to k values in the rule binary vector whose modification will result in the best possible rule that can be obtained by modifying up to k values of the current rule. This is repeated iteratively until no modification to a rule with better quality is possible. The variants of the algorithm depend on the parameter k, for example, the algorithm 1-Opt with k=1 checks rules with one modified value, 2-Opt checks rules with one or two modified values, and so on. A faster variant of this are the algorithms k-OptF, which only look for the first modification of up to k values that provides a better rule quality than that of the current rule, and not necessarily the best. (cf. [Mla93], p. 3)

2.4.5 The algorithms JoJo, Frog and JoJo-FOL

The JoJo algorithm (cf. [FW93]) is another bidirectional algorithm. It can both generalize and specialize a rule and is part of a four step procedure to incrementally refine a rule set. (cf. [FW93], p. 1)

The search space consists of classification rules which can be generalized or specialized in every step. (cf. [FW93], p. 1). The JoJo algorithm can start at any point in the search space. Several criterions to choose a good starting position are given that estimate the description length, which means the number of conditions a rule contains, or the attributes that are chosen. It can also start with rules generated in some previous run of the JoJo algorithm. (cf. [FW93], p. 3)

The JoJo algorithm consists of a generalizer and a specializer, which compute the rules that can be reached by one generalization or specialization step respectively. In the simplest version, a generalization step consists of deleting a condition from the rule's body, while a specialization step consists of adding a condition to the rule's body. The generalizer and specializer also order the rules according to some preference criterion, which must not necessarily be equal for both. After all generalizations and specializations have been computed and sorted, the third part of the algorithm, called the scheduler, comes into play. It orders the rules according to some total preference criterion, which can be different from the other two, and chooses the best rule. In order to prevent loops because the same rule is considered twice, one can specify that the value of the preference criterion has to increase in every step. To get a rule set for all of the training examples, the whole procedure can be applied iteratively to get at first one rule that covers part of the training data, then delete the positive examples covered by this rule and apply the procedure again to get a second rule, and so on, until no more positive examples than specified as a threshold remain. (cf. [FW93], p. 3-4)

This algorithm is then extended to a four step procedure, which is provided with a rule set and positive and negative examples and transforms the rule set into one which covers all positive and no negative examples. In the first step, rules that cover too many negative examples are replaced by a set of rules that is obtained by applying JoJo to the subspace of the search space which the old rule covers. In the second step, JoJo is applied to the set of the uncovered positive examples and all negative examples in order to find rules that cover the remaining positive examples. In the third step, rules are deleted if there exists another rule of which the body is a subset of the first rule's body. This means that the rules which cover only a subset of what another rule already covers will be removed. The fourth step is to minimize the rule set, which is done by an heuristic search that takes the best rule, adds it to the final rule set and deletes all positive examples that are covered by this rule. If more uncovered examples than specified as a threshold remain, it will repeat this for the second best rule, and so on. (cf. [FW93], p. 4-5)

Tests performed on the JoJo algorithm show that JoJo is able to produce very short rules as description of the target concept. (cf. [FW93], p. 5)

Later, this algorithm is extended to the algorithm Frog, which functions like JoJo but additionally has the possibility to delete, add or change more than one condition in the rule's body in one step. (cf. [FW94], p. 1)

Like the JoJo algorithm, it has a search space consisting of rules and can start with an arbitrary rule. From one rule, it can reach a sucessor rule by adding a condition to it, which means the rule is specialized, or by deleting a condition from it, which means the rule is generalized. In order to increase the speed of the search, however, Frog also allows for the so-called sidesteps. A sidestep consists either of changing the value of an attribute to obtain a new rule, or of replacing one condition by another. In each step, several sidesteps can be performed, depending on the rate of change which decreases as the rule quality increases. In order to prevent infinite loops, the algorithm stores rules that have already been regarded so as to not regard them twice. The advantage of Frog over JoJo is that the search becomes faster using the sidesteps, and that the sidesteps give Frog the possibility to leave a local optimum in which JoJo would remain stuck. (cf. [FW94], p. 5-6)

On the basis of the algorithms JoJo and Frog, another algorithm called JoJo-FOL is developed that uses a bidirectional search strategy to learn clauses of a restricted first-order language. The search space of the algorithms consists of clauses in first-order logic that do not contain function symbols and contain only a limited number of variables, in order to obtain a finite search space. (cf. [Wie96], p. 1)

The algorithm works similar as JoJo does. It executes the following strategy as long as some preference criterion can be improved: While too many uncorrectly classified examples remain, it generalizes the current rule by adding a literal to it and calculates appropriate variable bindings. If the best new rule has a higher evaluation than the current rule, it is used as the new current rule. If not, it is discarded and the refinement process is started anew with a different initial rule. If the number of incorrectly classified examples is below a threshold, the algorithm specializes the current rule by deleting a literal from it and calculates appropriate variable bindings. If the new rule is better than the old one, it is taken as the new current rule. If the new rule is not better, the current rule is saved in the rule set and not further refined. (cf. [Wie96], p. 3-4)

3 Implementation of a Bidirectional Rule Learner within the SeCo-framework

The following chapter gives an overview of the things that need to be considered when implementing a bidirectional rule learner. First of all, some theoretical considerations are made with regard to the bidirectional algorithm, comparing both its possibilities and its runtime complexity to that of a top down approach and presenting solutions for the challenges discovered. Afterwards, the SeCo-framework is introduced as a basis for the implementation of the bidirectional rule learner. The framework's components that are relevant for this paper are explained. With both the theoretical thoughts from the first part and the basic introduction to the framework in mind, the implementation of the bidirectional rule learner is presented. Some alternatives to the components that were actually implemented are also discussed, some of them to explain why they have been discarded in favour of better solutions, others because they might provide a basis for further extensions of the bidirectional rule learner.

3.1 Theoretical Considerations

The search space for all three search strategies consists of all rules that can be formed using the available conditions. The difference between the strategies is that top down starts with the empty rule and can only add conditions, as figure 3.1(a) shows. Bottom up search, on the other hand, starts with a most specific rule, which is a rule that contains all available conditions, and can then only delete conditions, as can be seen in figure 3.1(b). Bidirectional search combines all the possibilities of the other two approaches and can therefore be depicted as an undirected graph containing the same edges as the directed graphs for top down and bottom up do, but without being restricted to just one of the directions. This graph for bidirectional search is shown in figure 3.1(c).

The graphs in figure 3.1 show the search space for the an example with the distinct conditions a, b c and d, where d is the head of the rules. All possible rules that can be formed with these conditions are displayed as nodes in the search space graph. Rules that are syntactically different but semantically equivalent to the rules present, for example because the conditions in their bodies appear in a different order or because a condition appears more than once, are not specifically displayed because it would not add additional possibilities in terms of expressitvity to an algorithm to consider these rules. Furthermore, it is assumed that in each step, every possible condition that has not yet been used can be added to a rule. If less conditions can be added due to additional restrictions, some of the edges in the graphs do not exist. Additionally, it is always assumed that the number of possible conditions and therefore the search space is finite. Otherwise, all algorithms could get lost in an infinite path of the search graph and might not terminate.

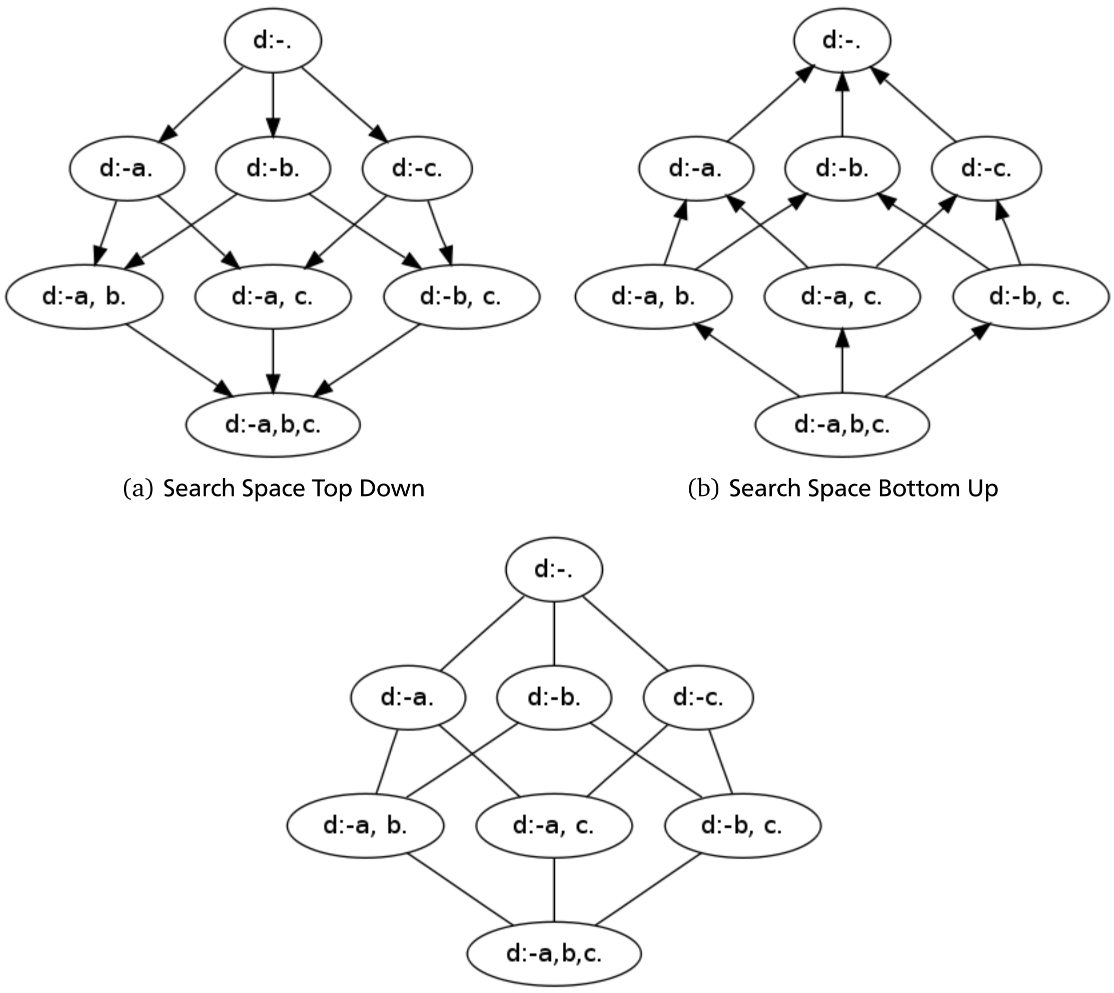
In each of the graphs, a rule node *r1* is said to be a refinement of another rule node *r1* if *r2* can be reached from *r1* in one step in the graph. The rule *d :- a, c.*, for example, is a refinement of *d :- a.* in figure 3.1(a), which is the top down search graph. In figure 3.1(b), which is the bottom up search graph, *d :- a, c.* is not a refinement of *d :- a.* because it cannot be reached in one step from *d :- a.*. In fact, *d :- a, c.* cannot be reached from *d :- a.* in the bottom up graph at all. The only possible refinement of *d :- a.* in the bottom up graph would be *d :- .*, which is the empty rule. In the bidirectional search graph as shown in figure 3.1(c), however, both *d :- a, c.* and *d :- .* are possible refinements of *d :- a.*, since they can both be reached from *d :- a.* in one step.

One iteration in the refinement process consists of computing all refinements that can be reached from a given rule in one step and returning those refinements.

