Solving classification problems using Infix Form Genetic Programming

Mihai Oltean and Crina Groşan

Department of Computer Science,
Faculty of Mathematics and Computer Science,
Babeş-Bolyai University, Kogălniceanu 1
Cluj-Napoca, 3400, Romania.
{moltean, cgrosan}@cs.ubbcluj.ro

Abstract. A new evolutionary technique, called Infix Form Genetic Programming (IFGP) is proposed in this paper. The IFGP individuals are strings encoding complex mathematical expressions. The IFGP technique is used for solving several classification problems. All test problems are taken from PROBEN1 and contain real world data. IFGP is compared to Linear Genetic Programming (LGP) and Artificial Neural Networks (ANNs). Numerical experiments show that IFGP is able to solve the considered test problems with the same (and sometimes even better) classification error than that obtained by LGP and ANNs.

1 Introduction

Classification is a task of assigning inputs to a number of discrete categories or classes [3, 9]. Examples include classifying a handwritten letter as one from A-Z, classifying a speech pattern to the corresponding word, etc.

Machine learning techniques have been extensively used for solving classification problems. In particular Artificial Neural Networks (ANNs) [3] have been originally designed for classifying a set of points in two distinct classes.

Recently Genetic Programming (GP) techniques [1, 4] have been used for classification purposes. GP has been developed as a technique for evolving computer programs. Originally GP chromosomes were represented as trees and evolved using specific genetic operators [4]. Later, several linear variants of GP have been proposed. Some of them are Linear Genetic Programming (LGP) [1], Grammatical Evolution (GE) [6], Gene Expression Programming (GEP) [2]. In all these techniques chromosomes are represented as strings encoding complex computer programs.

Some of these GP techniques have been used for classification purposes. For instance, LGP [1] has been used for solving several classification problems in PROBEN1. The conclusion was that LGP is able to solve the classification problems with the same error rate as a neural network.

A new evolutionary technique, called Infix Form Genetic Programming (IFGP), is proposed in this paper. IFGP chromosomes are strings encoding complex mathematical expressions using infix form. This departs from GE [6] and GADS

[7] approaches which encode expressions using a grammar [4] . An interesting feature of IFGP is its ability of storing multiple solutions of a problem in a chromosome.

In this paper IFGP is used for solving several real-world classification problems taken from PROBEN1.

The paper is organized as follows. Several machine learning techniques (Neural Networks and Linear Genetic Programming) for solving classification problems are briefly described in section 2. Section 3 describes in detail the proposed IFGP technique. The benchmark problems used for comparing these techniques are presented in section 4. Comparison results are presented in section 5.

2 Machine learning techniques used for classification task

Two of the machines learning techniques used for solving classification problems are described in this section.

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) [3, 9] are motivated by biological neural networks. ANNs have been extensively used for classification and clustering purposes. In their early stages, neural networks have been primarily used for classification tasks. The most popular architectures consist of multi layer perceptrons (MLP) and the most efficient training algorithm is backpropagation [3].

ANNs have been successfully applied for solving real-world problems. For instance, the PROBEN1 [8] data set contains several difficult real-world problems accompanied by the results obtained by running different ANNs architectures on the considered test problems. The method applied in PROBEN for training was RPROP (a fast backpropagation algorithm - similar to Quickprop). The parameters used were not determined by a trial-and-error search. Instead they are just educated guesses. More information about the ANNs parameters used in PROBEN1 can be found in [8].

2.2 Linear Genetic Programming

Linear Genetic Programming (LGP) [1] uses a specific linear representation of computer programs.

Programs of an imperative language (like ***C***) are evolved instead of the tree-based GP expressions of a functional programming language (like ***LISP***)

An LGP individual is represented by a variable-length sequence of simple ***C*** language instructions. Instructions operate on one or two indexed variables (registers) r or on constants c from predefined sets. The result is assigned to a destination register.

