Discovering New Rule Induction Algorithms with Grammar-based Genetic Programming

Gisele L. Pappa and Alex A. Freitas

Computing Laboratory
University of Kent
Canterbury, Kent, CT2 7NF, UK
`glp6, A.A.Freitas@kent.ac.uk`
`http://www.cs.kent.ac.uk/∼aaf`

Summary. Rule induction is a data mining technique used to extract classification rules of the form IF (conditions) THEN (predicted class) from data. The majority of the rule induction algorithms found in the literature follow the sequential covering strategy, which essentially induces one rule at a time until (almost) all the training data is covered by the induced rule set. This strategy describes a basic algorithm composed by several key elements, which can be modified and/or extended to generate new and better rule induction algorithms. With this in mind, this work proposes the use of a grammar-based genetic programming (GGP) algorithm to automatically discover new sequential covering algorithms. The proposed system is evaluated using 20 data sets, and the automatically-discovered rule induction algorithms are compared with four well-known human-designed rule induction algorithms. Results showed that the GGP system is a promising approach to effectively discover new sequential covering algorithms.

1 Introduction

In the classification task of data mining, one way of representing the knowledge discovered by the data mining algorithm consists of a set of classification rules, where each rule has the form: IF (conditions) THEN (predicted class). This knowledge representation has the advantage of being intuitively comprehensible to the user (Iglesia *et al.*, 1996).

There are a number of rule induction algorithms that have been proposed to discover such classification rules (Fürnkranz, 1999, Mitchell, 1997). A particularly popular strategy consists of the sequential covering approach, where in essence the algorithm discovers one rule at a time until (almost) all examples are covered by the discovered rules (i.e., match the conditions of at least one rule). Sequential covering rule induction algorithms are typically greedy, performing a local search in the rule space.

An alternative approach to discover classification rules consists of using an evolutionary algorithm (EA), which performs a more global search in the rule space.

Indeed, there are also a number of EAs for discovering a set of classification rules from a given data set (Freitas, 2002).

In this chapter, however, we propose a very different and pioneering way of using an EA in the context of classification rule discovery. We propose a grammar-based genetic programming (GGP) system that *automatically* discovers new sequential covering rule induction *algorithms* (rather than *rules*). The discovered rule induction algorithms are generic and robust enough to be applicable to virtually any classification data set in any application domain, in the same way that a manually-designed rule induction algorithm is generic and robust enough to be applicable to virtually any classification data set.

The proposed method allows the automatic discovery of new rule induction algorithms potentially quite different from conventional, manually-designed rule induction algorithms. Hence, the automatically-discovered rule induction algorithms can avoid some of the human preconceptions and biases embedded in manually-designed rule induction algorithms, possibly leading to more effective algorithms in challenging application domains.

The first version of the proposed GGP system and its corresponding computational results have been previously published in (Pappa and Freitas, 2006). Nonetheless, one limitation of that version is that the evolved rule induction algorithms could cope only with nominal (categorical) attributes, and not with continuous (real-valued) attributes. In this chapter we describe a new, extended version of the system, which can cope with both nominal and continuous attributes. This new characteristic introduced into the system makes it now suitable for a larger variety of classification data sets, and this fact is reflected in the greater number of data sets used to evaluate the system, namely 20 data sets, whereas our first experiments reported in (Pappa and Freitas, 2006) used only 11 data sets. In addition, this chapter shows and discusses in detail one of the rule induction algorithms automatically generated by the GGP system, a result that was not reported in (Pappa and Freitas, 2006).

The remainder of this paper is organized as follows. Section 2 briefly discusses rule induction algorithms. Section 3 gives a brief overview of GGP. Section 4 introduces the proposed GGP. Section 5 reports the results of several computational experiments. Finally, Section 6 presents the conclusions and describes future research directions.

2 Sequential Covering Rule Induction Algorithms

The sequential covering strategy (also known as separate and conquer) is certainly the most explored and most used strategy to induce rules from data. It was first employed by the algorithms of the AQ family (Michalski, 1969) in the late sixties, and over the years was applied again and again as the basic algorithm in rule induction systems.

The separate and conquer strategy works as follows. It learns a rule from a training set, removes from it the examples covered by the rule, and recursively learns another rule which covers the remaining examples. A rule is said to cover an example e when all the conditions in the antecedent of the rule are satisfied by the example e. For instance, the rule “IF (salary $>$ £ 100,000) THEN rich” covers all the examples in the training set in which the value of salary is greater than £100,000, regardless of the current value of the class attribute of an example. The learning process goes on until a pre-defined criterion is satisfied. This criterion usually requires that all or almost all examples in the training set are covered by a rule.

Algorithms following the sequential covering approach usually differ from each other in four main elements: the representation of the candidate rules, the search mechanisms used to explore the space of candidate rules, the way the candidate rules are evaluated and the pruning method, although the last one can be absent (Fürnkranz, 1999, Witten and Frank, 2005).

Considering the first of these four rule induction algorithms elements, the rule representation has a significant influence in the learning process, since some concepts can be easily expressed in one representation but hardly expressed in others. The most common rule representations used by rule induction algorithms are propositional or first order logic.

The next two elements found in rule induction algorithms determine how the algorithm will explore the space of candidate rules. The first of them, i.e., the search mechanism, is composed by two components: a search strategy and a search method. The search strategy determines the region of the search space where the search starts and its direction, while the search method specifies which specializations/generalizations should be considered.

Broadly speaking, there are three kinds of search strategies: bottom-up, top-down and bi-directional. A bottom-up strategy starts the search with a very specific rule, and iteratively generalizes it. A top-down strategy, in contrast, starts the search with the most general rule and iteratively specializes it. The most general rule is the one that covers all examples in the training set (because it has an empty antecedent, which is always satisfied for any example). At last, a bi-directional search is allowed to generalize or specialize the candidate rules.

