Genetic Programming of Prototypes for Pattern Classification

Hugo Jair Escalante[1], Karlo Mendoza[2], Mario Graff[3], and Alicia Morales-Reyes[1]

[1] Computational Sciences Department, INAOE, Luis Enrique Erro 1, Puebla, 72840, Mexico
hugojair@inaoep.mx
[2] Graduate Program on Systems Engineering, UANL, San Nicólas de los Garza, N.L., 66451, Mexico
[3] Facultad de Ingenieria Electrica Universidad Michoacana de San Nicolas de Hidalgo, Mexico

Abstract. This paper introduces a genetic programming approach to the generation of classification prototypes. Prototype-based classification is a pattern recognition methodology in which the training set of a classification problem is represented by a small subset of instances. The assignment of labels to test instances is usually done by a 1NN rule. We propose a new prototype generation method, based on genetic programming, in which examples of each class are automatically combined to generate highly effective classification prototypes. The genetic program aims to maximize an estimate of the generalization performance of a 1NN classifier using the prototypes. We report experimental results on a benchmark for the evaluation of prototype generation methods. Experimental results show the validity of our approach: the proposed method outperforms most of the state of the art techniques when using both small and large data sets. Better results are obtained for data sets with numeric attributes only, although the performance of our method on mixed data is very competitive as well.

1 Introduction

Among the most popular pattern classification methods are those based on similarity estimation. This type of methods rely on similarity measures to assign labels to new objects, a representative classifier of this methodology is KNN. Similarity-based methods have reported outstanding results on classical pattern classification tasks, including handwritten digit recognition and text categorization. However, despite their acceptable performance, they require computing similarity estimates with the whole objects when a new instance needs to be classified, which can be computationally expensive. Besides, this type of methods require large resources for storage and they can be sensitive to noisy instances.

Prototype-based classifiers aim at alleviating the computational cost by using only a subset of representative instances for classification instead the whole training set of objects. In this way, only a small number of similarity estimations have to

J.M. Sanches, L. Micó, and J.S. Cardoso (Eds.): IbPRIA 2013, LNCS 7887, pp. 100–107, 2013.
© Springer-Verlag Berlin Heidelberg 2013

be computed. Thus, the main goal of prototype-based classifiers is to achieve comparable performance to methods that use the whole data set of instances, while reducing the computational cost of classification. The key issue in prototype-based classification is that of determining what are the prototypes to be used for classification. There are two main alternatives for solving this problem: selection and generation of prototypes. In the former approach, a subset of the whole objects is selected to form the set of prototypes [8]. The second approach, consists of generating a subset of representative instances by using information in the data set of objects [9]. This paper focuses in the latter formulation.

A wide diversity of prototype generation methods have been proposed so far, Triguero et al. have summarized recently the most representative techniques [9]. Representative methods generate prototypes iteratively, starting from a set of points in the input space that are updated. Most of these methods start from randomly generated prototypes, while others use a subset of the training instances as initial prototypes. Initial prototypes are modified in different ways with the goal of maximizing criteria related to the (training) classification accuracy [1,2,3,7,9]. Whereas acceptable results have been reported with such approaches we think that prototype generation methods must better exploit the available labeled instances from the training set, as current approaches only either use training instances to evaluate the performance of prototypes or use (some) training instances to initialize the generation process. Besides, in most prototype generation techniques users must specify the number of prototypes they want to select. This implies the user must have some knowledge regarding the classification problem being addressed, limiting the applicability of these methods.

We propose a genetic programming approach to the prototype generation problem. The proposed method combines instances from the original data set to generate prototypes. The combination strategy is automatically determined via genetic programming, where the genetic program aims at generating prototypes that maximize generalization performance (using a hold-out cross-validation estimate) of an 1NN classification rule. The proposed strategy is able to automatically select the number of prototypes per class, thus no requiring input from the user in this aspect. We report experimental results obtained with the proposed strategy in a suite of benchmark pattern classification problems [9]. The considered benchmark allows us to compare the performance of our approach to that of the most representative prototype generation methods. In terms of accuracy and percentage of data set reduction, the method outperforms most alternative approaches when using both small and large data sets. Despite its effectiveness, the intuitive idea behind our method is very simple and there are many ways in which our approach can be extended in such a way that better prototypes can be obtained.

