Data Min Knowl Disc (2009) 18:392–418
DOI 10.1007/s10618-008-0121-2

A divide-and-conquer recursive approach for scaling up instance selection algorithms

Aida de Haro-García · Nicolás García-Pedrajas

Received: 9 June 2008 / Accepted: 10 November 2008 / Published online: 12 December 2008

Abstract Instance selection is becoming more and more relevant due to the huge amount of data that is being constantly produced. However, although current algorithms are useful for fairly large datasets, scaling problems are found when the number of instances is of hundreds of thousands or millions. In the best case, these algorithms are of efficiency $O(n^2)$, n being the number of instances. When we face huge problems, scalability is an issue, and most algorithms are not applicable. This paper presents a divide-and-conquer recursive approach to the problem of instance selection for instance based learning for very large problems. Our method divides the original training set into small subsets where the instance selection algorithm is applied. Then the selected instances are rejoined in a new training set and the same procedure, partitioning and application of an instance selection algorithm, is repeated. In this way, our approach is based on the philosophy of divide-and-conquer applied in a recursive manner. The proposed method is able to match, and even improve, for the case of storage reduction, the results of well-known standard algorithms with a very significant reduction of execution time. An extensive comparison in 30 datasets form the UCI Machine Learning Repository shows the usefulness of our method. Additionally, the method is applied to 5 huge datasets with from 300,000 to more than a million instances, with very good results and fast execution time.

Responsible editor: Eamonn Keogh.

A. de Haro-García · N. García-Pedrajas (✉)
Department of Computing and Numerical Analysis, University of Córdoba, 14071 Córdoba, Spain
e-mail: npedrajas@uco.es

A. de Haro-García
e-mail: i22hagaa@uco.es

Keywords Instance selection · Instance based learning · Divide-and-conquer · Scalability

1 Introduction

The overwhelming amount of data that is available nowadays in any field of research poses new problems for data mining and knowledge discovery methods. This huge amount of data makes most of the existing algorithms inapplicable to many real-world problems. Two approaches have been used to face this problem: scaling up data mining algorithms (Provost and Kolluri 1999) and data reduction. Nevertheless, scaling up a certain algorithm is not always feasible. Data reduction consists of removing missing, redundant, and/or erroneous instances to get a tractable amount of data. Data reduction techniques use different approaches: feature selection (Liu and Motoda 1998), feature-value discretization (Hussain et al. 1999), and instance selection (Blum and Langley 1997). This paper deals with instance selection for instance based learning.

Instance selection (Liu and Motoda 2002) consists of choosing a subset of the total available data to achieve the original purpose of the data mining application as if the whole data were used. Different variants of instance selection exist. Many of the approaches are based on some form of sampling (Cochran 1977; Kivinen and Mannila 1994). There are other more modern methods that are based on different principles, such as, Modified Selective Subset (MSS) (Barandela et al. 2005), entropy-based instance selection (Son and Kim 2006), Intelligent Multiobjective Evolutionary Algorithm (IMOEA) (Chen et al. 2005), and LVQPRU method (Li et al. 2005).

Our aim is focused on instance selection for instance-based learning. We can distinguish two main models (Cano et al. 2003): instance selection as a method for prototype selection for algorithms based on prototypes (such as k-Nearest Neighbors) and instance selection for obtaining the training set for a learning algorithm that uses this training set (such as classification trees or neural networks).

The problem of instance selection for instance based learning can be defined as (Brighton and Mellish 2002) "the isolation of the smallest set of instances that enable us to predict the class of a query instance with the same (or higher) accuracy than the original set". It has been shown that different groups of learning algorithms need different instance selectors in order to suit their learning/search bias (Brodley 1995). This may render many instance selection algorithm useless, if their philosophy of design is not suitable to the problem at hand.

In the best case, existing instance selection algorithms are of efficiency $O(n^2)$, n being the number of instances. For huge problems, with hundreds of thousands or even millions of instances, these methods are not applicable. Trying to develop algorithms with a lower efficiency order is likely to be a fruitless search. Obtaining the nearest neighbor of a given instance is $O(n)$. To test whether removing an instance affects the accuracy of the nearest neighbor rule, we must measure the effect on the other instances of the absence of the removed one. Measuring this effect involves recalculating, directly or indirectly, the nearest neighbors of the instances. The result is a process of $O(n^2)$. In this way, the attempt to develop algorithms of an efficiency order below $O(n^2)$ is not very promising.

Thus, the alternative is reducing the size n of the set to which instance selection algorithms are applied. In the construction of ensembles of classifiers the problem of learning from huge datasets has been approached by means of learning many classifiers from small disjoint subsets (Chawla et al. 2004). In that paper, the authors showed that it is also possible to learn an ensemble of classifiers from random disjoint partitions of a dataset, and combine predictions from all those classifiers to achieve high classification accuracies. They applied their method to huge datasets with very good results. Furthermore, the usefulness of applying instance selection to disjoint subsets has also been shown in García-Pedrajas et al. (2008). In that work, a cooperative evolutionary algorithm was used. The training set was divided into several disjoint subsets and an evolutionary algorithm was performed on each subset of instances. The fitness of the individuals was evaluated only taking into account the instances in the subset. To account for the global view needed by the algorithm a global population was used. This method is scalable to medium/large problems but cannot be applied to huge problems. Zhu and Wu (2006) also used disjoint subsets in a method for ranking representative instances.

Following that philosophy, we can develop a methodology based on applying the instance selection algorithm to subsets of the whole training set. A simple approach consists of using a stratified random sampling (Liu and Motoda 2002; Cano et al. 2005), where the original dataset is divided into many disjoint subsets, and then apply instance selection over each subset independently. However, due to the fact that to select the nearest neighbor of an instance we need to know the whole dataset, this method is not likely to produce good results. In fact, in practice its performance is poor. However, the divide-and-conquer principle of this method is an interesting idea for scaling up instance selection algorithms. Furthermore, divide-and-conquer methodology has the additional advantage that we can adapt the size of the subproblems to the available resources.

One way to improve that method is, instead of using a random partition of the dataset, to construct the subsets considering adjacent regions. In this way, the locality of nearest neighbor rule may work in our favor. The application of this idea produces good results in terms of testing error, but storage reduction is still too small.[1] A way of improving reduction storage is applying the above idea in a recursive manner, after the first application of the instance selection algorithm the subsets of selected instances are rejoined and the method is repeated. This is the methodology proposed in this paper.

Thus our algorithm first step is the partition of the whole dataset into small disjoint subsets. Then, the instance selection algorithm is performed over each subset independently. Then the selected instances are rejoined in a new dataset and the procedure is repeated. This recursive instance selection proceeds until a certain reduction is achieved or any other stop criterion is met.

For an $O(n^2)$ algorithm, if we divide the dataset into n' subsets of size s, $n' = n/s$, we must apply the algorithm n' times, with a complexity proportional to $n' \times (s^2)$. The gaining in execution time would be greater as the size of the original dataset is larger. If the complexity of the instance selection algorithm is greater, the reduction

[1] Experiments with this method, not shown in this paper, obtained testing error results better than standard instance selection algorithms but with a very limited storage reduction.

of the execution time will be even better. In fact, for a given subset size, the complexity of our method is the complexity of applying $n' = n/s$ times a given instance selection algorithm, which is linear in the number of instances. The method has the additional advantage of allowing an easy parallel implementation. As the application of the instance selection algorithm to each subset is independent from all the other subsets, the subsets can be processed at the same time. Besides, the communication between the nodes of the parallel execution is small. Only the selection performed by each node must the transmitted.

This paper is organized as follows: Sect. 2 presents the proposed method for instance selection based on our recursive procedure; Sect. 3 states the experimental setup; Sect. 4 shows the results of the experiments; and Sect. 5 states the conclusions of our work and future research lines.

2 Recursive method

Our method is applicable to any instance selection algorithm, as the instance selection algorithm is a parameter of the method. First, our method divides the whole training set, T, into disjoint subsets, t_i, of size s such as $T = \bigcup_i t_i$. s is the only parameter of the algorithm. The way the dataset is divided is relevant, and it is explained in Sect. 2.1. Then, the instance selection algorithm of our choice is performed over every subset independently. The selected instances in each subset are joined again. With this new training set constructed with the selected instances, the process is repeated until a certain stop criterion is fulfilled. The process of combining the instances selected by the execution of the instance selection algorithm over each dataset can be performed in different ways. We can just repeat the partition process as in the original dataset. However, as the first partition is performed using spatial properties of the instances (see Sect. 2.1) we can take advantage of this performed task. In this way, instead of repeating the partitioning process, we join together the subsets of selected instances until new subsets of approximately size s are obtained. An example of the whole algorithm is shown in Fig. 1, and the detailed process is shown in Algorithm 1.

Stop criterion The stop criterion is of importance for the method. If the recursive process is repeated too many times, the reduction is too large and the testing error very poor. Fixing a number of iterations for every dataset is difficult, as it depends on the specific features of each problem. Thus, we use a cross-validation approach. We divide the training set into two parts, using one of them for performing the instance selection algorithm and the other one for obtaining the validation error. The number of iterations is obtained as the last iteration before the validation error starts to grow. Then, we perform the algorithm using the whole training set for the number of iterations obtained in the cross-validation process.

The most important advantage of our method is the large reduction in the execution time. The experiments show a large difference when using standard widely used instance selection algorithms. Additionally, the method is easy to implement in a parallel environment, as the execution of the instance selection algorithm over each subset is performed independently. As shown in Fig. 6, the behavior of our method is almost linear in the number of instances.