Any of these search strategies is complemented with a search method. The search method is a very important part of a rule induction algorithm since it determines which specializations/generalizations will be considered at each specialization/generalizations step. Too many specializations/generalizations are not allowed due to computational time, but too few may disregard good conditions and reduce the chances of finding a good rule. Among the available search methods are the greedy search and the beam search.

The search method is guided by a rule evaluation heuristic, which is the second component found in rule induction algorithms which has influence in the way the rule space is explored. The regions of the search space being explored by a rule induction algorithm can drastically change according to the heuristic chosen to assess a rule while it is being built. Among the heuristics used to estimate the quality of a rule are the information content, information gain, Laplace correction, m-estimate, ls-content, among others (Fürnkranz, 1999).

The first algorithms developed using the sequential covering approach were composed by the three components described so far: a rule representation, a search strategy and a evaluation heuristic. However, these first algorithms searched the data for complete and consistent rule sets. It means they were looking for rules that covered all the examples in the training set (complete) and that covered no negative examples (consistent). These are not common characteristics of any real-world data sets. Hence, pruning methods were introduced to sequential covering algorithms to avoid over-fitting and to handle noisy data.

Pruning methods are divided in two categories: pre- and post-pruning. Pre-pruning methods stop the refinement of the rules before they become too specific or over-fit the data, while post-pruning methods first produce a complete and consistent rule or rule set, and later try to simplify it. When comparing pre- and post-pruning techniques, each of them has its advantages and pitfalls. Though pre-pruning techniques are faster, post-pruning techniques usually produce simpler and more accurate models (at the expense of inefficiency, since some rules are learned and then simply discarded from the model).

In an attempt to solve the problems caused by pre- and post-pruning techniques, some methods combine or integrate them to get the best of both worlds. Some of these systems, for instance, prune each rule right after it is created, instead of waiting for the complete model to be generated (Cohen, 1995).

This section briefly described the main elements which compose a sequential covering rule induction algorithm. Knowledge about these elements and the variety of ways they can be implemented was the base to build the grammar used by the GGP system described in Section 4. For a more complete survey of sequential covering rule induction algorithms and its components the user is referred to (Fürnkranz, 1999, Mitchell, 1997, Witten and Frank, 2005).

3 Grammar-based Genetic Programming

Genetic Programming (GP) (Koza, 1992) is an area of evolutionary computation which aims to automatically evolve computer programs. It works by following Darwin's principle of selection and survival of the fittest, and can be easily adapted to solve a variety of problems. GP's success is backed up by a list of 36 human-competitive solutions, where two created patentable new inventions (gp.org).

Grammar-based Genetic Programming (GGP) is a variation of the classical GP method and, as its name indicates, the main difference among GGP and GP is that the former uses a grammar to create the population of candidate solutions for the targeted problem. The main advantage of using a grammar together with a GP system is that it can include previous knowledge about how the target problem is solved, and so be used to guide the GP search. Moreover, GGP solves a well-known problem in the GP literature, called closure[1].

Grammars (Aho *et al.*, 1986) are simple mechanisms capable of representing very complex structures. Their formal definition was first given by Chomsky in 1950. According to Chomsky, a grammar can be represented by a four-tuple $\{N, T, P, S\}$, where N is a set of non-terminals, T is a set of terminals, P is a set of production rules, and S (a member of N) is the start symbol. The production rules define the language which the grammar represents by combining the grammar symbols.

In this work we are specially interested in context-free grammars (CFG). CFGs are the class of grammars most commonly used with genetic programming, and they are usually described using the Backus Naur Form (BNF) (Naur, 1963).

[1] A traditional GP system creates individuals by combining a set of functions and terminals. The closure property states that every function in the GP function set has to be able to process any value it receives as input. For further details see (Banzhaf *et al.*, 1998).

According to the BNF notation, production rules have the form <expr> ::= <expr><op><expr>, and symbols wrapped in "<>" represent the non-terminals of the grammar. Three special symbols might be used for writing the production rules in BNF: "|","[ ]" and "( )". "|" represents a choice, like in <var> ::=$x|y$, where <*var*> generates the symbol x or y. "[ ]" wraps an optional symbol which may or may not be generated when applying the rule. "( )"is used to group a set of choices together, like in $x ::= k(y|z)$, where x generates k followed by y or z. The application of a production rule from $p \in P$ to some non-terminal $n \in N$ is called a derivation step, and it is represented by the symbol $\Rightarrow$.

Once a grammar that includes background knowledge about the target problem has been defined by the user, a GGP system follows the pseudo-code defined in Algorithm 1.

Algorithm 1: Basic pseudo-code for a GGP system

```
Define a representation for the individuals
Define parameters such as population size, number of generations, crossover,
mutation and reproduction rates
Generate the first GGP population based on the production rules of the
grammar
while maximum number of generations not reached do
    Evaluate the individuals using a pre-defined fitness function
    Select the best individuals, according to the fitness function, to breed
    Perform crossover, mutation and reproduction operations with the
    selected individuals, always producing offspring which are also valid
    according to the defined grammar
Return individual with best fitness
```

Note that each individual represents a candidate solution to the target problem. Hence, since we use a GGP to automatically evolve rule induction algorithms, each individual is a rule induction algorithm, and the grammar gathers knowledge about how rule induction algorithms were previously developed.

4 Discovering New Rule Induction Algorithms

This section describes an extended version of the GGP method proposed in (Pappa and Freitas, 2006). As explained before, the main difference between the method described here and the one described in (Pappa and Freitas, 2006) is that, while the latter could only cope with nominal attributes, the former can cope with both nominal and numerical attributes.

In summary, the proposed GGP works as follows. It creates the first population of individuals based on the production rules of a grammar, which is used to guide the search for new rule induction algorithms. In this population, each GGP individual represents a complete sequential-covering rule induction algorithm, such as CN2 (Clark and Boswell, 1991). As the GGP system is based on the principle of survival of the fittest, a fitness function is associated with each individual in the population, and used to selected a subset of them (through a tournament selection of size 2) to breed and undergo crossover, mutation and reproduction operations. The individuals generated by these operations (which also have to be valid according to the grammar being used) are inserted into a new population, representing a new generation of evolved individuals. The evolution process is carried out until a maximum number of generations is reached.