An example of LGP program is the following:

```
void LGP_Program(double v[8])
    {
    ...
    v[0] = v[5] + 73;
    v[7] = v[4] - 59;
    if (v[1] > 0)
    if (v[5] > 21)
    v[4] = v[2] *v[1];
    v[2] = v[5] + v[4];
    v[6] = v[1] * 25;
    v[6] = v[4] - 4;
    v[1] = sin(v[6]);
    if (v[0] > v[1])
    v[3] = v[5] * v[5];
    v[7] = v[6] * 2;
    v[5] = [7] + 115;
    if (v[1] ≤ v[6])
    v[1] = sin(v[7]);
    }
```

A linear genetic program can be turned into a functional representation by successive replacements of variables starting with the last effective instruction. Variation operators are crossover and mutation. By crossover continuous sequences of instructions are selected and exchanged between parents. Two types of mutations are used: micro mutation and macro mutation. By micro mutation an operand or operator of an instruction is changed. Macro mutation inserts or deletes a random instruction.

Crossover and mutations change the number of instructions in chromosome.

The population is divided into demes which are arranged in a ring. Migration is allowed only between a deme and the next one (in the clockwise direction) with a fixed probability of migration. Migrated individuals replace the worst individuals in the new deme. The deme model is used in order to preserve population diversity (see [1] for more details).

3 IFGP technique

In this section the proposed *Infix Form Genetic Programming* (IFGP) technique is described. IFGP uses linear chromosomes for solution representation (i.e. a chromosome is a string of genes). An IFGP chromosome usually encodes several solutions of a problem.

3.1 Prerequisite

We denote by F the set of function symbols (or operators) that may appear in a mathematical expression. F usually contains the binary operators {+, −, *,

/}. By *Number_of_Operators* we denote the number of elements in F. A correct mathematical expression also contains some terminal symbols. The set of terminal symbols is denoted by T. The number of terminal symbols is denoted by *Number_of_Variables*.

Thus, the symbols that may appear in a mathematical expression are $T \cup F \cup \{$'(', ')'$\}$. The total number of symbols that may appear in a valid mathematical expression is denoted by *Number_of_Symbols*.

By C_i we denote the value on the i^{th} gene in a IFGP chromosome and by G_i the symbol in the i^{th} position in the mathematical expression encoded into an IFGP chromosome.

3.2 IFGP Algorithm

A steady-state [10] variant of IFGP is employed in this paper. The algorithm starts with a randomly chosen population of individuals. The following steps are repeated until a termination condition is reached. Two parents are chosen at each step using binary tournament selection [5]. The selected individuals are recombined with a fixed crossover probability p_c. By recombining two parents, two offspring are obtained. The offspring are mutated and the best of them replaces the worst individual in the current population (only if the offspring is better than the worst individual in population). The algorithm returns as its answer the best expression evolved for a fixed number of generations.

3.3 IFGP individual representation

In this section we describe how IFGP individuals are represented and how they are decoded in order to obtain a valid mathematical expression.

Each IFGP individual is a fixed size string of genes. Each gene is an integer number in the interval [0 .. *Number_Of_Symbols* - 1]. An IFGP individual can be transformed into a functional mathematical expression by replacing each gene with an effective symbol (a variable, an operator or a parenthesis).

Example

If we use the set of functions symbols $F = \{+, *, -, /\}$, and the set of terminals $T = \{a, b\}$, the following chromosome

C = 7, 3, 2, 2, 5

is a valid chromosome in IFGP system.

3.4 IFGP decoding process

We will begin to decode this chromosome into a valid mathematical expression. In the first position (in a valid mathematical expression) we may have either a variable, or an open parenthesis. That means that we have *Number_Of_Variables* + 1 possibilities to choose a correct symbol on the first position. We put these possibilities in order: the first possibility is to choose the variable x_1, the second possibility is to choose the variable x_2 ... the last possibility is to choose the

closed parenthesis ')'. The actual value is given by the value of the first gene of the chromosome. Because the number stored in a chromosome gene may be larger than the number of possible correct symbols for the first position we take only the value of the first gene *modulo* $[6, 7]$ number of possibilities for the first gene.

Generally, when we compute the symbol stored in the i^{th} position in expression we have to compute first how many symbols may be placed in that position. The number of possible symbols that may be placed in the current position depends on the symbol placed in the previous position. Thus:

(i) if the previous position contains a variable (x_i), then for the current position we may have either an operator or a closed parenthesis. The closed parenthesis is considered only if the number of open parentheses so far is larger than the number of closed parentheses so far.