2 *GP*^{2}*:* Genetic Programming of Classification Prototypes

The proposed method, called GP^2, automatically combines instances from a particular class to generate classification prototypes for that class. The combination

strategy is determined by a genetic program that aims at maximizing an estimate of the generalization performance of an 1NN classifier. Although the prototypes of a class are determined only by examples of its class, under the proposed approach the prototypes for all of the classes associated to the classification problem are dependent (the fitness function evaluates sets of prototypes). Hence making prototypes suitable for discrimination.

Let $T = \{(\mathbf{x}_1, y_1) \dots, (\mathbf{x}_N, y_N)\}$ be a training set of labeled instances, with $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \mathcal{C} = \{1, \dots, K\}$, where d is the dimensionality of the problem and K is the number of classes in the considered problem. Our goal is to build a set of prototypes $P = \{(\mathbf{w}_1, y_1) \dots, (\mathbf{w}_L, y_L)\}\}$, such that $L << N$, where $\mathbf{w}_i \in \mathbb{R}^d$ and there is at least one prototype associated to each class in $\mathcal{C}$.

In order to determine the set of prototypes P, we propose a genetic programming formulation where each individual j of the population is itself a set of prototypes P_j, and where each prototype in P_j is build with combinations of training examples of a single class. That is, we assume that if the label associated to $\mathbf{w}_i$ is $y_i = r$, then $\mathbf{w}_i$ is obtained by combining instances in T labeled with r. Each prototype is represented by a tree structure, as depicted in Figure 1, thus an individual is formed by a set of trees (each tree associated to a single prototype). The nodes of the tree can contain either operators (e.g., $+, -, \max$, etc.) by which instances are combined, or the instances themselves ($\mathbf{x}_i \in T$). When a tree is evaluated it generates a prototype, which is a point in the $d-$dimensional space obtained by the combination of other points in $\mathbb{R}^d$. We evolve a population of such forest-like individuals using a standard evolutionary computation approach and select the best individual after a number of generations.

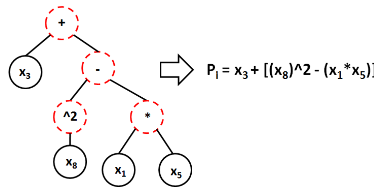

Fig. 1. Representation for prototypes in GP^2

The pseudocode of the proposed method is shown in Algorithm 1. A population of individuals is randomly generated using a standard *Ramped-Half-and-Half* strategy [5]. Each individual is generated in such a way that the same number of initial prototypes is considered for each of the classes, where the initial number of prototypes is a parameter fixed by the user[1]. After initialization the evolutionary process starts. We evaluate the fitness of individuals, and apply genetic operators with certain probabilities until a maximum number of generations are reached.

[1] One should note that the considered operators allow the GP^2 approach to automatically determine the number of prototypes that will be considered per each class.

Algorithm 1. Genetic programming algorithm used by GP^2.

Require: D : Development and V : validation data sets; g : number of iterations;
$\mathcal{P} \leftarrow$ Initialize population with s individuals;
$i \leftarrow 0$; $P^* = []$; $f^* = -Inf$;
while $i \leq g$ **do**
  $i \leftarrow i + 1$;
  for $j = 1 \rightarrow s$ **do**
    $f(j) \leftarrow fitness(P_j)$
    if $f(j) \geq f^*$ **then**
      $f^* \leftarrow f(j)$;
      $P^* \leftarrow P_j$;
    end if
  end for
  Apply selection operator;
  $\mathcal{P} \leftarrow$ Apply large cross-over operator with probability P_l;
  $\mathcal{P} \leftarrow$ Apply cross-over operator with probability P_s;
  $\mathcal{P} \leftarrow$ Apply Karyokinesis operator with probability P_k;
  $\mathcal{P} \leftarrow$ Apply Mutation operator with probability P_k;
end while
return Individual P^*, the one with lowest fitness value;

