university of groningen

University Medical Center Groningen

University of Groningen

Genetic algorithms in data analysis

Lankhorst, Marc Martijn

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
1996

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Lankhorst, M. M. (1996). Genetic algorithms in data analysis. Groningen: s.n.

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 12-11-2019

5

Grammatical Inference II: Context-Free Grammars

As we have seen in the previous chapter, grammatical inference is the problem of learning an acceptor for a language based on a set of samples from that language. In that chapter, genetic algorithms were used to find one such type of acceptors, pushdown automata. This chapter presents a different approach. It introduces a genetic algorithm that directly infers context-free grammars from legal and illegal examples of a context-free language.

First, Section 5.1 outlines the adaptation of a genetic algorithm to infer context-free grammars. It discusses the representation of grammars in the form of chromosomes and the genetic operators used. Furthermore, it gives different fitness functions to evaluate the performance of the grammars.

Next, Section 5.2 describes the experiments conducted to test the algorithm's performance. In these experiments, the same test languages are used as in Chapter 4. Finally, Section 5.3 presents a discussion of the results of the experiments, and compares the inference of context-free grammars with the results on pushdown automata of the previous chapter.

This chapter is based on [96, 98, 99, 100].

5.1 A Genetic Algorithm for the Induction of Grammars

5.1.1 Representation

In representing the production rules of a context-free grammar in the form of a chromosome, we have two options. The first is to use a low-level binary representation of these rules, preferably in such a way that each bitstring is guaranteed to represent a legal production rule. The second is to take a high-level representation in which each chromosome is a list of barely encoded production rules, such as the one used by [157], or the marker-based encoding of [77].

To encode grammars as bitstrings, an interval encoding is used that represents a vector of integers in one bitstring-encoded number. Suppose we want to encode an integer vector $\mathbf{p} = [p_0, \ldots, p_n]$ with $\forall i : 0 \leq p_i < m_i$. We can do this by successively subdividing the interval $[0, 1)$. At each level k, the subinterval corresponding to p_k is used as the basis for the next stage, i.e., if we want to encode an integer p_k at stage k, we divide the interval $[a_k, b_k)$ into m_k equal subintervals, and we choose the p_k'th subinterval $[a_{k+1}, b_{k+1}) \subset [a_k, b_k)$ with

$a_{k+1} = a_k + p_k \cdot (b_k - a_k)/m_k$ and $b_{k+1} = a_k + (p_k + 1) \cdot (b_k - a_k)/m_k$. The number $E(\mathbf{p})$ that encodes the complete vector is contained in the final interval and is computed as follows:

$$E(\mathbf{p}) = \sum_{i=0}^{n} \left(p_i \cdot \prod_{j=0}^{i} \frac{1}{m_j} \right) . \tag{5.1}$$

This yields a real number in the interval $[0, 1)$, which can easily and unambiguously be decoded into the corresponding vector. It is easy to verify that we need at least

$$b = \left\lceil \log_2 \prod_{j=0}^{n} m_j \right\rceil \tag{5.2}$$

bits to represent $E(\mathbf{p})$.

If we want to encode a grammar rule, the vector $\mathbf{p}$ consists of the size of the rule and the numbers of its symbols. Suppose we want to encode the rules of a grammar $G = \langle N, \Sigma, P, S \rangle$ with a set of nonterminal symbols $N = \{X_i \mid 0 \leq i < n_N\}$, a set of terminal symbols $\Sigma = \{X_j \mid n_N \leq j < n_N + n_\Sigma\}$, and rules with a maximum number of n_{RHS} right-hand side symbols. Let $\rho = X_{q_0} \to X_{q_1} \dots X_{q_k}$ $(0 \leq k \leq n_{RHS})$ be a grammar rule, with q_i being the index in $N \cup \Sigma$ of the i'th symbol of the rule, $X_{q_0} \in N$, and $X_{q_i} \in N \cup \Sigma$ $(1 \leq i \leq k)$. This rule is encoded as follows:

$$E(\rho) = \frac{1}{n_{RHS} + 1} \left(k + \frac{1}{n_N} \left(q_0 + \sum_{i=1}^{k} q_i \left(\frac{1}{n_N + n_\Sigma} \right)^i \right) \right) . \tag{5.3}$$

Finally, the bitstrings encoding the individual rules are concatenated to form a chromosome that represents the grammar.