(ii) if the previous position contains an operator, then for the current position we may have either a variable or an open parenthesis.

(iii) if the previous position contains an open parenthesis, then for the current position we may have either a variable or another open parenthesis.

(iv) if the previous position contains a closed parenthesis, then for the current position we may have either an operator or another closed parenthesis. The closed parenthesis is considered only if the number of open parentheses so far is larger than the number of closed parentheses.

Once we have computed the number of possibilities for the current position it is easy to determine the symbol that will be placed in that position: first we take the value of the corresponding gene modulo the number of possibilities for that position. Let p be that value ($p = \mathrm{C}_i \ mod \ Number\_Of\_Possibilities$). The p^{th} symbol from the permitted symbols for the current is placed in the current position in the mathematical expression. (Symbols that may appear into a mathematical expression are ordered arbitrarily. For instance we may use the following order: x_1, x_2, ..., +, -, *, /, '(', ')'.)

All chromosome genes are translated but the last one. The last gene is used by the correction mechanism (see below).

The obtained expression usually is syntactically correct. However, in some situations the obtained expression needs to be repaired. There are two cases when the expression needs to be corrected:

The last symbol is an operator (+, -, *, /) or an open parenthesis. In that case a terminal symbol (a variable) is added to the end of the expression. The added symbol is given by the last gene of the chromosome.

The number of open parentheses is greater than the number of closed parentheses. In that case several closed parentheses are automatically added to the end in order to obtain a syntactically correct expression.

Remark: If the correction mechanism is not used, the last gene of the chromosome will not be used.

Example

Consider the chromosome $C = 7, 3, 2, 0, 5, 2$ and the set of terminal and function symbols previously defined ($T = \{a, b\}$, $F = \{+, -, *, /\}$).

For the first position we have 3 possible symbols (a, b and '('). Thus, the symbol in the position C_0 ***mod*** $3 = 1$ in the array of possible symbols is placed in the current position in expression. The chosen symbol is b, because the index of the first symbol is considered to be 0.

For the second position we have 4 possibilities (+, -, *, /). The possibility of placing a closed parenthesis is ignored since the difference between the number of open parentheses and the number of closed parentheses is zero. Thus, the symbol '/' is placed in position 2.

For the third position we have 3 possibilities (a, b and '('). The symbol placed on that position is an open parenthesis '('.

In the fourth position we have 3 possibilities again (a, b and '('). The symbol placed on that position is the variable a.

For the last position we have 5 possibilities (+, -, *, /) and the closed parenthesis ')'. We choose the symbol on the position 5 **mod** $5 = 0$ in the array of possible symbols. Thus the symbol '+' is placed in that position.

The obtained expression is $E = b\ /\ (a+$.

It can be seen that the expression E it is not syntactically correct. For repairing it we add a terminal symbol to the end and then we add a closed parenthesis. Now we have obtained a correct expression:

$$E = b\ /\ (a + a).$$

The expression tree of E is depicted in Fig. 1.

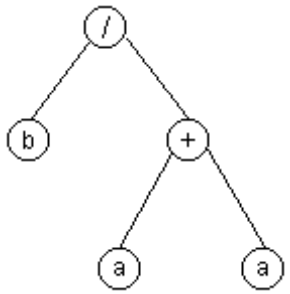

Fig. 1. The expression tree of $E = b\ /\ (a + a)$.

3.5 Using constants within IFGP model

An important issue when designing a new GP technique is the way in which the constants are embaded into the proposed model.

Fixed or ephemeral random constants have been extensively tested within GP systems [4]. The interval over which the constants are initially generated

is usually problem-dependent. For a good functionality of the program a priori knowledge about the required constants is usually needed.

By contrast, IFGP employs a problem-independent system of constants.

We know that each real number may be written as a sum of powers of 2.

In the IFGP model each constant is a power of 2. The total number of constants is also problem-independent. For instance, if we want to solve problems using double precision constants, we need 127 constants: 2^{-63}, 2^{-62},..., 2^{-1}, 2^{0}, 2^{1},..., 2^{63}. Particular instances of the classification problems may require fewer constants. As can be seen in section 5 a good solution for some classification problems can be obtained without using constants. However, if we do not know what kind of constants are required by the problems being solved it is better the use the double precision system of constants (as described above).