Note that the main modifications introduced to the system in order to cope with numerical attributes are related to the terminals of the grammar and the way they are implemented. Hence, in this section, we first present the grammar introduced in (Pappa and Freitas, 2006), emphasizing its components which cannot be found in traditional rule induction algorithms, and then we present the modifications necessary to make it cope with numerical attributes. Following the grammar description, we show an example of an individual which can be evolved by the system, and then describe the individuals' evaluation process. Finally, we explain how the evolutionary operators were implemented.

4.1 The grammar

In a GGP system, the grammar is the element which determines the search space of the candidate solutions for a target problem. Hence, in the GGP system proposed here, the grammar contains previous knowledge about how humans design rule induction algorithms, plus some new concepts which were borrowed from other data mining paradigms or created by the authors (and that to the best of the authors' knowledge were never used in the design of sequential-covering rule induction algorithms).

The proposed grammar is presented in Table 1. It uses the BNF terminology introduced earlier, and its *Start* symbol is represented by the non-terminal with the same name. Recall that non-terminals are wrapped into <> symbols, and each of them originates a production rule. Grammar symbols not presented between <> are terminals. In the context of rule induction algorithms, the set of non-terminals and terminals are divided into two subsets. The first subset includes general programming elements, like *if* statements and *for/while* loops, while the second subset includes components directly related to rule induction algorithms, such as *RefineRule* or *PruneRule*.

The non-terminals in the grammar represent high-level operations, like a while loop (*whileLoop*) or the procedure performed to refine a rule (*RefineRule*). The terminals, in turn, represent a very specific operation, like *Add1*, which adds one condition-at-a-time to a candidate rule during the rule refinement process (*RefineRule*). Terminals are always associated with a building block. A building block represents an "atomic operation" (from the grammar's viewpoint) which does not need any more refinements. Building blocks will be very useful during the phase of rule induction code generation, as each of them is associated with a chunk of Java code.

As observed in Table 1, the grammar contains 26 non-terminals (NT), where each NT can generate one or more production rules. Recall that in the BNF notation, used to describe the grammar in Table 1, the symbol "|" separates different production rules, and the symbol "[ ]" wraps an optional symbol (which may or may not be generated when applying the rule). For

Table 1. The grammar used by the GGP (adapted from (Pappa and Freitas, 2006))

```
1-  <Start> ::= (<CreateRuleSet>|<CreateRuleList>) [<PostProcess>].
2-  <CreateRuleSet> ::= forEachClass <whileLoop> endFor
                        <RuleSetTest>.
3-  <CreateRuleList> ::= <whileLoop> <RuleListTest>.
4-  <whileLoop>::= while <condWhile> <CreateOneRule> endWhile.
5-  <condWhile>::= uncoveredNotEmpty |uncoveredGreater
                (10| 20| 90%| 95%| 97%| 99%) trainEx.
6-  <RuleSetTest> ::= lsContent |confidenceLaplace.
7-  <RuleListTest>::= appendRule | prependRule.
8-  <CreateOneRule>::= <InitializeRule> <innerWhile> [<PrePruneRule>]
                       [<RuleStoppingCriterion>].
9-  <InitializeRule> ::= emptyRule| randomExample| typicalExample |
                         <MakeFirstRule>.
10- <MakeFirstRule> ::= NumCond1| NumCond2| NumCond3| NumCond4.
11- <innerWhile> ::= while (candNotEmpty| negNotCovered)
                    <FindRule> endWhile.
12- <FindRule> ::= (<RefineRule>|<innerIf>) <EvaluateRule>
                   [<StoppingCriterion>] <SelectCandidateRules>.
13- <innerIf> ::= if <condIf> then <RefineRule> else <RefineRule>.
14- <condIf> ::=  <condIfExamples> | <condIfRule>.
15- <condIfRule> ::= ruleSizeSmaller (2| 3| 5| 7).
16- <condIfExamples> ::= numCovExp ( >| <)(90%| 95%| 99%).
17- <RefineRule> ::= <AddCond>| <RemoveCond>.
18- <AddCond> ::= Add1| Add2.
19- <RemoveCond>::= Remove1| Remove2.
20- <EvaluateRule>::= confidence | Laplace| infoContent| infoGain.
21- <StoppingCriterion> ::= MinAccuracy (0.6| 0.7| 0.8)|
                            SignificanceTest (0.1| 0.05| 0.025| 0.01).
22- <SelectCandidateRules> ::= 1CR| 2CR| 3CR| 4CR| 5CR| 8CR| 10CR.
23- <PrePruneRule> ::= (1Cond| LastCond| FinalSeqCond) <EvaluateRule>.
24- <RuleStoppingCriterion> ::= accuracyStop (0.5| 0.6| 0.7).
25- <PostProcess> ::= RemoveRule EvaluateModel| <RemoveCondRule>.
26- <RemoveCondRule> ::= (1Cond| 2Cond| FinalSeq) <EvaluateRule>.
```

instance, the NT *Start* generates four production rules: *CreateRuleList*, *CreateRuleSet*, *CreateRuleList PostProcess* and *CreateRuleet PostProcess*. In total, the grammar has 83 production rules, which were carefully generated after a comprehensive study of the main elements of the pseudo-codes of basic rule induction algorithms, which follow the basic process described in Section 2.