The symbolic encoding is more straightforward. A special symbol, say ξ, is used to denote the end of the right-hand side of a rule, and a vector of symbol numbers represents each rule. These vectors are concatenated to form a chromosome. If each rule consists of a left-hand and at most n_{RHS} right-hand symbols, a grammar of r rules is encoded as a vector of $r \cdot (n_{RHS} + 1)$ symbols. In decoding, we first decode the left-hand symbol, and decode the right-hand symbols from left to right until we observe a ξ.

5.1.2 Genetic Operators

Selection

Selection is carried out by a stochastic universal sampling algorithm [9], using rank-based selection [153]. Furthermore, an elitist strategy is employed in which the best individual of the population always survives to the next generation.

Mutation

In the case of a binary encoding, a mutation rate of $1/\ell$ is used throughout all experiments (ℓ being the number of bits in the chromosome), following [116].

For the symbolic encoding, this heuristic cannot be employed. Instead, an analogous mutation probability of $1/m$ is used, where $m = r \cdot (n_{RHS} + 1)$ is the length of the chromosomes (see Section 5.1.1). This mutation is purely random: every symbol can be mutated to every other symbol in the gene's domain.

Recombination

Both the binary and the symbolic representation schemes presented in Section 5.1.1 allow crossover operations to break the right-hand side of production rules. From (5.3), it follows that lower order bits encode grammar symbols at the end of the right-hand side of the rule, whereas higher order bits encode symbols closer to the beginning, or even the left-hand symbol. However, bits are not guaranteed to be part of the encoding of a single symbol. Hence, symbols surrounding a crossover point might be influenced.

Crossover within an individual right-hand side of a production rule has the advantage of creating a larger variety of right-hand sides in the population. Moreover, it might be beneficial to construct a new right-hand side from parts of the parents' right-hand sides. Not allowing this crossover to occur, [157] had to rely on mutation alone to change the right-hand sides of production rules.

In all experiments, standard two-point crossover (see Section 2.4.2) is used with a crossover probability of 0.9.

5.1.3 Fitness Evaluation

The most important issue in constructing a genetic algorithm is the choice of a particular fitness function. Suppose we have sets $\mathcal{S}_+$ of positive and $\mathcal{S}_-$ of negative examples of a language $\mathcal{L}$, and a grammar $G = \langle \Sigma, V, P, S \rangle$. Defining the fraction of correctly analyzed sentences as follows,

$$
\begin{aligned}
cor(G,\sigma) &= \begin{cases} 1 & \text{if } \sigma \in \mathcal{L} \cap L(G) \text{ or } \sigma \in \overline{\mathcal{L}} \cap \overline{L(G)}\ , \\ 0 & \text{otherwise}\ , \end{cases} \\
cor(G,S) &= \frac{1}{|S|} \sum_{\sigma \in S} cor(G,\sigma)\ ,
\end{aligned} \tag{5.4}
$$

a simple fitness function would be

$$
F_1(G, \mathcal{S}_+, \mathcal{S}_-) = cor(G, \mathcal{S}_+) \times cor(G, \mathcal{S}_-)\ , \tag{5.5}
$$

which yields fitness values between 0 and 1.

However, to evaluate the fitness of a particular grammar with respect to the positive and negative training examples, it does not suffice to simply count the correctly accepted (rejected) positive (negative) examples. Thus a grammar that can analyze large parts of the examples correctly, but fails to recognize the complete sentences, would receive a very low fitness value. Although recombination of such a partially correct grammar might yield a better result, this low fitness will cause it to be thrown away, thereby destroying valuable information.