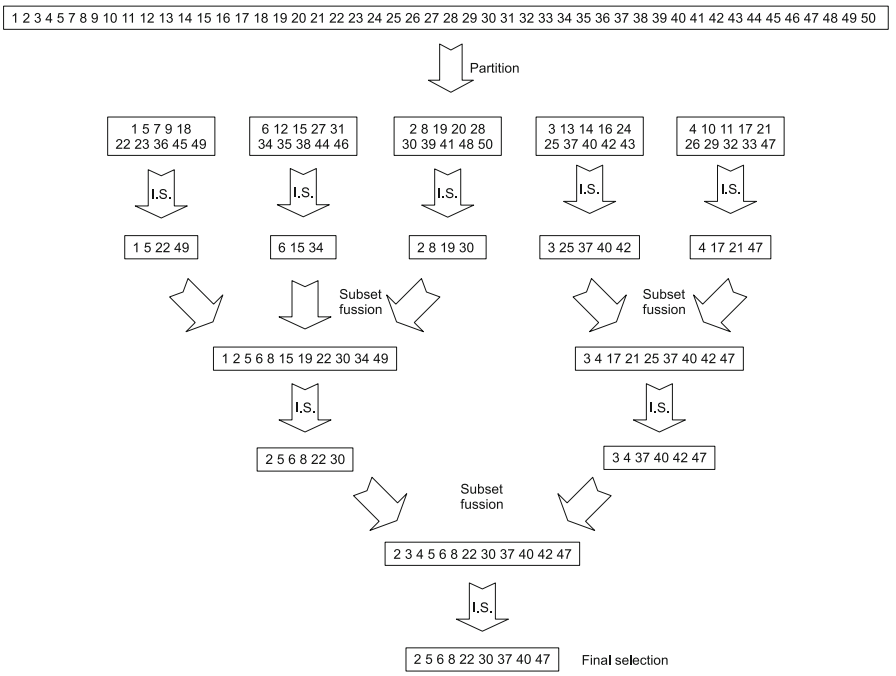

Fig. 1 Example of the method for a dataset of 50 instances and subsets of 10 instances. The instance selection (I.S.) algorithm can be any one of the many available methods

Algorithm 1: Recursive instance selection algorithm

Data : A training set $T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, and subset size s.
Result : The reduced selector $S \subset T$.
1 $S = T$
2 Partitionate instances into disjoint subsets $t_i : \bigcup_i t_i = S$ of size s
repeat
  foreach *subset* $t_i \subset S$ **do**
3 Apply instance selection algorithm in t_i to obtain selected subset $s_i \subset t_i$
4 Remove from S instances removed from t_i
  end
5 Fussion subsets s_i to obtain new subsets t_j of size s
until *until stop criterion*
6 Return S

There is a last useful feature of the proposed approach. As the problem size grows the requirements for memory use also grow. For some problems it is not feasible to keep all data in memory due to the need to use the disk for memory swapping. However, existing methods need all the training set in memory during the execution of the algorithm. Our method requires each subset to be in memory only while it is processed, but it is not needed during the processing of the remaining subsets. In this way, our method is scalable both in time and storage requirements.

2.1 Partition of the dataset

The first step of our method is to partition the training set into a number of disjoint subsets, t_i, which comprise the whole training set, $\bigcup_i t_i = T$. The size of the subsets is fixed by the user. The actual size has no relevant influence over the results, provided it is small to avoid large execution time. Furthermore, the time spent by the algorithm highly depends on the size of the larger subset, so it is important that the partition algorithm would produce subsets of approximately equal size.

The most simple method is just a random partition, where each instance is randomly assigned to one of the subsets. However, k-NN is a local learning algorithm, so a partition that produces, at least partially, subsets of instances near to each other is likely to produce better results. A simple method is dividing the input space into regions of equal size, and using the instances within each region as subsets. However, this method does not produce subsets of equal size. Another procedure based on the same idea can be constructed that produces subsets of equal size.

This procedure selects a random input, without replacement, and divides the set into two halves using the median of the values of the input, thus assuring that the two subsets are of the same size. The number of subsets must be a power of two. As obtaining the median is of $O(N)$, the partition can be made efficiently. The process is repeated using a new selected random variable until the desired number of subsets is produced.

The partition performed in a training set of 2000 instances is depicted in Fig. 2b. The dataset, which is made up of three classes is shown in Fig. 2a. The figure shows how the partition is local, in the sense that the instances in each subset correspond to adjacent instances in the space. This partition is performed only in the first iteration of the algorithm. In the subsequent steps (see Algorithm 1 step 1) the subsets are joined into new subsets. The figure also shows a problem with the proposed method, as the partition is performed without using class labels, some subsets have only instances of one class. This fact happens in the real-world datasets of our experiments. When an instance selection algorithm receives a subset with instances of only one class its performance is poor, as removing instances has no effect on the error of the nearest neighbor classifier.

To avoid this effect we perform the partition taking class labels into account. The training set is divided into subsets that have approximately the same distribution of classes of the original set. This is accomplished applying the described algorithm to each class separately. Algorithm 2 shows the complete procedure for performing the partition. The result of such a partition, which is used in all the reported experiments, is shown in Fig. 2c.

Partitioning in a similar way, using just one variable, has been used before in for learning ensembles of classifiers (Banfield et al. 2005).

2.2 Evaluating instance selection algorithms

The evaluation of a certain instance selection algorithm is not a trivial task. We can distinguish two basic approaches: direct and indirect evaluation (Liu and Motoda 2002).

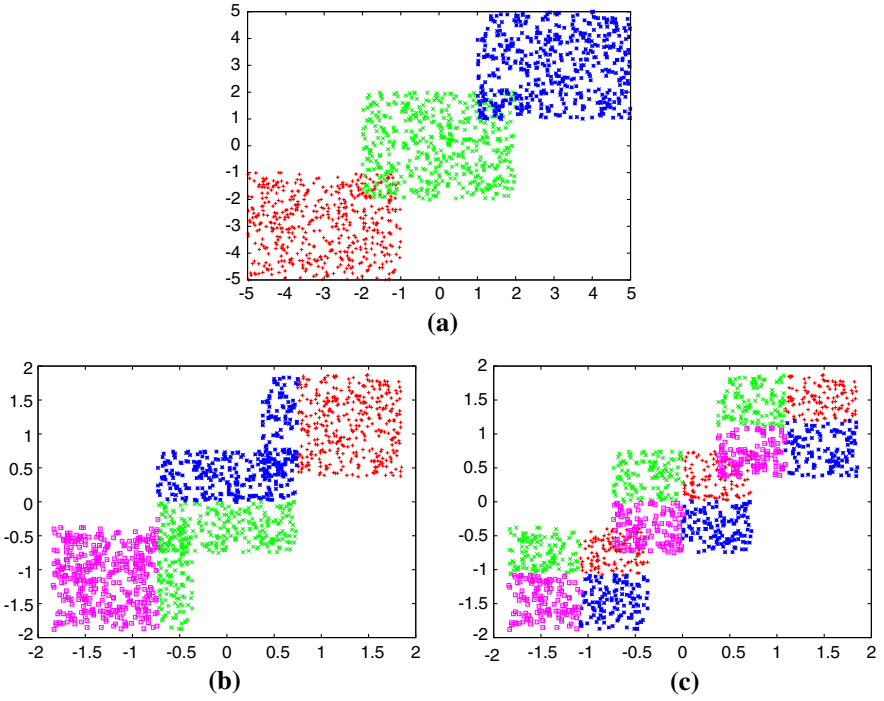

Fig. 2 Example of the method for partitioning a dataset with 2000 instances, three classes and two features. **a** Original training set with three classes. **b** Partition ignoring class labels into four subsets. **c** Partition using class labels into four subsets

Algorithm 2: Algorithm for partitioning the training sets into disjoint subsets

Data : A training set $T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, and subset size s.
Result : The partition into disjoint subsets $t_i : \bigcup_i t_i = T$.
Adjust the number of subsets n_s to a power of 2 using size s
foreach *Class* **do**
  for $i = 0$ *to* $\log_2(n_s)$ **do**
    Select an input j randomly without replacement
    for $k = 0$ *to* 2^i **do**
      Divide every subset into two halves using median of input j within the subset
    end
  end
end
Return $t_i : \bigcup_i t_i = T$

Direct evaluation evaluates a certain algorithm based exclusively on the data. The objective is to measure at which extent the selected instances reflect the information present in the original data. Some proposed measures are entropy (Cover and Thomas 1991), moments (Smith 1998), and histograms (Chaudhuri et al. 1998).

Indirect methods evaluate the effect of the instance selection algorithm on the task at hand. So, if we are interested in classification we evaluate the performance of the used classifier when using the reduced set obtained after instance selection as learning set.

Therefore, when evaluating instance selection algorithms for instance learning, the most usual way of evaluation is estimating the performance of the algorithms on a set of benchmark problems. In those problems several criteria can be considered, such as (Wilson and Martinez 2000): storage reduction, generalization accuracy, noise tolerance, and learning speed. Speed considerations are difficult to measure, as we are evaluating not only an algorithm but also a certain implementation. However, as the main aim of our work is scaling up instance selection algorithms, execution time is a basic issue. To allow a fair comparison, we have performed all the experiments in the same machine, a bi-processor computer with two Intel Xeon QuadCore at 1.60 GHz.

3 Experimental setup

In order to make a comprehensive comparison between the standard algorithms and our proposal we have selected a set of 30 problems from the UCI Machine Learning Repository (Hettich et al. 1998). A summary of these data sets is shown in Table 1. We have selected datasets with, at least, 1000 instances. For estimating the storage reduction and generalization error we used a k-fold cross-validation method. In this method the available data is divided into k approximately equal subsets. Then, the method is learned k times, using, in turn, each one of the k subsets as testing set, and the remaining $k-1$ subsets as training set. The estimated error is the average testing error of the k subsets. A fairly standard value for k is $k = 10$. The table shows the generalization error of 1-NN classifier using all instances, which can be considered a baseline measure of the error of each dataset. These datasets are representative of problems from medium to large size.

The use of t-tests (Anderson 1984) for the comparison of several methods has been criticized in several papers (Dietterich 1998). This test can provide an accurate evaluation of the probability of obtaining the observed outcomes by chance, but it has limited ability to predict relative performance even on further data set samples from the same domain, let alone on other domains. Moreover, as more data sets and algorithms are used, the probability of type I error, a true null hypothesis incorrectly rejected, increases dramatically. Multiple comparison tests can be used in order to circumvent this last problem, but these tests are not usually able to establish differences between the algorithms.

To avoid these problems several authors perform a sign test on the win/draw/loss record of the two algorithms across all datasets. If the probability of obtaining the observed results by chance, the p-value of the sign test, is below 5%, they conclude that the observed performance is indicative of a general underlying advantage to one of the algorithms with respect to the type of learning task used in the experiments.