In this section, we focus on the new components of the grammar, which are usually not found in traditional rule induction algorithms. The major "new" components inserted to the grammar are:

- The terminal *typicalExample*, which creates a new rule using the concept of typicality, borrowed from the instance-based learning literature (Zhang, 1992). An example is said to be typical if it is very similar to the other examples belonging to the same class it belongs to, and not similar to the other examples belonging to other classes. In other words, a typical example has high intra-class similarity and low inter-class similarity.
- The non-terminal *MakeFirstRule*, which allows the first rule to be initialized with one, two, three or four attribute-value pairs, selected probabilistically from the training data in proportion to their frequency. Attribute-value pairs are selected subject to the restriction that they involve different attributes (to prevent inconsistent rules such as "sex = male AND sex = female").
- The non-terminal *innerIf*, which allows rules to be refined in different ways (e.g. adding or removing one or two conditions-at-a-time to/from the rule) according to the number of conditions they have, or the number of examples the rule list/set covers.
- Although some methods do use rule look-ahead, i.e., they do insert more than one condition-at-a-time to a set of candidate rules, we did not find in the literature any rule induction algorithm which removes two conditions-at-a-time from a rule. This is implemented by the terminal *Remove2*.

Note that the list above shows a set of single components which are new "building blocks" of rule induction algorithms. These components increase the diversity of the candidate rule induction algorithms considerably, but it is the combination of the "standard" and new components which will potentially contribute to the creation of a new rule induction algorithm different from conventional algorithms.

As it will be discussed in Section 4.3, the individuals generated by following the production rules of the grammar are converted into executable rule induction algorithms by using a GGP/Java interface, which reads out an individual and puts together chunks of code associated with the terminals and non-terminals of the grammar contained in that individual.

Hence, in order to modify the grammar and make it cope with data sets containing numerical attributes, the main modifications are introduced in some chunks of Java code associated with the terminals of the grammar. The terminals whose implementation went through major extensions were the ones responsible for refining rules by adding/removing conditions to/from it. They were extended in a way that they can produce algorithms that represent rule conditions of the form "<attribute, operator, value>", where operator is "=" in the case of nominal attributes, and operator is "$\geq$" or "$\leq$" in the case of numerical attributes.

The approach followed by these terminals to generate rule conditions with numerical attributes is similar to the one implemented by the Ripper and C4.5 algorithms, where the values of a numerical attribute are sorted, and all threshold values considered. The best threshold value is chosen according to

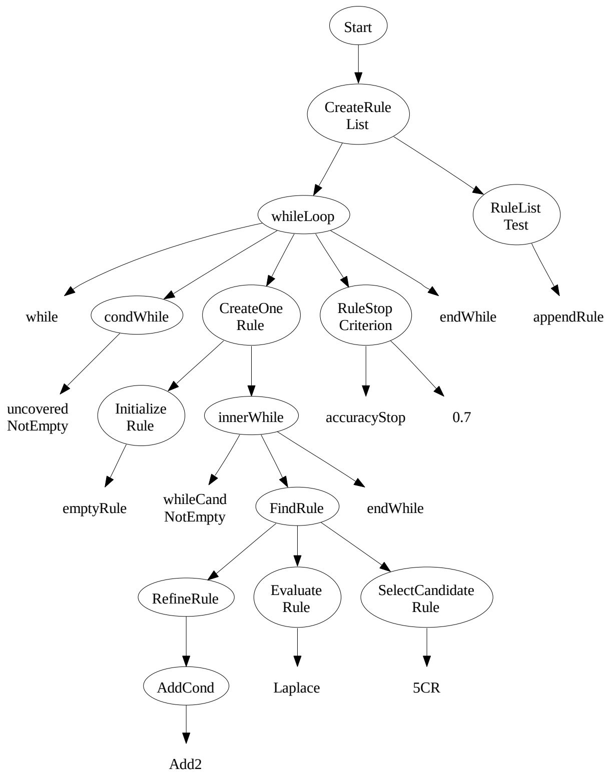

Fig. 1. Example of a GGP Individual (a complete rule induction algorithm)

the information gain associated with that attribute-value pair - see (Witten and Frank, 2005) or (Quinlan, 1993) for details.

By applying the production rules defined by the grammar, the GGP system can generate up to approximately 5 billion different rule induction algorithms (Pappa, 2007). Each of these rule induction algorithms can be represented by an individual in the GGP population. The next section explains how the individuals extracted from the grammar are represented in the proposed GGP system.

4.2 Individual Representation

In a GGP system, each individual represents a candidate solution for the problem being tackled. In this work, each individual represents a complete rule induction algorithm following the sequential covering approach, which can be applied to generate rules for any classification data set.

Figure 1 shows an example of an individual generated by the grammar presented in the previous section. The root of the tree is the non-terminal *Start*. The tree is then derived by the application of production rules for each non-terminal. For example, *Start* (NT 1) generates the non-terminal *CreateRuleList* (NT 3), which in turn produces the non-terminals *whileLoop* and *RuleListTest*. This process is repeated until all the leaf nodes of the tree are terminals.

In order to extract from the tree the pseudo-code of the corresponding rule induction algorithm, we read all the terminals (leaf-nodes) in the tree from left to right. The tree in Figure 1, for example, represents the pseudo-code described in Alg. 2 (shown at the end of Section 5), expressed at a high level of abstraction.

4.3 Individual's Evaluation

An evolutionary algorithm works by selecting the fittest individuals of a population to reproduce and generate new offspring. Individuals are selected based on how good their corresponding candidate solutions are to solve the target problem. In our case, we need to evaluate how good a rule induction algorithm is.

In the rule induction algorithm literature, comparing different classification algorithms is not a straightforward process. There is a variety of metrics which can be used to estimate how good a classifier is, including classification accuracy, ROC analysis (Fawcett, 2003) and sensitivity/specificity. There are studies comparing these different metrics, and showing advantages and disadvantages in using each of them (Flach, 2003, Caruana and Niculescu-Mizil, 2004). Nevertheless, as pointed out by Caruana and Niculescu-Mizil (Caruana and Niculescu-Mizil, 2004), in supervised learning there is one ideal classification model, and "we do not have performance metrics that will reliably assign best performance to the probabilistic true model given finite validation data".

