Towards a Genetic Programming Algorithm for Automatically Evolving Rule Induction Algorithms

Gisele L. Pappa, Alex A. Freitas

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

Abstract. Rule induction is one of the techniques most used to extract knowledge from data, since the representation of knowledge as if/then rules is very intuitive and easily understandable by problem-domain experts. Existing rule induction algorithms have been *manually* designed. In this era of increasing automation, Genetic Programming (GP) represents a powerful tool for automatically evolving computer programs. This work proposes a genetic programming algorithm for automatically evolving rule induction algorithms. Hence, the evolved rule induction algorithm will, to a large extent, be free from the human biases that are implicitly incorporated in current manually-designed algorithms (such as the typical use of a greedy search method). This is a very ambitious, adventurous goal, which, if successful, will pave the way for a new generation of more robust, considerably less greedy rule induction algorithms. In particular, an automatically evolved rule induction algorithm can be designed to cope with attribute interaction better than current greedy rule induction algorithms, which will tend to lead to an improved performance in complex data sets.

1 Introduction

"…We consider this trend to increase the automation of science to be both inevitable and desirable …". With these words King et al [16] concluded their article about a new "robot scientist" for scientific discovery. The system works with biological data, and automatically generates hypotheses to explain observations, plans and physically executes experiments to test the hypotheses - using a laboratory robot, and interprets the results to falsify hypotheses inconsistent with the data.

A task can be automated by programming a machine to follow, step by step, the process a trained human would follow to execute it, obtaining, at the end, the same or similar results. Despite the significant progress in the automation of data analysis tasks, it should be noted that the design of a machine learning or data mining algorithm is still an essentially *manual* task. This holds true even for the above-mentioned robot scientist, where the machine learning algorithm used by the robot was manually designed by the researchers.

In this era of increasing automation, it seems timely to ask the question: why not going one step further? Why not using a computer program to produce, as its output, a full machine learning algorithm? This is the intriguing topic of this paper, which proposes to develop a Genetic Programming (GP) algorithm for automatically evolving a rule induction algorithm.

GP is a powerful tool for automatically evolving computer programs. In general, the program evolved by GP can produce the same solution humans use to solve the target problem, or something completely new, perhaps better than the "conventional" manually-designed solution.

GP can be defined as "the direct evolution of programs or algorithms for the purpose of inductive learning" [2]. GP is a kind of evolutionary algorithm (EA) that uses a set of functions and terminals, related to the domain of the problem to be solved, to represent candidate solutions for the problem. Nevertheless, while in most kinds of EAs an individual corresponds to a solution for one particular problem instance, in GP an individual should correspond to a general "recipe" for solving a given kind of problem. GP also uses a fitness function to evaluate the quality of the candidate solutions, and the principle of natural selection to evolve better and better candidate solutions to the target problem. GP will be reviewed in Section 3.

Rule induction is one of the methods most used to extract knowledge from data. Although there are many other methods, like instanced based learning (e.g. k-nearest neighbor), statistical techniques (e.g. naïve bayes classifier), neural networks and support vector machines [28], they do not normally return to the user, as the output of the system, comprehensible knowledge. Therefore, rule induction algorithms are widely used because the representation of knowledge as if/then rules is very intuitive and easily understandable by problem-domain experts.

In addition to the previously cited methods that are often used for classification, in the last few years there has been extensive research using GP for rule induction [12], [29]. However, the GP algorithms used in that kind of research, like many others proposed in the literature, do not really evolve computer programs. The outputs of those GPs are sets of rules *specific* to a given database, and cannot be applied to other databases. Hence, in the context of data mining, GP has been widely used for other purposes than evolving *generic* computer programs or algorithms for inductive learning – which, despite being the original goal of GP, is still a very open problem.

This work proposes a GP algorithm for automatically evolving a rule induction algorithm. The motivation for this goal is as follows.

All current rule induction algorithms were manually developed by a human being, and so they inevitably incorporate a human bias. In particular, the idea of developing greedy algorithms was partially derived from the classical view of concepts in cognitive psychology [11]. According to this classical view, categories are defined by a small set of attributes. All members of a category share these defining attributes, and no non-member shares them. In general the defining attributes are supposed to be largely independent (uncorrelated) of each other - i.e. there is little or no attribute interaction. (By attribute interaction we mean: consider three attributes Y, X_1 and X_2, where Y is the goal (class) attribute to be predicted and X_1 and X_2 are predictor attributes. X_1 and X_2 interact when the direction or magnitude of the relationship between Y and X_1 depends on the value of X_2 [11].)

This view is not currently the most acceptable in cognitive psychology, but it did influence the development of early rule induction algorithms, which were based on selecting one-attribute-value-at-a-time, in a greedy fashion, ignoring attribute interactions. A machine-developed algorithm could completely change this kind of algorithm bias, since "its bias" would be very different from the kind of algorithm bias imposed by a human algorithm designer. In particular, a machine-developed rule induction algorithm could cope better with attribute interaction, avoiding the greedy strategy of the vast majority of human-developed algorithms. This is important, because many complex real-world data sets are plagued by attribute interaction problems [9], [23], [24].