We considered the following operators: $\{+, -, *, \div, cosine, sin, tan, \mathbf{x}^2, \mathbf{x}^3, minT, Abs\}$. For the evolutionary process we considered traditional genetic operators[2] and also we propose a new operator specifically designed for prototype generation called *mitosis*. The considered genetic operators are the following (recall an individual is composed by a set of trees, one per prototype, and that each tree is associated to a single class):

- **Large crossover [1].** Two individuals of the current generation are selected via tournament. From both selected individuals we randomly select one of the prototypes composing the individuals. Next, all of the prototypes at the right of the selected prototype in the first individual are moved to the left side (from the selected prototype) of the second individual, and viceversa. Trees within and individual are sorted in lexicographical order.
- **Crossover.** Two individuals are selected with tournament and a class is randomly selected. A subset of trees (i.e., prototypes) associated with the selected class are interchanged between the selected individuals.
- **Mutation.** We implement the point-mutation, where an individual is randomly selected, and then a node of the selected individual is also randomly selected. The value of the selected node is modified (if the node has an operator it is replaced by another operator, if the node is associated to an instance, the instance is replaced by another instance).
- **Mitosis.** We proposed this operator to correct prototypes that are very close to instances from multiple classes. The underlying idea is to split a prototype into two other prototypes associated to two different classes, when the initial prototype

[2] We preliminary evaluated several genetic operators and combinations of them, here we report the combination with which we obtained the best results.

is making too many mistakes. We select an individual with a tournament process. Next we estimate the false-positives rate per each prototype in the selected individual, the prototype with the highest error rate is selected to apply the mitosis operator. Next, we identify the training instances that are associated to the selected prototype (according to the 1NN rule) for each of the classes. The instances labeled with the two classes with more hits by the prototype are used to generate two new prototypes. The new prototypes are generated by adding the centroid of instances of each class plus the difference vector between the centroids, see [6].

We use as fitness function to the accuracy obtained by a 1NN classifier using the set of prototypes (i.e., trees) associated to an individual. Clearly, if we attempt to maximize accuracy in the training set we would end up with an overfitted set of prototypes in a certain number of iterations. Instead, we split the available data into development and validation subsets. The development subset will be used to find generate prototypes and the validation data set will be used to estimate the performance of prototypes. Since data may be limited we would like to use all of the available training data to generate the prototypes. Therefore, we propose to run the genetic programming approach several times in order to generate multiple sets of prototypes. Each run we use a different partition of development and validation data, which makes the prototypes effective for classifying different partitions of the training set. The proposed approach for the generation of prototypes is described in Algorithm 2. All of the parameters of the proposed approach were fixed empirically using a hold-out approach, see [6].

Algorithm 2. Overview of the *(GP^2)* algorithm.

Require: Training data set $T = \{(\mathbf{x}_1, y_1) \ldots, (\mathbf{x}_N, y_N)\}$;
  Set $\mathcal{S} = \{\}$;
  for $i = 1 \rightarrow k$ **do**
    Split training data (T) into development (D) and validation (V) sets;
    {where $T = D \cup V$, $D \cap V = \emptyset$;}
    $\mathcal{U} \leftarrow GPGP(D, V)$;
    $\mathcal{S} = S \cup U$
  end for
  return $\mathcal{S}$

3 Experiments and Results

We report experimental results obtained by the GP^2 method using a suite of benchmark data sets for the evaluation of prototype generation techniques. In [9], Triguero et al. proposed a taxonomy of the most representative methods for prototype generation proposed so far and presented a comparative study of these techniques, a total of 24 prototype generation methods were considered. For the comparative study, Trigero et al. used a 59 data sets associated to different classification problems. The number of classes was between 2 and 28, the number of instances between 101 and 12, 960, and the number of features ranged between

5 and 180. The considered data sets were split into small (40 data sets, data sets with less than 2000 instances) and large data sets (19 data sets). The data sets comprised both numeric-valued and nominal attributes. The benchmark allows us to evaluate the performance of our method under different settings and to compare its performance to state of the art approaches for prototype generation.