Intuitively, one expects that a bidirectional learner generates better rules than a top down or bottom up learner. The bidirectional learner is able to revert decisions it has made at some point of the refinement process because it can delete a condition it has added earlier in the process, or add one it has deleted earlier. This means that it does not have to stick with bad decisions made early in the process like top down and bottom up approaches do, which is clearly an advantage of the bidirectional approach.

Unfortunately, this also presents a problem. The bidirectional approach can get stuck in an infinite loop if it keeps adding and deleting the same conditions, since the undirected graph in which it searches for a solution contains cycles, as can be seen in figure 3.1(c).

Both top down and bottom up search do not have this problem, since their search graphs do not contain any cycles. In each step, they always increase or decrease the rule length, respectively, and therefore cannot go back to a rule they already considered in an earlier iteration. The longest possible path for the algorithm in the top down graph is from the node representing the empty rule to the node representing the rule with all possible conditions. Correspondingly, the longest path for the bottom up search is from the rule containing all possible conditions to the empty rule. At this point, the algorithm must terminate in any case because no further refinement of the most specific rule in top down or the empty rule in bottom up is possible. With the number of possible conditions excluding the rule's head being denoted by a positive integer n, the length of this path is n at maximum - that is the path where all possible conditions are added to (or deleted from, respectively) the rule one after another. This means that after at most $n + 1$ iterations, the refinement



(a) Search Space Top Down (b) Search Space Bottom Up

(c) Search Space Bidirectional

Figure 3.1: The search space for a top down, bottom up and bidirectional strategy. a, b, c and d are distinct conditions.

process is finished for top down or bottom up, where the first n iterations are needed to get to the most specific rule or the empty rule respectively, and the last one to find out that no further refinements exist. This implies that the refinement process for top down and bottom up will eventually terminate, and will do so after at most $n+1$ iterations.

For bidirectional search, however, the maximum number of iterations is unlimited as its search graph contains cycles. Since in practice, the algorithm is required to terminate, one must add additional restrictions to the bidirectional approach in order to guarantee that it will eventually return a result.

When bidirectional search does get stuck in a cycle, it means that it considers rules twice that it had already discarded in an earlier iteration. If the algorithm were smarter, it would remember which rules it had analysed before and found not good enough. That way, it could detect when no new rules can be generated that are better than the ones it has seen so far, and stop its computations. Thus an improvement of the algorithm is pretty straightforward: While refining rules, mark those that have been generated as refinements as visited. If in a later iteration, a rule is generated as refinement again that has been visited earlier, it is discarded and not returned again. (cf. [FW94], p. 5)

That way, bidirectional search will consider each node in the search graph at most once. Since there are 2^n rule nodes in the graph, with n being the number of possible conditions excluding the head, this means that after at most 2^n+1 iterations, the refinement process is finished. With this restriction, the bidirectional algorithm will eventually terminate. In this case, it is possible to return the best rule of all the rules that exist in the search space.

If this approach proves to be too restrictive, it would even be possible to allow the algorithm to visit up to k nodes again although they have been visited earlier, where k is a parameter specified when starting the learning process. That way, the algorithm would still terminate and the maximum number of iterations in the refinement process would only increase by a constant, namely k. However, this should not be necessary since the algorithm already has the chance to visit every node in the search space once and it should not need to return to one it discarded earlier if it always chooses the best refinement as the next rule.

Another possibility to make sure bidirectional search will eventually terminate is to require the rule refined in one iteration to have a higher value than its predecessor, which is the rule it is a refinement of. (cf. [FW93], p. 4) That way, the refinement process can only advance to rules with a higher value than the rule in the iteration before, which makes it impossible to consider a rule twice. Additionally, it is not necessary to keep track of visited nodes, since only the predecessor's value must be stored in order to check that only rules with a strictly greater value can be be used for the next iteration. Furthermore, in this way not only the rules that have actually been visited but potentially more rules are excluded from further refinement, which speeds up the search process. The upper bound for the number of necessary iterations is still $2^n + 1$ as in the case of the nodes marked as visited.

In practice, however, the algorithm would only need to consider all 2^n nodes if they were ordered in a chain ascendingly sorted after their value so that the algorithm never has a choice between several nodes but just one option. Since the structure of the graph is defined by the number of available conditions, only one case is possible where there exists just one successor to a node: If there is only one possible condition that can be added and the rule in question is the empty rule, which means that no condition can be deleted yet, this rule has only one successor. And if there is only one condition, the in this case most specific rule with that condition also has only one successor, namely the empty rule, since only this condition can be deleted and no other condition added. In this case, if the rule at which the algorithm starts has the lower value and the other rule has a higher value, the algorithm would indeed have to look at all nodes before it terminates. In this case, $n = 1$ holds, because just one possible condition exists, and the algorithm will indeed need $2 + 1 = 2^1 + 1 = 3$ iterations, but this case is negligible because n is so small that the exponential complexity does not matter. If there is more than one condition, the nodes cannot be arranged as a chain and therefore the bidirectional search with the strictly greater restriction will need less than $2^n + 1$ iterations, although it is impossible to tell exactly how many nodes are likely to score below a given node's value without knowing the scoring mechanism, the structure of the rules in question and the set of examples they are evaluated on. Only for a specific example one can compute the exact number.

However, both of these restrictions limit the possibilities of the bidirectional search. A rule which is better than the current rule might be unreachable from the current rule if the only path to the better rule includes other rules that have already been visited or have a lower value than the current rule, because this would mean that the algorithm is prohibited from using this path. On the other hand, the bidirectional search still has more possibilities than top down or bottom up search have, and it is necessary to impose these restrictions upon the algorithm's freedom if only by these means it can be guaranteed that it terminates. And it is not a big restriction since it is unlikely that returning to a rule that has already been considered and has proven not to be good enough to fire the stopping criterion will be necessary in order to reach the best possible rule. The real problem is that the algorithm cannot reach the best possible rule anymore if the value of this rule is indeed higher than that of the current rule but the path from the current rule to the best possible rule leads through other rules with a value lower than that of the current rule. With the restriction that the new candidates have to have a higher value than their predecessor did, this path is not accessible to the algorithm anymore. This means that the search for the best possible rule can get stuck in local optimums instead of reaching the global optimum, which is a common problem of hill-climbing approaches.

On the other hand, this also means that, even if the generation of the refinements in each step was only taking constant time, the whole refinement process had exponential run time complexity, whereas top down and bottom up search both have only linear run time complexity. As can be seen, one has to trade off between run time complexity and giving the algorithm all the possibilities the bidirectional approach allows for. However, algorithms with a bad complexity class have been known to work well in practice, so this does not speak against the bidirectional approach. Furthermore, those are only considerations with regard to the algorithm's run time which do not take into account the quality of the rules the bidirectional algorithm produces. The rules might still be of better quality than those produced by a top down or bottom up strategy. Since the complexity given here is only an upper bound, it is also possible that the bidirectional algorithm finds rules of better quality in less time than the top down or bottom up algorithm. In the evaluations performed for this paper, the run time of the bidirectional algorithm has never been noticeably longer than that of the top down or bottom up algorithm with the same configurations. However, the considerations with regard to run time complexity might be kept in mind for very large data sets.

For the implementation of the bidirectional strategy in this paper, the approach which requires refinements to have a strictly greater value than their predecessor has been chosen, because its runtime is lower than that of the approach marking visited nodes. Furthermore, it is easier to implement, because it is not necessary to keep track of all the visited nodes. It still leaves the bidirectional search much more possibilities to revert earlier decisions than top down or bottom up search posess and is therefore considered a good compromise between runtime and implementation complexity and the quality of the results that can be expected.

3.2 Overview of the SeCo-framework

A general overview of the way the SeCo-framework learns a classifier for a rule learning task is given in in figure 1. With respect to this paper, the most important parts of the SeCo-framework are the methods findBestRule and refineRule. The pseudo code for findBestRule is given in figure 2, the implementation within the SeCo framework can be found in seco.learners.core.AbstractSeco. What findBestRule does is: It initializes a rule and iteratively refines it until it fulfills a certain stopping criterion. It then returns the best rule it has seen during the whole refinement process. In order to calculate the refinements, it calls refineRule, as can be seen in line nine of the pseudocode in figure 2. In the case of a bidirectional rule learner, refineRule would return all rules that can be obtained by either deleting or adding a condition from or to the current rule. In the case of a top down refiner, it would only return the specializations, and in the case of a bottom up refiner, only the generalizations. Within the framework, the implementation of refineRule can be found in the corresponding RuleRefiner class, for example in seco.learners.bidirectional.BidirectionalRefiner for the newly implemented bidirectional learner.

The class AbstractSeco, where findBestRule is located, is the one containing the classifier with all its components, one of them being the rule refiner. The different refiners are all implementations of the interface IRuleRefiner, which specifies that they must have, among other things, a refineRule method.

Apart from the refiner, the IRuleInitializer can be important for bidirectional rule learning as well. It is used to create an initial rule which findBestRule will use as the first rule it passes to refineRule for further refinement. The corresponding line in the pseudocode in figure 2 is line one. When using a top down strategy, the initial rule will be an empty rule, because if you can only add conditions it makes sense to start with an empty rule since otherwise you could not reach all possible rules by adding conditions anymore. The class returning this empty rule is seco.learners.components.TopDownRuleInitializer. When using a bidirectional strategy, one can start with the empty rule as initial rule but one can also start with any other rule, since the algorithm can both add or delete conditions and therefore, in theory, reach every other rule from each rule in the search space. [1] Since starting with a rule that is already close to the optimum solution can make the algorithm faster and prevent it from getting stuck in an unfavourable part somewhere else in the search space, the initial rule is one of the things influencing the quality of the bidirectional rule learning algorithm.

In order to evaluate how good a given rule is, the framework uses several heuristics to calculate a value for each rule. The component that takes care of that is called IRuleEvaluator. It needs to be passed an heuristics with which it can evaluate the rules. The evaluation of the rule appears in line two (for the initial rule) and in line eleven (for the refinements) of the pseudocode in figure 2.

If one needs to impose some restrictions on which of the current refinements may be passed to the next iteration, the interfaces ICandidateSelector and IRuleFilter are also important. ICandidateSelector is responsible for selecting those rules that will be refined in the current iteration from the set of all rules available at this point in the iteration, which corresponds to line six of the pseudocode in figure 2. The ICandidateSelector allows for prioritizing rules above others. IRuleFilter decides which of the rules that have been created as refinements will be passed on to the next iteration. This corresponds to line before the last one in figure 2. An example for an IRuleFilter is the BeamWidthFilter, which allows for a beam search within the framework. Beam search means that not all refinements are passed on to the next iteration but only the n best refinements, where n is a positive integer called the beam size. In the case of a beam search, the filter would make sure that only as many rules as corresponding to the beam size will be passed on.

In order to let the SeCo framework know which type of classifier one would like to build, one provides it with a so-called *configuration file* in XML format detailing which class to use for each of the components.

This was a brief introduction to the most important components of the SeCo-framework with regard to this paper. A detailed description of the other components can be found in [JF10b].

3.3 Implementation of the Bidirectional Rule Learner within the SeCo-framework

A first implementation of a bidirectional search strategy is pretty straightforward. Given a rule c, create all rules that can be reached from c by deleting one condition (the generalizations of c), and all rules that can be reached by adding one condition (the specializations of c), then return the set of all generalizations and specializations as possible refinements of c.