We emphasize that the automatic design of a full rule induction algorithm is a very ambitious, adventurous goal, which, if successful, will pave the way for a new generation of more robust, considerably less greedy rule induction algorithms. In particular, summarizing the above discussion, an automatically-evolved algorithm will, to a large extent, be free from the human biases that are implicitly incorporated in current manually-designed rule induction algorithms. This is expected to lead to an improved performance in complex data sets plagued by strong attribute interactions.

The remainder of this paper is organized as follows. Section 2 presents a survey of rule induction algorithms. Section 3 presents the main concepts of Genetic Programming, and Section 4 explains how the concepts of Sections 2 and 3 can be joined to automatically evolve a rule induction algorithm. Section 5 presents the conclusions and research directions to be followed.

2 A Survey of Rule Induction Algorithms

Rule induction algorithms, like most concept learners, were designed to acquire general concepts from a set of training examples. In the context of the well-known classification task, which is the focus of this paper, each training example is represented by a set of predictor attributes and a goal (or class) attribute. The algorithm tries to find relationships between the predictor and the goal attributes, creating a model that can be later used to predict the value of the goal attribute of new examples.

In the case of rule induction algorithms, the classification model is represented by a set of rules. A rule has the format IF *cond1* AND *cond2* … THEN *conseq*, where the conditions in the antecedent are described by associations between attributes and their values, and the consequent represents the predicted value for the goal attribute.

Research about rule induction algorithms has been carried out for more than 30 years. During this period, many inductive learners were developed. Almost all of them follow one of the three most common strategies used to induce rules from data [19]. The first strategy consists of generating a decision tree, using the divide and conquer strategy, and then extracting one rule for each leaf node of the tree, as in C4.5 Rules [21]. The second strategy is separate and conquer [28], found in the AQ algorithms [18], CN2 [5] and RIPPER [7]. The third is the use of evolutionary algorithms, like genetic algorithms and genetic programming, to extract rules from data, such as GABIL [8], and the GP algorithms described in [12] and [29]. Some hybrid algorithms combining the first two techniques can also be found, like in PART [10].

The divide and conquer strategy [3], used by decision tree algorithms, constructs a decision tree using a top down, greedy search. It evaluates all the predictor attributes to verify how well each of them, individually, classifies the examples in the training set. The best attribute is selected as the root of the tree. For each possible value of the chosen attribute a sub-tree is generated. For each sub-tree, the next attribute is chosen considering only the examples of the training set whose attributes values satisfy the condition associated with the corresponding branch of the tree. The process is recursively repeated until a stopping criterion is satisfied, e.g., until all examples in a leaf node belong to the same class or until the number of examples in a leaf node is smaller than a user-defined threshold. The class predicted by each leaf is determined by the most common class value found among the examples at that leaf.

After the tree is generated, it is mapped to a set of rules, creating a new rule for each of the tree paths from the root to a leaf. After this step, the rules should be refined, to avoid the major problem of decision tree representation: the replicate subtree problem [13]. This problem arises because decision tree learning algorithms cannot represent overlapping rules. Consequently, in some cases, the same subtree has to be learnt many times in different points of the tree. This problem also tends to make decision trees less comprehensible and more complex than sets of rules.

The separate and conquer strategy generates sets of rules, instead of decision trees. It learns a rule from a training set, removes from it the examples covered by the rule, and recursively learns another rule that covers the remaining examples, until all examples are covered. It is the most common strategy used for rule induction algorithms, and the methods based on this approach differ from each other in four main points: 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. Note that this strategy is also greedy, learning one rule at a time, and it typically generates a rule by adding (removing) one condition at a time to (from) a current partial rule, as will be discussed later.

Evolutionary algorithms (EA) are a completely different approach by comparison with the previous ones. Instead of using a greedy search, they use a global search to evolve a population of candidate solutions (each of them representing a rule or a set of rules), and explore the search space using the principle of natural selection (reproduction of the best individuals, i.e. of the best rules) and “genetic” operators (loosely inspired by natural genetics) such as crossover and mutation.

Comparing the three strategies presented above, EAs have the advantage of performing a more global search, avoiding one of the biggest problems of the greedy search: be trapped in a local optimal solution. In the context of a learning problem, this global search makes EAs cope better with attribute interaction [9], [12], [20]. On the other hand, EAs have the disadvantage that they are usually considerably slower than conventional, greedy rule induction algorithms.

Although the majority of separate and conquer methods use a hill climbing approach, it is not difficult to extend them to use less greedy methods, like a beam search or a best-first search, improving the way they cope with attribute interaction. However, searching more exhaustively the search space increases the chances that the discovered rules overfit the training data. Studies reported in [22], [26] verified in different domains that a large beam width can lead to worse results than small ones.