Nevertheless, the comparison using sign tests has two problems: Firstly, the differences between the two algorithms compared must be very marked for the test to find significant differences (Demšar 2006); secondly, on some occasions the p-value of the test can be above or below the critical value due to a single modification of the outcome of one experiment, making the result of the test less reliable. So, as the main test we have used the Wilcoxon test for comparing pairs of algorithms for several reasons (Demšar 2006). The Wilcoxon test assumes limited commensurability. It is safer

Table 1 Summary of data sets

Data set	Cases	Features			Classes	Variables	1-NN error
		C	B	N			
Abalone	4177	7	–	1	29	10	0.8034
Adult	48842	6	1	7	2	105	0.2005
Car	1728	–	–	6	4	16	0.1581
Gene	3175	–	–	60	3	120	0.2767
German	1000	6	3	11	2	61	0.3120
Hypothyroid	3772	7	20	2	4	29	0.0692
Isolet	7797	617	–	–	26	617	0.1443
krkopt	28056	6	–	–	18	6	0.4356
kr vs. kp	3196	–	34	2	2	38	0.0828
Letter	20000	16	–	–	26	16	0.0454
Magic	19020	10	–	–	2	10	0.2084
mfeat-fac	2000	216	–	–	10	216	0.0350
mfeat-fou	2000	76	–	–	10	76	0.2080
mfeat-kar	2000	64	–	–	10	64	0.0435
mfeat-mor	2000	6	–	–	10	6	0.2925
mfeat-pix	2000	240	–	–	10	240	0.0270
mfeat-zer	2000	47	–	–	10	47	0.2140
Nursery	12960	–	1	7	5	23	0.2502
Optdigits	5620	64	–	–	10	64	0.0256
Page-blocks	5473	10	–	–	5	10	0.0369
Pendigits	10992	16	–	–	10	16	0.0066
Phoneme	5404	5	–	–	2	5	0.0952
Satimage	6435	36	–	–	6	36	0.0939
Segment	2310	19	–	–	7	19	0.0398
Shuttle	58000	9	–	–	7	9	0.0010
Sick	3772	7	20	2	2	33	0.0430
Texture	5500	40	–	–	11	40	0.0105
Waveform	5000	40	–	–	3	40	0.2860
Yeast	1484	8	–	–	10	8	0.4879
Zip	9298	256	–	–	10	256	0.0292

The features of each data set can be *C* continuous, *B* binary or *N* nominal. The variables column shows the number of variables, as it depends not only on the number of features but also on their type

than parametric tests since it does not assume normal distributions or homogeneity of variance. Thus, it can be applied to error ratios, storage requirements and execution time. Furthermore, empirical results (Demšar 2006) show that it is also stronger than other tests.

The formulation of the test (Wilcoxon 1945) is the following: Let d_i be the difference between the results of the two methods on i-th dataset. These differences are ranked according to their absolute values; in case of ties an average rank is assigned. Let R^+ be the sum of ranks for the datasets on which the second algorithm outperformed the

first, and R^- the sum of ranks where the first algorithm outperformed the second. Ranks of $d_i = 0$ are split evenly among the sums:

$$R^+ = \sum_{d_i > 0} \text{rank}(d_i) + \frac{1}{2} \sum_{d_i = 0} \text{rank}(d_i), \tag{1}$$

and,

$$R^- = \sum_{d_i < 0} \text{rank}(d_i) + \frac{1}{2} \sum_{d_i = 0} \text{rank}(d_i). \tag{2}$$

Let T be the smaller of the two sums and N be the number of datasets. For a small N, there are tables with the exact critical values for T. For a larger N, the statistic

$$z = \frac{T - \frac{1}{4}N(N+1)}{\sqrt{\frac{1}{24}N(N+1)(2N+1)}} \tag{3}$$

is distributed approximately according to $N(0, 1)$.

3.1 Standard algorithms for the comparison

Our model is tested against three of the most successful state-of-the-art algorithms. We have used the classical algorithms DROP3 (Wilson and Martinez 2000), and ICF (Brighton and Mellish 2002). DROP3 (*Decremental Reduction Optimization Procedure 3*) is shown in Algorithm 3. This algorithm represents one of the examples of a new generation of algorithms that were designed taking into account the effect of the order of removal on the performance of the algorithm. So, this algorithm is designed to be insensitive to the order of presentation of the instances. It includes a noise filtering step using a method similar to Wilson's *Edited Nearest-Neighbor* Rule (Wilson 1972). Then, the instances are ordered by the distance to their nearest neighbor. The instances are removed beginning with the instances furthest from its nearest neighbor. This tends to remove the instances furthest from the boundaries first.

ICF is shown in Algorithm 4. For ICF algorithm *coverage* and *reachability* are defined as follows:

$$Coverage(c) = \{c' \in T\text{: } LocalSet(c)\} \tag{4}$$

$$Reachable(c) = \{c \in T\text{: } LocalSet(c')\}. \tag{5}$$

The Local-set of a case c is defined as "the set of cases contained in the largest hypersphere centred on c such that only cases in the same class as c are contained in the hypersphere" (Brighton and Mellish 2002). In Case Base Reasoning (CBR) framework (Smyth and Keane 1995) a case c can be adapted to a case c' if c is relevant to the correct prediction of c'. That means that c is a member of the neighborhood of c', bounding the neighborhood of c' by the first instance of a different class

Algorithm 3: DROP3 algorithm

```
   Data   : A training set T = {(x_1, y_1), ..., (x_n, y_n)}.
   Result : The reduced selector S ⊂ T.
1  S = T
2  Noise filtering: Remove any instance in S missclassified by its k neighbors
3  Sort instances in S by distance to their nearest enemy
   foreach Instance P ∈ S do
4      Find P.N_{1..k+1}, the k + 1 nearest neighbors of P in S
5      Add P to each of its neighbors' list of associates
   end
   foreach Instance P ∈ S do
6      Let with = # of associates of P classified correctly with P as a neighbor
7      Let without = # of associates of P classified correctly without P
       if without ≥ with then
8          Remove P from S
           foreach Associate A of P do
9              Remove P from A's list of nearest neighbors
10             Find a new nearest neighbor for A
11             Add A to its new neighbor's list of associated
           end
       end
   end
```

Algorithm 4: Iterative Case Filtering (ICF) algorithm

```
   Data   : A training set T = {(x_1, y_1), ..., (x_n, y_n)}.
   Result : The reduced selector S ⊂ T.
1  S = T
2  Noise filtering: Remove any instance in S missclassified by its k neighbors
   repeat
       forall x ∈ S do
3          Compute reachable(x)
4          Compute coverage(x)
       end
5      progress = false
       forall x ∈ S do
           if |reachable(x)| > |coverage(x)| then
6              Flag x for removal
7              progress = true
           end
       end
       forall x ∈ S do
           if x flagged for removal then
8              S = S − {x}
           end
       end
   until not progress
```

[see (Brighton and Mellish 2002) for details]. The algorithm is based on repeatedly applying a deleting rule to the set of retained instances until no more instances fulfill the deleting rule.

The concept of reachable and coverage sets used by ICF are similar to the neighborhood and associate sets used by RT algorithms (Wilson and Martinez 1997). The

difference is that the sets defined in ICF are not of fixed size, but bounded by the first instance belonging to another class. This difference is considered *crucial* by the authors of ICF.

We have chosen ICF and Drop3 as representative of the methods we can consider "classical", as they have been around for quite a long time and are widely used. In the experimental section we compare the performance of these two methods when they are applied to the whole training set, we will call this application just ICF and Drop3, with the application of both methods using our recursive approach being called recursive ICF/Drop3.

As an alternative to these "classical" methods, evolutionary computation algorithms have been applied for instance selection, considering this task to be a search problem. Evolutionary computation (EC) (Holland 1975; Goldberg 1989; Michalewicz 1994) is a set of global optimization techniques that have been widely used in the last few years for almost every problem within the field of Artificial Intelligence. In evolutionary computation a population (set) of individuals (solutions to the problem faced) are codified following a code similar to the genetic code of plants and animals. This population of solutions is evolved (modified) over a certain number of generations (iterations) until the defined stop criterion is fulfilled. Each individual is assigned a real value that measures its ability to solve the problem, which is called its *fitness*.

In each iteration new solutions are obtained combining two or more individuals (crossover operator) or randomly modifying one individual (mutation operator). After applying these two operators a subset of individuals is selected to survive to the next generation, either by sampling the current individuals with a probability proportional to their fitness, or by selecting the best ones (elitism). The repeated processes of crossover, mutation and selection are able to obtain increasingly better solutions for many problems of Artificial Intelligence.

The application of evolutionary computation to instance selection is easy and straightforward. Each individual is a binary vector that codes a certain sample of the training set. The evaluation is usually made considering both data reduction and classification accuracy. Examples of applications of genetic algorithms to instance selection can be found in Kuncheva (1995), Ishibuchi and Nakashima (2000) and Reeves and Bush (2001).

Brighton and Mellish (2002) argued that the structure of the classes formed by the instances can be very different, thus, an instance selection algorithm can have a good performance in one problem and be very inefficient in another. They state that the instance selection algorithm must gain some insight into the structure of the classes to perform an efficient instance selection. However, this insight is not usually available or very difficult to acquire, especially in real-world problems with many variables and complex boundaries between the classes. In such a situation, an approach based on EC may be of help. The approaches based on EC do not assume any special form of the space, the classes or the boundaries between the classes, they are only guided by the ability of each solution to solve the task. In this way, the algorithm learns the relevant instances from the data without imposing any constraint in the form of classes or boundaries between them.

Thus, one of the most interesting advantages of the application of evolutionary computation to instance selection is that evolutionary approaches do not depend on

specific classifiers, and can be used with any instance based classifier. This is in contrast with most standard instance selection algorithms that are specifically designed for k-NN classifiers. For instance, Reeves and Bush (2001) used a genetic algorithm to select instances for RBF neural networks.

Cano et al. (2003) performed a comprehensive comparison of the performance of different evolutionary algorithms for instance selection. They compared a generational genetic algorithm (Goldberg 1989), a steady-state genetic algorithm (Whitley 1989), a CHC genetic algorithm (Eshelman 1990), and a population based incremental learning algorithm (Baluja 1994). They found that evolutionary based methods were able to outperform classical algorithms in both classification accuracy and data reduction. Among the evolutionary algorithms, CHC was able to achieve the best overall performance.