When implementing this algorithm within the SeCo framework, one can make use of the existing TopDownRefiner class in order to calculate the specializations of the rule. Therefore, it is only necessary to implement a BottomUpRefiner class that creates all the generalizations of the rule. One can then use both the TopDownRefiner's and the BottomUpRefiner's

[1] This is only true in theory because in practice, one must impose some restrictions upon the algorithm to make sure it terminates, as discussed in chapter 3.1.

Algorithm 1 AbstractSeCo(*Examples*)

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

Pseudo code of the SeCo-framework, taken from [JF10b], p. 3.

Algorithm 2 FindBestRule(*Growing, Pruning*)

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

Pseudocode for findBestRule, taken from [JF10b], p. 4.

refineRule method within the BidirectionalRefiner's refineRule method to calculate all generalizations and specializations of a given rule and then return a set containing both the specializations and the generalizations. [2]

However, this simple approach has one major problem, as already discussed in chapter 3.1: If a situation occurs in which a rule r1, which is a generalization of a rule r2, is selected as best refinement in one iteration, and in the next iteration r2 is selected as best refinement (obtained from r1 by adding one condition), and in the next iteration r1 (obtained from r2 by deleting the condition that was added in the last step), and so on, the algorithm never terminates. In test runs of the implementation described above, this happened with almost all data sets, so further improvement of the algorithm is necessary in order to avoid this behaviour.

A visited marker could be implemented within the BidirectionalRefiner class as follows: A list of all rules that have been refined so far (or alternatively, of all rules that have been refined or calculated as refinements) must be maintained. Before returning the refinements in refineRule, check for each refinement if it is contained in the list of visited rules, and if it is, delete it from the list of refinements and do not return it. The BidirectionalRefiner class does not know yet when to reset the list of visited rules because it does not know whether it is still refining the same rule or whether findBestRule let it start with a new rule, in which case the list of visited rules should be emptied. This extra information must be provided in some way when findBestRule calls refineRule. It could be implemented by an additional parameter of

[2] Actually, the TopDownRefiner in the SeCo-framework does not return all specializations of a rule but applies a certain quality control via the filterUsableConditions method before adding a condition to the current rule. This is left as it is for the BidirectionalRefiner, since an improvement of rule quality in the TopDownRefiner is likely to improve the rule quality in the BidirectionalRefiner as well.

refineRule which is set to true whenever refineRule shall reset the visited list before calculating refinements. Alternatively, one can implement a new method in the BidirectionaleRefiner class which resets the visited list when called. In both cases, findBestRule must be adapted to specify the parameter correctly or call the reset method at the right point in the refinement process. The tracking of visited rules requires some changes to the framework and may slow down the algorithm, since maintaining the visited list and checking if refinements are contained in said list before returning them requires additional time. Therefore, another approach to avoid infinite loops which can be implemented more elegantly is considered, although it is slightly more restrictive than the tracking of visited rules.

This strategy to avoid infinite loops works as follows: In any iteration, make sure that from the set of all refinements of a candidate rule c, only those rules that have a value higher than c can pass to the next iteration, i.e. can be used for further refinement. This implies that the values of the candidate rules that are considered must be strictly ascending from one iteration to the next, which makes it impossible to select a rule twice for refinement. Therefore, an infinite loop ist not possible. (cf. [FW93], p. 4) This strategy was chosen for the implementation of the bidirectional refiner used in this paper, as its implementation is more straightforward. If it proves to be too restrictive, the tracking of visited rules can still be added to the framework in order to compare the results of both strategies.

In theory, one can implement the strategy requiring rules to be strictly greater inside the refineRule method, similar to the implementation of the visited strategy. One can change the refineRule method in BidirectionalRefiner so that it only returns those rules with a greater value than their predecessor as refinements. An additional parameter in the configuration file could specify when to return only the refinements that are strictly greater and when to return all refinements. The latter might be relevant in some scenarios, for example for testing purposes. The parameter from the configuration file would be passed to the BidirectionalRefiner via the setProperty method that all SeCo components have for this purpose. The rule's predecessor is directly available within the refineRule mehtod, since it is the rule that is just being refined. Additionally, the getPredecessor method of seco.models.CandidateRule can be used to access this information. The heuristics used to evaluate the rule can also be obtained, using the method getHeuristic of seco.models.CandidateRule. Therefore, all the necessary information is available within the refineRule method.

This solution is possible but not the most elegant one. It would be better to use existing components of the SeCo-framework, because intuitively, the refineRule method in BidirectionalRefiner is only expected to refine rules, like the corresponding one in TopDownRefiner does.

Unfortunately, the strategy cannot be implemented using a CandidateSelector, since this results in another infinite loop. The candidate selector would at the beginning of the iteration select only those rules as candidates that have a higher value than their predecessors. Thus, it would prevent rules with a value lower or equal to that of their predecessor from further refinement, i.e. they will not be included in the list of candidates. It would, however, not delete them from the list of rules. However, the outer loop only terminates if rules is empty or if a stopping criterion fires at some point within the refinement process. This implies that if only rules with a value lower than that of their predecessor remain and no rule has fired a stopping criterion so far, findBestRule will continue infinitely in the following manner: It will select no candidates from rules, since no rule is better than its predecessor. Therefore, there will be no refinement taking place, hence no new rules that might fire a stopping criterion will be created. Since rules is then exactly the same as before the iteration, findBestRule will start with the next iteration and do the same thing. The conclusion is that although one might assume that CandidateSelector is the right place for the implementation, it is not, since it never results in the deletion of unfavourable candidates but just prioritizes which ones will be refined in the current iteration.

The best place to implement this strategy would be an IRuleFilter, which actually does remove rules directly from the list of rules that are passed to the next iteration. At the moment, the IRuleFilter is already used for something different than applying a certain quality criterion, like the strictly-greater-requirement would be one, to the rule set. In a beam search scenario with the class BeamWidthFilter as the implementation of an IRuleFilter used, it selects only the top n rules from the set of refinements, where n is the beam size. Thanks to the class MultiRuleFilter, however, one can combine the BeamWidthFilter with a filter selecting only rules that are strictly greater than their predecessors with a beam search. This filter can make use of the methods getPredecessor and getHeuristic of seco.models.CandidateRule just like the refineRule method could in order to evaluate the rules and compare their value to that of their predecessor. Since the refinements are usually already evaluated, it might not even need to evaluate the rules again. It just needs to compare the rule values and delete the rule if its value is lower or equal than that of its predecessor, or leave the rule in the list if its value is greater. This solution seems to be the most elegant one. When it was clear that this solution would also work out, however, a simpler version was already implemented in the framework which is not as elegant but does essentially the same thing. For the evaluations performed in this paper, this version was used due to time restrictions. But for a further improvement of the BidirectionalRefiner in the framework, the more elegant solution using an IRuleFilter should be kept in mind.

The solution that was chosen for implementation does let refineRule generate all possible refinements but within findBestRule in AbstractSeco adds only those to the list of rules that will be passed to the next iteration that have a greater value than their predecessor. To implement this approach within the framework, a new general property *strictlyGreater* was introduced. This property can be set to true within the configuration file in order to require refinements that will be passed to the next iteration to have a strictly greater value than their predecessor. When *strictlyGreater* is not specified, it has a default value of false, which means that this solution will still work with existing configuration files and behave in the same way for those files as it did before the new property was introduced. In findBestRule, at the end of the loop in which one candidate (or a set of candidates) is refined, after each refinement has been evaluated and before it is added to the list of rules that are to be further refined in the next iteration, there is now a new condition which holds true if *strictlyGreater* is false or if the value of the refinement is indeed strictly greater than that of the rule of which it is a refinement. This means that *strictlyGreater* implies that the value of the refinement must be strictly greater than that of its predecessor. If this condition holds, the refinement is added to the list of the refinements for the next iteration, if not, it is discarded.

This approach solves the problem of the occuring infinite loops. It works well with existing configuration files and requires only small changes within the framework - AbstractSeco needs a new class variable, and findBestRule needs to check another condition before adding refinements to the list of rules that will be passed to the next iteration. It still lets refineRule in BidirectionalRefiner return all refinements, instead of only those better than the current rule. Therefore, it appears to be a good solution that works well for the evaluations performed.

In order to make the most of the BidirectionalRefiner's possibilities, another feature was implemented, namely a new IRuleInitializer. The default IRuleInitializer is the one for a TopDownRefiner which creates an emtpy rule as the initial rule. Since the BidirectionalRefiner can start with any rule, another IRuleRefiner was implemented that creates the initial rule by randomly choosing one of the instances from the data set and transforming it into a rule. This is done by creating conditions for all attributes with the respective value the attribute has in the given example. The condition created from the attribute for which a classifier is to be learned is taken as head, the other conditions form the rule's body. The idea behind this strategy is that such a rule may already be closer to an optimum solution and therefore the search process might be speeded up. Additionally, if it is closer to an optimum solution, it may prevent the BidirectionalRefiner from getting stuck in a local optimum further away from the optimum solution and therefore the rules in the final rule set might be of better quality.

4 Evaluation of the Bidirectional Rule Learner

4.1 Overview of evaluation methods

The evaluation part of the SeCo-framework performs the learning processes with several algorithms on several data sets. Which algorithm configurations and which data sets are used is specified in a configuration file. For the evaluations within this paper, the data sets in the folder `/data/test/` in the SeCo-framework have been used. Table 4.1 gives an overview of the data set's names, number of examples and the number of classes that are to be learned. The algorithms which are evaluated against each other are described in detail in section 4.2. For each algorithm and data set, the SeCo-framework returns the configuration details of the algorithm, the rule set that was learned and statistics on the classifiers performance, such as the number of rules, the number of used conditions, the number of referred attributes within the rule set, the average rule length and the classifiers accuracy on the training set and in the cross validation.

The classifier's accuracy is the probability with which it classifies a randomly selected instance correctly, i.e. it assigns the instance the class it actually belongs to. (cf. [Koh95], p. 2)

Cross validation is a method to estimate the accuracy of a classifier. In order to learn a classifier on one data set and test it on another, the full data set is split into k mutually exclusive subsets, called the k folds. Cross validation then runs k times and learns a classifier on all but one of the folds. It then tests the classifier on the fold it had omitted from the data set during learning. Each fold is omitted from the learning set exactly once. To compute the classifier's accuracy it is summed up how many times an instance is classified correctly and this is divided by the total number of instances. The number of times an instance is classified correctly is, more exactly, the number of times that the instance is assigned the class it actually belongs to by the classifier learned on the full data set excluding the test set containing the instance. The partitioning into folds can be done so that the examples in each fold are distributed among the different classes in approximately the same way as in the full data set, which is called stratified cross-validation. (cf. [Koh95], p. 3) In the evaluations performed for this paper, stratified ten-fold cross validation is used as recommended in [Koh95] (p. 7) for selecting one classifier over another.

Additionally, the SeCo-framework provides a comparison of all the algorithms tested against in each other in one evaluation run. For this, it states the average rank the algorithm has reached on average over all data sets. Comparing the ranks, one can see which algorithms performed better than others in terms of accuracy.

In addition to the ranks, the framework also computes macro and micro average of the accuracy on the data sets. For computing the macro average, all classes are assigned the same weight, whereas for the micro average, all classes are assigned a weight corresponding to their number of instances.

Whether the performance of the algorithms is significantly different is determined by using a Friedman Test with a post-hoc Nemenyi Test (cf. [Dem06], p. 11-13).

The null hypothesis of the Friedman test is that the performance of the classifiers is not significantly different. When the null hypothesis can be rejected, it means that the classifiers are significantly different. In this case, a post-hoc Nemenyi test can be applied in order to determine which classifiers differ significantly in their performance.