We now describe in more detail separate and conquer algorithms, since this is the kind of algorithm that will be automatically evolved by the GP proposed in this work. The separate and conquer approach was chosen because it is simple and easy to adapt, the generated rules are usually simpler and more comprehensible than those generated by divide and conquer methods, and it is more efficient (faster) than the EA-based approach.

2.1 Separate and Conquer Algorithms

As mentioned before, most of the rule induction algorithms based on the divide and conquer approach differ from each other with respect to four points, which will be discussed in the next items: the representation of the candidate rules, the search mechanism used to explore the space of candidate rules, the way the created rules are evaluated and the pruning method.

The representation of the candidate rules. 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. In particular, rules can be represented using propositional or first order logic. Propositional rules are composed by selectors, which are associations between pairs of attribute-values, like *age > 10*, *salary < 2000* or *sex = male*. CN2, C4.5 rules, and RIPPER are examples of propositional rule algorithms. First order rules are more sophisticated, and can express relations between two attributes, generating rules with conditions such as $x > y$. FOIL, REP and PROGOL use this representation.

When using a first order representation, the concepts are usually represented as Prolog relations, like father(x,y). Methods that use this Prolog representation are classified as Inductive Logic Programming (ILP) systems [17]. ILP uses the same principles of rule induction algorithms, essentially replacing the concepts of conditions and rules by literals and clauses. In addition, ILP techniques allow the user to incorporate background knowledge about the problem, which helps to focus the search in promising areas of the search space.

Moreover, the rule sets generated to describe a concept can be ordered or unordered. Ordered rule sets are also know as decision lists. In the context of large rules sets, ordered rules are usually considered more difficult to understand than unordered ones, since in order to comprehend the last rule of a list all the previous ones must also be taken in consideration [5]. Since the knowledge generated by rule induction algorithms is usually analysed and validated by an expert, rules at the end of the list become very difficult to understand, particularly in very long lists. Hence, unordered rules are often favoured over ordered ones.

The Search Mechanism. The search mechanism is composed by a search strategy and a search method. Broadly speaking, there are three kinds of search strategies, namely bottom-up, top-down or bi-directional strategy.

A bottom-up strategy starts the search with a very specific rule, and iteratively generalizes it [14]. A top-down one, in contrast, starts with the most general rule and

iteratively specializes it. A bi-directional search is allowed to generalize or specialize the candidate rules, according to the situation.

The most common strategy used by separate and conquer algorithms is the top-down one. The algorithm starts with a rule that covers all the training examples, and iteratively specializes it while some rule quality measure is optimized. This technique is used by algorithms of the AQ family, CN2, and Foil, among others. DLG [13] is one of the few algorithms that uses propositional logic and a bottom-up approach. SWAP-1 [27] follows the bi-directional strategy.

Regarding to the search method, greedy and beam search are the most commonly applied methods, although best-first and stochastic methods can also be used. Greedy search is the most popular method. Greedy algorithms create a rule using one attribute, generalize/specialize it, evaluate the extended rules created by the generalization/specialization operation, and keep just the best extended rule. This process is repeated until a stopping criterion is satisfied. Although they are fast and easy to implement, they have the well-known myopia problem: at each rule extension step, they make the best local choice, and cannot backtrack if later in the search the chosen path is not good enough to discriminate examples belonging to different classes. As a result, they do not cope well with attribute interaction.

Beam search methods try to eliminate the drawbacks of greedy algorithms selecting, instead of 1, the *b* best extended rules at each iteration, where *b* is the width of the beam. Hence, they explore a larger portion of the search space than greedy methods, coping better with attribute interaction. Nevertheless, learning problems involving very complex attribute interactions (like parity problems [25]) are still a very difficult problem for beam search algorithms.

Rule Evaluation. The way the candidate rules are evaluated can change completely the regions of the search space that are being explored. There are many heuristics current used by rule induction algorithms, each one with their own advantages and disadvantages. Fürnkranz [13] classified these evaluation heuristics in 4 categories:

1. Heuristics that favor rules that cover as many positive examples and as few negative examples as possible. Some examples of these heuristics are accuracy, information content, entropy, Laplace estimate, etc.
2. Heuristics that measure the complexity of the candidate solutions, like rule length, positive coverage or minimum description length.
3. Heuristics based on gain, which compute the difference in the value of some heuristic function measured between the current rule and its predecessor, such as information gain or coverage gain.
4. Weighted heuristics, which combines the previously described heuristics or adjusts the behavior of a single heuristic in a certain direction, e.g., J-measure and weighted information gain.

Pruning Methods. Combining the three elements discussed above, one can have many rule induction algorithms. But current methods, besides these elements, have other means to generate simpler and more accurate rules. Almost all of them implement a pruning technique, which helps to avoid over-fitting and to handle noisy data. These pruning techniques can be used during the production of the rules (pre-pruning) or in a post processing step (post-pruning).

