GP Ensembles for Large Scale Data Classification

Gianluigi Folino, Clara Pizzuti, and Giandomenico Spezzano
ICAR-CNR,
Via P.Bucci 41/C,
Univ. della Calabria
87036 Rende (CS), Italy
{folino,pizzuti,spezzano}@icar.cnr.it

July 6, 2005

Abstract

An extension of Cellular Genetic Programming for data classification to induce an ensemble of predictors is presented. The method is able to deal with large data sets that do not fit in main memory since each classifier is trained on a subset of the overall training data. The predictors are then combined to classify new tuples. Experiments on several data sets show that, by using a training set of reduced size, better classification accuracy can be obtained, but at a much lower computational cost.

Index terms: data mining, genetic programming, classification, bagging, boosting.

Corresponding author: Clara Pizzuti

1 Introduction

Genetic programming (GP)[18] is a general purpose method that has been successfully applied to solve problems in different application domains. In the data mining field [10], GP showed to be a particularly suitable technique to deal with the task of data classification [15, 22, 25, 20, 11] by evolving decision trees. Many data mining applications manage databases consisting of a very large number of objects, each of which having several attributes. This huge amount of data (gigabytes or even terabytes of data) is too large to fit into the memory of computers, thus it causes serious problems in the realization of predictors, such as decision trees [23]. One approach is to partition the training data into small subsets, obtain an ensemble of predictors on the base of each subset, and then use a voting classification algorithm to predict the class label of new objects [6, 4, 7].

Bagging [2] and boosting [27] are well known ensemble techniques that repeatedly run a learning algorithm on different distributions over the training data. Bagging builds *bags* of data of the same size of the original data set by applying random sampling with replacement. Unlike bagging, boosting draws tuples randomly, according to a distribution, and tries to concentrate on harder examples by adaptively changing the distributions of the training set on the base of the performance of the previous classifiers. It has been shown that bagging and boosting improve the accuracy of decision tree classifiers [2, 24, 1].

The combination of Genetic Programming and ensemble techniques has been receiving a lot of attention because of the improvements that GP obtains when enriched with these methods [17, 28, 19, 5, 12, 14]. The first proposal of using bagging and boosting in Genetic Programming is due to Iba [17]. The main features of his approach are the following. First divide the whole population in a set of subpopulations, then evolve each subpopulation sequentially and independently on a training set of the same size of the whole data set, obtained by applying resampling techniques, finally select the best individuals from each of the subpopulations to vote on the test set. Iba applied his approach to well-known problems used in GP literature, like discovery of trigonometric identity, and boolean concept formation.

However, when the data set is large, constructing and elaborating a fixed number of training

sets of the same size of the entire data set does not seem a feasible approach. In particular, for the task of data classification, if the training set contains a high number of tuples with many features, large decision trees are requested to accurately classify them. Thus a decision tree generator based on genetic programming should cope with a population of large sized trees. It has been pointed out [25] that, in order to obtain the same classification accuracy of a decision tree generated by C4.5 [23], small population size is inadequate. Processing large populations of trees that contain many nodes considerably degrades the execution time of GP and requires an enormous amount of memory.

In this case data reduction through the partitioning of the data set into smaller subsets seems a good approach, though an important aspect to consider is which kind of partitioning has the minimal impact on the accuracy of the results. Furthermore, to speed up the overall predictor generation process it seems straightforward to consider parallel implementations of bagging and boosting.

Cellular Genetic Programming for data classification ($CGPC$) enhanced with ensemble techniques [12, 14] showed to enhance both the prediction accuracy and the running time of $CGPC$. In [12] and [14] the algorithms $BagCGPC$ (*Bag Cellular Genetic Programming Classifier*) and $BoostCGPC$ (*Boost Cellular Genetic Programming Classifier*), that implement the bagging technique of Breiman [2] and the $AdaBoost.M1$ boosting algorithm of Freund and Shapire [16], respectively, by using $CGPC$ as base classifier, have been presented.

This paper extends the previous works by implementing the algorithm $AdaBoost.M2$ to efficiently deal also with multi-class problems, and presents several experiments that show how Cellular Genetic Programming enriched with these voting algorithms obtains enhancements in both accuracy and execution time. More interestingly, it is experimentally shown that higher accuracy can be obtained by using a small subset of the training set at a much lower computational cost. The main contributions of the paper can be summarized as follows:

- a parallel cellular implementation of genetic programming extended with bagging and boosting techniques is described and applied for the task of data classification;
- two algorithms $BagCGPC$, implementing the bagging technique, and $BoostCGPC$, im-

plementing the boosting one, are reported and compared with *CGPC*, that realizes the base cellular genetic programming approach for data classification;

- to asses the effectiveness of the approach, experiments on several data sets, having different sizes, and number of attributes and classes are presented;

- the influence of the training set size on the accuracy of the methods has been evaluated by executing *BagCGPC*, *BoostCGPC* and *CGPC* on training sets, built from the overall training set by using random sampling with replacement, of size 5%, 10%, 20%, 50%, and 100% of the size of the training data;

- the accuracy obtained by *BagCGPC* and *BoostCGPC* has been studied when both the number of classifiers and the sample sizes are increased and the error rates obtained have been compared with that produced by *CGPC* running on all the data set;

- a scalability analysis of the algorithms when the number of available processors augments has been performed. The experiments pointed out that *BagCGPC* and *BoostCGPC* outperform *CGPC* both in accuracy and execution time. More interestingly, higher accuracy can be obtained by using a small sample, often only of size the 5% of the overall data set, at a much lower computational cost. The approaches presented can thus deal with large data sets that do not fit in main memory since each classifier can be trained on a subset of the overall training data.

The paper is organized as follows. In section 2 the standard approach to data classification through genetic programming is explained. In section 3 the cellular genetic programming method is presented. Section 4 reviews the bagging and boosting techniques. Section 5 describes the extension of cellular genetic programming with the Boosting technique. Section 6 describes the extension of cellular genetic programming with the bagging algorithm. In section 7, finally, the results of the method on some standard problems are presented.

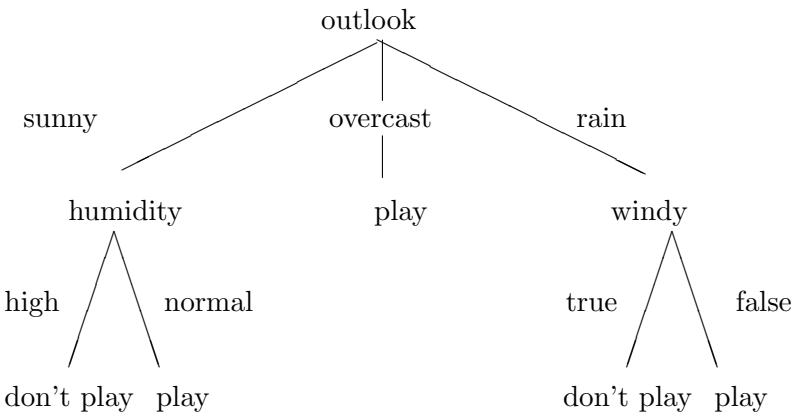


Figure 1: An example of decision tree with Terminal set=$\{play, don't\ play\}$ and Function set = $\{\ f_{outlook}(sunny, overcast, rain),\ f_{humidity}(high, normal),\ f_{windy}(true, false)\}$

2 Data Classification through Genetic Programming

Genetic programming [18], unlike other evolutionary strategies, is particularly apt to inductively generate decision trees for the task of data classification. In fact, decision trees can be interpreted as composition of functions where the function set is the set of attribute tests and the terminal set are the classes. The function set can be obtained by converting each attribute into an attribute-test function. Thus there are as many functions as attributes. For each attribute A, if $A_1, \ldots A_n$ are the possible values A can assume, the corresponding attribute-test function f_A has arity n, and if the value of A is A_i then $f_A(A_1, \ldots A_n) = A_i$. When a tuple has to be evaluated, the function at the root of the tree tests the corresponding attribute and then executes the argument that outcomes from the test. If the argument is a terminal, then the class name for that tuple is returned, otherwise the new function is executed.