First of all, the average ranks of the different classifiers are determined using the evaluation framework. With those average ranks the statistic by Iman and Davenport is calculated (cf. [Dem06], p. 11) and used to determine whether the classifiers are significantly different. If the null hypothesis can be rejected, which means the classifiers are significantly different, the post-hoc Nemenyi test is performed. For the post-hoc Nemenyi test, the critical difference is computed (cf. [Dem06], p. 11). If the average ranks of two classifiers have a difference of at least the critical difference, they are considered significantly different. The exact formulas to compute the respective values can be found in [Dem06].

The bidirectional strategy and the RandomRuleInitializer used in this evaluation are the ones described in chapter 3, while the top down strategy and all other components are from the Seco-Framework.

4.2 Configurations selected for testing

In this section, it is described which algorithms are compared to each other and what insights are expected from the tests.

The heuristics used in the evaluation are Correlation, Laplace, m-estimate, Weighted Relative Accuracy (WRA) and FoilGain. These heuristics are chosen to test whether the bidirectional strategy works well with all of them or better with some of them than others. With each of the heuristics having different advantages and disadvantages, it is expected that the ones chosen will provide a useful insight into what works well with a bidirectional strategy and what does not. While WRA sometimes prefers too general models, Laplace has a tendency towards overfitting of the data if no pruning strategy is applied (cf. [JF10a], p. 350-351). M-estimate assigns an a priori coverage to a rule even if it covers zero examples,

Table 4.1: Names, size and number of classes of the data sets the evaluation was performed on. The files can be found in the folder `data/test/` within the SeCo-framework.

Name of Data Set	File Name	Number of Instances	Number of Classes to be learned
anneal	anneal.arff	798	6
audiology	audiology.arff	226	24
auto-mpg	auto-mpg.arff	398	4
breast-cancer-data	breast-cancer.arff	286	2
wisconsin-breast-cancer	breast-w.arff	699	2
cleveland-14-heart-disease	cleveland-heart-disease.arff	303	5
german-credit	credit-g.arff	1000	2
credit	credit.arff	490	2
glass-database	glass.arff	214	7
hypothyroid	hypothyroid.arff	3163	2
ionosphere	ionosphere.arff	351	2
kr-vs-kp	krkp.arff	3196	2
mushroom	mushroom.arff	8124	2
segment	segment.arff	2310	7
sick-euthyroid	sick-euthyroid.arff	3163	2
soybean	soybean.arff	638	19
tic-tac-toe	tic-tac-toe.arff	958	2
relation	titanic.arff	2201	2
vowel	vowel.arff	990	11

which is controlled by the parameter m (cf. [JF10a], p. 352). In the evaluation runs performed here, the default value for m from the SeCo-framework is used, which is its optimum value (cf. [JF10a], p. 360). While Correlation, Laplace, m-estimate, Weighted Relative Accuracy (WRA) are Value Heuristics giving an absolute estimation of the quality of one rule, FoilGain is a Gain Heuristics evaluating the rule's quality relative to its predecessor. (cf. [JF10b], p. 12) More details on the heuristics used can be found in the papers cited above. The formulas for the heuristics are given in table 4.2.

4.2.1 Comparison of basic bidirectional approach and basic top down approach

The aim of this test is to find out whether the bidirectional strategy yields better results than the top down strategy, and to find out if the bidirectional strategy works better with one heuristics than with another. In order to answer these questions, several versions of bidirectional algorithms have been evaluated against several versions of top down algorithms. All algorithms tested here are based on the default configuration given within the SeCo-framework. From this default configuration, four top down algorithms are obtained by varying the heuristics between Correlation, Laplace, m-estimate and WRA. From this, four corresponding bidirectional algorithms are obtained by changing the refiner to a bidirectional one and setting the property strictlyGreater to true. An overview of the configurations used in this test can be found in table 4.3.

4.2.2 Comparison of empty rule or random rule as initial rule in bidirectional approach

This test is performed in order to find out if the quality of the results of the bidirectional rule learner can be improved by using an initial rule that may be closer to a possible solution than the empty rule. For this task, the four bidirectional algorithms from test 1 are evaluated against four corresponding bidirectional algorithms which are the same except that the RuleInitializer is now RandomRuleInitializer instead of TopDownRuleInitializer, which means the refinement process is started with a rule generated from a randomly chosen example instead of with the empty rule. A list of all configurations used in this test is given in table 4.4.

This test is performed only once because the initial rule is randomly chosen for each of the four algorithms with a random starting rule and each of the 19 data sets, which means that in total, the test is performed with at minimum 76

Table 4.2: Formulas for the heuristics used in the evaluation. Each formula evaluates one rule on a data set. The total number of positive or negative examples in the data set is denoted by P or N respectively. The number of positive or negative examples the specific rule covers is denoted by p or n respectively. With p' and n', the corresponding values of the rule's predecessor are denoted.

Name	Formula	Source
Correlation	$\frac{pN-Pn}{\sqrt{PN(p+n)(P-p+N-n)}}$	[JF10a], p. 21
FoilGain	$p(\log_2 \frac{p}{p+n} - \log_2 \frac{p'}{p'+n'})$	[JF10a], p. 18 and p. 12
Laplace	$\frac{p+1}{p+n+2}$	[JF10a], p.15
m-estimate	$\frac{p+m+\frac{p}{P+N}}{p+n+m}$	[JF10a], p. 15
Weighted Relative Accuracy (WRA)	$\frac{p}{P} - \frac{n}{N}$	[JF10b], p. 12

randomly chosen starting rules, maybe more, depending on the number of times findBestRule is called. It is assumed that this number is sufficiently large that an unfavourable choice of the starting rule is unlikely to occur every time. For this reason, a second test run was performed but not evaluated in detail, because the results observed where approximately the same as in the first run.

4.2.3 Comparison of bidirectional without and top down with post processing

Since the objective of this paper is to show whether a bidirectional strategy performs equally well or better than a top down strategy pruning the rules after learning, another test is needed that compares a bidirectional algorithm to a top down algorithm that uses post processing. As top down algorithm with pruning, Ripper is chosen.

In the Ripper algorithm, the example data set is split into a growing and a pruning set. The growing set is used to learn a rule by applying a top down strategy during the refinement process, and the pruning set is used to prune the rule directly after learning it. Additionally, Ripper applies a post processing to the rule set it has learned. Of each rule, three variants are created: The rule itself, a variant of the rule that is learned completely new, and the original rule with more conditions added to it. Out of these three versions, the best rule is chosen and the original rule is replaced by it. Ripper does this optimization of the learned rule set two times as post processing. As opposed to the default top down variant from the first evaluation run (see chapter 4.2.1), Ripper uses FoilGain as heuristics and the MDLStoppingCriterion as rule stopping criterion, which means that the refining of a rule is stopped if it would result in a rule set of which the description length is greater than the description length of the example data set, i.e. the rule set is more complicated than the examples itself. (cf. [JF10b], p. 14 and p. 16) The details of the configuration are shown in table 4.5.

This algorithm is compared to two bidirectional algorithms, both based on the configuration of Ripper. They use the bidirectional refiner instead of the top down refiner, post processing is not used and strictlyGreater is set to true. One of the algorithms uses the same heuristics as Ripper, FoilGain, while the other one uses Correlation. The variation of heuristics is important in order to avoid that the bidirectional strategy obtains a bad result only because it was used with a heuristics it does not work well with while it could have returned a better result when used with another heuristics. An overview of the configurations used in this test can be found in table 4.5.

4.3 Discussion of the results

In this chapter, results of the evaluation runs described in chapter 4.2 are given, both as an overview of the average ranks, macro and micro average the different algorithms obtained and as a visualization of the results of the significance tests. Possible explanations for and consequences of the results are then discussed. To look at the data in more detail, some sample data sets and algorithms are selected and more detailed information about them is given, like number of rules of the classifier, number of conditions, number of referred attributes and average rule length. Selected were the two largest, the two smallest and two intermediate data sets, keeping in mind that the number of classes that have to be learned also varies among these data sets. The data sets selected are mushroom, kr-vs-kp, anneal, wisconsin-breast-cancer, audiology and glass-database. Their size and number of classes to be learned is shown in table 4.1. With this selection, it is made sure that a reasonable variety among the chosen data sets exist so as not to generalize an effect that maybe only appears in very specific data sets, for example only very large data sets. At the same time, the detailed data given is small enough that patterns can be easily observed and understood.

Table 4.3: Configurations used to compare the performance of a basic top down and a basic bidirectional approach with varying heuristics, as described in chapter 4.2.1.

(a) Specification of the parts that were not varied in the test configurations.

Name	Component used
Rule Evaluator	RuleEvaluator
Rule Initializer	TopDownRuleInitializer
Rule Filter	BeamWidthFilter (beam size = 1)
Stopping Criterion	NoNegativesCoveredStop
Rule Stopping Criterion	CoverageRuleStop
Post Processor	NoOpPostProcessor
Reduced Error Pruning	not used
Minimum Number of examples a rule has to cover	1
Weighted Covering	not used

(b) Configuration numbers and specification of the parts that were varied in different configurations.

Number	Rule Refiner	Heuristics	strictly Greater
1	Bidirectional	m-estimate	true
2	TopDown	m-estimate	false
3	Bidirectional	Laplace	
4	TopDown	Laplace	false
5	Bidirectional	Correlation	
6	TopDown	Correlation	false
7	TopDown	WRA	false
8	Bidirectional	WRA	true

4.3.1 Results of the comparison of basic bidirectional and basic top down approach

The results of the test described in chapter 4.2.1 can be found in table 4.6. The results of the post-hoc Nemenyi test are visualized in figure 4.1. As can be concluded from the ranks given in table 4.6, the bidirectional variants do not perform better than the top down variants. However, there is only a significant difference between Top Down with m-estimate and Bidirectional with WRA, with the top down algorithm being the better one. This difference occurs only at the 0.1 level. At the 0.05 level, the algorithms are significantly different in the Friedman test but no significant difference between either two of them can be detected in the post-hoc Nemenyi test. On the 0.01 level, the null hypothesis of the Friedman test cannot be rejected.

Looking closer at the results, an interesting thing becomes obvious: Even in the case where top down performs significantly better than bidirectional, there are many data sets on which the accuracy of bidirectional is only a little bit smaller than that of the top down algorithm, but the rule set contains less and much simpler rules. Examples of this are given in table 4.2.

Detailed results for some of the algorithms and data sets can be found in table 4.7. A visualization of these details is shown for three of the algorithms in figure 4.2. Between the two variants using m-estimate as heuristics almost no differences can be detected both in terms of rule set complexity and accuracy. What is interesting, however, is that the variant which performed worst in the test, namely Bidirectional with WRA, obtains an accuracy which is almost as good as that of the other two algorithms but with a smaller rule set. For the two smallest data sets, which are audiology and glass-database, the accuracy of Bidirectional with WRA is even better than that of the other two algorithms. Ideally, one would prefer simpler rules in the hopes that they are better suited for unknown examples since they are more general. It does not seem that this is the case here; according to the ranks, the simpler rules perform almost as well but not better than the more complex rules in the cross validation. Looking at the results obtained with Bidirectional using Laplace one can observe that this algorithm tends to learning classifiers with many more rules than the other algorithms, so it does not seem to be a general property of bidirectional search that very general rule sets are learned in Bidirectional using

Table 4.4: Configurations used to compare the performance of bidirectional approaches with varying initial rules and heuristics, as described in chapter 4.2.2.