Pre-pruning methods try to stop the refinement of the rules before they become too specific or over-fit the data. A statistical significance test is one of the criteria used to stop rule generalization/specification. It compares the observed class distribution among examples satisfying the rule with the expected distribution that would result if the rule had selected examples randomly. Other pruning criteria are minimum purity, encoding length restriction and cutoff stopping criterion [13].

Post-pruning methods try to improve the learned model after it has been constructed. It removes rules or rule conditions from the model, preserving or improving the predictive accuracy in the training set. Among the most used post-pruning techniques are reduced pruning error (REP) [4] and GROW [6].

I-REP and its improved version, RIPPER, are well known rule induction methods, and they work integrating pre- and post-pruning techniques. Their rule pruning techniques follow the same principles of REP, but they prune each rule after it is created, instead of waiting for the complete model to be generated.

Pre-pruning techniques are more efficient (faster) than post-pruning techniques, but post-pruning usually finds models with higher accuracy and simpler rules than pre-pruning. Intuitively, this is due to the fact that post-pruning has more information (the complete learned model) available to make decisions, and so it tends to be less "shortsighted" than pre-pruning. In any case, many post-pruning techniques are still greedy, by removing one condition at a time from a rule.

2.2 A Generic Specification of Rule Induction Algorithms

The concepts described in the previous section represent the essential elements of a rule induction algorithm. Considering these elements, it is possible to specify a general, high-level pseudo-code describing rule induction algorithms based on the separate and conquer approach. This task was previously executed by Fürnkranz [13], which proposed the generic pseudo-code shown in Algorithm 1 for separate and conquer rule induction algorithms:

```
Procedure SeparateAndConquer (Examples)
Theory = ∅
While POSITIVE(Examples) ≠ ∅
   Rule = FindBestRule(Examples)
   Covered = Cover(Rule, Examples)
   If RuleStoppingCriterion(Theory, Rule, Examples)
          Exit while
   Examples = Examples \ Covered
   Theory = Theory ∪ Rule
Theory = PostProcess (Theory)
return(Theory)
```

```
Procedure FindBestRule(Examples)
InitRule = InitializeRule(Examples)
InitVal = EvaluateRule(InitRule)
BestRule = <InitVal, InitRule>
Rules = {BestRule}
While Rules ≠ ∅
   Candidates = SelectCandidates(Rules, Examples)
   Rules = Rules \ Candidates
   For Candidate ∈ Candidates
     Refinements = RefineRule(Candidate, Examples)
     For Refinement ∈ Refinements
        Evaluation = EvaluateRule(Refinement, Examples)
        unless StoppingCriterion(Refinement, Evalua-
     tion, Examples)
           New Rule = <Evaluation, Refinement>
           Rules = InsertSort(NewRule, Rules)
           If NewRule > BestRule
              NewRule = BestRule
   Rules = FilterRules(Rules, Examples)
return(BestRule)
```

Algorithm 1: A general pseudo-code for rule induction algorithms [13]

The vast majority of rule induction algorithms that follow the separate and conquer approach can be instantiated following the pseudo-code of Algorithm 1. Below we describe how this pseudo-code can be implemented to originate the 4 basic elements of rule induction algorithms.

1. Search mechanism – This involves both the search strategy and the search method. The search strategy is implemented through the procedures:
   - *InitializeRules*, which specifies if the initial rule should be a very generic rule (with an empty antecedent), a very specific rule (derived from a "seed" example) or another possibility in between those two.
   - *RefineRules*, which determines if the current rule should be generalized or specialized, where the chosen operation should be consistent with the kind of initial rule specified in the *InitializeRules* procedure.

   The search method(s) is(are) defined using *SelectCandidates* and *FilterRules*.
   - *SelectCandidates* – This procedure selects the subset of rules that will be generalized/specialized. A somewhat generic way of specifying this procedure consists of referring to a beam search. Then a specific search method can be obtained by instantiating the parameter *b*, the beam width. For instance, if *b* is set to 4, the best 4 rules will be selected to be refined. Note that a greedy search method can be obtained by setting the parameter *b* to 1.
   - *FilterRules* – This procedure can use either the same search method as *SelectCandidates* or a different search method. The specification of the search method can be done in similar ways in both procedures.

2. Representation of candidate rules –This element is implemented by the *RefineRules* procedure, since it determines which kind of conditions can be added to the candidate rules.
3. Rule Evaluation – This element is directly defined by the procedure *EvaluateRule*, which determines the rule-quality measure that will be used in rule evaluation.
4. Pruning Methods – This element is determined by two procedures, namely *StoppingCriterion*, which implements pre-pruning methods – i.e., it determines when to stop refining the rules – and *PostProcessing*, which implements post-pruning methods.

This general pseudo-code summarizes, in a very concise fashion, the general structure of the vast majority of rule induction algorithms manually designed by machine learning researchers. Hence, this "prior knowledge" could be given to a GP algorithm, which would work out how to automatically develop new rule induction algorithms based on this information, as explained in Section 4.

