Prototype Generation for Nearest Neighbor Classification: Survey of Methods

Isaac Triguero, Joaquín Derrac, Salvador García, and Francisco Herrera

Abstract

Prototype generation techniques have arisen as very competitive methods for enhancing the nearest neighbor classifier through data reduction. A great number of methods tackling the prototype generation problem have been proposed in the literature.

This technical report provides a survey of the most representative algorithms developed so far. A previously proposed categorization has been used to present them and describe their main characteristics, thus providing a first insight into the prototype generation field which may be useful for every practitioner who needs a quick reference about the existing techniques and their particularities.

Index Terms

Prototype generation, nearest neighbor, taxonomy, classification.

I. Introduction

The nearest neighbor (NN) rule is one of the most successfully used techniques for resolving classification and pattern recognition tasks. Despite its high classification accuracy, this rule suffers from several shortcomings in time response, noise sensitivity and high storage requirements. These weaknesses have been tackled from many different approaches, among them, a good and well-known solution that we can find in the literature consists of reducing the data used for the classification rule (training data). Prototype reduction techniques can be divided into two different approaches, known as prototype selection and prototype generation or abstraction. The former process consists of choosing a subset of the original training data, whereas prototype generation builds new artificial prototypes to increase the accuracy of the NN classification. This technical report provides a survey of the prototype generation methods proposed in the literature.

II. Survey of Prototype Generation Algorithms

Prototype generation builds new artificial examples from the training set, a formal specification of the problem is the following: Let $\mathbf{x}_p$ be an instance where $\mathbf{x}_p = (\mathbf{x}_{p1}, \mathbf{x}_{p2}, ..., \mathbf{x}_{pm}, \mathbf{x}_{p\omega})$, with $\mathbf{x}_p$ belonging to a class ω given by $\mathbf{x}_{p\omega}$ and a m-dimensional space in which X_{pi} is the value of the i-th feature of the p-th sample. Then, let us assume

I. Triguero, J. Derrac and F. Herrera are with the Department of Computer Science and Artificial Intelligence, CITIC-UGR (Research Center on Information and Communications Technology), University of Granada, 18071 Granada, Spain.
E-mails: triguero@decsai.ugr.es, jderrac@decsai.ugr.es, herrera@decsai.ugr.es

S. García is with the Department of Computer Science, University of Jaén, 23071, Jaén, Spain.
E-mails: sglopez@ujaen.es

that there is a training set TR which consists of n instances $\mathbf{x}_p$ and a test set TS composed of s instances $\mathbf{x}_t$, with $\mathbf{x}_{t\omega}$) unknown. The purpose of prototype generation is to obtain a prototype generate set TG, which consists of r, $r < n$, prototypes, which are either selected or generated from the examples of TR. The prototypes of the generated set are determined to represent efficiently the distributions of the classes and to discriminate well when used to classify the training objects. Their cardinality should be sufficiently small to reduce both the storage and evaluation time spent by a NN classifier.

Based on [1], prototype generation algorithms can be classified into four main groups depending on the generation mechanism adopted: Positioning Adjustment, Class re-labeling, Centroid based and Space Splitting.

III. Positioning Adjustment

The methods that belong to this family aim to correct the position of a subset of prototypes from the initial set by using an optimization procedure. New positions of prototype can be obtained using the movement idea in the m-dimensional space, adding or subtracting some quantities to the attribute values of the prototypes. This mechanism is usually associated to a fixed or mixed type of reduction.

- **Learning Vector Quantization 3 (LVQ3)** [2]

  Learning Vector Quantization can be understood as a special case of artificial neural network in which a neuron corresponds to a prototype and a competition weight based is carried out in order to locate each neuron in a concrete place of the m-dimensional space to increase the classification accuracy [2].