To evaluate a grammar, we would like to credit grammars for accepting part of an input sentence. The longer the part of the input sentence that is consumed in a left-to-right scan of the input (as is performed by Earley's context-free parsing algorithm [45] that is used in the fitness computation), the higher the grammar's fitness should be. This leads us to the following definitions:

$$
pref(G, a_1 \ldots a_n) = \max\left\{ k \,\middle|\, S \Rightarrow_G^* a_1 \ldots a_k \delta, \delta \in (\Sigma \cup V)^* \right\}\ , \tag{5.6}
$$

and

$$pref(G, \mathcal{S}) = \frac{\Sigma_{w \in \mathcal{S}} pref(G, w)}{\Sigma_{w \in \mathcal{S}} |w|} , \tag{5.7}$$

which yields 1 if all input sentences are parsed correctly. Since this is only meaningful for legal examples, the new evaluation function is defined as

$$F_2(G, \mathcal{S}_+, \mathcal{S}_-) = (cor(G, \mathcal{S}_+)/2 + pref(G, \mathcal{S}_+)/2) \times cor(G, \mathcal{S}_-) , \tag{5.8}$$

which again yields values between 0 and 1.

Additionally, we might include information on the generative capacity of a grammar. We can use the grammar G to generate a set of strings $\mathcal{S}_{GEN}(G)$ and test whether these strings belong to the language. The evaluation function can be augmented with this information, yielding

$$F_3(G, \mathcal{S}_+, \mathcal{S}_-) = F_2(G, \mathcal{S}_+, \mathcal{S}_-) \times cor(\mathcal{S}_{GEN}(G)) \tag{5.9}$$

Unfortunately, this requires a teacher with prior knowledge of the underlying structure of the language for which we want to infer a grammar.

5.2 Experiments

5.2.1 Test Data

To assess the performance of the genetic algorithm, it was tested on the same languages as employed in the experiments on pushdown automata of Chapter 4, listed in Table 4.2. For a lengthier discussion of the use of these languages by other researchers, see Section 4.5.1.

For languages 1–6, we used Tomita's training cases, $\mathcal{S}'_+$ and $\mathcal{S}'_-$, as listed in Table 4.4. For each of the other languages, we randomly generated two training sets $\mathcal{S}_+$ and $\mathcal{S}_-$ of 100 positive and 100 negative examples, with no duplication, a maximum sentence length of 30, and a Poisson-like length distribution (see Figure 4.1). Two similarly generated test sets, $\mathcal{T}_+$ and $\mathcal{T}_-$, disjoint with $\mathcal{S}_+$ and $\mathcal{S}_-$, were used to assess the quality of the solutions found by the GA for all experiments, including those on languages 1–6. These training and test sets are the same as those used in the experiments of the previous chapter (see Section 4.5.1).

Domain knowledge was used to determine the terminal symbols and the size—i.e. the maximum size of the right-hand sides of rules and the number of rules[1] and nonterminals—of the grammars to be inferred. These features could also be encoded on the chromosomes, but that would impose an extra computational burden upon the genetic algorithm. Table 5.1 lists the parameters of the algorithm for the different test cases.

5.2.2 Results

A first experiment with the micro-NL language, using 100 positive and 100 negative example sentences, a population of 100 individuals, and fitness function F_2 (5.8), did not result in a correct grammar. Neither enlarging the population nor using more examples could improve the results significantly.

[1] An additional rule containing the start symbol S was added to each grammar after decoding.

No.	Population	Rules	Symbols	RHS	Bits	Fitness func.
1	50	4	4	2	44	F_2
2	50	14	8	2	154	F_2
3	50	14	7	2	154	F_2
4	50	14	10	2	154	F_2
5	50	8	6	2	88	F_2
6	50	8	7	2	88	F_2
7	50	8	5	3	88	F_2
8	50	4	4	2	28	F_2
9	50	4	4	3	40	F_2
10	100	10	8	2	140	F_3

Table 5.1: Parameters of the algorithm.

S	$\rightarrow$	A
A	$\rightarrow$	(B
A	$\rightarrow$	ε
B	$\rightarrow$	A)
B	$\rightarrow$	A B

Table 5.2: Grammar inferred for the parentheses language.