3. Overview of Genetic Programming

Evolutionary Algorithms (EAs) are stochastic search methods inspired by the Darwinian concepts of evolution and survival of the fittest. They became very popular in many kinds of problems, like function optimization and several machine learning tasks, due to their domain-independent nature and their robust global search mechanism - with its associated implicit parallelism and noise tolerance [1], [15].

In essence, an EA evolves a population of individuals, where each individual is a candidate solution for the target problem. At each generation, the individuals are evaluated according to a fitness function. The best individuals are selected to reproduce, and undergo crossover and mutation procedures in order to produce new offspring (new candidate solutions) that inherit some features from their parents. The evolutionary process is iteratively performed until a stopping criterion is satisfied, such as a maximum number of generations is reached or an optimal solution is found.

Four major kinds of EA are genetic algorithms, genetic programming, evolutionary strategies and evolutionary programming [1]. This section briefly discusses genetic programming (GP), which is the main kind of EA designed to evolve programs.

Hence, GP is a kind of EA where the individuals being evolved are computer programs. As defined by Banzhaf [2], GP is "the direct evolution of programs or algorithms for the purpose of inductive learning".

When designing a GP system, some elements have to be considered: the set of functions and terminals that will be used to create the GP population, the representation of the individuals, and the fitness function used to measure the quality of the candidate solutions. In addition, crossover and mutation operators have to be designed according to the individual's representation, and a method for selection of the best individuals has to be implemented. The next sections explain the main concepts involved in the design of these elements.

3.1 Functions and terminals

The functions and terminals are the primitives with which a program in GP is built. Terminals provide a value to the system while functions process a value already in the system [2].

The terminals are usually constants, variables and/or zero-argument functions. The function set can be composed of many types of functions. The most common ones are the boolean and arithmetical functions, but the function set can also have conditional and/or loop statements and subroutines. The subroutines allow any kind of operation to be added to the function set.

No matter which or how many functions there are in the function set, all of them have to respect the closure property. This property states that every function has to be able to handle all the values it receives as input. Thus a division operator, for example, has to be modified to cope with division by zero – this is often implemented by making the operator return a given value, rather than an error, in case of division by zero.

Although the programmer has a lot of freedom to choose the function set, it should not have many functions, because the more functions the greater the search space.

3.2 Individual Representation

Recall that GP algorithms evolve a population of individuals, where each individual represents a solution for the target problem. If we are developing, for example, a GP for discovering rules in a specific data set, each GP individual will represent a set of candidate rules for that data set. There are two types of representations that are most used in the literature [2]: the first one represents an individual as a linear structure and the second as a tree.

A linear representation is simply a sequence of commands that are executed from left to right, while in a tree representation the execution of the tree is usually made in postfix order (reading the leftmost node of the tree). These conventions can be changed depending on the functions included in the function set.

Figure 1 shows individual representations using linear and tree-based genomes for a regression problem (where the goal would be to find the equation of the curve with the best fit to a set of data points). In the example of that figure the individuals represent the candidate solution (equation) $x^2 + 2$. In this example, *x* and *2* are terminals, and * and + are functions.

3.3 Fitness function and Selection Methods

After the GP population is initialized (usually randomly), individuals are evaluated using a fitness function. This function measures how well the individual solves the target problem. It is used to determine which individuals will reproduce and have parts of their genetic material (i.e., parts of their candidate solution) passed onto the next generation.

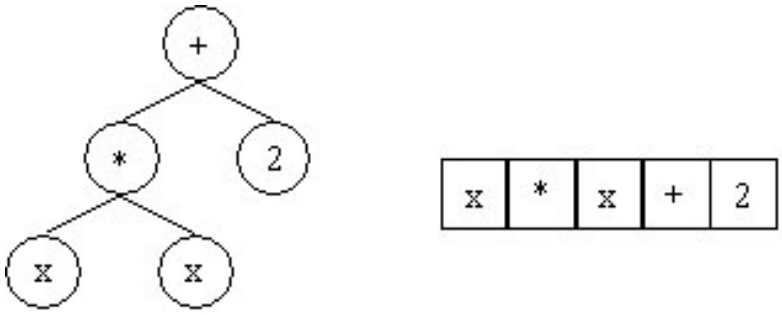

Figure 1: Examples of tree-based and linear-based individual representation

The better the fitness of an individual, the higher the probability of that individual being selected for reproduction. There are many selection methods, such as fitness-proportional selection, ranking selection and tournament selection. Tournament selection, for example, randomly gets a pre-defined number of individuals from the population and simulates a tournament among them. Typically, the individual with the best fitness is declared the winner of the tournament and is therefore selected for reproduction, crossover and mutation.

3.4 Crossover and Mutation Operators

Crossover swaps genetic material (parts of candidate solutions) between two individuals, whereas mutation replaces some part of the genetic material of an individual with a new randomly-generated genetic material. These two operations are applied with user-specified probabilities. The basic idea of these operators is as follows.