Classification accuracy is still the most common metric used to compare classifiers, although some authors tried to show the pitfalls of using classification accuracy when evaluating induction algorithms (Provost *et al.*, 1998) – specially because it assumes equal misclassification costs and known class distributions – and others tried to introduce ROC analysis as a more robust standard measure. Based on these facts and on the idea of using a simpler measure when first evaluating the individuals produced by the GGP, we chose to use a measure based on accuracy to compose the fitness of the GGP system.

In this framework, a rule induction algorithm RI_A is said to outperform a rule induction algorithm RI_B if RI_A has better classification accuracy in a set of classification problems. Thus, in order to evaluate the rule induction algorithms being evolved, we selected a set of classification problems, and created a meta-training set. The meta-training set consists of a set of data sets, each of them divided as usual into (non-overlapping) training and validation sets.

As illustrated in Figure 2, each individual in the GGP population is decoded into a rule induction algorithm using a GGP/Java interface. The Java code is then compiled, and the resulting rule induction algorithm run in all the

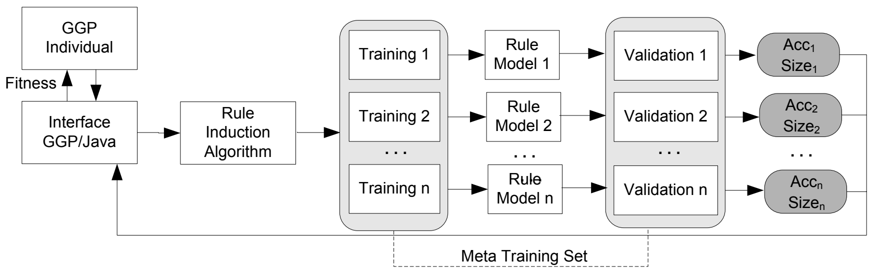

Fig. 2. Fitness evaluation process of a GGP Individual

data sets belonging to the meta-training set. It is a conventional run where, for each data set, a set or list of rules is built using the set of training examples and evaluated using the set of validation examples. It is important to observe that, during preliminary experiments with the GGP, we noticed that the it was suffering from a common problem found when solving predictive data mining tasks: over-fitting. As the same training sets were being accessed by the GGP over and over, the produced rule induction algorithms were over-fitting these data sets. We solved this problem with a simple and effective solution borrowed from the literature on GP for data mining (Bhattacharyya, 1998): at each generation, the data used in the training and validation sets of the data sets in the meta-training set are merged and randomly redistributed. This means that, at each generation, the GGP individuals are evaluated in a different set of validation data, helping to avoid over-fitting.

After the rule induction algorithm is run in all the data sets in the meta-training set, the accuracy in the validation set and the rule lists/sets produced for all data sets are returned. These two measures can be used to calculate a fitness function. In this work, we used as the fitness function the average values of the function described in Eq.(1) over all the data sets in the meta-training set.

$$fit_i = \begin{cases} \frac{Acc_i - DefAcc_i}{1 - DefAcc_i}, \text{ if Acc}_i > \text{DefAcc}_i \\ \frac{Acc_i - DefAcc_i}{DefAcc_i}, \text{ otherwise} \end{cases} \tag{1}$$

According to the definition of fit_i, where i denotes the id of a given data set, if the accuracy obtained by the classifier is better than the default accuracy, the improvement over the default accuracy is normalized, by dividing the absolute value of the improvement by the maximum possible improvement. In the case of a drop in the accuracy with respect to the default accuracy, this difference is normalized by dividing the negative value of the difference by the maximum possible drop (the value of *DefAcc*$_i$). The default accuracy for a given data set is simply the accuracy obtained when using the most frequent class in the training set to classify new examples in the validation (or test) set.

The fitness values obtained by the process just described are used for selecting the best individuals in the population, and passing them onto the new generations. At the end of the evolution process, the individual with the best fitness value is returned as the GGP's final solution.

However, in order to verify if the newly created rule induction algorithm is really effective, we have to test it in a new set of data sets, which where unseen during the GGP's evolution. This new set of data sets was named meta-test set, and it is the accuracy obtained by the discovered rule induction algorithms in these data sets which has to be taken into account when evaluating the GGP system.

4.4 Evolutionary Operators

After individuals are generated and evaluated, they are selected to undergo reproduction, crossover and mutation operations, according to used defined probabilities. The reproduction operator simply copies the selected individual to the new population, without any modifications. The crossover operator, in contrast, involves two selected individuals, and swaps a selected subtree between them. The mutation operator also selects a subtree of one selected individual, and replace it by a new, randomly generated tree.

However, in GGP systems, the new individuals produced by the crossover and mutation operators have to be consistent with the grammar. For instance, when performing crossover the system cannot select a subtree with root *EvaluateRule* to be exchanged with a subtree with root *SelectCandidateRules*, because this would create an invalid individual according to the grammar.

Therefore, crossover operations have to exchange subtrees whose roots contain the same non-terminal. Mutation can be applied to a subtree rooted at a non-terminal or applied to a terminal. In the former case, the subtree undergoing mutation is replaced by a new subtree, produced by keeping the same label in the root of the subtree and then generating the rest of the subtree by a new sequence of applications of production rules, so producing a new derivation subtree. When mutating terminals, the terminal undergoing mutation is replaced by another "compatible" symbol, i.e., a terminal or non-terminal which represents a valid application of the production rule whose antecedent is that terminal's parent in the derivation tree. The probability of mutating a non-terminal is 90%, while the probability of mutating a terminal is 10%.