Nevertheless, the major problem that has to be addressed when applying genetic algorithms to instance selection is the scaling of the algorithm. As the number of instances grows, the time needed for the genetic algorithm to reach a good solution increases exponentially, making it totally useless for large datasets. As this paper is mainly concerned with this problem, we have used as third instance selection method a genetic algorithm using CHC methodology. The execution time of CHC is clearly longer than the time spent by ICF and Drop3, so it gives us a good benchmark to test our methodology on an algorithm that has an important scalability problem.

The source code, in C and licensed under the GNU General Public License, used for all methods, as well as the partitions of the datasets, are freely available upon request to the authors.

4 Experimental results

The same parameters were used for the standard version of every algorithm and its application within our methodology. For Drop3 and ICF we used $k = 3$ neighbors, and for CHC we used $k = 1$. For our model we used a subset size of 100 instances. Cross-validation was used for stopping the selection using a random 10% of the training set as validation set. For CHC we used a population of 100 individuals that were evolved over 100 generations. The evaluation of the individuals was made considering two criteria: reduction of storage, ρ, and classification error, ϵ. The fitness of individual j, F_j, is given by:

$$F_j = w(1 - \epsilon) + (1 - w)\rho, \tag{6}$$

where $0 \leq w \leq 1$. The weight w is needed to avoid a negative effect that may occur due to the asymmetry of the two values of the fitness function: the reduction value can be made arbitrarily high, up to the maximum value 1, by just removing more instances. In our experiments $w = 2/3$. The fitness value is the usual one in applying genetic algorithms to instance selection (Cano et al. 2003).

Table 2 shows the results using Drop3 as base algorithm. These results are plotted in Fig. 3. The testing error of our approach is slightly worse than the error obtained

Table 2 Testing error, storage requirements and execution time (in seconds) for standard Drop3 algorithm and our approach

Dataset	Drop3					Recursive Drop3				
	Storage		Error		Time	Storage		Error		Time
	Mean	SD	Mean	SD		Mean	SD	Mean	SD	
Abalone	0.3069	0.0027	0.7782	0.0979	1.8	0.0730	0.0326	0.7767	0.0804	0.6
Adult	0.1248	0.0011	0.1714	0.0051	22853.9	0.0570	0.0023	0.1977	0.0158	112.3
Car	0.2668	0.0135	0.2378	0.0955	1.9	0.1665	0.0232	0.2680	0.0886	0.2
Gene	0.3877	0.0067	0.2776	0.0230	35.6	0.0989	0.0231	0.3186	0.0295	1.2
German	0.3073	0.0080	0.2870	0.0743	0.9	0.0500	0.0228	0.3170	0.0508	0.3
Hypo	0.0514	0.0041	0.0610	0.0111	11.8	0.0113	0.0033	0.1180	0.0812	0.9
Isolet	0.2852	0.0025	0.1770	0.0403	208.9	0.1956	0.0054	0.2233	0.0494	2.3
krkopt	0.4431	0.0032	0.4803	0.0098	1533.0	0.1950	0.0036	0.5178	0.0091	13.1
kr vs. kp	0.2229	0.0069	0.1016	0.0171	10.1	0.1243	0.0076	0.1693	0.0159	0.7
Letter	0.1744	0.0008	0.1037	0.0053	1849.8	0.3675	0.0128	0.1161	0.0116	17.4
Magic	0.1789	0.0083	0.1978	0.1333	199.8	0.0599	0.0248	0.2533	0.0807	4.8
mfeat-fac	0.1208	0.0052	0.0600	0.0175	39.3	0.1499	0.0229	0.0630	0.0123	2.3
mfeat-fou	0.2473	0.0077	0.2320	0.0423	5.6	0.1347	0.0282	0.2815	0.0250	0.9
mfeat-kar	0.1655	0.0064	0.0835	0.0292	6.5	0.1594	0.0258	0.0930	0.0310	1.0
mfeat-mor	0.2062	0.0043	0.2885	0.0315	1.4	0.1138	0.0368	0.3435	0.0303	0.3
mfeat-pix	0.1095	0.0031	0.0480	0.0155	76.5	0.1442	0.0260	0.0560	0.0171	2.6
mfeat-zer	0.2231	0.0041	0.2375	0.0236	4.0	0.1467	0.0301	0.2390	0.0231	0.8
Nursery	0.2934	0.0049	0.3327	0.0454	337.4	0.1274	0.0191	0.2612	0.0660	3.0
Optdigits	0.0911	0.0027	0.0420	0.0069	161.0	0.1825	0.2453	0.0453	0.2865	3.0
Page-bl	0.0430	0.0020	0.0437	0.0078	15.7	0.0170	0.0035	0.0859	0.0247	1.0
Pendigits	0.0451	0.0017	0.0168	0.0028	175.0	0.1737	0.0073	0.0135	0.0044	3.2
Phoneme	0.1852	0.0020	0.1383	0.0147	11.5	0.0675	0.0089	0.2645	0.0266	0.8
Satimage	0.1366	0.0034	0.1101	0.0130	57.7	0.0828	0.0035	0.1608	0.0162	1.8
Segment	0.1219	0.0076	0.0784	0.0204	4.1	0.1261	0.0206	0.1087	0.0314	0.6
Shuttle	0.0028	0.0008	0.0016	0.0006	7543.4	0.0197	0.0047	0.0063	0.0017	20.9
Sick	0.0625	0.0045	0.0509	0.0127	14.8	0.0216	0.0091	0.1056	0.0657	1.0
Texture	0.0878	0.0018	0.0329	0.0062	97.0	0.2411	0.0073	0.0320	0.0075	2.2
Waveform	0.2961	0.0043	0.2276	0.0286	28.8	0.0951	0.0287	0.2352	0.0223	1.5
Yeast	0.3193	0.0087	0.4500	0.0253	0.6	0.0777	0.0243	0.4642	0.0274	0.3
Zip	0.1040	0.0022	0.0497	0.0062	601.7	0.1824	0.0064	0.0532	0.0074	4.9

Mean and standard deviation values are shown

using the standard algorithm. The results of storage requirements are favorable to our method which is able to significantly improve the results of standard Drop3. In fact, for abalone, car, gene, german, isolet, krvskp, krkopt, magic, nursery, phoneme, waveform and yeast datasets, our method is even able to show a clear improvement over standard Drop3. However, it is in execution time that our algorithm shows its better face. The reduction in the time spent is very marked, with an extreme case in shuttle

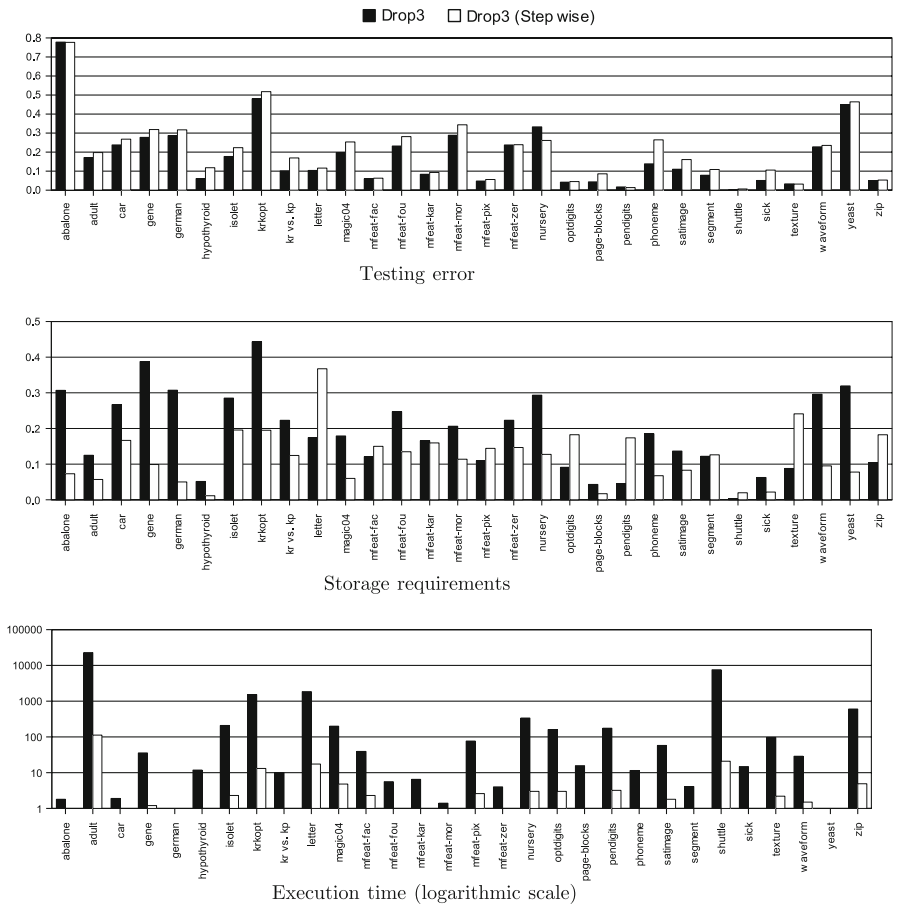

Fig. 3 Testing error, storage requirements and execution time (in seconds) for standard Drop3 algorithm and our approach

dataset where our algorithm took less than 0.3% of the time needed by standard Drop3. As a summary, we can say that for Drop3 our procedure is able to improve the storage reduction of Drop3, with a worse testing error but with a large reduction of execution time. Wilcoxon test finds significant differences in terms of testing error (p-value of 0.0001) in favor of standard Drop3, and in terms of storage (p-value of 0.0104) in favor of our approach. Regarding standard deviation, our method has higher values, but still within moderate limits.