S	$\rightarrow$	X
X	$\rightarrow$	X Y
X	$\rightarrow$	Y X
X	$\rightarrow$	ε
Y	$\rightarrow$	X
Y	$\rightarrow$	a Y b
Y	$\rightarrow$	b X a

Table 5.3: Grammar inferred for the a's & b's language.

Some grammars evolved that scored a high fitness value by analyzing all examples correctly, but generated many illegal sentences and scored low on the negative examples of test set $\mathcal{T}_{-}$. The cause of this problem is that the language is sparse, i.e., the set of correct sentences forms a very small part of the total set of sentences that can be formed from the given nonterminals. Therefore, restricting the grammar just by training it on illegal sentences is a very hard job. To overcome this problem, evaluation function F_3 as defined in (5.9) was used. This function includes information on the generative capacity of the grammar at hand, and yielded a dramatic improvement of the results.

Examples of the grammars inferred for the parenthesis and the a's & b's language are shown in Tables 5.2 and 5.3, respectively. To illustrate the discontinuous character of the convergence of the GA, a graph of the fitness function during one of the runs is shown in Figure 5.1. Small changes to a grammar can cause it to parse many more of the examples correctly, which explains the sometimes quite substantial jumps in the convergence process.

Tables 5.6 and 5.7 show the results of the genetic algorithm with binary and symbolic representation, tested on all 10 example languages, averaged over 5 runs[2] of at most 2000 generations each.

[2] The experiments on both binary and symbolic representations used the same 5 (randomly generated) initial populations.

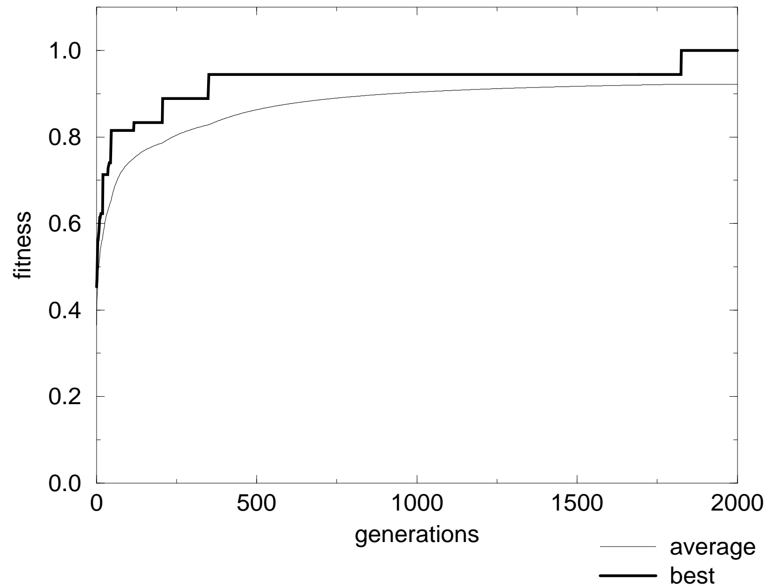

Figure 5.1: Convergence of the binary GA for language no. 4 of Table 4.2. Average and best fitness vs. number of generations.

The best individual obtained in each of the 5 runs was saved. The tables give the best and average fitness of these 5 individuals, the average number of generations until these individuals were found, and their average scores on the four sentence sets. In these and subsequent tables and figures, the error bounds are determined from the 95% confidence interval given by a standard t-test. No significant difference between the binary and symbolic approach can be distinguished in these results.

5.2.3 Comparison with Results on Pushdown Automata

If we compare the results of this chapter with those on the inference of pushdown automata (Chapter 4), a number of similarities and differences catch the eye. First, the grammar-based GA does a better job extracting the exact structure of the regular languages 1–6 from Tomita's training sentences (see Table 5.8). For all six languages, a correct grammar was found in at least one of the runs, whereas a correct PDA was not found for languages 2 and 6. However, both the grammars and the PDAs show the same difficulty in generalizing from the training examples, as demonstrated by their scores on the test sets ($\mathcal{T}_+$ and $\mathcal{T}_-$). Since this difficulty results from the small sets of training examples ($\mathcal{S}'_+$ and $\mathcal{S}'_-$), the similarity is not surprising.