Figure 3 shows an example of a crossover operation. Note that just part of the individuals are shown, for the sake of simplicity. The process works as follows. Parent 1 has a node probabilistically selected for crossover. In the example illustrated, the chosen node is *RefineRule*. The node *RefineRule* is then searched in the derivation tree of parent 2. As parent 2 has a node named *RefineRule*, their subtrees are swapped, generating child 1 and child

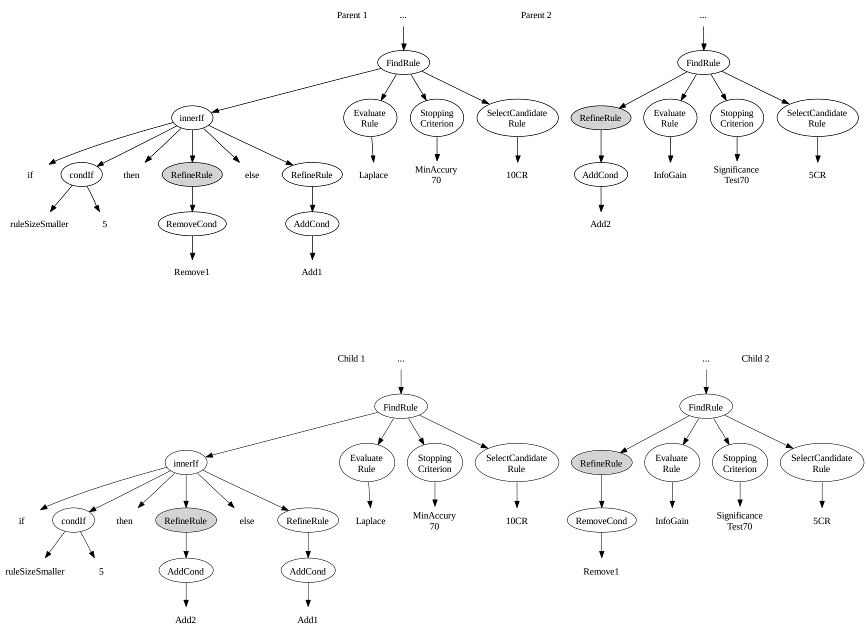

Fig. 3. Example of Crossover in the proposed GGP

2. If *RefineRule* is not present in the tree of parent 2, a new non-terminal is selected from the tree of parent 1. The GGP performs at most 10 attempts to select a node which can be found in both parents. If after 10 attempts it does not happen, both individuals undergo mutation operations.

5 Computational Results and Discussion

In order to test the effectiveness of the proposed GGP system to discover new rule induction algorithms, we have to define two different sets of parameters: (1) the parameters for the GGP system and (2) the data sets used during the training phase of the system.

Table 2 shows the 20 data sets used in the experiments. The figures in the column *Examples* indicate the number of examples present in the training and validation data sets – numbers before and after the "/", respectively, followed by the number of nominal attributes, numerical attributes and classes. The last column shows the default accuracy. It is important to note that during the evolution of the rule induction algorithm by the GGP, for each data set in the meta-training set, each candidate rule induction algorithm (i.e., each GGP individual) is trained with 70% of the examples, and then validated in the

Table 2. Data sets used by the GGP

Data set	Examples	Attributes		Classes	Def. Acc.
		Nomin.	Numer.		(%)
monks-2	169/432	6	-	2	67
monks-3	122/432	6	-	2	52
bal-scale-discr	416/209	4	-	3	46
lymph	98/50	18	-	4	54
zoo	71/28	16	-	7	43
glass	145/69	-	9	7	35.2
pima	513/255	-	8	2	65
hepatitis	104/51	14	6	2	78
vehicle	566/280	-	18	4	26
vowel	660/330	3	10	11	9
crx	461/229	9	6	2	67.7
segment	1540/770	-	19	7	14.3
sonar	139/69	-	60	2	53
heart-c	202/101	7	6	2	54.5
ionosphere	234/117	-	34	2	64
monks-1	124/432	6	-	2	50
mushroom	5416/2708	23	-	2	52
wisconsin	456/227	9	-	2	65
promoters	70/36	58	-	2	50
splice	2553/637	63	-	3	52

remaining 30%. In contrast, in the meta-test set, the evolved rule induction algorithms are evaluated using a well-known 5-fold cross validation procedure (Witten and Frank, 2005).

As our priority was to investigate the influence the GGP parameters have in the quality of the rule induction algorithms produced, we first defined the data sets which will be used in the GGP meta-training and meta-test sets. However, it is not clear how many data sets should be used in each of these meta-sets of data, or what would be the best criteria to distribute them into these two meta-sets. Intuitively, the larger the number of data sets in the meta-training set, the more robust the evolved rule induction algorithm should be. On the other hand, the smaller the number of data sets in the meta-test set, the less information we have about the ability of the evolved rule induction algorithm to obtain a high predictive accuracy for data sets unseen during the evolution of the algorithm.

As a reasonable compromise, the data sets in Table 2 were divided into 2 groups of 10 data sets each. The top 10 sets listed in Table 2 were inserted into the meta-training set, while the bottom 10 data sets formed the meta-test set. We selected the data sets which compose the meta-training set based on the execution time of rule induction algorithms, so that we included in the meta-training set the data sets leading to faster runs of the rule induction algorithms.

After creating the meta-training and meta-test sets, we turned to the GGP parameters: population size, number of generations and crossover, mutation and reproduction rates. In all the experiments reported in this section the population size is set to 100 and the number of generations to 30. These two figures were chosen when evaluating the GGP evolution in preliminary experiments, but are not optimized. Regarding crossover, mutation and reproduction rates, GPs usually use a high rate of crossover and low rates of mutation and reproduction. However, the balance between these three numbers is an open question, and may be very problem dependent (Banzhaf *et al.*, 1998).

In previous experiments, we set the value for the reproduction rate parameter to 0.05, and run the GGP with crossover/mutation rates of 0.5/0.45, 0.6/0.35, 0.7/0.25, 0.8/0.15 and 0.9/0.05, respectively. The results obtained by the GGP when run with these different parameters configurations showed that the system was robust to these variations, producing very similar results with all the configurations. In the experiments reported in this section, the crossover rate was set to 0.7 and the mutation rate to 0.25.