Figure 1 shows a simple decision tree, well known in the literature [21], for deciding to play tennis on the base of the weather conditions, with the corresponding terminal and function sets. For example, if a tuple has the value of the attribute *outlook* equal to *sunny* and that of *humidity* equal to *normal*, then it is classified as *play*. In order to evaluate the accuracy of the decision tree, the standard measure used in the machine learning community is adopted that computes the fraction of tuples classified into the correct class. Thus the fitness function [18] is

defined as the number of training examples classified into the correct class. Both crossover and mutation must generate syntactically correct decision trees. This means that an attribute can not be repeated more than once in any path from the root to a leaf node. In order to balance the accuracy against the size of tree, the fitness is augmented with an optional parameter, the *parsimony*, which measures the complexity of the individuals. Higher is the parsimony, simpler is the tree, but accuracy diminishes.

Approaches to data classification based on genetic programming can be found in [22, 15, 20, 25, 11, 9].

In the next section the cellular genetic programming approach to data classification, introduced in [11] is presented. The method uses cellular automata as a framework to enable a fine-grained parallel implementation of GP through the diffusion model.

3 Data Classification using Cellular Genetic Programming

Approaches to data classification through genetic programming involve a lot of computation and their performance may drastically degrade when applied to large problems because of the intensive computation of fitness evaluation of each individual in the population. High performance computing is an essential component for increasing the performance and to obtain large-scale efficient classifiers. To this purpose, several approaches have been proposed. The different models used for distributing the computation and to easily parallelize genetic programming, cluster around two main approaches [29]: the well-known *island model* and the *cellular* ($diffusion$) *model*. In the island model several isolated subpopulations evolve in parallel by executing a standard sequential evolutionary algorithm, and periodically exchanging by migration their best individuals with the neighboring subpopulations. In the cellular model each individual has a spatial location on a low-dimensional grid and the individuals interact locally within a small neighborhood. The cellular model considers the population as a system of active individuals that interact only with their direct neighbors. Different neighborhoods can be defined for the cells and the fitness evaluation is done simultaneously for all the individuals. Selection, reproduction and mating take place locally within the neighborhood.

```
Let p_c, p_m be crossover and mutation probability
for each point i in grid do in parallel
   generate a random individual t_i
   evaluate the fitness of t_i
end parallel for
while not MaxNumberOfGeneration do
   for each point i in grid do in parallel
      generate a random probability p
      if (p < p_c)
         select the cell j, in the neighborhood of i,
         such that t_j has the best fitness
         produce the offspring by crossing t_i and t_j
         evaluate the fitness of the offspring
         replace t_i with the best of the two offspring
         evaluate the fitness of the new t_i
      else
      if ( p < p_m + p_c) then
         mutate the individual
         evaluate the fitness of the new t_i
      else
         copy the current individual in the population
      end if
      end if
end parallel for
end while
```

Figure 2: The algorithm CGPC

In [13] a comparison of cellular genetic programming with both canonical genetic programming and the island model using benchmark problems of different complexity is presented and the the superiority of the cellular approach is shown.

Cellular genetic programming (*CGP*) for data classification has been proposed in [11]. The method uses cellular automata as a framework to enable a fine-grained parallel implementation of GP through the diffusion model. The algorithm, in the following referred as *CGPC* (*Cellular Genetic Programming Classifier*), is described in figure 2.

At the beginning, for each cell, an individual (i.e. a tree) is randomly generated and its fitness is evaluated. Then, at each generation every tree undergoes one of the genetic operators (reproduction, crossover, mutation) depending on a probability test, i.e. a random number in the

interval [0,1] is generated and compared with crossover and mutation probability. If crossover is applied, the mate of the current individual is selected by picking the neighbor having the best fitness. Then one random point in each parent is selected as the crossover point, and the subtrees rooted at these points are exchanged to generate the offspring. The current tree is replaced by the best of the two offspring if the fitness of the latter is better than that of the former. If mutation is applied, a random point in the tree is selected, and the subtree rooted at that point is substituted by a newly generated subtree. After the execution of the number of generations defined by the user, the individual with the best fitness represents the classifier.

In the CGPC algorithm two types of parallelism can be exploited: *inter-individual* parallelism, that refers to the ability of evaluating the fitness of the individuals of the population simultaneously, and *intra-individual* parallelism, that enables the computation of the fitness of each individual by handling the data in parallel. The majority of the parallel evolutionary algorithms exploit only the inter-individual parallelism because they are designed to solve problems which are more CPU-bound rather than than I/O bound. In these cases, a typical solution adopted is to divide the population into P subpopulations and to assign each subpopulation to a different node of the parallel machine. In this way, different subsets of individuals have their fitness computed in parallel by different processors. In the data mining context, however, applications are data-intensive, thus inter-individual parallelism does not give good results because of the size of the data sets, which are very large. In this case, data should be managed in parallel by exploiting intra-individual parallelism that distributes the partitions of the data being mined across the processors.

An efficient implementation of the CGPC algorithm that realizes both these types of parallelism is shown in figure 3. The parallel implementation of the algorithm has been done by using a partitioning technique based upon a domain (in our case the population) decomposition in conjunction with the Single-Program-Multiple-Data (SPMD) programming model (i.e. all the processors use the same program, though each has its own data) to support coarse-grain inter-individual parallelism and a Parallel File System (PFS) that realizes the intra-individual parallelism and provides fast, reliable data access from all the nodes in an homogenous or het-

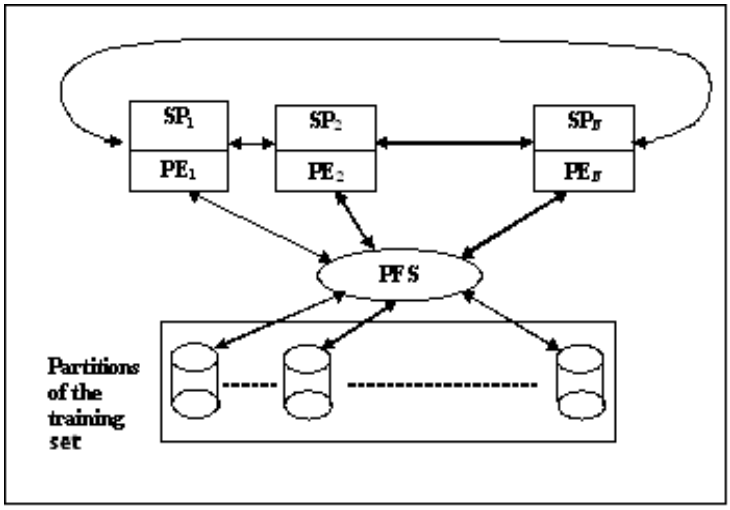


Figure 3: Software architecture of CGPC

erogeneous cluster of processors. PFS enables parallel applications to simultaneously access a set of files (or a single file) from any node that has the PFS file system mounted. The PFS solution allows to partition data being mined on the processors and guarantees to meet the needs of data-intensive applications in terms of scalability and performance. The parallelization scheme is based on a subdivision of the population in P subpopulations, each having the same number of individuals and a partitioning of the data being mined in P parts. The population is divided among the P processors by taking slices on the x-direction. The number of columns in the x-direction must be divisible by the number of the processing elements (PEs), in order to balance the computational load among the processors and ensuring that the size of each subpopulation (SP_i) be greater than a threshold determined from the granularity supported by the processor. On each PE, a slice process is allocated. It executes the CGPC algorithm by using its own subpopulation SP_i and the overall training set to compute the fitness. Each slice process accesses to the partitioned data through the PFS, that transfers the data set into the computer memory in blocks. In this way all the individuals of the subpopulations can operate on the training data more efficiently. Slice processes can be imagined as islands where each island is structured as a grid of individuals interacting locally. To take advantage of the cellular model of genetic programming, subpopulations are not independently evolved, but they exchange the individuals on the borders in an asynchronous way. Each slice process updates the individuals

belonging to its subpopulation sequentially and exchanges asynchronously the outermost individuals with the neighbors. In our implementation, the slice processes form a logical ring and each process determines its right- and left-processes.

Although CGPC allows the construction of accurate decision trees, the performance of the algorithm strongly depends on the size of the training set. Moreover, also the use of PFS introduces overheads and limits the performance and the scalability of the algorithm because of the necessity of each processing node to access data contained on the others nodes, through the PFS, to evaluate the fitness of the individuals. One approach to improve the performance of the model is to build an ensemble of classifiers, where each classifier works locally on a different subset of the training data set and then combines them together to classify the test set.