Crossover re-combines the genetic material of two parent individuals (which have been previously selected, as discussed in section 3.3), in order to produce two new children. If the individuals are represented by trees, randomly-selected subtrees are swapped between the two parents. In the case of linear genomes, randomly-selected linear segments of code are swapped.

Unlike crossover, mutation acts on a single parent individual of the population. It randomly selects a subtree of the tree-based genome or a segment of code in linear genomes and replaces it by a new randomly-generated subtree or code segment.

Both crossover and mutation operations can be implemented in many ways – see [2] for a detailed review of these operators.

4. Designing a Genetic Programming Algorithm for Automatically Evolving Rule Induction Algorithms

As mentioned earlier, this work proposes the use of GP to automatically evolve a rule induction algorithm. The most important aspects of the design of a new GP algorithm to solve this problem are: a) the definition of a function set and a terminal set suitable for representing a candidate rule induction algorithm, b) the design of a fitness func-

tion suitable for evaluating a rule induction algorithm. These aspects are discussed in the next subsections.

4.1 The Function Set and the Terminal Set

The function set will be divided into two subsets. The first one will have functions that control the processing flow of the algorithm being evolved (like IF-THEN statements and FOR/WHILE loops), and the second one will include functions that are specific to rule induction algorithms.

This second subset of functions will be developed in two stages. In the first stage, very general, high-level functions will be used. This set of general functions will be based on pseudo-codes that specify a base algorithm for rule induction, and that can be instantiated to produce virtually any method based on the separate and conquer approach. A general pseudo-code with this characteristic was already designed by [13], as reviewed in subsection 2.2, and it will be used as a starting point for this task. In this first phase, functions like selectCandidates(rules,examples), refineRule(rule) and evaluateRule(rule) will compose the function set. In a second stage, these general functions will be refined, and replaced by lower level ones, such as calculateInformationGain(r), generateContingenceTable(r), setBeamWidth(b), countCoveredExamples(r), etc.

The terminal set will be composed by a set of reserved words of programming languages, such as "*if, while, for, repeat*", a set of operators, like "+, =, - >, <", a set of variables, some delimiters like "*(,)*" and a set of constants, that will be used to set the values of the parameters of some functions, like the width of the beam search.

In our system, each individual will correspond to a candidate rule induction algorithm. An individual will be represented by a tree structure, and grammar-based GP will be used to evolve the rule induction algorithm. In essence, grammar-based GP is a kind of GP where a grammar's set of production rules is used to create the initial population of individuals and genetic operators produce new individuals respecting the grammar's production rules [29]. Grammar-based GP is highly recommended when one has some kind of prior knowledge of the problem domain to guide the GP search. Hence, in our system each individual will consists of a derivation tree, created by applying the production rules defined in the grammar. An example of a fragment of an individual is shown in Figure 2.

In Figure 2, the productions of the grammar appear in bold, and the terminal symbols in italic. The functions belonging to the function set, as FindBestRule, are specified through grammar productions.

The derivation tree of Figure 2 represents the code

"*while (PositiveExamples <> 0)*
    Rule = FindBestRule…
endwhile",

which would be part of a larger individual representing an entire rule induction algorithm. Note that FindBestRule is not a terminal, and will be also extended.

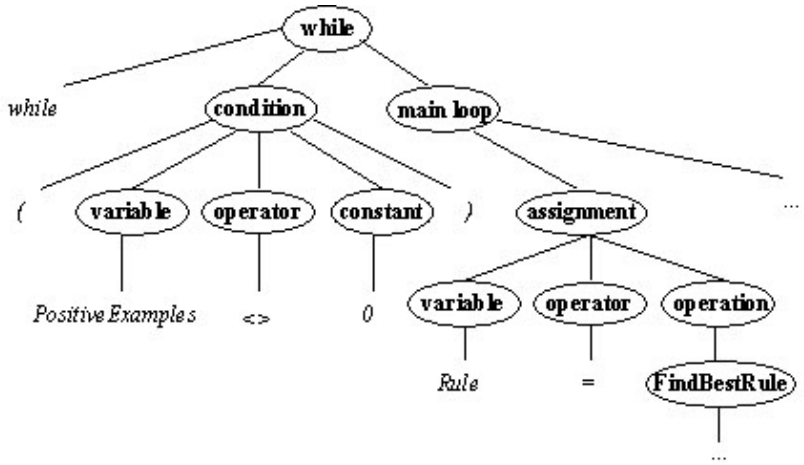

Figure 2: Fragment of an Individual, represented as a derivation tree

4.2 Fitness Evaluation

We emphasize that the goal of this paper is to evolve a generic rule induction algorithm, rather than just a rule set for a particular data set. This implies that the fitness function has to involve a measure of performance (e.g. predictive accuracy) of an individual (a candidate rule induction algorithm) across many data sets. Also, after the GP has run, the best evolved rule induction algorithm will have its performed evaluated on a separate set of data sets (unseen during the training of the GP).