(a) Specification of the parts that were not varied in the test configurations.

Name	Component used
Rule Evaluator	RuleEvaluator
Rule Refiner	BidirectionalRefiner
Rule Filter	BeamWidthFilter (beam size = 1)
Stopping Criterion	NoNegativesCoveredStop
Rule Stopping Criterion	CoverageRuleStop
Post Processor	NoOpPostProcessor
Reduced Error Pruning	not used
Minimum Number of examples a rule has to cover	1
Weighted Covering	not used
strictly Greater	true

(b) Configuration numbers and specification of the parts that were varied in different configurations.

Number	Heuristics	Rule Initializer
1	*m*-estimate	TopDown
2	Laplace	TopDown
3	Correlation	TopDown
4	WRA	TopDown
5	*m*-estimate	Random
6	WRA	Random
7	Correlation	Random
8	Laplace	Random

WRA. Most likely, the result is due to WRA's tendency towards more general rules and Laplace's tendency to overfit the data.

All in all, the difference between the basic bidirectional and top down approaches does not appear to be big. Most of the the observed differences seem to be due to the different heuristics used rather than the different search strategies. This means that in the basic version, bidirectional performs neither better nor worse than top down search.

4.3.2 Results of the comparison of empty rule or random rule as initial rule in bidirectional approach

The results of the test described in chapter 4.2.2 can be found in table 4.9. The visualization of the results of the post-hoc Nemenyi tests is shown in figure 4.5. The results show that the bidirectional variants with a random starting rule perform significantly worse than their counterparts with an empty starting rule. This result is surprising, since the introduction of a starting rule generated from a random example was meant to be an improvement of the algorithm. A second test run shows approximately the same results. For a closer look on the results, the two algorithms with the heuristics Laplace, one with RandomRuleInitializer and the other with TopDownRuleInitializer, are taken as an example. Their detailed results on selected data sets can be seen in table 4.8. From these statistics it is obvious that the variant using the RandomRuleInitializer learned rule sets consiting of no rules at all, or very few very specific rules. For the cases where the RandomRuleInitializer variant actually did learn rules, the number of attributes of the data sets might be interesting: There are 39 attributes in anneal, 10 in wisconsin-breast-cancer and 10 in glass-database, including the class attribute. This implies that in those cases, the algorithm did not get very far in refining the rules it has learned. As can be seen in the case of anneal, where the learned classifiers for the RandomRuleInitializer and the TopDownRuleInitializer variant can be found in figures 4.3.2 and 4.3.2 respectively, the RandomRuleInitializer stopped after learning one rule with a value of 0.667 although better, more general rules can be found, for example the rule *Class = c5 :- family = TN.* with a value of 0.984, which is part of the classifier the TopDownRuleInitializer variant learned.

Table 4.5: Configurations used to compare the performance of a top down approach with post processing (Ripper) and the bidirectional approach, as described in chapter 4.2.3. Configuration number 1 is the default Ripper configuration that can be found in the SeCo-framework (`config/new_config/fully_specified/Ripper.xml`)

(a) Specification of the parts that were not varied in the test configurations.

Name	Component used
Rule Evaluator	RuleEvaluator
Rule Initializer	TopDownRuleInitializer
Rule Filter	BeamWidthFilter (beam size = 1)
Stopping Criterion	NoNegativesCoveredStop
Rule Stopping Criterion	MDLStoppingCriterion
Minimum Number of examples a rule has to cover	2
Weighted Covering	not used

(b) Configuration numbers and specification of the parts that were varied in different configurations.

Number	Rule Refiner	Heuristics	Post Processor	Pruning Set	strictly Greater
1	TopDown	FoilGain	Ripper (2 optimizations)	33.34 %	false
2	Bidirectional	Correlation	NoOp	none used	true
3	Bidirectional	FoilGain	NoOp	none used	true

Table 4.6: Results of test 4.2.1: All configurations and their macro and micro average and ranks in the cross validation.

Number	Rule Refiner	Heuristic	strictly Greater	Macro Average	Micro Average	Average Rank
1	Bidirectional	*m*-estimate	true	86.516	93.581	3.474
2	TopDown	*m*-estimate	false	86.426	93.649	3.421
3	Bidirectional	Laplace	true	85.587	92.921	4.447
4	TopDown	Laplace	false	85.517	92.914	4.553
5	Bidirectional	Correlation	true	84.890	92.343	4.500
6	TopDown	Correlation	false	84.884	92.356	4.500
7	TopDown	WRA	false	83.222	90.292	5.474
8	Bidirectional	WRA	true	83.198	90.228	5.632

As the results indicate, starting from the empty rule seems to be preferable to starting from a rule generated from an example. It appears that the empty rule is closer to the optimum solution than one of the examples. Assuming that more general rules are better than very specific rules because they may fit unknown examples better, this might be a reasonable conclusion. Looking closer at the results, one can observe that the learners using the RandomRuleInitializer often learn a rule set consisting of only a few very specific rules, or in many cases, consisting of no rules at all. This indicates that the search for a good classifier has been stopped too early, possibly because a stopping criterion fired or because the search got stuck in a local optimum before finding good rules due to the strictly greater restriction, or a combination of those two.

In the case where the rule set consists of no rules at all, the outer loop of the Separate-and-Conquer algorithm as seen in figure 1 must have terminated the while loop right after the first call to findBestRule because the rule stopping criterion fired for the rule that was returned in this one call to findBestRule. Therefore, the rule did not get added to the rule set and an empty set was returned. The rule stopping criterion used in this configuration is CoverageRuleStop, which fires for a rule that covers more negative examples than positive examples. The rule for which it fired is then not added to the theory and no further search for more rules is conducted. One problem with the RandomRuleInitializer that could result in the learning of an empty theory is that when choosing the random example, the RandomRuleInitializer does not differ between positive and negative examples. It is possible that the initial rule is created from a negative example and therefore covers at least this negative example and maybe no positive example. It is likely that the refinements of this initial rule will have similar coverage statistics as their predecessor. The specializations cannot cover more examples

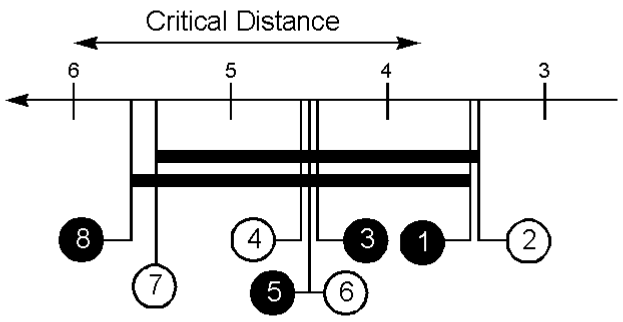
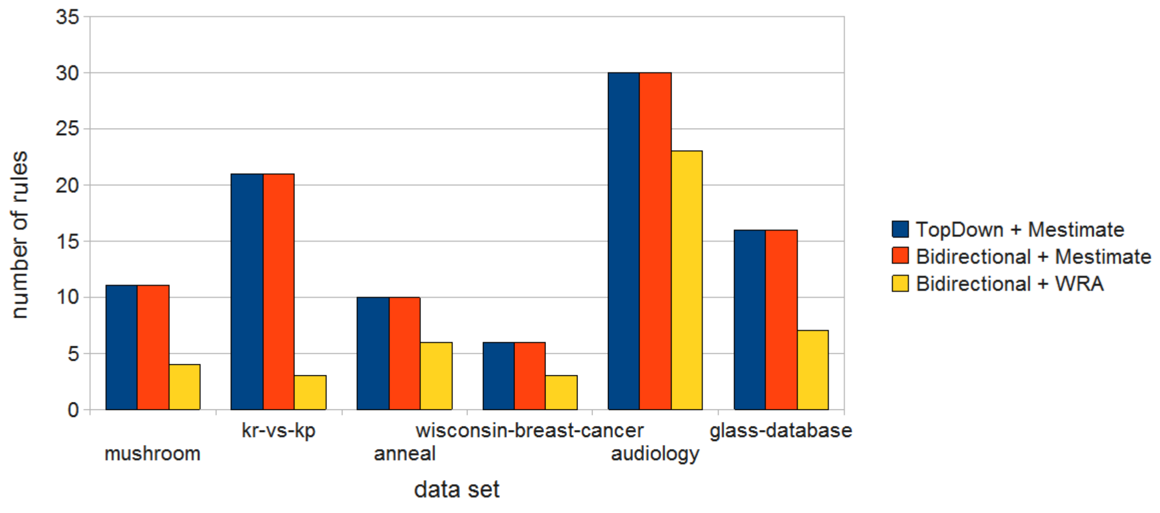


Figure 4.1: Results of significance test on evaluation run 4.2.1. The numbers of the algorithms correspond to the ones given in table 4.6. Bidirectional variants are displayed with a white number in a black circle, top down variants with a black number in a white circle. Groups of algorithms are connected if their performance does not differ significantly on the 0.1 level. In this case, only the algorithms 2 (TopDown with *m*-estimate) and 8 (Bidirectional with WRA) differ significantly in their performance, with 2 being the better one.

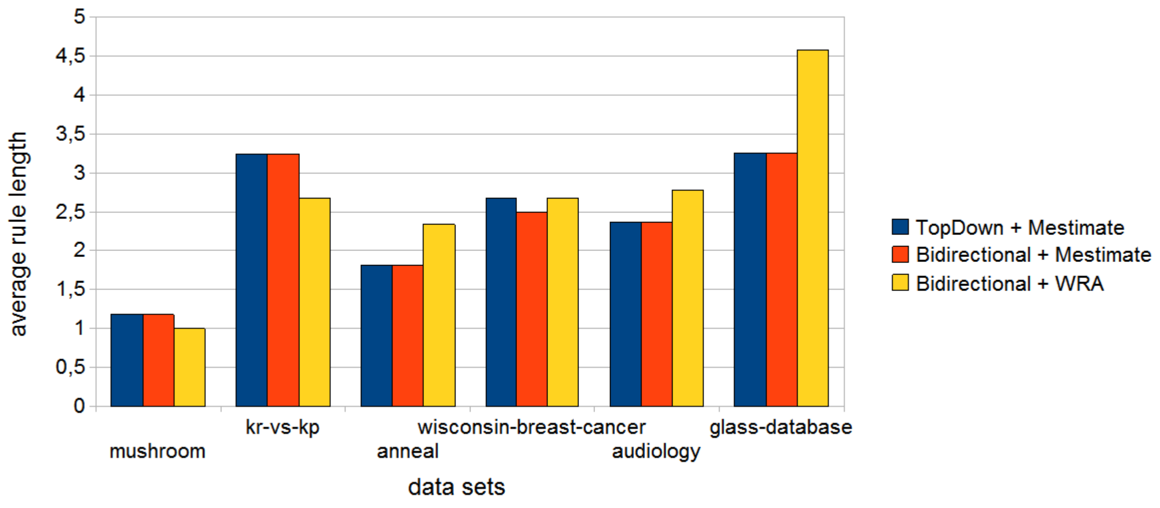
than their predecessor, so if their predecessor does not cover any positive examples, neither will the specializations, so it is impossible to get to a rule that covers more positive than negative examples by specialization. The generalizations do potentially cover more examples than their predecessor but they cannot cover a lower number of negative examples than their predecessor covered. Hopefully they cover more positive examples without increasing the number of negative examples covered, but even then it will probably take some refinement steps to reach a rule that actually does cover more positive than negative examples. In the case where an empty rule set is learned, it seems that findBestRule did not reach any of these rules, assuming that if it had reached them, it had returned one of those instead of one which does not cover more positive than negative examples because the latter should not have a better evaluation than the rules which do cover more positive than negative examples. There are two possible reasons why the better rules could not be reached: If every path to the rules covering more positive than negative examples requires that the algorithm at one point proceeds to a rule having a lower evaluation than the current rule, it cannot reach these rules because of the strictly greater restriction. The second possibility why the path to better rules could be blocked is that a rule in the path fires the stopping criterion within findBestRule and is therefore never added to the list of refinements that can be used in the next iteration. The latter, however, is unlikely to be the source of the problem here, since the stopping criterion used is NoNegativesCoveredStop, which means that a rule is not further refined if it covers no negative examples. Starting with a rule that covers only one negative and no positive examples, this will only hold true for the rule's specializations, which should not be necessary in getting to a rule which covers more positive than negative examples. The most plausible explanation is therefore that findBestRule was not able to proceed to better rules due to the strictly greater restriction. It then returns a rule covering more negative than positive examples, which leads to the immediate termination of the whole learning process because the rule stopping criterion CoverageRuleStop fires, and results in an empty rule set being returned as classifier.