  In the initialization step, the prototypes are placed within the training set, by maintaining the same number of representatives in each class. Then, let $\mathbf{x}_p \in TR$ be an input sample, let $\mathbf{y}_q$ be the nearest codebook vector to $\mathbf{x}_p$, and let $\mathbf{y}_q$(t) represent the codebook vector $\mathbf{y}_q$ at the step t. The learning process in the basic version of the LVQ, i.e. the LVQ1 algorithm, consists of updating the position of $\mathbf{y}_q$.

  If the class label of the codebook vector $\mathbf{y}_q$ matches the class label of the training instance $\mathbf{x}_p$, then the codebook vector is moved towards $\mathbf{x}_p$. Otherwise, it is moved away from the given input sample. The modifications to the codebook vector $\mathbf{y}_q$ are performed according to the following general rule:

$$\begin{cases} \mathbf{y}_q(t+1) = \mathbf{y}_q + \alpha(t)[\mathbf{x}_p(t) - \mathbf{y}_q(t)], \text{ if } class(\mathbf{x}_p) == class(\mathbf{y}_q) \\ \mathbf{y}_q(t+1) = \mathbf{y}_q - \alpha(t)[\mathbf{x}_p(t) - \mathbf{y}_q(t)], \text{ if } class(\mathbf{x}_p) != class(\mathbf{y}_q) \end{cases} \tag{1}$$

  where $0 < \alpha(t) < 1$ denotes the corresponding learning rate, which may be either constant or decrease monotonically with time.

  In the case of LVQ2, two codebook vectors, $\mathbf{z}_p$ and $\mathbf{r}_q$, the nearest neighbors to an input sample $\mathbf{x}_p$, are updated simultaneously. While $\mathbf{z}_p$ must belong to the correct class, $\mathbf{r}_q$ must belong to a wrong class. Furthermore, $\mathbf{x}_p$ must fall into a window defined around the mid-plane of $\mathbf{z}_p$ and $\mathbf{r}_q$. The positioning adjustment in the LVQ2 algorithm can be expressed as follows:

$$\begin{cases} \mathbf{z}_p(t+1) = \mathbf{z}_p(t) + \alpha(t)[\mathbf{x}_p(t) - \mathbf{z}_p(t)] \\ \mathbf{r}_q(t+1) = \mathbf{r}_q - \alpha(t)[\mathbf{x}_p(t) - \mathbf{r}_q] \end{cases} \tag{2}$$

The LVQ3 approach is similar to the LVQ2 with the addition of one restriction. If $\mathbf{x}_p$, $\mathbf{z}_p$ and $\mathbf{r}_q$ belong to the same class, both are positive corrected in the following way:

$$\begin{cases} \mathbf{z}_p(t+1) = \mathbf{z}_p(t) + \epsilon\alpha(t)[\mathbf{x}_p(t) - \mathbf{z}_p(t)] \\ \mathbf{r}_q(t+1) = \mathbf{r}_q + \epsilon\alpha(t)[\mathbf{x}_p(t) - \mathbf{r}_q] \end{cases} \tag{3}$$

where ϵ is a constant which ranges between 0.1 and 0.5 and depends on the size of the window.

The algorithm can be briefly summarized as follow:

1) t=0
2) TG(t) = Extract N Instances($\mathbf{x}_p$, TR) (* learning stage *)
3) repeat
4) x = ExtractOneInstance(TR)
5) TG(t + 1) = TG(t) + x
6) t=t+1
7) until (t = total learning steps)

- **Decision Surface Mapping (DSM)** [3]

The adaptive decision surface mapping algorithm is a variation of the LVQ method in which the authors dropped the requirement that the prototypes reflect the probability distribution of the classes. Instead, the algorithm modifies the location of the prototypes to closely map the decision surface separating classes.