In the following we first present the most known ensemble approaches in the literature. Then we show how we extended CGPC to generate ensemble of classifiers by bagging and boosting techniques. According to this approach, the classifiers of each subpopulation are trained by using CGPC on a different subset of the overall data and, finally, combined together to classify new tuples by applying a majority voting scheme.

4 Ensemble techniques

Let $S = \{(x_i, y_i)|i = 1, \ldots, N\}$ be a training set where x_i, called example, is an attribute vector with m attributes and y_i is the class label associated with x_i. A predictor, given a new example, has the task to predict the class label for it. Ensemble techniques build T predictors, each on a different subset of the training set, then combine them together to classify the test set.

Bagging (bootstrap aggregating) was introduced by Breiman in [2] and it is based on bootstrap samples (replicates) of the same size of the training set S. Each bootstrap sample is created by uniformly sampling instances from S with replacement, thus some examples may appear more than once while others may not appear in it. T bags $B_1, \ldots, B_T$ are generated and T classifiers $C^1, \ldots, C^T$ are built on each bag B_i. The number T of predictors is an input parameter. A final classifier classifies an example by giving as output the class predicted most often by $C^1, \ldots, C^T$, with ties solved arbitrarily.

Boosting was introduced by Schapire [26] and Freund [27] for boosting the performance of any "weak" learning algorithm, i.e. an algorithm that "generates classifiers which need only be a little bit better than random guessing" [16]. The boosting algorithm, called *AdaBoost*, adaptively changes the distribution of the sample depending on how difficult each example is to classify. Given the number T of trials to execute, T weighted training set $S_1, S_2, \ldots, S_T$ are sequentially generated and T classifiers $C^1, \ldots, C^T$ are built to compute a weak hypothesis h_t. Let w_i^t denote the weight of example x_i at trial t. At the beginning $w_i^1 = 1/n$ for each x_i. At each trial $t = 1, \ldots, T$, a weak learner C^t, whose error ϵ^t is bounded to a value strictly less than $1/2$, is built and the weights of the next trial are obtained by multiplying the weight of the correctly classified examples by $\beta^t = \epsilon^t/(1-\epsilon^t)$ and renormalizing the weights so that $\Sigma_i w_i^{t+1} = 1$. Thus "easy" examples get a lower weight, while "hard" examples, that tend to be misclassified, get higher weights. This induces AdaBoost to focus on examples that are hardest to classify. The boosted classifier gives the class label y that maximizes the sum of the weights of the weak hypotheses predicting that label, where the weight is defined as $ln(1/\beta^t)$. Freund and Schapire [16] showed theoretically that AdaBoost can decrease the error of any weak learning algorithm and introduced two versions of the method, *AdaBoost.M1* and *AdaBoost.M2*. AdaBoost.M1, when the number of classes is two, requires that the prediction be just slightly better than random guessing. However, when the number of classes is more than 2, a more sophisticated error-measure, called pseudo-loss, is introduced. In this case the boosting algorithm can focus the weak learner not only on the hard-to-classify examples, but also on the the incorrect labels that are hardest to discriminate. In the next section we present the extension of *GP* by using AdaBoost.M2.

More complex techniques such as arching [3] adaptively change the distribution of the sample depending on how difficult each example is to classify. Bagging, boosting and variants have been studied and compared, and shown to be successful in improving the accuracy of predictors [8, 1]. These techniques, however, requires that the entire data sets be stored in main memory. When applied to large data sets this kind of approach could be impractical.

Breiman in [4] suggested that, when the data sets are too large to fit into main memory, a

possible approach is to partition the data in small pieces, build a predictor on each piece and then paste these predictors together. Breiman obtained classifiers of accuracy comparable if all the data set had been used. Similar results were found by Chan and Stolfo in [6]. In [7], Chawla et al. on a very large data set with a committee of eight classifiers trained on different partitions of the data attained accuracy higher than one classifier trained on the entire data set.

Regarding the application of ensemble techniques in Genetic Programming, Iba in [17] proposed to extend Genetic Programming to deal with bagging and boosting. A population is divided in a set of subpopulations and each subpopulation is evolved on a training set sampled with replacement from the original data. Best individuals of each subpopulation participate to voting to give a prediction on the testing data. Experiments on some standard problems using ten subpopulations showed the effectiveness of the approach.

Soule [28] demonstrated that evolving teams that cooperate by voting on the solution found by each separate member can improve the GP's performance on problems than normally do not require a cooperation approach to be solved, like the even-7-parity problem.

Langdon and Buxton [19] studied the combination of classifiers to produce one classifier which is better than each. The performance of a classifier is measured by computing the ROC curve and experiments on three benchmarks show that GP compared with different classifiers can automatically do better than these.

Cantu-Paz and Kamath [5] applied evolutionary algorithms to the induction of oblique decision trees and combined such trees to build ensembles of evolutionary trees. They found that oblique trees obtained with evolutionary techniques show better accuracy than those obtained by using traditional methods and that ensembles of oblique trees have better accuracy than a single tree. Furthermore, some of the ensembles created by using a sample of the data set instead of the overall one had higher accuracy than the single tree obtained by using the overall data set.

In the next two sections we present the parallel algorithms that implement the boosting and bagging techniques through cellular genetic programming.

Given $S = \{(x_1, y_1), \ldots (x_N, y_N)\}$, $x_i \in X$
with labels $y_i \in Y = \{1, 2, \ldots, k\}$, and a population Q of size q
Let $B = \{(i, y), i \in \{1, 2, \ldots, k\}, y \neq y_i\}$
For j = 1, 2, ..., P (for each processor in parallel)
Draw a sample S_j with size n for processor j
Initialize the weights $w^1_{i,y} = \frac{1}{|B|}$ for $i = 1, \ldots, n, y \in Y$,
where n is the number of training examples on each processor j.
Initialize the subpopulation Q_i, for $i = 1, \ldots, P$
with random individuals
end parallel for
For t = 1,2,3, ..., T
For j = 1, 2, ..., P (for each processor in parallel)
Train $CGPC$ on the sample S_j using a weighted
fitness according to the distribution w^t
Compute a weak hypothesis $h_{j,t} : X \times Y \rightarrow [0, 1]$
Exchange the hypotheses $h_{j,t}$ among the P processors
Compute the error $\epsilon^t_j = \frac{1}{2} \sum_{(i,y) in B} w^t_{i,y} \cdot (1 - h_{j,t}(x_i, y_i) + h_{j,t}(x_i, y))$
if $\epsilon^t_j \geq 1/2$ **break loop**
Set $\beta^t_j = \epsilon^t_j / (1 - \epsilon^t_j)$,
Update the weights $w^t : w^{t+1}_{i,y} = \frac{w^t_{i,y}}{Z_t} \cdot \beta^{(\frac{1}{2}) \cdot (1 + h_{j,t}(x_i, y_i) - h_{j,t}(x_i, y))}$
where Z_t is a normalization constant (chosen so that $w^t_{i,y}$ will be a distribution)
end parallel for
end for t
output the hypothesis :
$h_f = arg\ max\ (\sum^p_j \sum^T_t log(\frac{1}{\beta^t_j}) h_{j,t}(x, y))$

Figure 4: The algorithm parallel *BoostCGPC* version AdaBoost.M2

5 BoostCGPC

Boost Cellular Genetic Programming Classifier, is described in figure 4. Given the training set $S = \{(x_1, y_1), \ldots (x_N, y_N)\}$ and the number P of processors to use to run the algorithm, we partition the population of classifiers in P subpopulations, one for each processor and draw P sets of tuples of size $n < N$, by uniformly sampling instances from S with replacement. Each subpopulation is evolved for k generations and trained on its local sample by running $CGPC$.