For our experiments we used all of the data sets as provided by Triguero et al. as well as the evaluation methodology they proposed. Specifically, we performed 10-fold cross validation (using exactly the same partitions used in [9]). In each experiment we ran Algorithm 2 using the training partition ($9-$folds) and the selected prototypes were evaluated in the test partition, we repeated this process 10 times, changing the test partition. We report the average accuracy (obtained in the test partitions) per data set and the percentage of data reduction in the training set.

Table 1 presents a summary of the obtained results in terms of test accuracy and data set reduction. For comparison, we show the average performance of the best methods in terms of accuracy (GENN[3]) and data set reduction (PSCSA[4]), as reported in [9], as well as the performance of 1NN. We can see from this table that GP^2 offers the best tradeoff between accuracy and data set reduction. GENN obtained the best performance over all, although it was the worse in terms of data set reduction. On the other hand, PSCSA achieved the highest data set reduction rates, but its performance was the worst among the considered methods. Our approach on the other hand, obtained acceptable performance in terms of both accuracy and data set reduction. In terms of accuracy, our method obtained better results in the large data sets, see Figure 2. This is a positive result, since for the prototype generation methods target large data sets mainly, because we want to make more efficient the classification process.

Figure 3 shows a plot comparing the accuracy and reduction effectiveness of our method and that obtained with the 24 methods considered in the study of Triguero et al. [9]. We can clearly see that our method achieved the best tradeoff in large data sets and still obtained decent performance in small data sets.

Table 1. Average performance and percentage of reduction obtained by GP^2 for all of the data sets, also we show the performance obtained in small and large sets. For reference we show the performance obtained by GENN [4], PSCSA [2] and 1NN.

	Test accuracy			**Training set reduction**		
Measure	**All**	**Small**	**Large**	**All**	**Small**	**Large**
GP^2	75.07%±5.02	72.13%±6.73	81.23%±1.41	83.29%	76.69%	97.17%
GENN	78.48%±18.57	75.64%±15.45	81.33%±21.70	17.19%	18.62%	15.76%
PSCSA	66.94%±20.39	66.82%±18.74	67.07%±22.05	99.23%	98.58%	99.88%
1NN	77.04%±19.44	73.48%±16.64	80.60%±22.24	0%	0%	0%

[3] GENN is an editing method that analyzes the label associated to neighboring instances in order to eliminate or edit the label of instances [4].

[4] PSCSA implements an artificial-immune-system heuristic which starts selecting a single instance per class as prototype[2].