More precisely, the entire *set of data sets* will be divided into two sets, which we will refer to as the *meta training set* and the *meta test set*. The meta training set will consist of several data sets, each of which will be divided into training and test sets. In order to compute the fitness of an individual, its corresponding candidate rule induction algorithm will be used to discover classification rules from each of the training sets in the meta training set, and the discovered rules will be used to classify the corresponding test sets in the meta training set. This will produce a measure of predictive accuracy for each data set in the meta training set. Those measures will then be combined (e.g. by computing an arithmetic average) to compute the overall measure of predictive accuracy associated with the candidate rule induction algorithm. This overall measure of accuracy will be used as the fitness value of the individual, for the purpose of selection in the evolutionary process. Therefore, the system will enforce an evolutionary pressure to produce a rule induction algorithm that is robust across all the data sets included in the meta training set.

The data sets that will compose the meta training set can be chosen in two different ways. First, one can choose data sets that are as diverse as possible, from many different domains. This will emphasize the generality of the evolved rule induction algorithm, which was, presumably, the same approach to manually design and evaluate most existing rule induction algorithms. Second, one could instead favor the evolu-

tion of a rule induction algorithm that is tailored for one kind of application domain, by including on the meta training set only data sets coming from that domain.

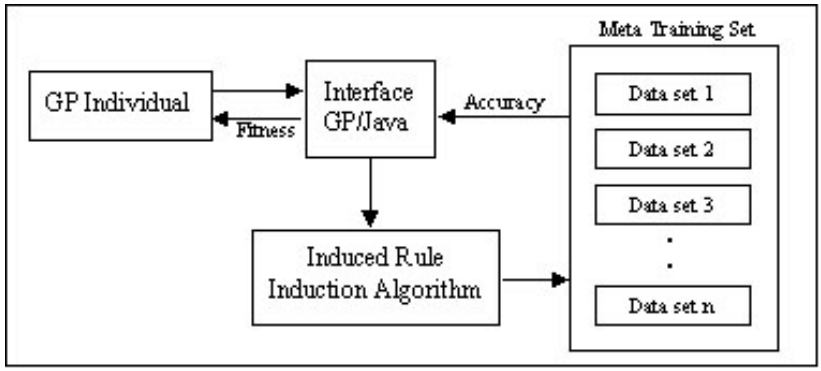

Figure 3: Fitness evaluation in the GP for evolving rule induction algorithms

This process is illustrated in Figure 3. In order to compute the individuals' fitness values, each individual will be converted into a rule induction algorithm, using an interface that implements the functions and terminals of the individual in Java code.

At the end of the evolution, the best evolved rule induction algorithm will be evaluated in another set of data sets, the meta test set, which contains data sets that were *not* used in the GP's meta training set. Again, each of the data sets in the meta test set will be divided into training and test sets. The evolved rule induction algorithm will be used to discover rules from each of the training sets in the meta test set, and the discovered rules will be used to classify the corresponding test sets in the meta test set, providing the final measure of performance (predictive accuracy) for the automatically evolved rule induction algorithm. Finally, the performance of the evolved rule induction algorithm will be compared with the performance of other rule induction algorithms applied to the same meta test set. The rule induction algorithms of WEKA [28] and/or Clementine will be used for comparison in the experiments.

The experimental methodology just described has a high computational cost. To tackle this problem, we can implement techniques to reduce computational costs, such as using just a subset of the training sets (in the meta training set) in the fitness evaluation – using random training-subset selection and/or "intelligent" training subset selection [12].

5 Conclusions

As discussed earlier, this paper proposes to pursue a very ambitious goal, namely to automatically design a full rule induction algorithm. We have identified GP as a promising paradigm to pursue this goal, since GP was explicitly designed to evolve *generic* computer programs or algorithms for inductive learning. However, this original goal of GP is still a distant goal, since current GP algorithms for rule induction are being used just to evolve a rule set for *a given specific* data set. This project aims to

go much further than this, by developing a GP for performing *algorithm induction*, rather than just *rule set* induction.

Hence, this is an adventurous, risky research project, which, if successful, will pave the way for a new generation of more robust, considerably less greedy rule induction algorithms. In particular, an automatically-evolved algorithm will, to a large extent, be free from the human biases that are implicitly incorporated in current manually-designed rule induction algorithms. This is expected to lead to an improved performance in complex data sets plagued by strong attribute interactions, since conventional greedy rule induction algorithms do not cope well with attribute interactions.

This paper has outlined the design of the proposed GP. In particular, the paper has shown that, using prior knowledge in the form of the general structure of rule induction algorithms developed by human designers, we can define a suitable set of functions and terminals that will be used by the GP to evolve rule induction algorithms. We have also specified how to compute the fitness function value of individuals (candidate rule induction algorithms) of the GP population, using the novel concept of a "meta training set" – formed by a *set of* data sets.

The next steps of this research will consist of refining the design of the GP (a work which is in hand), implementing it and evaluating the performance of the evolved rule induction algorithm. In particular, the automatically-designed algorithm will be extensively compared with conventional, manually-designed algorithms, in order to identify the strengths and weaknesses of the evolved algorithm.