After k generations, the individual with the best fitness is selected for participating to vote. In fact the P individuals of each subpopulation having the best fitness are exchanged among the P subpopulations and constitute the ensemble of predictors that will determine the weights

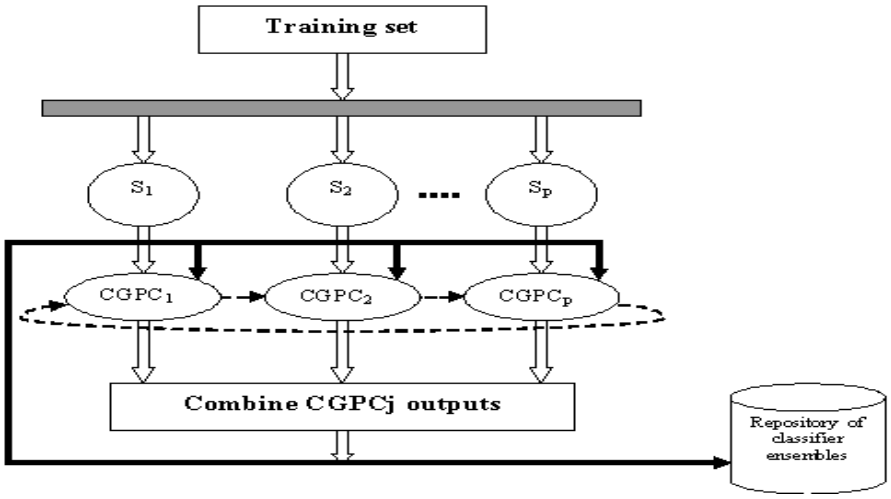


Figure 5: Implementation of *BoostCGPC* on a distributed memory parallel computer.

of the examples for the next round.

Figure 5 illustrates the basic framework for the parallel implementation of the *BoostCGPC* algorithm on a distributed memory parallel computer. We assume that each training sample $S_i, i = 1, \ldots, P$ resides on a different processor within the parallel computer. We use the diffusion model of GP to parallelize in a natural way the implementation of *BoostCGPC*. The size of each subpopulation $Q_i, i = 1, \ldots, P$ present on a node, must be greater than a threshold determined from the granularity supported by the processor. Each processor, using a training sample S_i and a subpopulation Q_i implements a classifier process $CGPC_i$ as a learning algorithm and generates a classifier.

During the boosting rounds, each classifier process maintains the local vector of the weights that directly reflect the prediction accuracy on that site. At every boosting round the hypotheses generated by each of these classifiers ($CGPC_i$ in Figure 5) are combined to produce the ensemble of predictors. Then, the ensemble is broadcasted to each classifier process to locally recalculate the new vector of the weights and a copy of the ensemble is stored in a repository. After the execution of the fixed number T of boosting rounds, the classifiers stored in the repository are used to evaluate the accuracy of the classification algorithm. Note that, the algorithm can also be used to classify distributed data which cannot be merged together, for example, in applications that deal with proprietary, privacy sensitive data, where it is not permitted moving raw data

Given $S = \{(x_1, y_1), \ldots (x_N, y_N)\}$, $x_i \in X$
with labels $y_i \in Y = \{1, 2, \ldots, k\}$, and a population Q of size q
For j = 1, 2, ..., P (for each processor in parallel)
Draw a sample S_j with size n for processor j
where n is the number of training examples on each processor j.
Initialize the subpopulation Q_i, for $i = 1, \ldots, P$
with random individuals
Train *CGPC* on the sample S_j
Compute a weak hypothesis $h_j : X \rightarrow Y$
Exchange the hypotheses h_j among the P processors
end parallel for
output the hypothesis :
$h_f = arg\ max\ (\sum_j^p D_j)$
where $D_j = 1$ if $h_j(x_i) = y_i$, 0 otherwise

Figure 6: The algorithm parallel *BagCGPC*

from different sites to a single central location for mining.

6 BagCGPC

Bag Cellular Genetic Programming Classifier, adopts the same parallelization strategy of *BoostCGPC* and it is described in figure 6. In such a case, given the training set $S = \{(x_1, y_1), \ldots (x_N, y_N)\}$ and the number P of processors to use, we partition the population in P subpopulations, one for each processor and we draw P samples from S of size $n < N$. Each subpopulation is evolved for k generations, trained on its local sample by running *CGPC* and generates a classifier working on a sample of the training data instead of using all the training set. The single classifier is always represented by the tree with the best fitness in the subpopulation. With P subpopulations we obtain P classifiers that constitute our ensemble. The output is the class predicted most often by the P classifiers.

Notice that our approach substantially differs from Iba's scheme [17] that extends genetic programming with bagging and boosting, since we use a parallel genetic programming model, we make cooperate the subpopulations to generate the classifiers and each subpopulation does not use the overall training set.

7 Experimental Results

In this section we compare *BagCGPC*, *BoostCGPC* and *CGPC* by using 8 data sets. Two of them (*Census* and *Covtype*) are from the UCI KDD Archive[1], four (*Pendigit*, *Segment*, *Satimage*, and *Adult*) are taken from the UCI Machine Learning Repository [2], one (*Phoneme*) is from the ELENA project [3], and one (*Mammography*) is a research data set used by [7] . The size and class distribution of these data sets are described in table 1. They present different characteristics in the number and type (continuous and nominal) of attributes, two classes versus multiple classes and number of tuples. In particular, *Cens* and *CovType* are real large data sets. The *Cens* data set contains weighted census data extracted from the 1994 and 1995 current population surveys conducted by the U.S. Census Bureau. The *CovType* data set comprises data representing the prediction of forest cover type from cartographic variables determined from US Forest Service and US Geological Survey. The *Pendigit* data set regards pen-based recognition of handwritten digits. The *Segment* data set contains image data described by high-level numeric-valued attributes. The *Satimage* data set is generated from Landsat Multi-Spectral Scanner image data. Each sample contains 36 pixel values and a number indicating one of the six class categories of the central pixel. The *Adult* data set describes data classified with respect to their income exceeding 50K/yr based on census data. The *Phoneme* data set contains data distinguishing between nasal and oral sounds. Finally, the *Mammography* data set contains information about mammography analysis and the classification attribute discriminates tuples with respect to the presence of calcifications in the tissues. This data set is very skewed since there are only 260 tuples among the 11183 having calcifications.

The experiments have been performed on a Linux cluster with 16 dual-processor 1,133 Ghz Pentium III nodes having 2 Gbytes of memory connected by Myrinet and running Red Hat v7.2.

The main objectives of the experiments have been the following:

- to investigate the influence of the training set size on the accuracy of the methods; to

[1] http://kdd.ics.uci.edu/
[2] http://www.ics.uci.edu/∼mlearn/MLRepository.html
[3] ftp.dice.ucl.ac.be in the directory pub/neural/ELENA/databases

Table 1: Data sets used in the experiments

Dataset	**Attr.**	**Tuples**	**Classes**
Adult	14	48842	2
Census	41	299285	2
Covtype	54	581012	7
Mammography	10	11183	2
Pendigits	16	10992	10
Phoneme	5	5404	2
Segment	19	2310	7
Satimage	36	6435	6

Table 2: Main parameters used in the experiments

Name	**Value**
max_depth_for_new_trees	6
max_depth_after_crossover	17
max_mutant_depth	2
grow_method	RAMPED
crossover_func_pt_fraction	0.7
crossover_any_pt_fraction	0.1
fitness_prop_repro_fraction	0.1
parsimony_factor	0

this end *BagCGPC*, *BoostCGPC* and *CGPC* were executed by using the 5%, 10%, 20%, 50%, and 100% of the training data, and an ensemble of 50 predictors.

- To study how the accuracy of *BagCGPC* and *BoostCGPC* varies when both the number of classifiers and the sample sizes are increased; the error rates obtained are compared with that produced by *CGPC* running on all the data set.

- To perform a speedup study when the size of the data set augments and the number of processors is fixed.

The parameters used for the experiments are shown in table 2. We used a replacement policy called *greedy* that replaces the current individual with the fittest of the two offspring only if the latter has a fitness value better that the former. All results were obtained by averaging 10-fold cross-validation runs, where at each run the 90% of the training set is used for training and the 10% remaining for testing. In order to do a fair comparison among *CGPC*, *BagCGPC*, and *BoostCGPC* we used 10 processors for all the three algorithms, population size 1000 and number of generations 500 for *CGPC*. To obtain the same parameters, *BagCGPC* was executed 5 times on ten processors in parallel, with population size 100 on each processor (for a total size of 100 $\times$ 10=1000) and number of generations 100 (for a total number of generations 100 $\times$ 5=500), thus generating 50 classifiers. On the other hand, the number T of rounds of *BoostCGPC* was 5, again on 10 processors, population size 100 on each processor, number of generations 100 for each round, thus generating the same total population size, number of generations, and number of classifiers, i.e. 500, 500, and 50, respectively.