The DSM algorithm starts by randomly selecting a small subset of prototype from TR, keeping the proportion of prototypes per class. In this variant, when a sample $\mathbf{x}_p \in TR$ is of the same class as the nearest prototype (codebook), no modification are applied. However, when misclassification occurs, modification take place to apply both punishment and reward. The punishment step takes the nearest neighbor ($\mathbf{y}_q$) and moves it away from $\mathbf{x}_p$ in this way:

$$\mathbf{y}_q(t+1) = \mathbf{y}_q - \alpha(t)[\mathbf{x}_p(t) - \mathbf{y}_q(t)] \tag{4}$$

The reward step search for the nearest correct prototype ($\mathbf{y}_c$) and moves it towards $\mathbf{x}_p$:

$$\mathbf{y}_c(t+1) = \mathbf{y}_c + \alpha(t)[\mathbf{x}_q(t) - \mathbf{y}_c(t)] \tag{5}$$

where $0 < \alpha(t) < 0.3$ denotes the corresponding learning rate, which may be either constant or decrease monotonically with time.

Furthermore, if there is no misclassification during a fixed number of iterations, the algorithm finishes.

- **Vector Quantization (VQ)** [4]

  The VQ for non-parametric data reduction algorithm combines the VQ technique and the NN method. It can be described as follows:

  1) For each class i, (i=1,..,Number Of Classes), find the M_i level optimal quantizers using a LVQ algorithm. Suppose the final quantiser reproduction symbols are $A_i = y_j^i; j = 1, ..., M_i$;
  2) Combine all the reproduction alphabets A_i of all classes into one single set A of M vectors, that is, $U = \cup_i A_i$ and $M = \sum_i M_i$. Output A as the set of final prototypes (TG).

- **Learning Vector Quantization with Training Counter (LVQTC)** [5]

  Learning vector quantization with training count (LVQTC) represents a modification of the original LVQ scheme, where additional attributesare appended to each prototype.

  Each prototype has a probability P_i, $i = 1..NumberofClasses$, per class. This probability stores the number of times vectors of that class have trained the prototype (training counters). Furthermore, each prototype knows the centroids of wrong classes.

  The initial TG is randomly selected from the TR with a small fraction (About 5% or less), keeping the class distributions.

  For each iteration, the following steps are taken:

  1) Prototype resetting: set all training counters to zero.
  2) Prototype training: An LVQ stage is performed. In this stage, the training counter P_i is appropriately incremented in each iteration. The punishment and reward steps as performed as:

$$\begin{cases} \mathbf{y}_q(t+1) = \mathbf{y}_q + (\alpha_r(t)/P_tot)[\mathbf{x}_p(t) - \mathbf{y}_q(t)], \text{ if } class(\mathbf{x}_p) == class(\mathbf{y}_q) \\ \mathbf{y}_q(t+1) = \mathbf{y}_q - (\alpha_w(t)/P_tot)[\mathbf{x}_p(t) - \mathbf{y}_q(t)], \text{ if } class(\mathbf{x}_p) != class(\mathbf{y}_q) \end{cases} \tag{6}$$

  where α_r and α_w are two distinct learning parameters, and P_tot=$P_1 + P_2 + ... + P_numberClasses$

  3) Prototype pruning and creation: Before a new iteration is started, prune under-trained prototypes and create new prototypes are applied, according to the following rules:

     a) Pruning: Eliminate prototypes with:

$$P_tot = P_1 + P_2 + ... + P_numberClasses < P_prn \tag{7}$$

        where P_prn is a user-modifiable parameter.

     b) Creation: Let us denote by P_w the maximum value of the wrong class training counters of the prototype; if P_w ¿ P_prn, create a prototype of a class whose training counter $P_i = P_w$ and with a

prototype equal to the original.

- **MSE (MSE)** [6]

  This algorithm is similar to the LVQ approach. In this approach, the initialization strategy is modified to generate a "reasonable" initial number of prototypes in TG. This strategy operates in three phases:

  1) A standard C-means is performed separately on each class, with a large number of training instances randomly chosen as initial centroids (10 or 20 per class).
  2) A edited nearest neighbor rule is applied to discard non-representative prototypes.
  3) A second elimination rule is used to discards redundant prototypes (the Van de Merckt rule [7]).