Table 3 shows the results using ICF as base algorithm. These results are plotted in Fig. 4. Our method is able to improve the testing error of standard ICF, with a better average error in 18 of the 30 datasets. However, the differences are not statistically significant (p-value of 0.8774). On the other hand, storage requirements are clearly better in our approach. In this way, Wilcoxon test finds significant differences between both algorithms at a confidence level of 95% (p-value of 0.0148). The differences in execution time are, as in the previous case, clearly marked as is shown in Fig. 4. As

Table 3 Testing error, storage requirements and execution time (in seconds) for standard ICF algorithm and our approach

Dataset	ICF					Recursive ICF				
	Storage		Error		Time	Storage		Error		Time
	Mean	SD	Mean	SD		Mean	SD	Mean	SD	
Abalone	0.2510	0.0047	0.8082	0.0865	1.7	0.0461	0.0187	0.7990	0.0680	0.8
Adult	0.1082	0.0009	0.2194	0.0032	9170.8	0.0275	0.0119	0.2125	0.0420	85.0
Car	0.3813	0.0557	0.2709	0.1407	1.1	0.1787	0.0239	0.3180	0.0891	1.0
Gene	0.2508	0.0080	0.3527	0.0146	26.1	0.0634	0.0048	0.4641	0.0273	2.0
German	0.1485	0.0093	0.3260	0.0544	0.4	0.0116	0.0044	0.3943	0.0869	1.4
Hypo	0.0398	0.0029	0.1156	0.0356	3.7	0.0129	0.0714	0.0696	0.0100	0.9
Isolet	0.1713	0.0028	0.2648	0.0541	103.1	0.0949	0.0040	0.2751	0.0578	2.3
krkopt	0.5290	0.0073	0.4032	0.0068	1109.8	0.0786	0.0027	0.6210	0.0096	7.4
kr vs. kp	0.2707	0.0080	0.1267	0.0330	5.3	0.0463	0.0045	0.2132	0.0125	1.0
Letter	0.1362	0.0025	0.2018	0.0154	760.3	0.1650	0.0062	0.1947	0.0084	7.3
Magic	0.1160	0.0073	0.2395	0.1499	138	0.0214	0.0029	0.2581	0.1290	2.9
mfeat-fac	0.0896	0.0052	0.0905	0.0171	17.5	0.1296	0.0125	0.0828	0.0176	3.2
mfeat-fou	0.1395	0.0055	0.3280	0.0256	2.4	0.0681	0.0078	0.4194	0.0340	1.6
mfeat-kar	0.1035	0.0056	0.1725	0.0334	2.5	0.1154	0.0084	0.1580	0.0328	1.2
mfeat-mor	0.2008	0.0088	0.3685	0.0563	0.6	0.0964	0.0201	0.3229	0.0384	0.7
mfeat-pix	0.0864	0.0047	0.1000	0.0218	27	0.1172	0.0089	0.0669	0.0160	2.9
mfeat-zer	0.1503	0.0062	0.2715	0.0193	1.7	0.0793	0.0074	0.2970	0.0223	1.3
Nursery	0.8752	0.0095	0.2414	0.1407	287.2	0.2932	0.0446	0.2291	0.1139	3.7
Optdigits	0.0606	0.0023	0.1103	0.0258	82.8	0.1104	0.0061	0.0626	0.0083	3.7
Page-bl	0.0307	0.0027	0.2185	0.0128	5.6	0.0557	0.0482	0.0569	0.0133	1.3
Pendigits	0.0348	0.0018	0.0651	0.0094	70.6	0.2071	0.0113	0.0251	0.0097	2.7
Phoneme	0.1392	0.0031	0.1941	0.0192	4.6	0.0391	0.0102	0.2774	0.0199	0.7
Satimage	0.0713	0.0023	0.1677	0.0259	25.4	0.0699	0.0039	0.1622	0.0216	2.2
Segment	0.1077	0.0060	0.1394	0.0285	1.6	0.1118	0.0091	0.1361	0.0258	0.7
Shuttle	0.0229	0.0026	0.0473	0.0129	2640	0.1902	0.0814	0.0034	0.0007	102.1
Sick	0.0452	0.0045	0.0912	0.0190	4.7	0.0103	0.1325	0.1244	0.0101	1.1
Texture	0.0725	0.0030	0.0973	0.0149	46.8	0.1695	0.0100	0.0578	0.0124	2.0
Waveform	0.1211	0.0050	0.2840	0.0231	15	0.0240	0.0112	0.2818	0.0264	1.6
Yeast	0.2137	0.0079	0.5095	0.0358	0.3	0.0259	0.0102	0.5453	0.0771	0.1
Zip	0.0497	0.0019	0.2549	0.0182	219.8	0.0923	0.0067	0.1793	0.0224	4.5

Mean and standard deviation values are shown

a summary, we can say that for ICF our procedure is able to match the testing error of ICF, with better average performance in terms of storage reduction, and with a large reduction of execution time. As was the case for Drop3, there are several datasets, namely abalone, adult, car, gene, german, krkopt, krvskp, mfeat-mor, nursery, waveform and yeast, for which the increment in storage reduction is clearly marked.

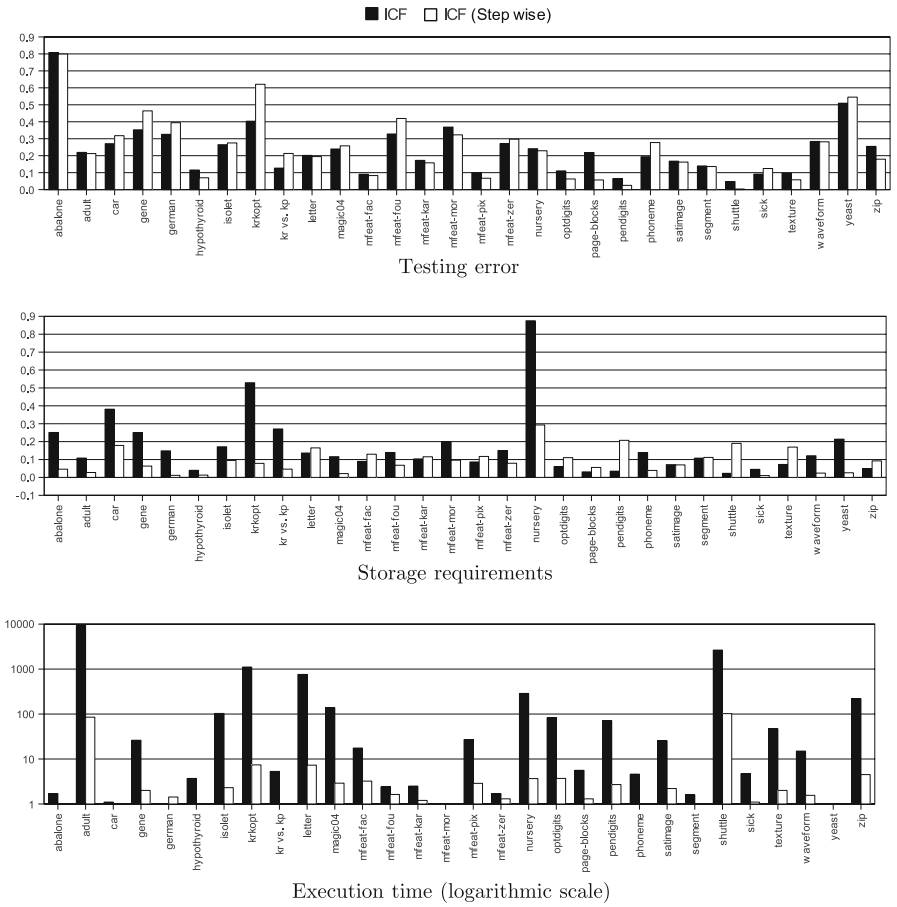

Fig. 4 Testing error, storage requirements and execution time (in seconds) for standard ICF algorithm and our approach

In terms of standard deviation, for storage reduction our algorithm achieves worse results, although the differences are small, and for testing error the deviations of both methods are similar.

Table 4 shows the results using CHC genetic algorithm as base method. These results are plotted in Fig. 5. A first interesting result is the problem of scalability of CHC algorithm, which is more accused for this algorithm than for Drop3 and ICF. In other works (Cano et al. 2003; García-Pedrajas et al. 2008), CHC algorithm is compared with Drop3 and ICF in small to medium problems. For those problems, the performance of CHC was better than the performance of Drop3 and ICF. However, as the datasets are larger, the scalability problem of CHC manifests itself. In our set of problems, CHC clearly performs worse than Drop3 and ICF and takes considerably more execution time. We must take into account that for CHC we need a bit in the chromosome for each instance in the dataset. This means that for large problems, such as adult, krkopt, letter, magic or shuttle, the chromosome has more than 10000 bits, making the convergence of the algorithm problematic.

Table 4 Testing error, storage requirements and execution time (in seconds) for standard CHC algorithm and our approach