One conclusion from this is that the effect of the strictly greater restriction seems to limit the choices of the algorithm much more when starting with a very specific rule than when starting with the most general rule possible. Or maybe it is just as restrictive when starting with an empty rule, but good solutions are closer to the empty rule than to the rule generated from an example, which means it is more likely that they can be reached when starting with the empty rule. Whichever one is the case, it can be concluded that maybe the strictly greater restriction does indeed limit the freedom of the bidirectional approach too much, especially when starting with a very specific rule. Dropping the strictly greater restriction in favour of the restriction which only prohibits the algorithm from examining already visited rules again might improve the quality of the rule sets the bidirectional rule learner can produce. On the other hand, the algorithm's run time might then be problematic for data sets with a large number of possible conditions. Further empirical analysis would be necessary to determine if bidirectional search can be improved by a less restrictive strategy to avoid infinite loops.

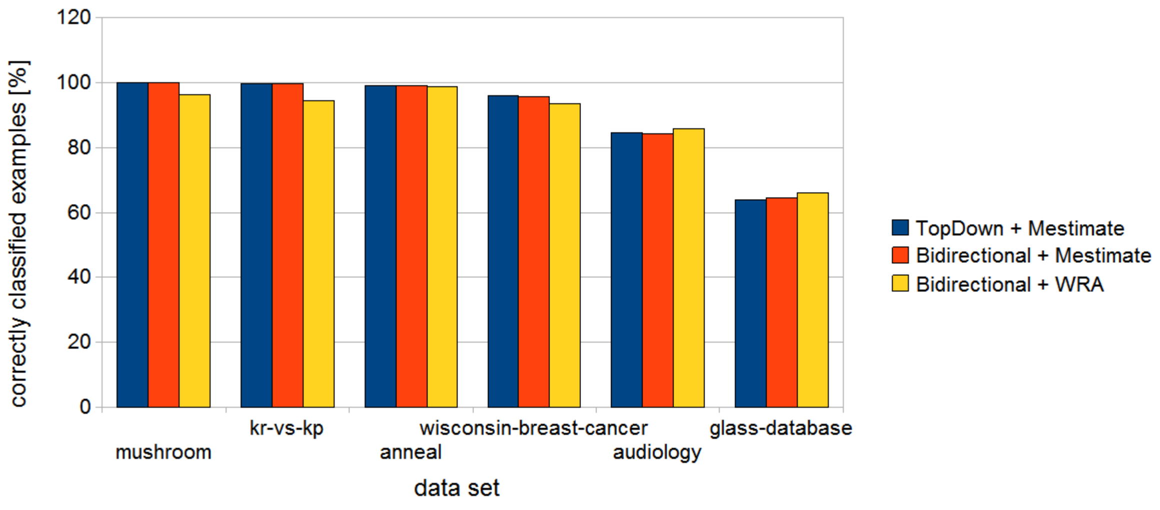
The second conclusion is that it is probably not wise to start the search with a rule generated from a negative example, since this rule is unlikely to be close to a good solution. It would be more sensible to change the implementation of the RandomRuleInitializer so that it chooses a positive example to create an initial rule. It is possible that even then, starting with a rule generated from an example is not a good idea when performing bidirectional search, especially when the data set in question contains many attributes, which results in an initial rule that contains many conditions. Since an optimum solution is probably more general than a specific example, it may be better to start with the empty rule, or some rule with a length in between those two extremes. Possibly other methods can be tried and evaluated to generate a good initial rule for bidirectional search. Finding a heuristics that produces good starting rules for the bidirectional algorithm might improve both its run time and the results it produces, therefore this is one of the possible topics for further research into



(a) Number of rules



(b) Average rule length



(c) Correctly classified examples (percentage of total number of examples in the data set)

Figure 4.2: Detailed statistics on selected data sets for the classifiers 1, 2 and 8 from the test described in chapter 4.2.1.

Figure 4.3: Classifier the Laplace/RandomRuleInitializer variant learned for the anneal data set

```
RuleSet:
Class = c5 :- family = TN, product_type = C, steel = -, carbon < 0, hardness < 0, temper_rolling = -,
condition = A, formability = 1, strength < 0, non_ageing = -, surface_finish = -, surface_quality = -,
enamelability = -, bc = -, bf = -, bt = -, bw_me = -, bl = -, m = -, chrom = C, phos = -, cbond = -,
marvi = -, exptl = -, ferro = -, corr = -, blue_bright_varn_clean = -, lustre = -, jurofm = -, s = -,
p = -, shape = SHEET, thick < 0.5, width < 609.9, len < 612, oil = -, bore = 0000, packing = -.
[1|0] Val: 0.667
```

Figure 4.4: Classifier the Laplace/TopDownRuleInitializer variant learned for the anneal data set

```
RuleSet:
Class = c1 :- exptl = Y.  [2|0] Val: 0.75
Class = c1 :- strength >= 650.  [1|0] Val: 0.667
Class = c1 :- blue_bright_varn_clean = V.  [1|0] Val: 0.667
Class = c1 :- steel = S, carbon < 0.  [3|0] Val: 0.8
Class = c1 :- carbon >= 1.5, carbon < 3.5.  [1|0] Val: 0.667
Class = cU :- hardness >= 75, width >= 50.  [31|0] Val: 0.97
Class = cU :- hardness >= 82.5.  [1|0] Val: 0.667
Class = cU :- thick >= 3.201, carbon >= 9.  [1|0] Val: 0.667
Class = cU :- carbon >= 9, carbon < 27.5.  [1|0] Val: 0.667
Class = c5 :- family = TN.  [60|0] Val: 0.984
Class = c2 :- surface_finish = P.  [8|0] Val: 0.9
Class = c2 :- strength >= 550.  [9|0] Val: 0.909
Class = c2 :- enamelability = 2.  [8|0] Val: 0.9
Class = c2 :- width >= 1410.05, condition = S.  [11|0] Val: 0.923
Class = c2 :- surface_quality = -, formability = 3.  [21|0] Val: 0.957
Class = c2 :- surface_quality = -, bl = Y.  [13|0] Val: 0.933
Class = c2 :- surface_quality = -, bw_me = B.  [3|0] Val: 0.8
Class = c2 :- surface_quality = -, condition = S, steel = R.  [15|0] Val: 0.941
```

bidirectional rule learning. It is possible, however, that the RandomRuleInitializer can be used for bottom up search. It does not return the most specific rule possible, but still very specific rules. If they are sufficiently specific to work well as a starting point for bottom up search could be determined by further empirical analysis.

A third conclusion is that maybe for the performance of bidirectional rule learning, the choice of the rule stopping criterion is also important. In the case of the top down search strategy starting with an empty rule, finding a rule that does not cover more positive than negative examples means that during the whole refinement process, findBestRule has not seen any better rule than this one, and even if it had specialized it any further, it would probably not have been able to improve its quality. For a rule that covers few or no positive examples, the set of examples will not be much different or not different at all after deleting the positive examples the rule does cover. Restarting findBestRule again with a data set that has not changed or only slightly will probably not return much better results if findBestRule applies the same strategy again to find new rules, starting with the empty rule. It is therefore reasonable to terminate the whole learning process at this point. When findBestRule starts with a rule generated from a random example, however, a new call to findBestRule could produce completely different results even if the data set it is given has not changed much or not at all. Therefore it is still possible that better rules will be generated after a rule not covering more positive than negative examples has been detected. Maybe one should not add this rule to the resulting rule set (or apply post processing that removes it later), but it would still make sense to continue searching for better rules. It may be interesting to evaluate the performance of bidirectional search when initializing findBestRule with a rule generated from a random example, or any randomly generated rule, with different rule stopping criterions.

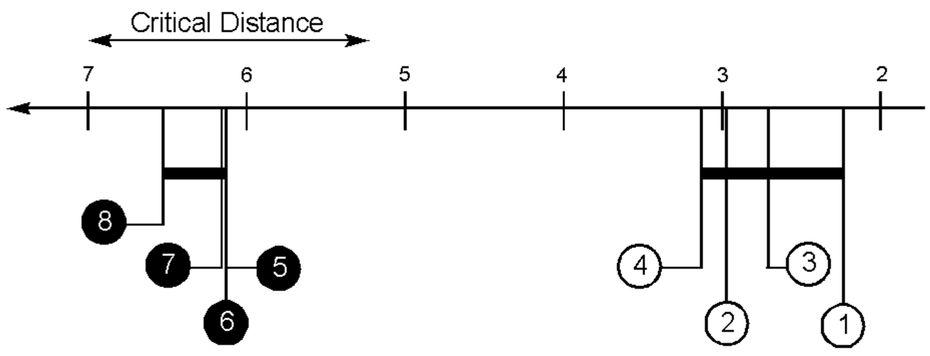


Figure 4.5: Results of significance test on evaluation run 4.2.2. The numbers of the algorithms correspond to the ones given in table 4.9. Algorithms using a RandomRuleInitializer are displayed as white numbers in a black circle while algorithms with TopDownRuleInitializer are displayed with a black number in a white circle. The critical distance shown is the one for the 0.01 level. Both the groups of algorithms with RandomRuleInitializer and with TopDownRuleInitializer are connected with all other algorithms in the group but not with the algorithms from the other group, because the performance of the two groups of algorithms differs significantly on the 0.01 level, with the group using the TopDownRuleInitializer being the better one.

4.3.3 Results of comparison of bidirectional without and top down with post processing

The results for the test described in chapter 4.2.3 can be found in table 4.10. The visualization of the results of the post-hoc Nemenyi test is shown in figure 4.6. The result of the evaluation is that the algorithms differ on the 0.01 level in the Friedman test. In the post-hoc Nemenyi test, Ripper and the bidirectional variant of Ripper which uses Correlation as heuristic are not significantly different on the 0.01, 0.5 and 0.1 level, but the bidirectional variant of Ripper using FoilGain (like the original Ripper configuration) performs significantly worse than those two on all significance levels.

Looking at the details given in table 4.11, one can observe that the bidirectional variant using FoilGain prefers classifiers with very few rules and a low average rule length. The accuracy it achieves with these classifiers is always worse than Ripper's accuracy on the same data set, and in all but one cases worse than that of Bidirectional with Correlation. Especially on the two smallest data sets, Bidirectional with FoilGain does not even achieve an accuracy of 50 %. It can be concluded that the bidirectional strategy does not work well in this configuration. Since the other bidirectional variant using Correlation as heuristics performs significantly better, it is very probable that the bidirectional strategy does not work well with FoilGain.