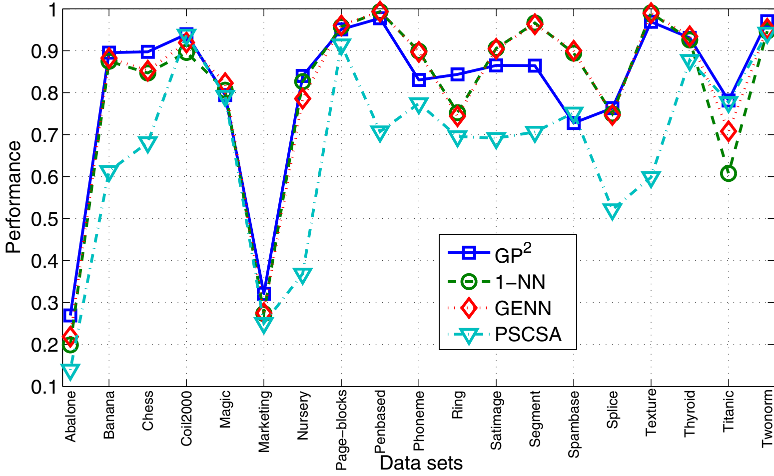

Fig. 2. Accuracy obtained by the proposed method in the large data sets

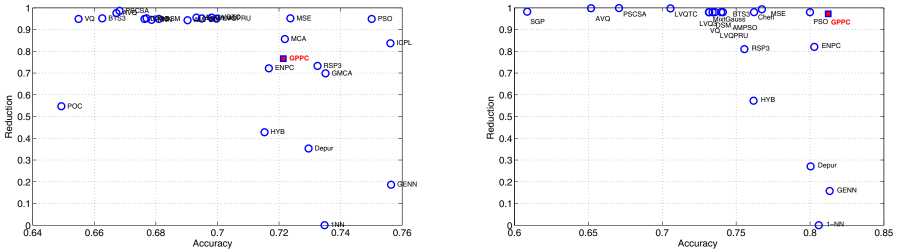

Fig. 3. Percentage of training set reduction ($y-$axis) vs test set accuracy, we report averages for the 24 methods considered in [9] for the small (left) and large (right) data sets.

Table 2. Average and standard deviation of accuracy obtained by the GP^2 when using different attirbutes

Measure	All	Small	Large
Nominal	69.73%±5.36	68.53%±4.19	74.02%±1.19
Numerical	78.48%±4.80	79.63%±8.80	83.80%±0.36

Table 2 shows the performance, in terms of accuracy, obtained by GP^2 separating data sets with numeric-only and mixed (numeric+nominal) valued attributes. Again, the best performance of our method was obtained in large data sets. Better results were obtained in data sets including only numeric attributes. This is a somewhat expected result as the optimization process of GP^2 generates attributes combining training instances, which makes sense in Euclidean spaces but not in spaces having nominal attributes.

4 Conclusions

We introduced GP^2 a method for the generation of classification prototypes. GP^2 uses genetic programming to find a set of prototypes by combining instances labeled with the same class. The goal of GP^2 is to maximize accuracy estimates obtained with a hold-out strategy. We report experimental results obtained with our method in benchmark data. Experimental results reveal that the proposed method offers beeeter tradeoff between accuracy and reduction than 24 other methods for prototype generation. Better results were obtained in large data sets with numeric-valued attributes. Although the performance of our method was also competitive in small data sets with mixed attributes. Future work includes a complete study on the performance of GP^2 under different parameter settings. Also we would like to devise strategies to make the GP^2 approach more efficient.

Acknowledgements. This work was supported by PROMEP under grant 103.5/07/2523 and by PAICYT-UANL program grant number 172-2010. Karlo M. Mendoza was supported by CONACyT.

References

1. Cordella, L.P., De Stefano, C., Fontanella, F., Marcelli, A.: Looking for prototypes by genetic programming. In: Zheng, N., Jiang, X., Lan, X. (eds.) IWICPAS 2006. LNCS, vol. 4153, pp. 152–159. Springer, Heidelberg (2006)
2. Garain, U.: Prototype reduction using an artificial immune system. Pattern Analysis and Applications 11(3-4), 353–363 (2008)
3. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning. Springer Series in Statistics. Springer New York Inc., New York (2001)
4. Koplowitz, J., Brown, T.: On the relation of performance to editing in nearest neighbor rules. Pattern Recognition 13(3), 251–255 (1981)
5. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer (2001)
6. Mendoza, K.M.: Programación genética para la generación automática de prototipos. Master's thesis, UANL, San Nicolas de los Garza, N. L., Mexico (2012)
7. Nanni, L., Lumini, A.: Particle swarm optimization for prototype reduction. Neurocomputing 72(4-6), 1092–1097 (2008)
8. Olvera, A., Carrasco-Ochoa, J.A., Martinez-Trinidad, J.F., Kittler, J.: A review of instance selection methods. Artificial Intelligence Reviews 34, 133–143 (2010)
9. Triguero, I., Derrac, J., García, S., Herrera, F.: A taxonomy and experimental study on prototype generation for nearest neighbor classification. IEEE Transactions on Systems, Man, and Cybernetics–Part C 42(1), 86–100 (2012)