Acknowledgement

The first author is supported by CAPES (Brazil), process number 1650-02-5.

References

1. Back, T, Fogel, D. B. and Michalewicz (Eds), Evolutionary Computation 1: Basic Algorithms and Operators, Institute of Physics Publishing, UK, 2000.
2. Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D., Genetic Programming: an Introduction: on the Automatic Evolution of Computer Programs and Its Applications. Morgan Kaufmann, 1998.
3. Bostrom, H., Asker, L., Combining Divide-and-Conquer and Separate-and-Conquer for Efficient and Effective Rule Induction, Proc. ILP-99, LNAI 1634, Springer, 1999.
4. Brink, C. A. and Pazzani, M. J., An investigation of noise-tolerant relational concept learning algorithms. In: Proc of 8th Int. Workshop on Machine Learning, 389-393, 1991.
5. Clark, P. and Boswell, R., Rule induction with CN2: Some recent improvements. In: Machine Learning - EWSL-91, Kodratoff Y. (Ed), 151 -163, Berlin, Springer-Verlag, 1991.
6. Cohen, W., Efficient pruning methods for separate and conquer rule learning systems. In. Proc 13th Int. Joint Conf. on AI, France, 988-994, 1993.
7. Cohen, W., Fast effective rule induction. In. Proc 12th ICML,Lake Taho, CA, Morgan Kaufmann, 1995.

8. DeJong, K. A., Spears, W. M, Gordon, D. F., Using genetic algorithms for concept learning. Machine Learning 13, 161-188, 1993.
9. Dhar, V., Chou, D., Provost, F., Discovering interesting patterns for investment decision making with GLOWER – a genetic learner overlaid with entropy reduction. Data Mining and Knowledge Discovery 4(4), 251-280, 2000
10. Frank, E. and Witten, I. H., Generating Accurate Rule Sets Without Global Optimization. In: Proc. of the 15th International Conference in Machine Learning, Shavlik, J.(Ed), Morgan Kaufmann Publishers, San Francisco, CA, 1998.
11. Freitas, A. A., Understanding the Crucial Role of Attribute Interaction in Data Mining. Artificial Intelligence Review 16(3), 177-199, 2001.
12. Freitas, A.A.: Data Mining and Knowledge Discovery with Evolutionary Algorithms. Springer-Verlag, 2002.
13. Fürnkranz, J. Separate-and-Conquer Rule Learning. A. I. Review 13(1), 3-54, 1999.
14. Fürnkranz, J., A Pathology of Bottom-Up Hill-Climbing in Inductive Rule Learning. In: Proc of the 13th Int. Conf. on Algorithmic Learning Theory, 263-277, Germany, Springer-Verlag, 2002.
15. Goldberg, D. E., Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, Reading, MA, 1989.
16. King, R. D., Whelan, K. E., Jones, F. M., Reiser, P. G., Bryant, C. H., Muggleton, S. H., Kell, D.B., Oliver, S. G., Functional genomic hypothesis generation and experimentation by a robot scientist. Nature. 2004 Jan 15, 247-52
17. Lavrac, N. and Dzeroski, S., Inductive Logic Programming: techniques and applications. Ellis Horwood, 1994.
18. Michalski, R. S., On the quasi-minimal solution of the general covering problem. In: Proc. Of 5th Int. Symposium on Information Processing, 125-128, 1969.
19. Mitchell, T. M., Machine Learning, Mc Graw Hill, 1997.
20. Papagelis, A. and Kalles, D., Breeding decision trees using evolutionary techniques. In. Proc of 18th Int. ICML-2001, 393-400, Morgan Kaufmann, 2001.
21. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann, 1993.
22. Quinlan, J. R. and Cameron-Jones, R. M., Oversearching and layered search in empirical learning. In: Proc. of 14th Joint Conf. on A. I., 1019-1024, Morgan Kaufmann, 1995.
23. Rendell, L., Seshu, R., Learning hard concepts through constructive induction: framework and rationale. Computational Intelligence 6, 247-270,1990.
24. Rendell, L., Ragavan, H., Improving the design of induction methods by analyzing algorithm functionality and data-based concept complexity. In:. IJCAI-93, 952-958, 1993.
25. Schaffer, C. , Overfitting avoidance as bias. Machine Learning 10, 153-178, 1993.
26. Webb, G. I., Systematic search for categorical attribute-value data-driven machine learning. In Proc. of 6th Joint Conf. on A. I., 342-347, World Scientific, 1993.
27. Weiss, S.M. and Indurkhya, N., Optimized Rule Induction. In: IEEE Expert: Intelligent Systems and Their Applications, 8(6), 61-69, 1993.
28. Witten, I. H., Frank, E., Data mining: Practical Machine Learning Tools and Techniques with Java Implementations, Morgan Kaufmann, 1999.
29. Wong, M. L., Leung, K. S., Data Mining using Grammar Based Genetic Programming and Applications, Kluwer Academic Publishers, 2000.