Within the IFGP chromosome each constant is represented by its exponent. For instance the constant 2^{17} is stored as the number 17, the constant 2^{-4} is stored as the number - 4 and the constant $1 = 2^{0}$ is stored as the number 0. These constants will act as terminal symbols. Thus the extended set of terminal symbols is $T = \{x_1, x_2, \ldots,$ -63, -62, .., -1, 0, 1, ..., 62, 63$\}$.

3.6 Fitness assignment process

In this section we describe how IFGP may be efficiently used for solving classification problems.

A GP chromosome usually stores a single solution of a problem and the fitness is normally computed using a set of fitness cases.

Instead of encoding a single solution, an IFGP individual is allowed to store multiple solutions of a problem. The fitness of each solution is computed in a conventional manner and the solution having the best fitness is chosen to represent the chromosome.

In the IFGP representation each sub-tree (sub-expression) is considered as a potential solution of a problem. For example, the previously obtained expression (see section 3.4) contains 4 distinct solutions (sub-expressions):

$E_1 = a$,
$E_2 = b$,
$E_3 = a + a$,
$E_4 = b$ / $(a + a)$.

Now we will explain how the fitness of a (sub)expression is computed.

Each class has associated a numerical value: the first class has the value 0, the second class has the value 1 and the m^{th} class has associated the numerical value $m-1$. Any other system of distinct numbers may be used. We denote by o_k the number associated to the k^{th} class.

The value $v_j(E_i)$ of each expression E_i (in an IFGP) chromosome for each row (example) j in the training set is computed. Then, each row in the training set will be classified to the nearest class (the class k for which the difference $|v_j(E_i) - o_k|$ is minimal). The fitness of a (sub)expression is equal to the number

of incorrectly classified examples in the training set. The fitness of an IFGP chromosome will be equal to the fitness of the best expression encoded in that chromosome.

Remarks :

(i) Since the set of numbers associated with the problem classes was arbitrarily chosen it is expected that different systems of number to generate different solutions.

(ii) When solving symbolic regression or classification problems IFGP chromosomes need to be traversed twice for computing the fitness. That means that the complexity of the IFGP decoding process it is not higher than the complexity of other methods that store a single solution in a chromosome.

3.7 Search operators

Search operators used within the IFGP model are recombination and mutation. These operators are similar to the genetic operators used in conjunction with binary encoding [5]. By recombination two parents exchange genetic material in order to obtain two offspring. In this paper only two-point recombination is used [5]. Mutation operator is applied with a fixed mutation probability (p_m). By mutation a randomly generated value over the interval [0, *Number_of_Symbols*–1] is assigned to the target gene.

3.8 Handling exceptions within IFGP

Exceptions are special situations that interrupt the normal flow of expression evaluation (program execution). An example of exception is *division by zero* which is raised when the divisor is equal to zero.

GP techniques usually use a *protected exception* handling mechanism [4, 6]. For instance if a division by zero exception is encountered, a predefined value (for instance 1 or the numerator) is returned. This kind of handling mechanism is specific for Linear GP [1], standard GP [4] and GE [6].

IFGP uses a new and specific mechanism for handling exceptions. When an exception is encountered (which is always generated by a gene containing a function symbol), the entire (sub) tree which has generated the exception is mutated (changed) into a terminal symbol. Exception handling is performed during the fitness assignment process.

4 Data sets

Numerical experiments performed in this paper are based on several benchmark problems taken from PROBEN1 [8]. These datasets were created based on the datasets from the UCI Machine Learning Repository [11].

Used problems are briefly described in what follows.

Cancer

Diagnosis of breast cancer. Try to classify a tumor as either benignant or malignant based on cell descriptions gathered by microscopic examination.

Diabetes

Diagnosis diabetes of Pima Indians based on personal data and the results of medical examinations try to decide whether a Pima Indian individual is diabetes positive or not.

Heartc

Predicts heart disease. Decides whether at least one of four major vessels is reduced in diameter by more than 50%. The binary decision is made based on personal data such as age sex smoking habits subjective patient pain descriptions and results of various medical examinations such as blood pressure and electro cardiogram results.

This data set was originally created by Robert Detrano from V.A. Medical Center Long Beach and Cleveland Clinic Foundation.