Dataset	CHC					Recursive CHC				
	Storage		Error		Time	Storage		Error		Time
	Mean	SD	Mean	SD		Mean	SD	Mean	SD	
Abalone	0.3818	0.0209	0.7998	0.0584	7722.2	0.1718	0.0511	0.8202	0.0387	263.9
Adult	0.1988	0.0030	0.2257	0.0060	91096.0	0.1393	0.0025	0.2104	0.0057	1201.0
Car	0.4192	0.0181	0.2639	0.0598	5483.6	0.1225	0.0354	0.2663	0.0690	45.1
Gene	0.3004	0.0175	0.2968	0.0288	7236.6	0.1364	0.0483	0.3640	0.0324	101.2
German	0.3483	0.0424	0.3290	0.0614	440.1	0.0976	0.2775	0.3430	0.2074	32.2
Hypo	0.2613	0.0187	0.0775	0.0100	7183.9	0.0997	0.0485	0.0923	0.0174	84.5
Isolet	0.2993	0.0186	0.2026	0.0433	10885.4	0.1824	0.0022	0.2252	0.0401	409.7
krkopt	0.5237	0.0056	0.4711	0.0066	137015.0	0.1701	0.0032	0.5045	0.0053	6258.7
kr vs. kp	0.2712	0.0274	0.1276	0.0195	7107.5	0.1393	0.0085	0.1687	0.0178	79.6
Letter	0.2952	0.0243	0.0905	0.0168	51024.0	0.1778	0.0036	0.1106	0.0059	1161.2
Magic	0.2952	0.0591	0.1225	0.0275	29327.0	0.1041	0.0556	0.3109	0.1253	644.3
mfeat-fac	0.3933	0.0357	0.0455	0.0136	6518.4	0.1052	0.0443	0.0735	0.0239	57.7
mfeat-fou	0.3384	0.0238	0.2280	0.0361	7026.9	0.1552	0.0294	0.2810	0.0296	80.1
mfeat-kar	0.3838	0.0448	0.0650	0.0170	7010.8	0.1500	0.0092	0.0945	0.0151	66.5
mfeat-mor	0.3573	0.0480	0.3265	0.0364	7054.2	0.1296	0.0339	0.3410	0.0335	66.6
mfeat-pix	0.3759	0.0037	0.0440	0.1058	6441.9	0.1259	0.0347	0.0645	0.0121	62.9
mfeat-zer	0.3439	0.0080	0.2205	0.0045	7076.3	0.1422	0.0276	0.2515	0.0424	73.0
Nursery	0.2941	0.0147	0.2427	0.0073	22397.3	0.1183	0.0417	0.2752	0.0815	326.0
Optdigits	0.2755	0.0136	0.0404	0.0048	7846.4	0.1395	0.0030	0.0504	0.0090	170.7
Page-bl	0.2786	0.0126	0.0408	0.0113	7662.8	0.1344	0.0250	0.0558	0.0116	126.1
Pendigits	0.2903	0.0033	0.0121	0.0128	11636.6	0.1357	0.0029	0.0170	0.0073	288.6
Phoneme	0.2846	0.0633	0.1457	0.0198	7772.9	0.1395	0.0032	0.1726	0.0095	173.0
Satimage	0.2825	0.0102	0.1157	0.0235	8491.8	0.1392	0.0029	0.1263	0.0137	127.3
Segment	0.3030	0.0293	0.0649	0.0115	7062.4	0.1400	0.0087	0.1078	0.0264	71.5
Shuttle	0.2638	0.0169	0.0055	0.0062	77089.0	0.1428	0.0025	0.0016	0.0006	1137.3
Sick	0.2578	0.0187	0.0514	0.0113	7179.8	0.1115	0.0336	0.0597	0.0134	80.1
Texture	0.2825	0.0260	0.0249	0.0243	7876.1	0.1425	0.0074	0.0415	0.0097	148.8
Waveform	0.2911	0.0035	0.2878	0.0062	7926.8	0.0954	0.0532	0.3118	0.0307	206.6
Yeast	0.3711	0.0087	0.5014	0.0253	2981.1	0.1363	0.0584	0.5439	0.0465	50.3
Zip	0.2871	0.0022	0.0510	0.0062	9748.2	0.1539	0.0052	0.0574	0.0065	371.7

Mean and standard deviation values are shown

Regarding the comparison with our method, the behavior of CHC is similar to the behavior of Drop3. Our method is able to significantly improve the storage reduction (p-value of the Wilcoxon test of 0.0000), although the testing error is worse (p-value of 0.0001). The reduction in execution time is dramatic, with our method usually running in about 1% of the time needed by the standard method. As a summary, we

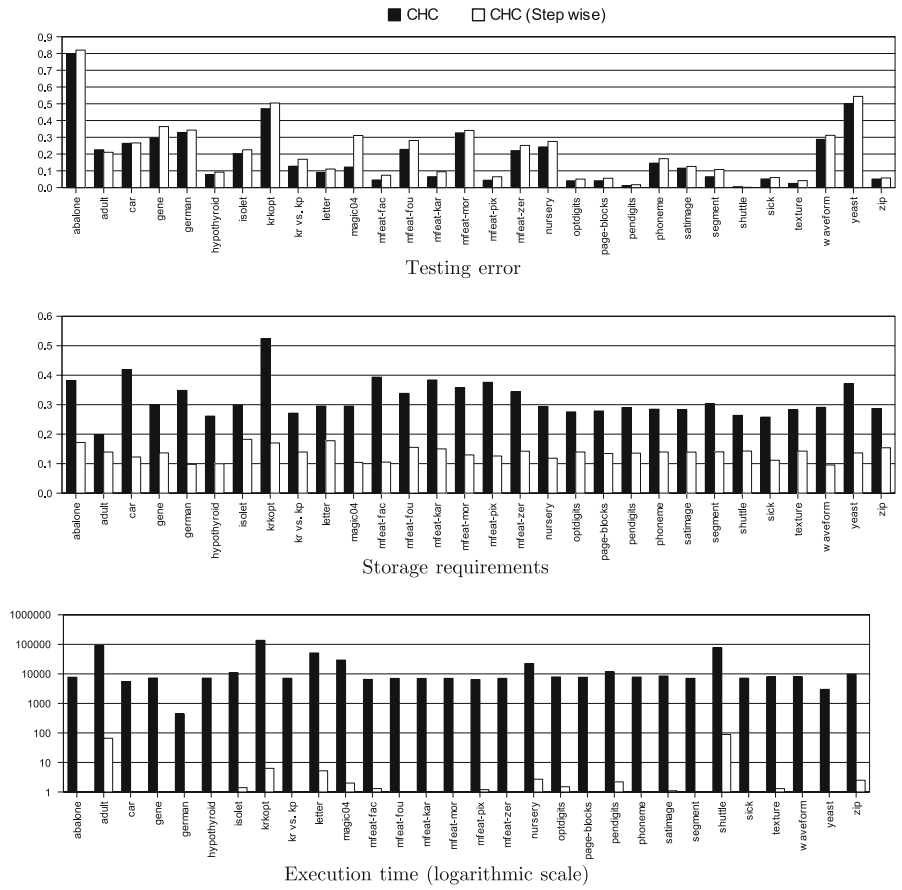

Fig. 5 Testing error, storage requirements and execution time (in seconds) for standard CHC genetic algorithm algorithm and our approach

can say that for CHC our procedure is able to improve the storage reduction of CHC, with a worse testing error, but with a very large reduction in execution time. The storage reduction is specially marked. Our method performs better, and with marked differences, in all 30 datasets. The table also shows that our method has a slightly worse deviation than standard CHC.

The behavior of the standard algorithms and our approach in terms of execution time in function of the number of instances is illustrated in Fig. 6. We plot the time spent by the algorithms as a function of the number of instances. The figure shows that standard methods have an execution time that is approximately quadratic with respect to the number of instances, for Drop3 and ICF, and even higher for CHC. So, for large problems the necessary time is substantial. On the other hand, our proposal is approximately linear, allowing the use of the methods even with hundreds of thousands of instances, as will be shown in Sect. 4.3.

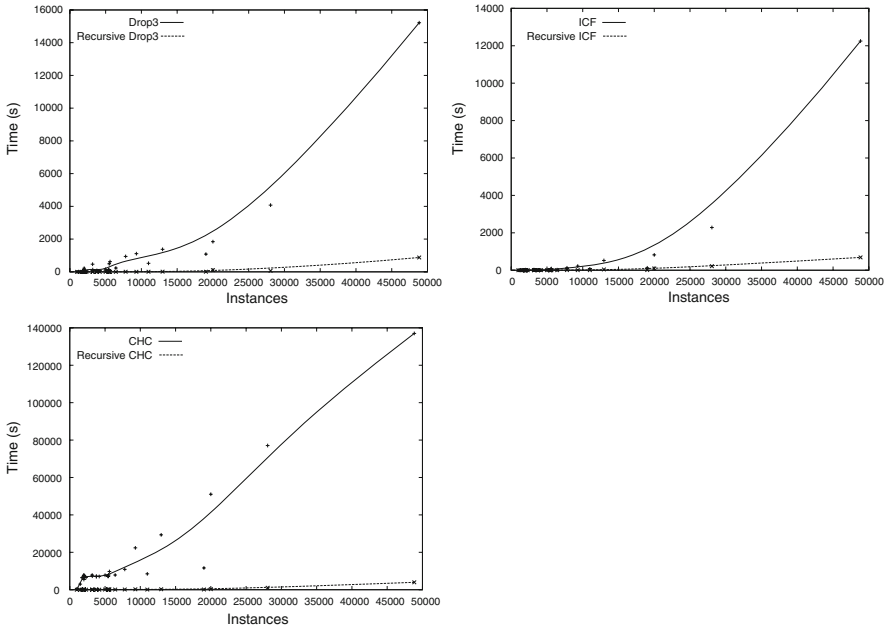

Fig. 6 Execution time (in seconds) for standard methods and our approach in function of the number of instances

4.1 Study of subset size effect

We have stated that the size of the subset is not relevant provided it is kept small, that is, of about a few hundreds of instances. However, this statement must be corroborated. We have performed experiments using Drop3 and ICF and subsets sizes of 250, 500 and 1000 instances. Figure 7 shows the average values of testing error, storage requirements and execution time with the four sizes. The trend is similar for both algorithms. As the size grows, the algorithm is more efficient in removing instances, achieving larger reductions to the storage. However, that reduction comes at the cost of deteriorating testing error, which is larger. Nevertheless, the increment in the reduction is larger than the increment in the error, so if we are mainly interested in storage reduction we can use a larger subset size, trading testing error for reduction.

Regarding execution time, the effect is clear. As the subset size grows the execution time grows. As the size is larger, the $O(N^2)$ of the algorithms begins to be relevant, and the processing time of each subset is more important than the reduction of the number of subsets to process. However, even for 1000 instances as subset size the reduction in execution time is still dramatic when compared to the standard algorithm.

4.2 Improving testing error

In previous sections, we have shown that our methodology is able to improve the performance of standard instance selection algorithms in terms of storage requirements

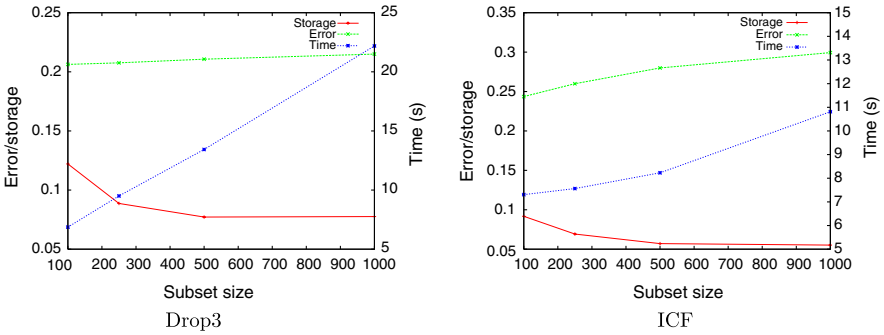

Fig. 7 Average testing error, storage requirements and execution time (in seconds) as a function of subset size

with a very significant reduction in execution time. However, for Drop3 and CHC algorithms, there is also a worsening in terms of testing error. We have developed two mechanisms for improving testing error. These mechanisms try to ameliorate the effect that the execution of the algorithm over disjoint subsets may have.