In table 3 we report the mean error rate over the 10-fold-cross-validations, execution time in seconds, and average size of the classifiers using a training set of size 5%, 10%, 20%, 50%, and 100% of the overall training set. The values in bold of the columns *Error* for *BagCGPC* and *BoostCGPC* highlights the percentage of training set needed by these two algorithms to obtain an error lower than that obtained by *CGPC* using the overall data set. Thus, for example, for the *Adult* data set, *BagCGPC* has an error of 16.41 with only the 5% of the data set, *BoostCGPC* has an error of 16.56 with the 20% of the data set, while the error obtained by

Table 3: Error, execution time and tree length of *BagCGPC*, *BoostCGPC*, and *CGPC*.

		BagCGPC			*BoostCGPC*			*CGPC*		
		Error	Time	Length	Error	Time	Length	Error	Time	Length
Adult	5%	**16.41**	269.44	357.08	18.03	278.46	395.69	18.45	388.55	574.97
	10%	15.74	385.08	332.27	17.20	340.11	300.56	17.86	595.10	903.66
	20%	15.33	710.64	334.21	**16.56**	631.63	262.34	16.92	881.84	700.39
	50%	15.23	1257.11	310.09	16.06	1178.034	221.73	16.83	1635.47	780.25
	100%	15.13	2561.16	303.48	15.57	2066.30	214.93	16.75	3349.51	856.01
Census	5%	**5.13**	2231.67	3162.88	6.44	1378.46	1163.9	5.55	1890.27	2546.47
	10%	5.05	3482.59	3082.84	6.15	2273.12	1048.3	5.37	2782.82	2490.08
	20%	5.01	5272.48	3059.85	5.42	4001.08	1021.80	5.29	5009.13	2322.44
	50%	4.98	10971.92	3023.12	**5.21**	7695.10	949.5	5.24	9879.22	2225.28
	100%	4.97	19790.31	3005.86	5.21	14127.65	936.6	5.22	19010.27	2166.27
Covtyped	5%	**34.164**	2871.63	76.47	**33.983**	2854.7	79.13	37.023	3145.23	224.42
	10%	33.962	5432.73	78.18	33.379	5593.5	84.05	36.653	7147.48	196.03
	20%	33.402	10432.06	78.33	32.637	10821.3	85.50	36.322	13490.67	185.87
	50%	32.956	20614.39	77.01	32.526	21480.6	84.56	36.185	25584.08	180.69
	100%	32.872	42832.13	76.57	32.186	43859.5	83.00	35.922	51063.77	173.42
Mammography	5%	2.01	54.88	42.87	2.37	65.21	79.89	2.82	62.85	64.06
	10%	**1.75**	69.30	46.64	2.16	96.93	94.67	2.35	87.91	81.24
	20%	1.68	111.30	52.64	2.07	156.65	118.94	2.28	149.94	91.81
	50%	1.59	205.51	57.122	**1.93**	296.86	133.89	1.99	274.11	103.72
	100%	1.57	401.16	61.04	1.93	479.43	108.53	1.94	482.93	118.92
Pendigits	5%	**18.57**	343.46	704.44	**18.46**	355.15	800.62	40.07	358.53	887.30
	10%	18.33	398.98	725.90	17.88	407.32	820.91	39.04	450.53	891.22
	20%	17.61	503.12	780.38	17.30	407.33	590.70	37.83	543.13	865.99
	50%	17.10	787.03	750.57	16.97	624.88	428.21	36.08	875.00	887.00
	100%	16.98	1410.59	734.34	16.84	1138.34	386.29	33.26	1564.61	1049.52
Phoneme	5%	**18.23**	112.23	116.99	19.89	128.08	204.73	27.87	143.25	254.51
	10%	17.66	140.27	127.93	**19.124**	144.62	212.67	24.37	173.46	279.43
	20%	17.24	186.28	134.95	18.74	181.08	197.31	22.11	229.40	295.17
	50%	17.05	248.19	145.60	18.07	234.12	152.51	20.80	330.58	312.76
	100%	16.95	343.90	158.45	17.96	293.93	141.47	19.70	449.76	324.53
Satimage	5%	22.24	78.12	125.85	**19.86**	136.45	300.31	27.81	149.28	318.25
	10%	**21.00**	105.02	135.31	17.98	172.33	339.10	26.49	202.43	392.09
	20%	20.78	153.59	127.33	17.16	225.98	354.22	23.33	260.12	397.27
	50%	20.70	240.84	126.57	16.80	309.59	281.27	22.24	429.66	441.77
	100%	20.65	418.55	136.34	16.66	574.05	243.14	22.02	745.04	411.98
Segment	5%	16.26	57.97	104.45	13.95	94.70	164.98	25.46	86.98	167.99
	10%	13.93	62.42	93.07	**11.07**	116.23	207.53	19.25	112.81	211.67
	20%	**12.24**	78.86	88.98	9.14	131.09	213.61	14.39	126.36	199.24
	50%	11.81	129.34	91.90	8.57	181.51	227.67	13.38	243.17	297.75
	100%	11.54	209.26	90.64	8.47	227.51	235.82	12.36	317.58	276.73

CGPC with all the data set is 16.75. The same behavior can be observed for all the data sets. *BagCGPC* and *BoostCGPC* achieve a better accuracy with respect to *CGPC* with at most the 20% of the data set, except for *BoostCGPC* on the *Mammography* and *Census* data sets, that needs the 50% of the data to attain a lower error than *CGPC*. In any case *BagCGPC*

and *BoostCGPC* obtain always a lower error than *CGPC* that, in some cases, like Pendigit, is remarkable.

The table also points out that *BagCGPC* achieves better results of almost 10% on two-class data sets with respect to *BoostCGPC*, but this improvement smooths as soon as the data set size increases. On the contrary *BoostCGPC* works much better on multi-class data sets, and this gain on accuracy remains constant when the data set size increases. Furthermore, the execution time and the average length of the trees of *BoostCGPC* are almost always less than those of *BagCGPC*.

The next experiment aimed at determining the minimum number of classifiers and the minimum sample size necessary to obtain an error rate lower than *CGPC* running on the overall data set. Thus figures 7, 8, 9, 10, 11, 12, 13, 14 show how the error rate of *BagCGPC* and *BoostCGPC* diminishes when both the number of classifiers and the sample sizes are increased; the error rates obtained are compared with those produced by *CGPC* running on all the data set, but with an increasing population size.

To this end we run *CGPC* with the overall data set, while *BagCGPC* and *BoostCGPC* were executed with 5%, 10%, and 20% of the tuples for 5 rounds, each round using an increasing number of classifiers, from 1 to 20, implying thus ensemble constituted by 5, 10, 15, 20, ..., 100 classifiers. The parameters used are the same of the previous experiments. *CGPC* used a population size equal to 100 $\times$ *number of classifiers* of the ensemble.

Figures 7(a), 8(a), 11(a), and 12(a) show that *BagCGPC* on the data sets *Adult*, *Census*, *Pendigits* and *Phoneme* has an error lower than *CGPC* for any number of classifiers, even with a sample of only the 5%. Figures 9(a), 10(a), 13(a), and 14(a), show that *BagCGPC* begins to overcome *CGPC*, on the *Covtype* data set, when 2 classifiers per round (that is an ensemble of 10 classifiers) and a sample of 20% are used; on the *Mammography* data set, when 3 classifiers per round (that is an ensemble of 15 classifiers) and a sample of 10% are used, or when the ensemble contains 10 classifiers (2 classifiers per round) and the size of the sample is 10%; on the *Satimage* data set when 2 classifiers per round (that is an ensemble of 10 classifiers) and a sample of 5% are used; and, finally, on the *Segment* data set when 10 classifiers per round (that