Comparing Ripper and the bidirectional variant with Correlation in detail reveals that in this case, bidirectional is not always the one which prefers simpler rules. In some cases, bidirectional uses the simpler rule set and in other cases Ripper uses the simpler rule set, with Ripper achieving a higher accuracy in the cross validation in three cases and Bidirectional with Correlation in two cases. In the case of wisconsin-breast-cancer, they achieve the same accuracy. In this case, the model of Bidirectional with Correlation has the smaller number of rules and conditions while Ripper's model refers to less attributes and has a smaller average rule length, so it is not possible to say that one of them is more general than the other. For the two smallest data sets, audiology and glass-database, Ripper uses more general models than Bidirectional with Correlation. For audiology, Bidirectional with Correlation achieves a higher accuracy while for glass-database, Ripper's classifier is slightly more accurate. For the two largest data sets, mushroom and kr-vs-kp, Ripper uses more or an equal number of rules, uses more conditions, refers to more attributes and the average rule length is higher. In this case, Ripper's model is less general but more accurate than that of Bidirectional with Correlation.

It can be concluded that neither Ripper nor Bidirectional with Correlation always prefers more general or more complex models. Ripper performs better than Bidirectional with Correlation, but not significantly better. It appears that the bidirectional strategy, when used with the right heuristics, is almost as good as Ripper. This indicates that Ripper's good performance might be due to the fact it prunes the rules after learning them, which gives it the same possibilities bidirectional search has. Since the difference in performance between Ripper and Bidirectional with Correlation is not big, the fact that the pruning is done on a data set that is independent from the training set does not seem to be the main factor of influence for the quality of the learned classifiers, and neither does the post processing. However, these two things nevertheless seem to be important since Ripper does perform better than Bidrectional with Correlation. Something else that might influence the performance of the bidirectional approach in a negative way could be the strictly greater restriction, since that is another difference between the two algorithms.

In summary, the results show that the bidirectional rule learner without post processing is not able to perform better than Ripper. However, neither does it perform significantly worse, although it has a lower rank than that of Ripper. Neither one of the strategies can be identified as preferring more general but accurate models.

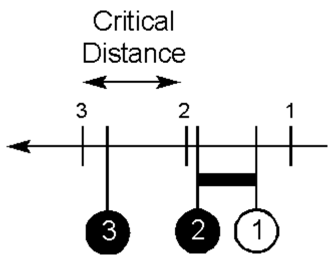


Figure 4.6: Results of significance test on evaluation run 4.2.3. The numbers of the algorithms correspond to the ones given in table 4.10. The two bidirectional variants are displayed with a white number in a black circle and Ripper is displayed with a black number in a white circle. Groups of algorithms are connected if their performance does not differ significantly on the 0.1 level. In this case, Ripper (algorithm 1) and its bidirectional variant using Correlation and no post processing do not differ significantly in their performance, but Bidirectional with FoilGain is significantly worse than those two on the 0.01 level. The critical distance displayed is that for the 0.01 level.

4.3.4 Results of comparison of the different bidirectional variants

In the following, results of the comparison between bidirectional algorithms using the TopDownRuleInitializer will be discussed. These results are obtained from all three tests.

Although no significant difference can be detected, it should be noted that both in the first (chapter 4.3.1) and in the second test (chapter 4.3.2), Bidirectional with m-estimate is the algorithm with the best rank of all bidirectional variants. In the first test, it is followed by Bidirectional with Laplace and then Bidirectional with Correlation while in the second test, Bidirectional with Correlation comes in second and is followed by Bidirectional with Laplace. The worst rank of all the Bidirectional variants with TopDownRuleInitializer occupies Bidirectional with WRA. Although the difference is not significant on any level, it appears that the Bidirectional approach works best with the heuristic m-estimate out of the heuristics that were tested. In the third test (chapter 4.3.3), Bidirectional with FoilGain performs significantly worse than Bidirectional with Correlation. It can be concluded that bidirectional search does not work well with FoilGain. Therefore, a different heuristics should be chosen when applying bidirectional search to rule learning tasks.

Table 4.7: Detailed results for some of the algorithms and data sets in test 4.2.1.

(a) TopDown with *m*-estimate

Data Set	Number of Rules	Number of Conditions	Referred Attributes	Average Rule Length	Accuracy in Cross Validation [%]
mushroom	11	13	13	1.18	100.00 ± 0.63
kr-vs-kp	21	68	68	3.24	99.41 ± 1.92
anneal	10	18	15	1.8	98.87 ± 1.05
wisconsin-breast-cancer	6	16	15	2.67	95.99 ± 1.97
audiology	30	71	71	2.37	84.51 ± 1.76
glass-database	16	52	45	3.25	64.02 ± 1.52

(b) Bidirectional with *m*-estimate

Data Set	Number of Rules	Number of Conditions	Referred Attributes	Average Rule Length	Accuracy in Cross Validation [%]
mushroom	11	13	13	1.18	100.00 ± 0.63
kr-vs-kp	21	68	68	3.24	99.41 ± 1.92
anneal	10	18	16	1.8	98.87 ± 1.05
wisconsin-breast-cancer	6	15	15	2.5	95.71 ± 1.97
audiology	30	71	71	2.37	84.07 ± 1.73
glass-database	16	52	46	3.25	64.49 ± 1.41

(c) Bidirectional with WRA

Data Set	Number of Rules	Number of Conditions	Referred Attributes	Average Rule Length	Accuracy in Cross Validation [%]
mushroom	4	4	4	1.00	96.27 ± 6.76
kr-vs-kp	3	8	8	2.67	94.34 ± 4.44
anneal	6	14	13	2.33	98.50 ± 0.77
wisconsin-breast-cancer	3	8	8	2.67	93.42 ± 2.63
audiology	23	64	64	2.78	85.84 ± 1.61
glass-database	7	32	29	4.57	65.89 ± 1.05

(d) Bidirectional with Laplace

Data Set	Number of Rules	Number of Conditions	Referred Attributes	Average Rule Length	Accuracy in Cross Validation [%]
mushroom	11	13	13	1.18	100.00 ± 0.63
kr-vs-kp	41	142	142	3.46	99.16 ± 1.76
anneal	18	29	27	1.61	99.37 ± 1.05
wisconsin-breast-cancer	22	48	44	2.18	96.14 ± 1.90
audiology	59	118	118	2.0	77.88 ± 1.84
glass-database	50	103	86	2.06	63.08 ± 1.52

Table 4.8: Detailed results for some of the data sets and algorithms in test 4.2.2.

(a) Results for Laplace with RandomRuleInitializer

Data Set	**Number of Rules**	**Number of Conditions**	**Referred Attributes**	**Average Rule Length**	**Correctly classified Examples in Cross Validation [%]**
mushroom	0	0	0	0.0	51.80 ± 0.89
kr-vs-kp	0	0	0	0.0	52.22 ± 0.95
anneal	1	38	38	38.0	76.19 ± 0.89
wisconsin-breast-cancer	2	20	18	10.0	66.24 ± 1.64
audiology	0	0	0	0.0	25.22 ± 0.84
glass-database	1	9	9	9.0	35.51 ± 0.77

(b) Results for Laplace with TopDownRuleInitializer

Data Set	**Number of Rules**	**Number of Conditions**	**Referred Attributes**	**Average Rule Length**	**Correctly classified Examples in Cross Validation [%]**
mushroom	11	13	13	1.18	100.00 ± 0.63
kr-vs-kp	41	142	142	3.46	99.16 ± 1.76
anneal	18	29	27	1.61	99.37 ± 1.05
wisconsin-breast-cancer	22	48	44	2.18	96.14 ± 1.90
audiology	59	118	118	2.0	77.88 ± 1.84
glass-database	50	103	86	2.06	63.08 ± 1.52

Table 4.9: Results of test 4.2.2: All configurations and their macro and micro average and ranks in the cross validation. All algorithms use a bidirectional refiner and strictlyGreater is set to true. The TopDownRuleInitializer uses the emtpy rule as initial rule, the RandomRuleInitalizer uses a rule generated from a randomly chosen example as initial rule.

Number	**Heuristic**	**Rule Initializer**	**Macro Average**	**Micro Average**	**Average Rank**
1	*m*-estimate	TopDown	86.516	93.581	2.237
2	Laplace	TopDown	85.587	92.921	2.974
3	Correlation	TopDown	84.890	92.343	2.711
4	WRA	TopDown	83.198	90.228	3.132
5	*m*-estimate	Random	55.091	59.229	6.132
6	WRA	Random	54.944	59.087	6.132
7	Correlation	Random	54.942	59.019	6.158
8	Laplace	Random	54.773	58.931	6.526

Table 4.10: Results of test 4.2.3: All configurations and their macro and micro average and ranks in the cross validation. The algorithms are all derived from the Ripper configuration, with the TopDown variant being the original Ripper configuration. The bidirectional variants are the same except that they use a bidirectional refiner, no post processing, strictlyGreater is set to true, and once FoilGain is used as heuristic like in the original Ripper configuration and once Correlation.

Number	Rule Refiner	Heuristic	Post Processor	Pruning Set	strictly Greater	Macro Average	Micro Average	Average Rank
1	TopDown	FoilGain	Ripper (2 optimizations)	33.34 %	false	85.937	92.985	1.342
2	Bidirectional	Correlation	NoOp	none used	true	84.990	92.414	1.895
3	Bidirectional	FoilGain	NoOp	none used	true	67.017	79.214	2.763

Table 4.11: Detailed results for all algorithms and some of the data sets in test 4.2.3.

(a) Ripper (Algorithm 1)

Data Set	Number of Rules	Number of Conditions	Referred Attributes	Average Rule Length	Correctly classified Examples in Cross Validation [%]
mushroom	7	12	12	1.71	100.00 ± 0.63
kr-vs-kp	13	38	38	2.92	99.28 ± 1.38
anneal	9	16	15	1.78	97.37 ± 1.38
wisconsin-breast-cancer	5	10	8	2.0	95.14 ± 2.35
audiology	16	24	24	1.5	73.45 ± 1.18
glass-database	7	16	15	2.29	69.63 ± 2.07

(b) Bidirectional with Correlation (Algorithm 2)

Data Set	Number of Rules	Number of Conditions	Referred Attributes	Average Rule Length	Correctly classified Examples in Cross Validation [%]
mushroom	7	8	8	1.14	98.18 ± 3.32
kr-vs-kp	7	19	19	2.71	98.22 ± 2.92
anneal	6	11	10	1.83	97.99 ± 0.77
wisconsin-breast-cancer	3	9	9	3.0	95.14 ± 2.39
audiology	21	55	55	2.62	75.66 ± 1.76
glass-database	9	32	31	3.56	69.16 ± 1.73

(c) Bidirectional with FoilGain (Algorithm 3)

Data Set	Number of Rules	Number of Conditions	Referred Attributes	Average Rule Length	Correctly classified Examples in Cross Validation [%]
mushroom	6	9	9	1.5	99.98 ± 0.89
kr-vs-kp	1	2	2	2.0	72.78 ± 22.64
anneal	4	6	6	1.5	87.97 ± 0.77
wisconsin-breast-cancer	0	0	0	0.0	65.52 ± 0.89
audiology	12	20	20	1.67	41.15 ± 1.30
glass-database	3	6	6	2.0	45.79 ± 1.10

5 Summary and Future Prospects

5.1 Summary

While there exist earlier approaches to apply a bidirectional search strategy to rule learning tasks, they have not yet been extensively tested. In order to test the bidirectional search strategy and to enhance the SeCo rule learning framework by a bidirectional learner, such a learner has been developed and implemented. It makes use of the existing top down learner and a newly implemented bottom up learner in order to compute both the specializations and the generalizations of a given rule. To prevent it from getting stuck in an infinite loop, an additional restriction is added that requires the candidate rule used in one iteration to have a higher rule value than its predecessor, which means that during the refinement process, the rule learner can only proceed to the next rule in a strictly ascending fashion and has to stop if the current rule has no successors with a higher value than the rule itself. That way, infinite loops are avoided but the bidirectional rule learner may find a local optimum rather than a global optimum because once it has reached a local optimum, it cannot leave it since all neighbouring rules have a lower value than the current rule. In order to give the algorithm more possibilities, it has been provided with the possibility to not only start with the empty rule as the top down learner has to but also to start with a rule generated by randomly converting on of the examples from the data set into a rule, which might be closer to a good solution and therefore speed up the learning process and maybe avoid certain local optima.