The results obtained by the GGP-derived rule induction algorithms (GGP-RIs) were compared with four well-known rule induction algorithms: the ordered (Clark and Niblett, 1989) and unordered (Clark and Boswell, 1991) versions of CN2, Ripper (Cohen, 1995) and C4.5Rules (Quinlan, 1993). Out of these four algorithms, C4.5Rules is the only one which does not follow the sequential covering approach, which is the approach followed by the GGP-RIs. However, as C4.5Rules has been used as a benchmark algorithm for classification problems for many years, we also included it in our set of baseline comparison algorithms.

It is also important to observe that the current version of the grammar does not include all the components present in Ripper, but does include all the components present in both versions of CN2. In other words, the space of candidate rule induction algorithms searched by the GGP includes CN2, but it does not include C4.5Rules nor the complete version of Ripper.

Table 3 shows the average accuracy obtained by the rule induction algorithms produced by the GGP in 5 different runs, followed by the results of runs of Ordered-CN2, Unordered-CN2, Ripper and C4.5Rules (using default parameter values in all these algorithms). Note that the results reported in Table 3 are the ones obtained in the data sets belonging to the meta-test set (containing data sets unseen during the GGP evolution), and were obtained using a 5-fold cross-validation procedure for each data set. The results obtained by the GGP in the data sets belonging to the meta-training set are not reported here because, as these data sets were seen by the GGP many time during evolution, it is not fair to compare them with any other algorithms. The numbers after the symbol "±" are standard deviations. Results were compared using a statistical t-test with significance level 0.01. Cells in dark gray represent significant wins of the GGP-RIs against the corresponding baseline algorithm, while light gray cells represent significant GGP-RIs' losses.

Table 3. Comparing predictive accuracy rates (%) for the data sets in the meta-test set

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
crx	77.46±3.8	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
segment	95.06±0.26	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	72.34±1.91	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
heart-c	76.72±1.5	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43
ionosphere	87.04±2.2	87.6 ± 2.76	90.52 ± 2.03	89.61 ± 1.75	89.06 ± 2.71
monks-1	99.93±0.07	100 ± 0	100 ± 0	93.84 ± 2.93	100 ± 0
mushroom	99.98±0.02	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
wisconsin	95.58±0.74	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
promoters	78.98±2.93	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	88.68±0.31	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78

In total, Table 3 contains 40 comparative results between GGP-RIs and baseline algorithms – 10 data sets × 4 baseline classification algorithms. Out of theses 40 cases, the accuracy of GGP-RIs was statistically better than the accuracy of a baseline algorithm in three cases, whilst the opposite was true in one case. In the other 36 cases there was no significant difference.

The GGP-RIs' predictive accuracies are statistically better than the C4.5Rules' accuracy in *mushroom* and Unordered-CN2's accuracy in *segment* and *splice*. Naturally, these three cases involve algorithms with the worst accuracy for the respective data set. It is also in a comparison among Ripper and the GGP-RIs in *splice* where the GGP-RIs obtain a significantly worse accuracy than Ripper.

Hence, these experiments lead us to conclude that the GGP-RIs can easily outperform classifiers which are not competitive with the other baseline algorithms. For example, in *splice* the predictive accuracy of Unordered-CN2 is 74.82 ± 2.94, while the other algorithms obtain accuracies close to 90%. In this case, the GGP-RIs can easily find a better accuracy than the one found by Unordered-CN2.

On the other hand, we can say that the GGP was not able to find a rule induction algorithm good enough to outperform the predictive accuracies of Ripper in *splice* because it did not have all the components necessary to do that in its grammar. However, note that the accuracy obtained by Ripper in *splice* is also statistically better than the ones obtained by C4.5Rules and CN2- Ordered when applying a t-test with 0.01 significance level.

Finally, recall that the search space of the GGP includes both Unordered and Ordered CN2. Hence, it seems fair to expect that the GGP-RIs would never obtain a predictive accuracy significantly worse than either version of CN2. Indeed, this was the case in the experiments reported in Table 3, where the GGP-RIs significantly outperformed Unordered-CN2 in two cases (dark

Algorithm 2: Example of a Decision List Algorithm created by the GGP

```
RuleList = $\emptyset$
repeat
    bestRule = an empty rule
    candidateRules = $\emptyset$
    candidateRules = candidateRules $\cup$ bestRule
    while candidateRules $\neq \emptyset$ do
        newCandidateRules = $\emptyset$
        for each candidateRule CR do
            Add 2 conditions-at-a-time to CR
            Evaluate CR using the Laplace estimation
            newCandidateRules = newCandidateRules $\cup$ CR
        candidateRules = 5 best rules selected from newCandidateRules
        bestRule' = best rule in candidateRules
        if Laplace(bestRule')> Laplace(bestRule) then bestRule = bestRule'
    if accuracy(bestRule) $< 0.7$ then break
    else RuleList = RuleList $\cup$ bestRule
until all examples in the training set are covered
```

gray cells in that table), and there was no case where either version of CN2 significantly outperformed the GGP-RIs.

So far we have shown that the evolved GGP-RIs are competitive to traditional human-designed rule induction algorithms. But how similar the former are to the latter? Out of the 5 GGP-RIs produced by the experiments described in this section (corresponding to 5 runs of the GGP with a different random seed in each run), 3 shared one relatively uncommon characteristic: they added two conditions instead of one condition at-a-time to an empty rule, as shown in Alg. 2. Alg. 2 starts to produce rules with an empty condition, adds two condition-at-a-time to it, evaluates the rule with the new conditions using the Laplace estimation and selects the best 5 produced rules to go on into the refinement process. The algorithm keeps inserting new conditions to the best selected rules until all the examples in the training set are covered, or while the rules found have accuracy superior to 70%.

In other words, Alg. 2 is a variation of CN2 where two conditions are added to a rule at-a-time. The other difference with respect to CN2 lies on the condition used to stop inserting rules to the model (predictive accuracy superior to 70%). But why most of the algorithms produced by the GGP are similar to CN2?