  Then the algorithm enters into a optimization process of the intialized prototypes of TG. The probability that a prototype $\mathbf{x}_p$ belongs to each cluster is defined as follows:

$$Z_i(X) = \frac{exp[-d_i^2/T]}{\sum_k exp[-d_k^2/T]} \tag{8}$$

  where T is the temperature parameter that decrease during the training, d_i is the Euclidean distance between the prototypes $\mathbf{y}_q$ and $\mathbf{x}_p$.

  The desired probabilities for the proposed model should be:

$$Z_i^*(X) = \begin{cases} 0 & \text{si } i \text{ no pertenece a C(x)} \\ \frac{exp[-d_i^2/T]}{\sum_k exp[-d_k^2/T]} & \text{si } i \text{ pertenece a C(x)} \end{cases} \tag{9}$$

  It defines the cost function, which determines the training process, and it is presented as the differences between the observed and desired probabilities:

$$E(X) = \frac{1}{2} T \sum_i (Z_i^*(X) - Z_i(X))^2 \tag{10}$$

  Finally, the optimization process is performed through an adaptive gradient procedure which minimizes the error function.

$$\mathbf{y}_q(t+1) = \mathbf{y}_q(t) + \Delta \mathbf{y}_q(t+1) \tag{11}$$

$$\Delta \mathbf{y}_q(t+1) = -\eta(t+1) \nabla_i E(\mathbf{x}_p(t+1)) + \mu \Delta \mathbf{y}_q(t) \tag{12}$$

  where μ is usually fixed to 0.9, $\eta(t)$ is a gradient step which decrease over time, and $\nabla_i E$ is the gradient of E with respect to $\mathbf{y}_q$.

  The algorithm can be outlined as follows:

  1) Choose an arbitrary set of patterns in each class as an initial set of prototypes.

2) Apply the initialization strategy.
3) Set gradient $\eta(0)$ at a low value and temperature T at a high one.
4) Make one iteration on the whole training set.
5) Decrease $\eta(t)$. If T ¿ T_min, decrease T
6) If the process has converged go to step 7. Otherwise go to step 4.
7) Detect and eliminate inactive prototypes.

- **Hybrid LVQ3 algorithm (HYB)** [8]

  This constitutes a hybridization of several prototype reduction techniques. Concretely, HYB essentially consists of two steps. First, initial prototypes are selected by a Support Vector Machines method. After this selection phase, HYB invokes a LVQ3 phase and executes a search in order to find the optimal positions [8].

  1) For every class, j, select an initial condensed prototype set $Y_{j,Test}$ by using any one of the reduction methods described earlier, and the entire training sets, $T_{i,t}$;
  2) Using $Y_{j,Test}$ as the set of condensed prototype vectors for class j, do the following using the Placement sets, $T_{i,p}$, and the Optimizing sets, $T_{i,0}$ for all the classes:
     a) Perform LVQ3 using the points in the Placement set, $T_{i,p}$. The parameters of the LVQ3 are spanned by considering increasing values of w from 0.0 to 0.5, in steps of $\triangle$w . The sets $Y_{j,Test}$ (for all j) and Y_{Test} are updated in the process. Select the best value w_0 after evaluating the accuracy of the classification rule on $T_{i,0}$, where the NN classification is achieved by the adjusted Y_{Test}.
     b) Perform LVQ3 using the points in the Placement set, Ti,P. The parameters of the LVQ3 are again spanned by considering increasing values of ϵ from 0.0 to 0.5, in steps of $\triangle$w. The sets $Y_{j,Test}$ (for all j) and Y_{Test} are updated in the process. Select the best value ϵ_0 after evaluating the accuracy of the classification rule on TR, where the NN classification is achieved by the adjusted Y_{Test};
     c) Repeat the above steps with the current w_0 and ϵ_0, till the best values w_* and ϵ_* are obtained;
  3) Determine the best prototype set Y_Final by invoking LVQ3, X times, with the data in $T_{i,P}$, and where the parameters are w_* and ϵ_*. Again, the pseudo-testing is achieved using the Optimizing set, TR

- **Evolutionary Nearest Prototype Classifier (ENPC)** [9]

  The ENPC algorithm is a genetic-based technique which generate an appropriate TG through an evolutionary process. Generally, genetic algorithms are very dependent on some crucial parameters that have to be found by a trial and error process. One of the most important characteristics of this method is the lack of initial conditions, ENPC does not need any parameter.