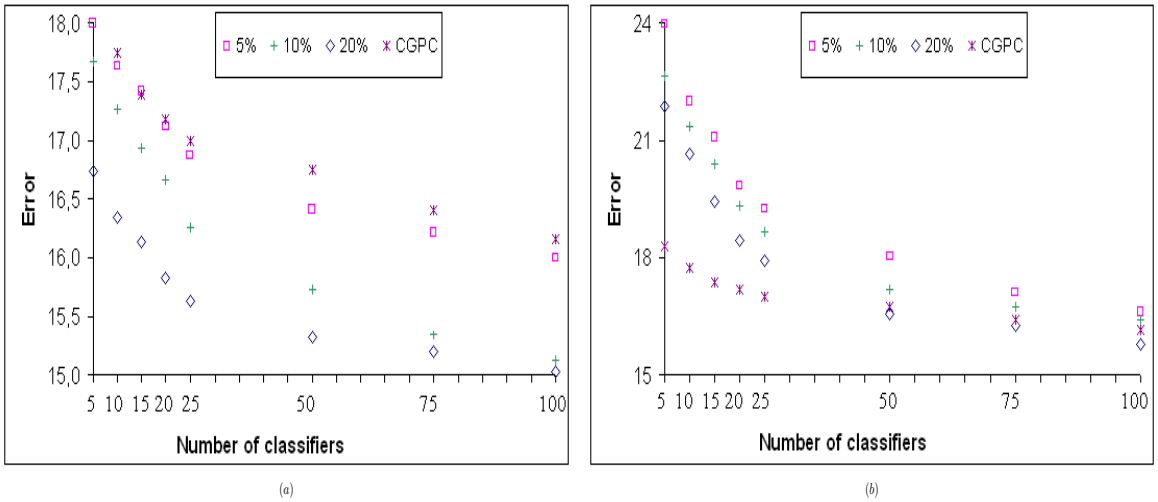


Figure 7: Mean error for different sample sizes of the training set vs number of classifiers for round (Adult dataset). (a) BagCGPC (b) BoostCGPC.

is an ensemble of 50 classifiers) and a sample of 20% are used.

The behavior of *BoostCGPC*, however, is different. On the two data sets *Census* and *Mammography*, it is not able to beat *CGPC* with these sizes of the training set, as figures 8(b) and 10(b) point out. In fact, table 3 shows that *BoostCGPC* needs the 50% of the data in order to overcome *CGPC* on these two data sets. As regard the others, Figure 7(b) shows that for the *Adult* data set *BoostCGPC* needs 50 classifiers and 20% of the data set, Figure 9(b) shows that for the *Covtype* data set it needs 10 classifiers and 10% of the data set, Figure 11(b) shows that *BoostCGPC* is always better than *CGPC* on *Pendigits*, Figure 12(b) shows that *BoostCGPC* on *Phoneme* needs at least the 10% of the data and 10×5 classifiers to be better than *CGPC*, Figure 13(b) shows that for the *Satimage* data set it needs 10 classifiers and 10% of the data set, finally, Figure 14(b) shows that *BoostCGPC* on *Segment* needs at least the 20% of the data and 2×5 classifiers. These experiments ulteriorly emphasize that the ensemble techniques can obtain better accuracy than their base classifiers even with a moderate number of predictors and a small training set.

Finally figure 15 shows for the Covtype data set how the CGPC, BagCGPC, and BoostCGPC behave when running on 20 processors an increasing number of tuples, that is 5%, 20%, 50%, and 100%. The figure points out that the speedup obtained is nearly linear for all the three methods,

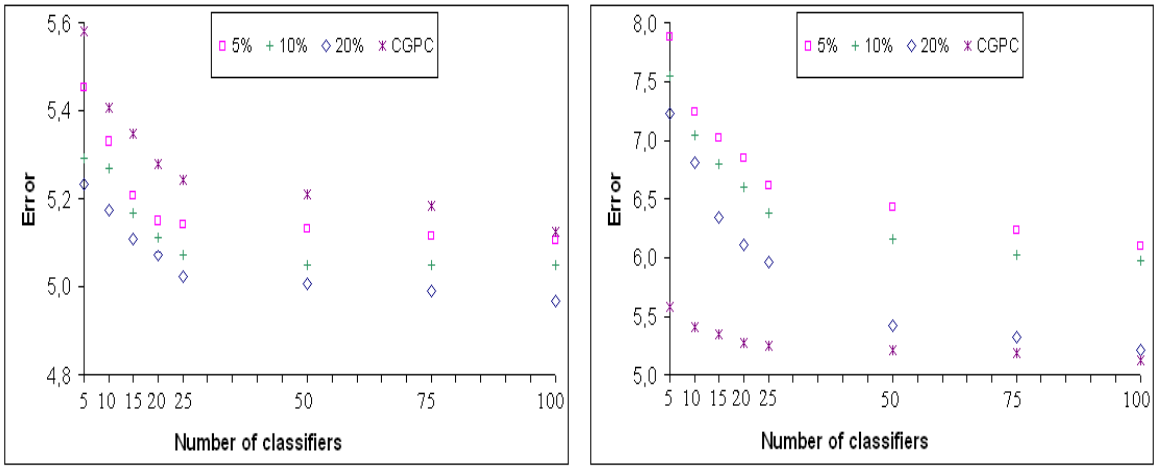


Figure 8: Mean error for different sample sizes of the training set vs number of classifiers for round (Census dataset). (a) BagCGPC (b) BoostCGPC.

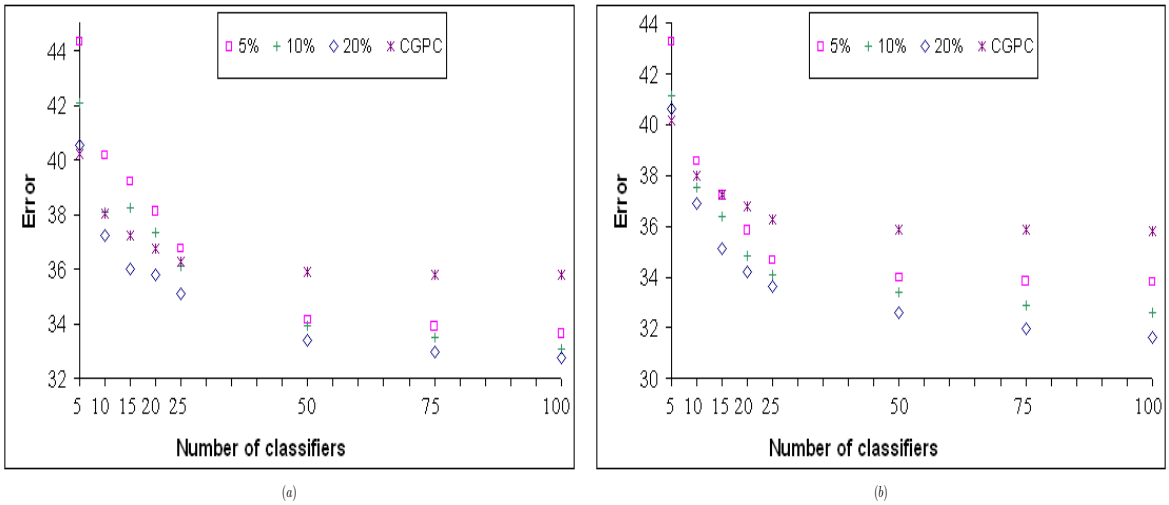
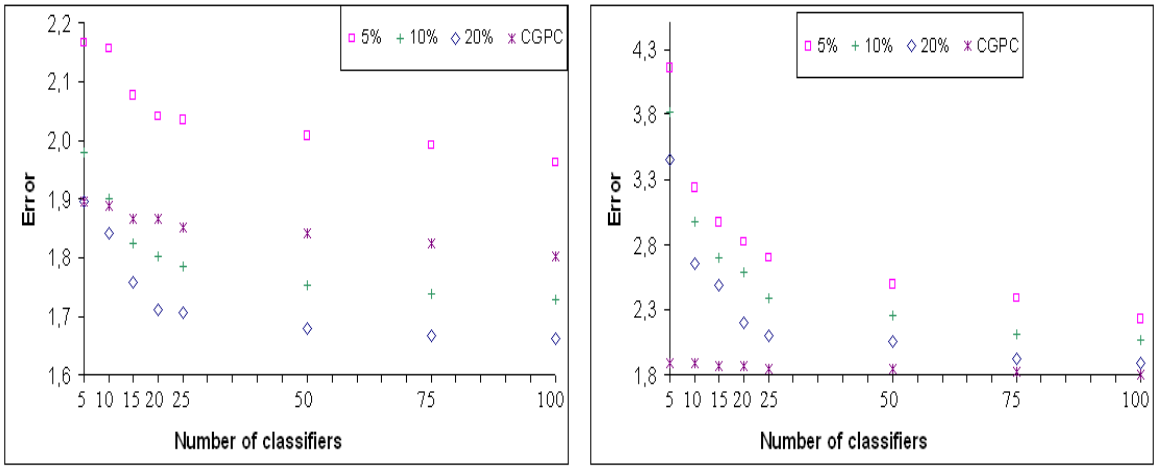
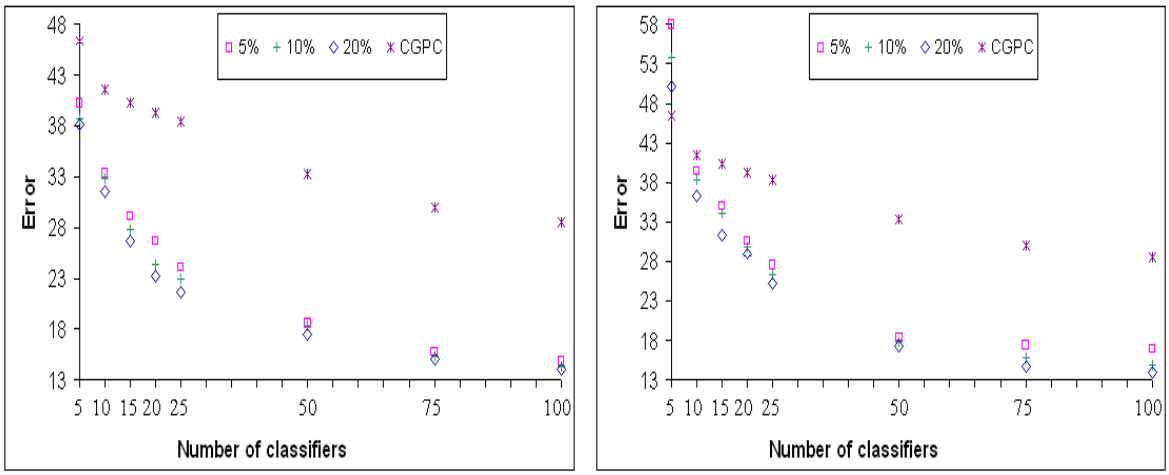