On languages 7–10, results of the grammar-based GA are clearly better, both in learning and in generalizing, as witnessed by the results on the training sets $\mathcal{S}$ and the test sets $\mathcal{T}$ (see Table 5.9). Although the results on the inference of grammars are somewhat better than those on PDAs, the grammar-inferring GA requires many more generations, and, due to the use of a full-blown Earley parser, the evaluation of a single grammar takes significantly more time than the evaluation of a PDA. The resulting computational burden is high and makes it difficult to scale up this approach to more difficult languages and larger numbers of input sentences.

No.	Best	Average	Generations	avg. % correct on $\mathcal{S}'_+$	$\mathcal{S}'_-$	$\mathcal{T}_+$	$\mathcal{T}_-$
1	1.000	1.000 ± 0.000	14 ± 5	100	100	100	100
2	1.000	0.975 ± 0.046	1534 ± 846	93	100	75	93
3	1.000	1.000 ± 0.000	68 ± 64	100	100	56	88
4	1.000	0.989 ± 0.031	1105 ± 952	98	100	94	83
5	1.000	0.912 ± 0.118	942 ± 932	83	100	100	71
6	1.000	1.000 ± 0.000	330 ± 718	100	100	89	49

Table 5.4: Results on languages 1–6, using binary encoding and operators, averaged over 5 runs, using the training examples of Table 4.4.

No.	Best	Average	Generations	avg. % correct on $\mathcal{S}'_+$	$\mathcal{S}'_-$	$\mathcal{T}_+$	$\mathcal{T}_-$
1	1.000	1.000 ± 0.000	6 ± 3	100	100	100	100
2	1.000	0.967 ± 0.093	925 ± 1005	93	100	79	89
3	1.000	1.000 ± 0.000	391 ± 524	100	100	53	82
4	1.000	0.956 ± 0.058	849 ± 1111	91	100	89	74
5	0.938	0.938 ± 0.000	814 ± 849	88	100	97	68
6	1.000	0.992 ± 0.023	123 ± 151	98	100	88	50

Table 5.5: Results on languages 1–6, using symbolic encoding and operators, averaged over 5 runs, using the training examples of Table 4.4.

No.	Best	Average	Generations	avg. % correct on $\mathcal{S}_+$	$\mathcal{S}_-$	$\mathcal{T}_+$	$\mathcal{T}_-$
7	1.000	1.000 ± 0.000	312 ± 268	100	100	100	100
8	1.000	1.000 ± 0.000	327 ± 220	100	100	100	100
9	1.000	1.000 ± 0.000	295 ± 454	100	100	100	100
10	0.980	0.976 ± 0.007	547 ± 225	100	98	100	97

Table 5.6: Results on languages 7–10, using binary encoding and operators, averaged over 5 runs, 100 positive and negative training examples.

No.	Best	Average	Generations	avg. % correct on $\mathcal{S}_+$	$\mathcal{S}_-$	$\mathcal{T}_+$	$\mathcal{T}_-$
7	1.000	1.000 ± 0.000	285 ± 187	100	100	100	100
8	1.000	1.000 ± 0.000	348 ± 306	100	100	100	100
9	1.000	0.924 ± 0.211	197 ± 184	91	96	97	95
10	1.000	1.000 ± 0.000	504 ± 283	100	100	100	99

Table 5.7: Results on languages 7–10, using symbolic encoding and operators, averaged over 5 runs, 100 positive and negative training examples.

	PDA				Grammar			
	binary		symbolic		binary		symbolic	
No.	$\mathcal{S}'$	$\mathcal{T}$	$\mathcal{S}'$	$\mathcal{T}$	$\mathcal{S}'$	$\mathcal{T}$	$\mathcal{S}'$	$\mathcal{T}$
1	100.0	100.0	100.0	94.5	100.0	100.0	100.0	100.0
2	90.5	73.5	87.5	71.5	96.5	84.0	96.5	84.0
3	100.0	77.5	100.0	77.0	100.0	72.0	100.0	67.5
4	99.5	58.5	98.0	60.5	99.0	88.5	95.5	81.5
5	100.0	84.0	100.0	87.5	91.5	85.5	94.0	82.5
6	88.5	57.5	87.5	63.5	100.0	69.0	99.0	69.0

Table 5.8: Average results of grammars and pushdown automata on languages 1–6, using the training examples of Table 4.4.