The first method is very simple. Before each step, we add a certain percentage of the instances removed in the previous step to the pool of selected instances. In our method this percentage is of 10%. With this method we try to avoid the damaging effect of removing too many instances.

The second method is more elaborated. The main source of deteriorated testing error in our method is the limited view that each execution of the algorithm has, due to the fact that it is applied to a small subset of the whole dataset. In this way, useful instances may be removed if they are not relevant in the subset they belongs to in a step of the algorithm. To avoid this effect we use a *"second chance"* approach. We maintain for the whole data set a list of marked instances. When the instance is selected for removing by the selection algorithm, it is removed only if it is marked in the list. Otherwise, the instance is marked but not removed. If the instance is marked, and not selected to be removed in the last step of the algorithm, it is unmarked. In this way, an instance is removed only if two consecutive steps of the method select it for removing. The chances of removing useful instances are decreased, as they must be selected for removal in two consecutive steps of the algorithms. The process, combining both methods, is shown in Algorithm 5.

In the first step, due to the large number of initial instances, the possibility of removing useful instances is small, so initially all the instances are marked. It means that all instances marked for removal in the first step of the algorithm are actually removed. Table 5 shows the results using both mechanisms, random addition of removed instances and second chance. These mechanisms are specially efficient for Drop3 and ICF. In this set of experiments our method is as good as Drop3 in terms of testing error and significantly better than ICF (p-values for the Wilcoxon test of 0.1650 and 0.0387 respectively). However, the cost is a worse storage reduction. For CHC, the techniques have a more limited effect. The testing error of our method is improved, but it is still significantly worse than the standard CHC algorithm.

Algorithm 5: Recursive instance selection algorithm with second chance and random addition of instances

```
Data   : A training set T = {(x_1, y_1), ..., (x_n, y_n)}, percentage of random addition p, and subset
         size s.
Result : The reduced selector S ⊂ T.
1  S = T
2  Partitionate instances into disjoint subsets t_i : ⋃_i t_i = S of size s
3  Mark all instances
   repeat
4      Add random p% of unselected instances to the set of selected instances
       foreach subset t_i ⊂ S do
5          Apply instance selection algorithm in t_i to obtain selected subset s_i ⊂ t_i
           forall Instances x selected for removing in t_i do
               if x is marked then
6                  Remove x from t_i
               else
7                  Mark x
               end
           end
           forall Instances x NOT selected for removing in t_i do
8              Unmark x
           end
9          Remove from S instances removed from t_i
       end
10     Fussion subsets s_i to obtain new subsets t_j of size s
   until until stop criterion
11 Return S
```

The experiments show that a better testing error can be achieved using second chance and a percentage of random addition of instances. It is up to the researcher to decide whether it is better for his/her application to focus either on testing error or storage reduction. Furthermore, other techniques may be developed, such as limiting the reduction performed by the application of the instance selection algorithm to a subset.[2] Table 6 shows the standard deviation of this method using the three algorithms of instance selection. In the table we can see that not only the testing error is improved, but also the deviation of the experiments is smaller. In fact, using second chance and random addition of instances, the deviation is very close to the results using the standard algorithms.

4.3 Huge problems

In previous experiments we have shown the performance of our methodology in problems that can be considered of medium to large size. In this section, we considered huge problems, from several hundreds of thousands of instances to more than a million. Table 7 shows the problems that are considered. These datasets will show whether our methodology allows scaling up standard algorithms to huge problems. We have tested our method using the three instance selection algorithms of the previous sections. As in the previous sections, for estimating the storage reduction and

[2] In fact, this procedure has been used with results similar to the presented second chance method.

Table 5 Testing error, storage requirements and execution time (in seconds) for the recursive approach using both mechanisms, random addition of removed instances and second chance, of improving testing error

Dataset	Recursive Drop3			Recursive ICF			Recursive CHC		
	Stor.	Error	Time	Stor.	Error	Time	Stor.	Error	Time
Abalone	0.2553	0.7935	1.0	0.1355	0.7868	0.4	0.3196	0.8086	237.4
Adult	0.1769	0.1803	335.3	0.0547	0.2020	223.1	0.2553	0.2013	1461.2
Car	0.2269	0.2360	0.4	0.1900	0.2936	0.2	0.2501	0.2314	40.6
Gene	0.2438	0.2855	2.5	0.1115	0.4205	2.9	0.2770	0.3398	91.1
German	0.2641	0.3060	0.3	0.0277	0.3270	0.3	0.2635	0.3630	31.4
Hypo	0.0857	0.0682	0.9	0.0308	0.0743	1.0	0.2499	0.0836	74.7
Isolet	0.4329	0.1743	4.4	0.1443	0.2497	5.7	0.3080	0.1982	383.6
krkopt	0.2943	0.4810	30.4	0.1298	0.5737	19.7	0.2906	0.4698	6494.1
kr vs. kp	0.2562	0.1282	1.0	0.0794	0.1806	0.8	0.2568	0.1367	83.4
Letter	0.4919	0.0925	21.0	0.2429	0.1526	16.0	0.2962	0.0879	1176.2
Magic	0.1294	0.2309	7.1	0.0413	0.2338	5.4	0.2529	0.2466	593.1
mfeat-fa	0.2496	0.0465	2.7	0.2025	0.0620	2.9	0.2454	0.0560	52.9
mfeat-fo	0.3137	0.2330	1.1	0.0986	0.3805	0.7	0.2776	0.2420	75.1
mfeat-ka	0.3055	0.0700	1.1	0.1782	0.1220	0.8	0.2628	0.0755	64.4
mfeat-mo	0.2478	0.3025	0.3	0.1821	0.3030	0.1	0.2601	0.3185	61.5
mfeat-pi	0.2754	0.0385	3.1	0.1833	0.0565	3.4	0.2563	0.0445	59.0
mfeat-ze	0.2735	0.2150	0.7	0.1222	0.2500	0.6	0.2637	0.2290	68.1
Nursery	0.1684	0.2515	4.0	0.2763	0.2202	5.5	0.2511	0.2394	369.3
Optdigits	0.2979	0.0361	3.5	0.1727	0.0571	3.7	0.2414	0.0468	185.3
Page-bl	0.0504	0.0634	1.0	0.1257	0.0559	0.4	0.2529	0.0479	132.1
Pendigits	0.2137	0.0117	4.3	0.2753	0.0162	4.1	0.2473	0.0133	293.2
Phoneme	0.1077	0.2143	0.9	0.0580	0.2361	0.5	0.2507	0.1546	180.4
Satimage	0.1304	0.1387	2.4	0.1348	0.1460	2.7	0.2475	0.1162	143.8
Segment	0.2037	0.0853	0.5	0.1623	0.1117	0.5	0.2526	0.0732	70.5
Shuttle	0.0333	0.0045	81.4	0.3029	0.0016	56.3	0.2613	0.0014	1057.4
Sick	0.0828	0.0533	0.6	0.0272	0.0759	1.0	0.2484	0.0520	72.1
Texture	0.2373	0.0307	2.6	0.2321	0.0420	3.4	0.2455	0.0295	150.6
Waveform	0.2958	0.2458	2.2	0.0578	0.2642	2.1	0.2583	0.2800	200.2
Yeast	0.2385	0.4676	0.2	0.0841	0.4899	0.1	0.2949	0.5372	42.5
Zip	0.3225	0.0389	8.6	0.1354	0.1191	6.6	0.2620	0.0450	359.1

generalization error we used a 10-fold cross-validation method. The size of the datasets prevents the execution of the standard algorithms in a reasonable time, so the validity of our approach will be tested against 10-fold cross-validation 1-NN testing error, which is shown in the table, together with the time needed to obtain that error.

In the previous experiments the number of steps of the algorithm was obtained by means of a cross-validation method which was described in Sect. 2 in the paragraph devoted to the stop criterion. However, due to the size of the datasets, and to make the

Table 6 Standard deviation of testing error and storage requirements for the recursive approach using both mechanisms, random addition of removed instances and second chance, of improving testing error

Dataset	Recursive Drop3		Recursive ICF		Recursive CHC	
	Storage	Error	Storage	Error	Storage	Error
Abalone	0.0124	0.0811	0.0167	0.0848	0.0075	0.0583
Adult	0.0047	0.0057	0.0096	0.0120	0.0030	0.0061
Car	0.0296	0.0631	0.0297	0.1041	0.0122	0.0659
Gene	0.0109	0.0275	0.0062	0.0331	0.0106	0.0187
German	0.0137	0.0709	0.0084	0.0603	0.0177	0.0585
Hypo	0.0042	0.0103	0.0073	0.0152	0.0100	0.0139
Isolet	0.0063	0.0442	0.0038	0.0558	0.0055	0.0416
krkopt	0.0079	0.0077	0.0033	0.0076	0.0025	0.0062
kr vs. kp	0.0072	0.0181	0.0083	0.0249	0.0085	0.0150
Letter	0.0051	0.0045	0.0061	0.0077	0.0035	0.0072
Magic	0.0098	0.0631	0.0043	0.1317	0.0031	0.1147
mfeat-fa	0.0050	0.0105	0.0060	0.0119	0.0163	0.0097
mfeat-fo	0.0107	0.0222	0.0102	0.0342	0.0097	0.0309
mfeat-ka	0.0063	0.0206	0.0123	0.0200	0.0103	0.0192
mfeat-mo	0.0180	0.0265	0.0141	0.0260	0.0077	0.0292
mfeat-pi	0.0070	0.0134	0.0141	0.0092	0.0110	0.0165
mfeat-ze	0.0055	0.0253	0.0075	0.0160	0.0105	0.0314
Nursery	0.0198	0.0842	0.0383	0.0915	0.0054	0.1097
Optdigits	0.0223	0.0065	0.0059	0.0121	0.0042	0.0115
Page-bl	0.0020	0.0112	0.0199	0.0091	0.0106	0.0073
Pendigits	0.0040	0.0049	0.0129	0.0043	0.0058	0.0038
Phoneme	0.0076	0.0302	0.0075	0.0221	0.0094	0.0117
Satimage	0.0034	0.0185	0.0065	0.0154	0.0078	0.0161
Segment	0.0075	0.0201	0.0150	0.0137	0.0126	0.0140
Shuttle	0.0010	0.0014	0.0127	0.0006	0.0019	0.0006
Sick	0.0081	0.0069	0.0120	0.0265	0.0069	0.0101
Texture	0.0194	0.0066	0.0069	0.0065	0.0091	0.0060
Waveform	0.0082	0.0180	0.0034	0.0222	0.0094	0.0157
Yeast	0.0118	0.0339	0.0064	0.0421	0.0093	0.0257
Zip	0.0052	0.0052	0.0093	0.0117	0.0071	0.0052

algorithm faster, no cross-validation is used for selecting the number of steps. Instead, we always perform two steps of the algorithm. This number of two steps was chosen as it was the most commonly chosen by the stop criterion in the previous experiments using medium to large datasets.