Figure 9: Mean error for different sample sizes of the training set vs number of classifiers for round (Covtype dataset). (a) BagCGPC (b) BoostCGPC.



(a) (b)

Figure 10: Mean error for different sample sizes of the training set vs number of classifiers for round (Mammography dataset). (a) BagCGPC (b) BoostCGPC.



(a) (b)

Figure 11: Mean error for different sample sizes of the training set vs number of classifiers for round (Pendigits dataset). (a) BagCGPC (b) BoostCGPC.

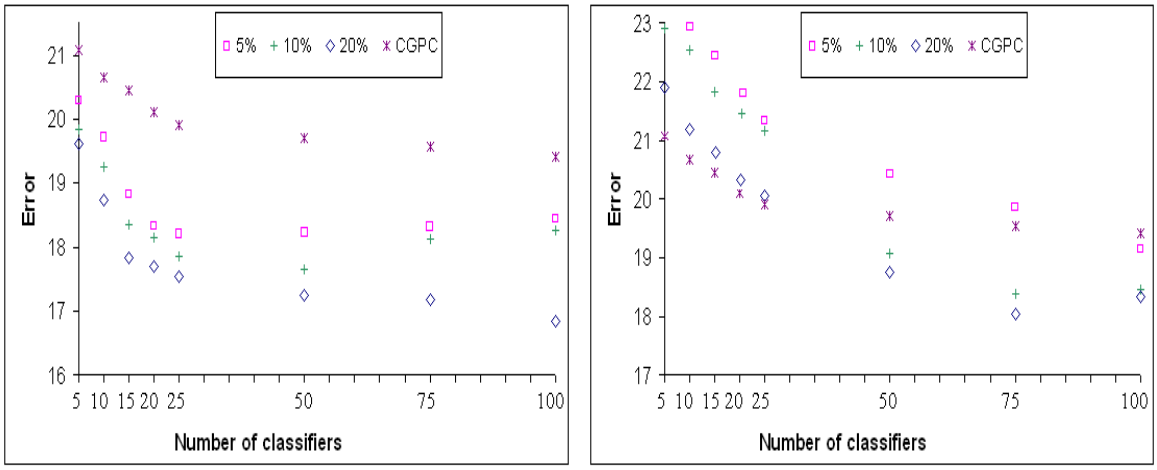


Figure 12: Mean error for different sample sizes of the training set vs number of classifiers for round (Phoneme dataset). (a) BagCGPC (b) BoostCGPC.

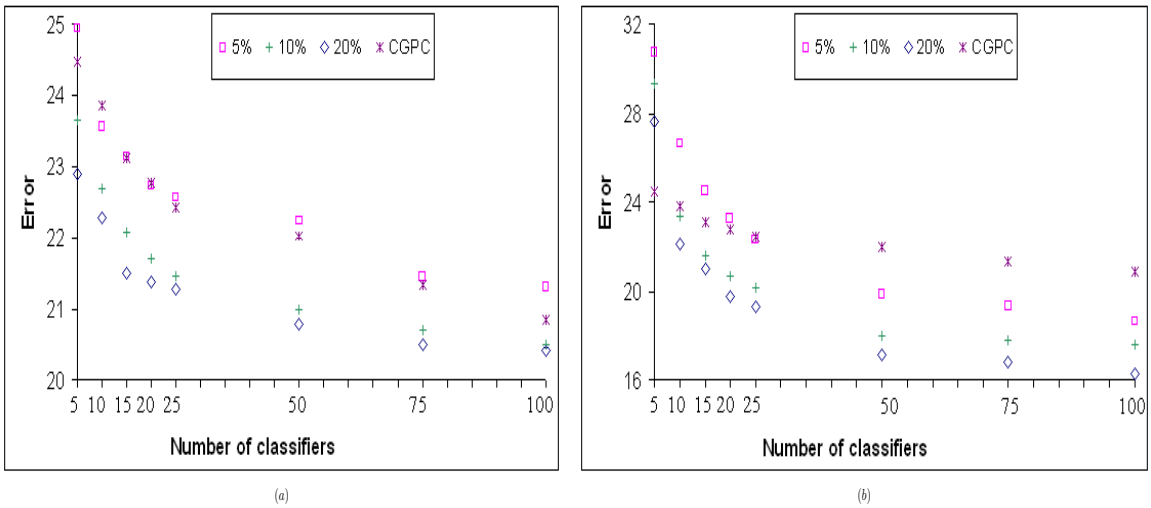


Figure 13: Mean error for different sample sizes of the training set vs number of classifiers for round (Satimage dataset). (a) BagCGPC (b) BoostCGPC.

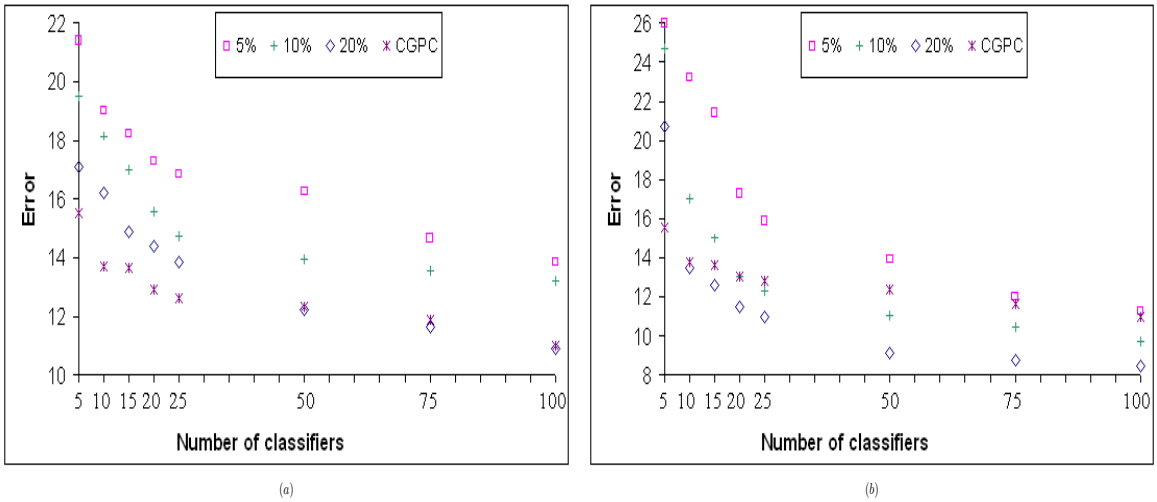


Figure 14: Mean error for different sample sizes of the training set vs number of classifiers for round (Phoneme dataset). (a) BagCGPC (b) BoostCGPC.

though that of CGPC is slightly better than that of BagCGPC and BoostCGPC. This is mainly due to the greater communication among the processors to exchange information in the latter two algorithms. However, it is worth noting that the execution time of BagCGPC and BoostCGPC is always lower than that of CGPC. For example, if we use the 20% of the tuples, CGPC needs 13490.67 seconds, while BagCGPC requires 10432.06 seconds and BoostCGPC 10821.3 seconds. The lower computation time is a direct consequence of the size of the trees generated by BagCGPC and BoostCGPC with respect to that of the trees generated by CGPC. In fact in the former case these sizes are much smaller, thus the application of ensemble techniques in Genetic Programming, as already observed by Iba, has the positive result of controlling the bloating problem, common in GP.