Horse

Predicts the fate of a horse that has colic. The results of a veterinary examination of a horse having colic are used to predict whether the horse will survive will die or should be euthanized.

The number of inputs, of classes and of available examples, for each test problem, are summarized in Table 1.

Table 1. Summarized attributes of several classification problems from PROBEN1

Problem	Number of inputs	Number of classes	Number of examples
cancer	9	2	699
diabetes	8	2	768
heartc	35	2	303
horse	58	3	364

5 Numerical experiments

The results of several numerical experiments with ANNs, LGP and IFGP are presented in this section.

Each data set is divided in three sub-sets (training set -50%, validation set - 25 %, and test set – 25%) (see [8]).

The test set performance is computed for that chromosome which had minim validation error during the search process. This method, called *early stopping*, is a good way to avoid overfitting [8] of the population individuals to the particular training examples used. In that case the generalization performance will be reduced.

In [1] Linear GP was used to solve several classification problems from PROBEN1. The parameters used by Linear GP are given in Table 2.

Table 2. Linear GP parameters used for solving classification tasks from PROBEN1

Parameter	**Value**
Population size	5000
Number of demes	10
Migration rate	5%
Classification error weight in fitness	1.0
Maximum number of generations	250
Crossover probability	90%
Mutation probability	90%
Maximum mutation step size for constants	± 5
Maximum program size	256 instructions
Initial maximum program size	25 instructions
Function set	{+, -, *, /, *sin*, *exp*, if >, if $\leq$}
Terminal set	{0,..,256} $\cup$ {input variables}

The parameters of the IFGP algorithm are given in Table 3.

Table 3. IFGP algorithm parameters for solving classification problems from PROBEN1

Parameter	**Value**
Population Size	250
Chromosome length	30
Number of generations	250
Crossover probability	0.9
Crossover type	Two-point Crossover
Mutation	2 mutations per chromosome
Number of Constants	41 $\{2^{-20}, \ldots, 2^{0}, 2^{20}\}$
Function set	{+, -, *, /, *sin*, *exp*}
Terminal set	{input variables} $\cup$ The set of constants.

The results of the numerical experiments are presented in Table 4.

From Table 4 it can be seen that IFGP is able to obtain similar performances as those obtained by LGP even if the population size and the chromosome length used by IFGP are smaller than those used by LGP. When compared to ANNs we can see that IFGP is better only in 3 cases (out of 12).

We are also interested in analysing the relationship between the classification error and the number of constants used by the IFGP chromosomes. For this purpose we will use a small population made up of only 50 individuals. Note that this is two magnitude orders smaller than those used by LGP. Other IFGP parameters are given in Table 3. Experimental results are given in Table 5.

Table 4. Classification error rates of IFGP, LGP and ANN for some date sets from PROBEN1. LGP results are taken from [1]. ANNs results are taken from [8]. Results are averaged over 30 runs

Problem	**IFGP–test set**			**LGP–test set**			**NN–test set**	
	best	**mean**	**stddev**	**best**	**mean**	**stddev**	**mean**	**stddev**
cancer1	1.14	2.45	0.69	0.57	2.18	0.59	1.38	0.49
cancer2	4.59	6.16	0.45	4.02	5.72	0.66	4.77	0.94
cancer3	3.44	4.92	1.23	3.45	4.93	0.65	3.70	0.52
diabetes1	22.39	25.64	1.61	21.35	23.96	1.42	24.10	1.91
diabetes2	25.52	28.92	1.71	25.00	27.85	1.49	26.42	2.26
diabetes3	21.35	25.31	2.20	19.27	23.09	1.27	22.59	2.23
heart1	16.00	23.06	3.72	18.67	21.12	2.02	20.82	1.47
heart2	1.33	4.40	2.35	1.33	7.31	3.31	5.13	1.63
heart3	12.00	13.64	2.34	10.67	13.98	2.03	15.40	3.20
horse1	23.07	31.11	2.68	23.08	30.55	2.24	29.19	2.62
horse2	30.76	35.05	2.33	31.87	36.12	1.95	35.86	2.46
horse3	30.76	35.01	2.82	31.87	35.44	1.77	34.16	2.32

Table 5. Classification error rates of IFGP (on the test set) using different number of constants. Results are averaged over 30 runs