Results are shown in Table 8. The first noticeable fact is the scalability of our method. The execution times are very low even for the two datasets that have more than a million instances. Even for CHC the time spent is within moderate limits. If we

Table 7 Summary of data sets

Data set	Cases	Features			Classes	Variables	1-NN	
		C	B	N			Error	Time (s)
Census	299285	7	–	30	2	409	0.0743	8723.2
Covtype	581012	54	–	–	7	54	0.3024	16980.3
kddcup99	494021	33	4	3	23	118	0.0006	12649.4
kddcup991M	1000000	33	4	3	21	119	0.0002	37093.5
Poker	1025010	5	–	5	10	25	0.4975	35460.7

The features of each data set can be *C* continuous, *B* binary or *N* nominal. The variables column shows the number of variables, as it depends not only on the number of features but also on their type

Table 8 Testing error, storage requirements and execution time (in seconds) for our approach for huge problems

Dataset	Storage	Error	Time
Recursive Drop3			
Census	0.1205	0.1101	510.7
Covtype	0.1381	0.3969	213.6
kddcup99	0.0499	0.0179	534.6
kddcup991M	0.0337	0.0087	1249.0
Poker	0.0941	0.4977	305.5
Recursive ICF			
Census	0.6047	0.0621	188.5
Covtype	0.1179	0.4047	134.5
kddcup99	0.2870	0.0181	222.0
kddcup991M	0.2918	0.0010	746.0
Poker	0.0474	0.5114	280.0
Recursive CHC			
Census	0.1410	0.0814	4200.7
Covtype	0.1416	0.3323	9682.3
kddcup99	0.1427	0.0009	5308.0
kddcup991M	0.1436	0.0004	9847.0
Poker	0.1671	0.5358	26468.0

compare these values with the estimation we can get from Fig. 6 of the time Drop3, ICF or CHC would need, the advantage of our proposal is clearly stated.

Regarding the performance, Drop3 is specially efficient in reducing storage. It achieves a large reduction for the five datasets, without damaging the performance very significantly. In fact, the increment of the testing error is similar to the increment experimented by the original Drop3 algorithm when applied to the 30 datasets of the previous experiments. In terms of testing error, CHC is the best one, although the reduction it achieves is worse than Drop3. ICF performs worse than CHC and Drop3, both in terms of testing error and storage reduction. It is, however, the fastest one.

5 Conclusions and future work

In this paper we have presented a new method for scaling up instance selection algorithms. The method is applicable to any instance selection method without any modification. The method consists of a recursive procedure, where the dataset is partitioned into disjoint subsets, an instance selection algorithm is applied to each subset, and then the selected instances are rejoined to repeat the process.

Using three well-known instance selection algorithms, Drop3, ICF and a CHC genetic algorithm, we have shown that our method is able to match the performance of the original algorithms with a considerable reduction in execution time. In terms of reduction of storage requirements, our approach is even better than the use of the original instance selection algorithm over the whole dataset. Additionally, our method is straightforwardly parallelizable without modifications.

Furthermore, we have also shown that our approach is able to scale up to huge problems with hundreds of thousands of instances. Using 5 of those huge datasets our method is able to execute fast, achieving a very significant reduction of storage while keeping the testing error similar to the 1-NN error obtained using 10-fold cross-validation and the whole dataset.

As a main research line we are working on the adaptation of our approach to algorithms that not only select instances but also modify prototypes. There are also new algorithms that combine feature and instance selection at the same time. Our method should also have to be adapted to allow feature selection.

Acknowledgements This work was supported in part by the Project TIN2008-03151 of the Spanish Comisión Interministerial de Ciencia y Tecnología.

References

Anderson TW (1984) An introduction to multivariate statistical analysis. Wiley Series in Probability and Mathematical Statistics, 2nd edn. Wiley, New York

Baluja S (1994) Population-based incremental learning. Technical Report CMU-CS-94-163. Carnegie Mellon University, Pittsburgh

Banfield RE, Hall LO, Bowyer KW, Kegelmeyer WP (2005) Ensembles of classifiers from spatially disjoint data. In: Lecture notes in computer science, vol 3541. Springer, pp 196–205

Barandela R, Ferri FJ, Sínchez JS (2005) Decision boundary preserving prototype selection for nearest neighbor classification. Int J Pattern Recognit Artif Intell 19(6):787–806

Blum A, Langley P (1997) Selection of relevant features and examples in machine learning. Artif Intell 97:245–271

Brighton H, Mellish C (2002) Advances in instance selection for instance-based learning algorithms. Data Min Knowl Discov 6:153–172

Brodley CE (1995) Recursive automatic bias selection for classifier construction. Mach Learn 20(1/2): 63–94

Cano JR, Herrera F, Lozano M (2003) Using evolutionary algorithms as instance selection for data reduction in KDD: an experimental study. IEEE Trans Evol Comput 7(6):561–575

Cano JR, Herrera F, Lozano M (2005) Stratification for scaling up evolutionary prototype selection. Pattern Recognit Lett 26(7):953–963

Chaudhuri S, Motwani R, Narasayya V (1998) Random sampling for histogram construction: how much is enough? In: Haas L, Tiwary A (eds) Proceedings of ACM SIGMOD, international conference on management of data. ACM Press, New York, USA, pp 436–447

Chawla NW, Hall LO, Bowyer KW, Kegelmeyer WP (2004) Learning ensembles from bites: a scalable and accurate approach. J Mach Learn Res 5:421–451
Chen JH, Chen HM, Ho SY (2005) Design of nearest neighbor classifiers: multi-objective approach. Int J Approx Reason 40(1–2):3–22
Cochran W (1977) Sampling Techniques. Wiley, New York
Cover TM, Thomas JA (1991) Elements of Information Theory. Wiley series in telecommunication. Wiley, New York
Demšar J (2006) Statistical comparisons of classifiers over multiple data sets. J Mach Learn Res 7:1–30
Dietterich TG (1998) Approximate statistical tests for comparing supervised classification learning algorithms. Neural Comput 10(7):1895–1923
Eshelman LJ (1990) The CHC adaptive search algorithm: how to have safe search when engaging in non-traditional genetic recombination. Morgan Kauffman, San Mateo
García-Pedrajas N, del Castillo JAR, Ortiz-Boyer D (2008) A cooperative coevolutionary algorithm for instance selection for instance-based learning. Mach Learn (in review)
Goldberg DE (1989) Genetic Algorithms in Search, Optimization and Machine Learning. Addison–Wesley, Reading, MA
Hettich S, Blake C, Merz C (1998) UCI repository of machine learning databases. http://www.ics.uci.edu/~mlearn/MLRepository.html
Holland JH (1975) Adaptation in natural and artificial systems. The University of Michigan Press, Ann Arbor
Hussain F, Liu H, Tan C, Dash M (1999) Discretization: an enabling technique. Technical Report TRC6/99, School of Computing, National University of Singapore
Ishibuchi H, Nakashima T (2000) Pattern and feature selection by genetic algorithms in nearest neighbor classification. J Adv Comput Intell Intell Inform 4(2):138–145
Kivinen J, Mannila H (1994) The power of sampling in knowledge discovery. In: Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems. ACM Press, Minneapolis, Minnesota, USA, pp 77–85
Kuncheva L (1995) Editing for the *k*-nearest neighbors rule by a genetic algorithm. Pattern Recognit Lett 16:809–814
Li J, Manry MT, Yu C, Wilson DR (2005) Prototype classifier design with pruning. Int J Artif Intell Tools 14(1–2):261–280
Liu H, Motoda H (1998) Feature selection for knowledge discovery and data mining. Kluwer, Norvell
Liu H, Motoda H (2002) On issues of instance selection. Data Min Knowl Discov 6:115–130
Michalewicz Z (1994) Genetic algorithms + data structures = evolution programs. Springer, New York
Provost FJ, Kolluri V (1999) A survey of methods for scaling up inductive learning algorithms. Data Min Knowl Discov 2:131–169
Reeves CR, Bush DR (2001) Using genetic algorithms for training data selection in RBF networks. In: Liu H, Motoda H (eds) Instances selection and construction for data mining. Kluwer, Norwell, pp 339–356
Smith P (1998) Into Statistics. Springer, Singapore
Smyth B, Keane MT (1995) Remembering to forget. In: Mellish CS (ed) Proceedings of the fourteenth international conference on artificial intelligence, vol 1, Montreal, Canada, pp 377–382
Son SH, Kim JY (2006) Data reduction for instance-based learning using entropy-based partitioning. In: Proceedigns of the international conference on computational science and its applications—ICCSA 2006, number 3982 in Lecture Notes in Computer Science, Springer, pp 590–599
Whitley D (1989) The GENITOR algorithm and selective pressure. In: Publishers MK (ed) Proceedings of the 3rd international conference on genetic algorithms. Los Altos, CA, pp 116–121
Wilcoxon F (1945) Individual comparisons by ranking methods. Biometrics 1:80–83
Wilson DL (1972) Asymptotic properties of nearest neighbor rules using edited data. IEEE Trans Syst Man Cybern 2(3):408–421
Wilson DR, Martinez AR (1997) Instance pruning techniques. In: Fisher D (ed) Proceedings of the fourteenth international conference on machine learning. Morgan Kaufmann, San Francisco, pp 404–411
Wilson DR, Martinez TR (2000) Reduction techniques for instance-based learning algorithms. Mach Learn 38:257–286
Zhu X, Wu X (2006) Scalable representative instance selection and ranking. In: Proceedings of the 18th international conference on patter recognition (ICPR'06), vol 3. IEEE Computer Society, pp 352–355