	PDA				Grammar			
	binary		symbolic		binary		symbolic	
No.	$\mathcal{S}$	$\mathcal{T}$	$\mathcal{S}$	$\mathcal{T}$	$\mathcal{S}$	$\mathcal{T}$	$\mathcal{S}$	$\mathcal{T}$
7	97.0	97.5	95.5	96.5	100.0	100.0	100.0	100.0
8	100.0	97.0	97.5	90.5	100.0	100.0	100.0	100.0
9	92.0	79.0	88.0	82.0	100.0	100.0	93.5	96.0
10	97.5	98.5	99.0	98.5	99.0	98.5	100.0	99.5

Table 5.9: Average results of grammars and pushdown automata on languages 7–10, 100 positive and negative training examples.

5.3 Discussion

In this chapter, it was shown that genetic algorithms can be used to infer context-free grammars from positive and negative examples of a language. Grammars for a set of regular languages, the language of correct parentheses expressions, the language of equal numbers of a's and b's, the two-symbol palindromes, and a tiny natural language subset were inferred. Further experimentation will have to show whether this technique is applicable to more complex languages. In the experiments, no significant difference between binary and symbolic encodings was observed.

It is clear that the grammar-based approach of this chapter differs substantially from the PDA-based approach of the previous chapter. Although the results on grammars are slightly better, the PDAs' more efficient implementation entails smaller computational requirements.

Several extensions of the approach of this and the previous chapter on PDA inference might be interesting. The evaluation functions used here weighed different aspects of grammars and condensed these into a single scalar. Instead of using such a scalar fitness, we could employ a multi-objective algorithm such as Schaffer's Vector Evaluated Genetic Algorithm (VEGA) [129], that uses multidimensional fitness vectors.

The fitness evaluation might further be enhanced by regarding the 'educational value' of the example sentences. If many of the grammars or automata in the population can judge an example correctly, this educational value is quite low. On the other hand, difficult sentences

that are often misclassified should be valued higher. To include these educational values in the fitness function, a weight factor could be assigned to each sentence, which is proportional to the number of grammars of the population that do not analyze this sentence correctly. A similar approach is the dynamic subset selection scheme of [56], that chooses a subset of difficult and 'disused' training examples from a larger set.

Another possibly useful approach, introduced by Hillis [71], is to use the concept of co-evolution. The example sentences form a population of parasites that compete with the population of grammars/automata. The fitness of a sentence is based on the difficulty with which the grammars can analyze this sentence, i.e., the more difficult a sentence is, the higher its fitness will be.

In this approach, the reproduction of correct sentences could be implemented using De Weger's tree crossover (TX) operator [42], or other recombination operators analogous to those of Koza's Genetic Programming paradigm [93] (see Section 2.6.3). In this paradigm, Lisp expressions are represented as parse trees, and crossover is implemented by taking suitable subtrees and exchanging them.

Reproducing incorrect sentences is even simpler, since there is no tree structure to be preserved. Therefore, a straightforward recombination of parts of incorrect examples (which is likely to result in new incorrect sentences), combined with a test whether the offspring is incorrect, will suffice.

As Wyard already pointed out [157], a bucket-brigade algorithm [74], in which the population consists of individual rules instead of complete grammars, might prove to be useful. In such an algorithm, a population member's fitness is determined by scoring its ability to correctly analyze the examples in conjunction with the other rules of the population. This approach has been employed successfully in the inference of classifier systems [62]. A possible advantage of this approach is that in a population of rules we only have to evaluate the merit of different grammar rules once, as opposed to a population of grammars, in which a single rule might appear in many different grammars. This might alleviate the computational burden of the algorithm.