In the empirical analysis performed, it showed that the bidirectional rule learner does not perform significantly better than the top down learner. There even are two variants, namely Top Down with m-estimate and Bdirectional with WRA, where the top down learner performs significantly better than the bidirectional one. When looking at the results in detail, however, it becomes obvious that the basic bidirectional and top down approaches work more or less equally well. Differences in performance are more likely caused by the use of different heuristics than by the different search strategies.

Furthermore, the empirical analysis showed that starting with a rule generated from a random example does not improve the bidirectional rule learner. Instead, the variants with a random starting rule perform significantly worse than their counterparts with the empty starting rule. This is the case because a rule generated from a randomly chosen example can also be one generated from a negative example, which implies that it is not a good starting point. From this point no rules with much better quality can be reached due to the strictly greater restriction, and the rule stopping criterion terminates the search for better rules too early so it cannot start a second time with a different starting rule. These results indicate that finding a good starting point is important and that using an example as starting rule does not work well for bidirectional search. Maybe it can be used for bottom up search. Additionally, the strictly greater restrictions may be too restrictive and should be evaluated against a visited marker. Another conclusion is that the choice of the rule stopping criterion is also important for bidirectional search to work well.

The result that Bidirectional with Correlation performs almost as good as Ripper and neither better nor significantly worse does neither advise nor discourage the use of a bidirectional refinement strategy. It does, however, indicate that Ripper's good performance is to an important part due to the fact that it prunes the rules it has learned.

The results from the empirical analysis indicate some points where the bidirectional strategy could be improved. Different initial rules, visited markers instead of the strictly greater restriction and different choices for stopping criteria could be regarded as possible improvements of the bidirectional strategy. It is possible that the top down strategy has a natural advantage over the bidirectional approach because top down search is more common and therefore, the components with which the search strategy is combined are likely to be more optimized towards a top down strategy. If that is the case, it may be overcome by fine tuning some of the components to bidirectional search or by developing new components, like a good rule initializer, to improve the bidirectional algorithm as a whole.

5.2 Future Prospects

Although the bidirectional learner did not perform significantly better than the top down learner in the tests performed, neither did it perform significantly worse. In many cases, it learned rule sets that are much simpler than those the top down learner produces and their accuracy in the cross validation was almost as good as that of the top down leaner's rule sets. From these results, it can be concluded that the bidirectional rule learner might have an advantage over the top down learner when provided with further improvements. A possible improvement would be to investigate different starting rules and their impact on the performance of the algorithm. When starting at a point which is already close to an optimum solution, the bidirectional rule learner might be both faster and able to reach a very good solution the top down learner might not have access to if deciding to go into a different direction early in the search process. Finding a good heuristics for choosing a good starting rule is one of the possible improvements that can be applied to the bidirectional rule learner. Another possibility to improve the algorithm is to compare the variant requiring rules to have a strictly greater value than their predecessor to the variant which only stores visited rules and requires them not to be visited again. Maybe requiring the bidirectional rule learner to advance only to rules with a strictly greater value does indeed restrict its freedom too much and makes it too susceptible to local optimums. Giving the algorithm more possibilities could improve its performance.

The implementation of the bidirectional rule learner tested within this paper is only a very basic one which is very similar to the top down rule learner. It is possible that it does not work well because many of its surrounding components are optimized for a top down strategy and need to be further adapted towards the bidirectional strategy to give the bidirectional learner the chance to fully play out its advantages.

Furthermore, some changes can be made in terms of elegance of the implementation. At the moment, it is checked with an extra condition in findBestRule whether the rules selected for the next refinement step have a strictly greater value than their predecessor. However, this can also be implemented using an IRuleFilter that filters out rules that do not fulfill this condition.

6 Appendix

6.1 Documentation of the Implementation

6.1.1 Changes within the framework

This section gives an overview of all the changes that were made within the framework with regard to this paper. Table 6.1 lists all changes that were necessary for the algorithm itself, table 6.2 lists those changes that were needed in order to test the implementation.

Table 6.1: Overview of all changes made to the SeCo framework in order to implement the bidirectional rule learner

Location	Description
seco.learners.bidirectional	added package for the bidirectional learner
seco.learners.bidirectional.BidirectionalRefiner	added bidirectional refiner class
seco.learners.bottomup	added package for the bottom up learner
seco.learners.bottomup.BottomUpRefiner	added bottom up refiner class
seco.learners.refiner	added package for an abstract refiner class
seco.learners.refiner.AbstractRefiner	added AbstractRefiner which functions as super class for all the RuleRefiners. Includes the method evaluateRule from TopDownRefiner because it is needed in all refiners, and other functions common to all refiners.
added method that generalizes a CandidateRule	seco.models.CandidateRule.generalize(int)
seco.learners.core.AbstractSeco	added new private class variable "m_strictlyGreater" which defaults to false. It specifies whether the value of refinements from one iteration must be strictly greater than their predecessor's value in order for the refinements to be allowed as candidates for the next iteration
seco.learners.core.AbstractSeco.setProperty	added another case in setProperty: if name is "strictlyGreater", set m_strictlyGreater to the boolean value that is given
seco.learners.core.AbstractSeco.findBestRule	at the end of the iteration a condition is added which checks, additionally to "t.getTruePositive() >= m_minNo && fprune", if m_strictlyGreater is false or if the refinement's value is indeed higher than that of it's predecessor - if this is true, the refinement may be added, depending on the other condition's values. If it is not true, the refinement will not be added to the list of candidates for the next iteration.
seco.learners.components.RandomRuleInitializer	IRuleInitializer that returns a rule generated from a randomly chosen example as initial rule
seco.models.CandidateRule	Added a constructor which takes an instance as parameter and transforms it into a rule.

6.1.2 Testing

In this section, the tests performed on the implementation will be described. Table 6.3 lists tested inputs for the function seco.models.Rule.equals(Object), table 6.4 lists tested inputs for seco.models.RuleSet.equals(Object). Positive inputs are those which should result in the method returning true, negative inputs are those for a result of false or an error message is expected.

Table 6.2: Overview of all changes made to the SeCo framework in order to test the implementation of the bidirectional rule learner

Location	Description
seco.learners.bidirectional.tests	added package for bidirectional learner's test cases
seco.learners.bottomup.tests.testBottomUpRefiner	added JUnit test case for bottom up refiner
seco.learners.bottomup.tests	added package for bottom up learner's test cases
seco.models.Rule.equals(Object)	added method that returns true if and only if both rules have the same condition as head, and contain the same conditions (in arbitrary order, duplicates counted as just one condition) within their body (Note: when both heads are null, they count as the same. This could be implemented differently, e.g. if one head is null, equals always returns null; one has to decide what makes the most sense in the given context. For the purpose of this paper, it was irrelevant, therefore it was left the way just described.)
seco.models.RuleSet.equals(Object)	added method that returns true if and only if each one of the RuleSets is a subset of the other, i.e. they are equal.
seco.models.tests	added test package for the tests for the equals-methods mentioned above
seco.models.tests.TestRule	class which tests the equals method of the Rule class
seco.models.tests.TestRuleSet	class which tests the equals method of the RuleSet class
seco.tests	package added for an abstract super class for all the tests implemented
seco.tests.SeCoTestCase	added abstract super class containing some methods common to all tests implemented

Table 6.3: Tested values for the input parameters of seco.models.Rule.equals(Object)

Value	positive or negative?
rule with empty head and empty body	positive
rule with empty body (default rule)	positive
rule with duplicate conditions	positive
permutations of conditions	positive
rules that are same object	positive
rules that are copy or clone of each other	positive
null (as only the rule to compare to can be null, this rule must be non-null or the method could not be called)	negative
some rules that are not equal	negative

Table 6.4: Tested values for the input parameters of seco.models.RuleSet.equals(Object)

Value	positive or negative?
empty set	positive
rule set with duplicate rules	positive
rule set with permutations of rules	positive
rule sets that are same object	positive
rule sets that are copy or clone of each other	positive
null (as only the rule set to compare to can be null - this rule set must be non-null or the method could not be called)	negative
some rule sets that are not equal	negative

7 Bibliography

[Dem06] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *JOURNAL OF MACHINE LEARNING RESEARCH*, 7:1–30, JAN 2006.

[Für99] Johannes Fürnkranz. Separate-and-Conquer Rule Learning. *Artificial Intelligence Review*, 13(1):3–54, February 1999.

[FW93] Dieter Fensel and Markus Wiese. Refinement of Rule Sets with JoJo. In Pavel Brazdil, editor, *ECML*, volume 667 of *Lecture Notes in Computer Science*, pages 378–383. Springer, 1993.

[FW94] Dieter Fensel and Markus Wiese. From JoJo to Frog: Extending a bi-directional Search Strategy to a more flexible three-directional Search. In *Beiträge zum 7. Fachgruppentreffen Maschinelles Lernen, Kaiserslautern*, 1994.

[JF10a] Frederik Janssen and Johannes Fürnkranz. On the Quest for Optimal Rule Learning Heuristics. *Machine Learning*, 78(3):343–379, March 2010.

[JF10b] Frederik Janssen and Johannes Fürnkranz. The SeCo-framework for rule learning. Technical Report TUD-KE-2010-02, Knowledge Engineering Group, Technische Universität Darmstadt, 2010.

[Koh95] Ron Kohavi. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *IJCAI*, pages 1137–1145, 1995.

[MIT82] TM MITCHELL. Generalization as Search. *ARTIFICIAL INTELLIGENCE*, 18(2):203–226, 1982.

[Mla93] Dunja Mladenić. Combinatorial Optimization in Inductive Concept Learning. In *ICML*, pages 205–211, 1993.

[WI91] Sholom M. Weiss and Nitin Indurkhya. Reduced complexity rule induction. In *IJCAI'91: Proceedings of the 12th international joint conference on Artificial intelligence*, pages 678–684, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[Wid93] Gerhard Widmer. Combining Knowledge-Based and Instance-Based Learning to Exploit Qualitative Knowledge. *Informatica (Slovenia)*, 17(4), 1993.

[Wie96] Markus Wiese. A Bidirectional ILP Algorithm. In *Proceedings of the MLnet Familiarization Workshop on Data Mining with Inductive Logic Programming (ILP for KDD)*, pages 61–72, 1996.

8 List of Figures

9 List of Tables

10 List of Algorithms