8 Conclusions

An extension of Cellular Genetic Programming for data classification to induce an ensemble of predictors that uses voting classification schemes based on bagging and boosting techniques has been presented. Experiments showed that the extension of *CGPC* with these voting algorithms reduces the size of the trees, enhances both accuracy and execution time and that higher accuracy can be obtained by using a small subset of the training set at a much lower computational cost.

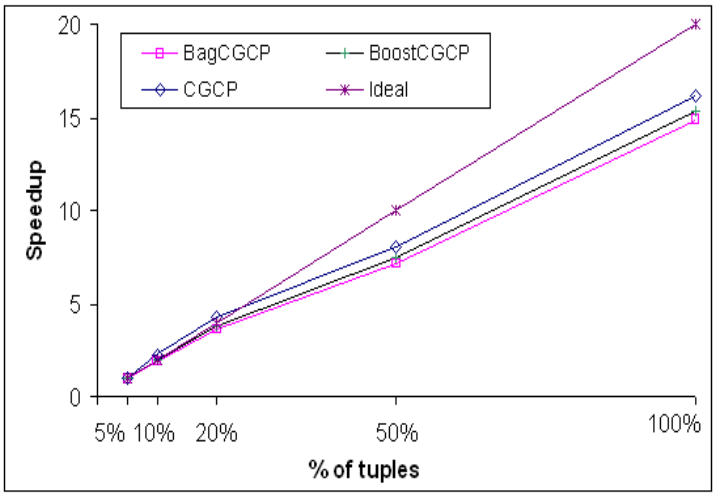


Figure 15: Speedup of CGPC, BagCGPC, and BoostCGPC

The approach is thus able to deal with large data sets that do not fit in main memory since each classifier is trained on a subset of the overall training data. The algorithm could also be used to classify distributed data which cannot be merged together. For example, in applications that deal with proprietary, privacy sensitive data, where it is not permitted moving raw data from different sites to a single central location for mining.

References

[1] Eric Bauer and Ron Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, (36):105–139, 1999.

[2] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[3] Leo Breiman. Arcing classifiers. *Annals of Statistics*, 26:801–824, 1998.

[4] Leo Breiman. Pasting small votes for classification in large databases and on-line. *Machine Learning*, 36(1,2):85–103, 1999.

[5] E. Cantú-Paz and C. Kamath. Inducing oblique decision trees with evolutionary algorithms. *IEEE Transaction on Evolutionary Computation*, 7(1):54–68, February 2003.

[6] P. K. Chan and S.J. Stolfo. A comparative evaluation of voting and meta-learning on partitioned data. In *International Conference on Machine Learning ICML95*, pages 90–98, 1995.

[7] N. Chawla, T.E. Moore, W. Bowyer K, L.O. Hall, C. Springer, and P. Kegelmeyer. Bagging-like effects for decision trees and neural nets in protein secondary structure prediction. In *BIOKDD01: Workshop on Data mining in Bioinformatics (SIGKDD01)*, 2001.

[8] Thomas G. Dietterich. An experimental comparison of three methods for costructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine Learning*, (40):139–157, 2000.

[9] J. Eggermont, J. N. Kok, and W.A. Kosters. Genetic programming for data classification: Partitioning the search space. In *Proceedings of ACM Symposium on Applied Computing, SAC'04*, pages 1001–1005. ACM Press, 2004.

[10] U.M. Fayyad, G. Piatesky-Shapiro, and P. Smith. From data mining to knowledge discovery: an overview. In U.M. Fayyad & al. (Eds), editor, *Advances in Knowledge Discovery and Data Mining*, pages 1–34. AAAI/MIT Press, 1996.

[11] G. Folino, C. Pizzuti, and G. Spezzano. A cellular genetic programming approach to classification. In *Proc. Of the Genetic and Evolutionary Computation Conference GECCO99*, pages 1015–1020, Orlando, Florida, July 1999. Morgan Kaufmann.

[12] G. Folino, C. Pizzuti, and G. Spezzano. Ensemble techniques for parallel genetic programming based classifiers. In E. Costa C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, editor, *Proceedings of the Sixth European Conference on Genetic Programming (EuroGP-2003)*, volume 2610 of *LNCS*, pages 59–69, Essex, UK, 2003. Springer Verlag.

[13] G. Folino, C. Pizzuti, and G. Spezzano. A scalable cellular implementation of parallel genetic programming. *IEEE Transaction on Evolutionary Computation*, 7(1):37–53, February 2003.

[14] G. Folino, C. Pizzuti, and G. Spezzano. Boosting technique for combining cellular gp classifiers. In M. Keijzer, U. O'Reilly, S.M: Lucas, E. Costa, and T. Soule, editors, *Proceedings of the Seventh European Conference on Genetic Programming (EuroGP-2004)*, volume 3003 of *LNCS*, pages 47–56, Coimbra, Portugal, 2004. Springer Verlag.

[15] A.A. Freitas. A genetic programming framework for two data mining tasks: Classification and generalised rule induction. In *Proceedings of the 2nd Int. Conference on Genetic Programming*, pages 96–101. Stanford University, CA, USA, 1997.

[16] Y. Freund and R. Scapire. Experiments with a new boosting algorithm. In *Proceedings of the 13th Int. Conference on Machine Learning*, pages 148–156, 1996.

[17] Hitoshi Iba. Bagging, boosting, and bloating in genetic programming. In *Proc. Of the Genetic and Evolutionary Computation Conference GECCO99*, pages 1053–1060, Orlando, Florida, July 1999. Morgan Kaufmann.

[18] J. R. Koza. *Genetic Programming: On the Programming of Computers by means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[19] W.B. Langdon and B.F. Buxton. Genetic programming for combining classifiers. In *Proc. Of the Genetic and Evolutionary Computation Conference GECCO'2001*, pages 66–73, San Francisco, CA, July 2001. Morgan Kaufmann.

[20] R.E. Marmelstein and G.B. Lamont. Pattern classification using a hybbrid genetic program - decision tree approach. In *Proceedings of the Third Annual Conference on Genetic Programming*, Morgan Kaufmann, 1998.

[21] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Internaltional Edition, 1997.

[22] N.I. Nikolaev and V. Slavov. Inductive genetic programming with decision trees. In *Proceedings of the 9th International Conference on Machine Learning*, Prague, Czech Republic, 1997.

[23] J. Ross Quinlan. *C4.5 Programs for Machine Learning*. Morgan Kaufmann, San Mateo, Calif., 1993.

[24] J. Ross Quinlan. Bagging, boosting, and c4.5. In *Proceedings of the 13th National Conference on Artificial Intelligence AAAI96*, pages 725–730. Mit Press, 1996.

[25] M.D. Ryan and V.J. Rayward-Smith. The evolution of decision trees. In *Proceedings of the Third Annual Conference on Genetic Programming*, Morgan Kaufmann, 1998.

[26] R. E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, 1990.

[27] R. E. Schapire. Boosting a weak learning by maiority. *Information and Computation*, 121(2):256–285, 1996.

[28] Terence Soule. Voting teams: A cooperative approach to non-typical problems using genetic programming. In *Proc. Of the Genetic and Evolutionary Computation Conference GECCO99*, pages 916–922, Orlando, Florida, July 1999. Morgan Kaufmann.

[29] M. Tomassini. Parallel and distributed evolutionary algorithms: A review. In P. Neittaanmki K. Miettinen, M. Mkel and J. Periaux, editors, *Evolutionary Algorithms in Engineering and Computer Science*, J. Wiley and Sons, Chichester, 1999.