The UCI data sets (Newman *et al.*, 1998) are very popular in the machine learning community, and they have been used to benchmark classification algorithms for a long time. To a certain extent, most of the manually-designed rule induction algorithms were first designed or later modified targeting these data sets. The fact that the evolved rule induction algorithms are similar to CN2 is evidence that CN2 is actually one of the best algorithms in terms of average predictive accuracy in a set of data sets available in the UCI repository. At the same time, as the rule induction algorithms produced by the GGP showed, there are many other variations of the basic sequential covering pseudo-code which obtain accuracies competitive to the ones produced by CN2, Ripper or C4.5Rules. In general, the evolved algorithms did not obtain significantly better accuracies than the baseline classification algorithms, but the former obtained slightly better results than the latter, overall. This can be observed in Table 3, which contains three significant wins (dark gray cells) and just one significant loss (light gray cell) for the evolved algorithms.

6 Conclusions and Future Directions

This work presented a grammar-based genetic programming system which automatically discovers new sequential covering rule induction algorithms. Computational results showed that the system can effectively evolve rule induction algorithms which are competitive in terms of accuracy with well-known human designed rule induction algorithms.

This work opens a whole new area of research, and there are many other directions which could be taken. Improvements to the current work include changing the fitness of the system to use the ROC framework, and studying the impacts this change would have in the created rule induction algorithms.

A more interesting direction, which at the moment is part of our ongoing work, is to automatically create rule induction algorithms tailored to a specific application domain. In other words, we can replace the meta-training and meta-test sets of the current GGP system by a single data set, corresponding to a target application domain, and produce customized rule induction algorithms. This would be a great contribution to the area of meta-learning, in particular, which is putting many efforts into finding which algorithms are the best to mine specific data sets.

Acknowledgments

The first author is financially supported by CAPES, the Brazilian Research Council, process number 165002-5.

References

Aho, A.V., Sethi, R., Ullman, J.D, (1986), Compilers: Principles, Techniques and Tools. 1st edn. Addison-Wesley.

Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D, (1998), Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann.

Bhattacharyya, S, (1998), Direct marketing response models using genetic algorithms. In: Proc. of 4th Int. Conf. on Knowledge Discovery and Data Mining (KDD-98). 144–148.

Caruana, R., Niculescu-Mizil, A, (2004), Data mining in metric space: an empirical analysis of supervised learning performance criteria. In: Proc. of the 10th ACM SIGKDD Int. Conf. on Knowledge discovery and data mining (KDD-04), ACM Press 69–78.

Clark, P., Boswell, R., (1991), Rule induction with CN2: some recent improvements. In Kodratoff, Y., ed, EWSL-91: Proc. of the European Working Session on Learning on Machine Learning, New York, NY, USA, Springer-Verlag 151–163.

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

Cohen, W.W., (1995), Fast effective rule induction. In Prieditis, A., Russell, S., eds, Proc. of the 12th Int. Conf. on Machine Learning (ICML-95), Tahoe City, CA, Morgan Kaufmann 115–123.

Fawcett, T, (2003), Roc graphs: Notes and practical considerations for data mining researchers. Technical Report HPL-2003-4, HP Labs.

Flach, P, (2003), The geometry of roc space: understanding machine learning metrics through roc isometrics. In: Proc. 20th International Conference on Machine Learning (ICML-03), AAAI Press 194–201.

Freitas, A.A, (2002), Data Mining and Knowledge Discovery with Evolutionary Algorithms. Springer-Verlag.

Fürnkranz, J, (1999), Separate-and-conquer rule learning. Artificial Intelligence Review **13**(1) 3–54.

de la Iglesia, B., Debuse, J.C.W., Rayward-Smith, V.J, (1996) Discovering knowledge in commercial databases using modern heuristic techniques. In: Proc. of the 2nd ACM SIGKDD Int. Conf. on Knowledge discovery and data mining (KDD-96), 44–49.

Koza, J.R, (1992), Genetic Programming: On the Programming of Computers by the means of natural selection. The MIT Press, Massachusetts.

Michalski, R.S, (1969), On the quasi-minimal solution of the general covering problem. In: Proc. of the 5th Int. Symposium on Information Processing, Bled, Yugoslavia 125–128.

Mitchell, T, (1997), Machine Learning. Mc Graw Hill.

Naur, P, (1963), Revised report on the algorithmic language algol-60. Communications ACM **6**(1) 1–17.

Newman, D.J., Hettich, S., Blake, C.L., Merz, C.J., (1998), UCI Repository of machine learning databases. University of California, Irvine, http://www.ics.uci.edu/~mlearn/MLRepository.html

Pappa, G.L., Freitas, A.A. (2006), Automatically evolving rule induction algorithms. In Fürnkranz, J., Scheffer, T., Spiliopoulou, M., eds, Proc. of the 17th European Conf. on Machine Learning (ECML-06). Volume 4212 of Lecture Notes in Computer Science., Springer Berlin/Heidelberg 341–352.

Pappa, G.L, (2007), Automatically Evolving Rule Induction Algorithms with Grammar-based Genetic Programming. PhD thesis, Computing Laboratory, University of Kent, Cannterbury, UK.

Provost, F., Fawcett, T., Kohavi, R, (1998), The case against accuracy estimation for comparing induction algorithms. In: Proc. of the 15th Int. Conf. on Machine Learning (ICML-98), San Francisco, CA, USA, Morgan Kaufmann Publishers

Inc. 445–453.

Quinlan, J.R, (1993), C4.5: programs for machine learning. Morgan Kaufmann.

Witten, I.H., Frank, E, (2005), Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. 2^{nd} edn. Morgan Kaufmann.

Zhang, J, (1992), Selecting typical instances in instance-based learning. In: Proc. of the 9^{th} Int. Workshop on Machine learning (ML-92), San Francisco, CA, USA, Morgan Kaufmann 470–479.

Genetic Programming, http://www.genetic-programming.org/ (2006)