Problem	**IFGP–41 constants**			**IFGP–0 constants**			**IFGP–81 constants**		
	best	**mean**	**stddev**	**best**	**mean**	**stddev**	**best**	**mean**	**stddev**
cancer1	1.14	2.45	0.60	1.14	3.18	1.06	1.14	2.41	0.77
cancer2	4.02	6.16	0.91	4.02	6.14	0.81	4.59	6.24	0.99
cancer3	3.44	5.17	1.10	2.87	5.07	1.50	2.87	5.15	1.07
diabetes1	22.39	25.74	2.19	21.87	26.04	1.76	21.87	25.34	2.08
diabetes2	26.04	29.91	1.65	25.00	29.21	2.21	25.52	29.82	1.30
diabetes3	21.87	25.34	1.76	19.79	24.79	1.91	22.91	25.88	3.60
heart1	17.33	24.44	3.76	18.67	23.28	3.33	18.66	25.28	3.64
heart2	1.33	6.97	4.07	1.33	4.97	3.16	1.33	7.06	4.60
heart3	12.00	14.00	2.51	10.67	15.42	3.40	9.33	15.11	4.25
horse1	26.37	32.16	3.12	25.27	30.69	2.49	27.47	31.57	1.91
horse2	30.76	35.64	2.37	30.76	35.49	2.81	31.86	35.71	2.23
horse3	28.57	34.13	3.14	28.57	35.67	3.90	28.57	34.90	3.53

From Table 5 it can be seen that the best results are obtained when the constants are not used in our IFGP system. For 8 (out of 12) cases the best result obtained by IFGP outperform the best result obtained by LGP. That does not mean that the constants are useless in our model and for the considered test problems. An explanation for this behaviour can be found if we have a look at the parameters used by IFGP. The population size (of only 50 individuals) and the chromosome length (of only 30 genes) could not be enough to obtain a

perfect convergence knowing that some problems have many parameters (input variables). For instance the '*horse'* problem has 58 attributes and a chromosome of only 30 genes could not be enough to evolve a complex expression that contains sufficient problem's variables and some of the considered constants. It is expected that longer chromosomes will increase the performances of the IFGP technique.

6 Conclusion and future work

A new evolutionary technique, Infix Form Genetic Programming (IFGP) has been proposed in this paper. The IFGP technique has been used for solving several classification problems. Numerical experiments show that the error rates obtained by using IFGP are similar and sometimes even better than those obtained by Linear Genetic Programming.

Further numerical experiments will try to analyse the relationship between the parameters of the IFGP algorithm and the classification error for the considered test problems.

References

1. Brameier, M., Banzhaf, W.: A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining. IEEE Transactions on Evolutionary Computation (2001) 5:17-26.
2. Ferreira, C.: Gene Expression Programming: a New Adaptive Algorithm for Solving Problems, Complex Systems (2001), 13(2):87-129.
3. Haykin S.: Neural Networks, a Comprehensive Foundation. second edition, Prentice-Hall, Englewood Cliffs (1999).
4. Koza, J. R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MA: MIT Press (1992).
5. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, Reading, MA (1989).
6. O'Neill, M., Ryan, C.: Grammatical Evolution. IEEE Transaction on Evolutionary Computation, Vol 5, (2001) 349-358.
7. Paterson, N.R., Livesey, M.J.: Distinguishing Genotype and Phenotype in Genetic Programming. In: Genetic Programming: Proceedings of the first annual conference, Koza,John R; Goldberg, David E; Fogel, David B; Riolo, Rick (Eds), The MIT Press (1996) 141-150.
8. Prechelt, L.: PROBEN1 – A Set of Neural Network Problems and Benchmarking Rules, Technical Report 21, University of Karlsruhe (1994).
9. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, Prentice Hall, Englewood Cliffs, NJ (1994).
10. Syswerda, G.: Uniform Crossover in Genetic Algorithms. In: Proc. 3rd Int. Conf. on Genetic Algorithms, Schaffer, J.D. (eds.), Morgan Kaufmann Publishers, San Mateo, CA (1989) 2-9.
11. UCI Machine Learning Repository, Available from www.ics.uci.edu/∼mlearn/MLRepository.html