  It employs as a basic structure a two-dimensional matrix, where each row is associated with a prototype of the whole classifier, and each column is associated with a class to define regions where the prototypes are

mapped. On its initialization, the algorithm only defines one prototype and it is introduced to TG. All the training instances are assigned to its regions, depending on their class attribute.

Then a whole evolutionary process starts: It carries out sequentially a set of evolutionary operations with the aim of generating a robust set of prototypes which will be able to correctly generalize the instances of the train set. The evolutionary operators defined are:

1) Mutation: TG suffers a mutation phase where prototypes are re-labeled with the most populate class in each region of the search space.
2) Reproduction: In order to refine TG, the reproduction operator function is to introduce new prototypes into the classifier, splitting the instances assigned to a prototype into two sets, where the second set is assigned to a new prototype. Each prototype $\mathbf{x}_p$ from the TG has the opportunity of introducing a new prototype to increase its own quality with the objective of defining different regions.
3) Fight: This operator allows prototypes to exchange their assigned instances. The fight can be performed in a cooperative or a competitive way, and it is ruled by the quality of the prototypes involved. This step does not modify the prototypes of TG it is only used to define again the different groups that we can find in TG.
4) Movement: When the different regions have been established, ENPC executes the move operator. It consists in relocating each prototype in the best expected place, i.e. it is moved to the centroid of its region.
5) Die: The current prototypes have a chance of being erased from the matrix, which is inversely proportional to its quality, removing some prototypes from TG that are not relevant.

In the whole process, the quality of each prototype is defined by the number of prototypes which it has currently assigned and its classification accuracy. When the evolutionary process is finished, the generated set of prototypes is employed to classify the tests set, by means of the 1-NN classifier.

- **Adaptive Vector Quantization (AVQ)** [10]

This method follows an incremental approach to build TG. It proceeds as:

1) With w_ij denoting the jth prototype of the ith class, perform clustering independently for each class of the training set by assigning every training sample to the nearest prototype of the same class.
2) Compute the number of ith class training samples whose closest ith class prototype is w_ij. Denote this number as R_ij.
3) Use the NN decision rule to classify TR by associating every sample to its nearest prototype regardless of the class distinction.
4) Compute the number of ith class training samples whose closest prototype is w_ij . Denote this number as Q_ij.
5) Denote the difference between R_ij and Q_ij as E_ij.

6) Denote the prototype that has the largest value of E_ij as w^* and find the training samples that were assigned to w^* in the clustering process. These training samples are then divided into two clusters by using the LBG method [11].
7) Increase the number of prototypes by one by replacing w^* with the two newly generated cluster centers.

- **Learning Vector Quantization with Pruning (LVQPRU)** [12]

  The LVQPRU approach is described as follows,

  1) Let N_pc be the number of prototypes per class.
  2) Randomly initialize TG with a DSM algorithm, obtaining N_pc*Number of Classes prototypes. Denote the number of instances closest to the kth prototype as $N_v(k)$, where $1 <= k <= size(TG)$.
  3) Delete the kth prototype if it does not contain any members (empty prototypes), i. e. $N_v(k)$ = 0, and decrease the total number of prototypes.
  4) Change the class label of a prototype if it disagrees with the plurality of the instances closest to it.
  5) Use LVQ2.1 to refine the locations of the prototypes.
  6) Apply the basic condensing algorithm (CNN) to the remaining prototypes to remove the internal ones.
  7) Prune the prototype whose removal causes the least increase in classification error.
  8) Use LVQ2.1 to fine-tune the positioning of the remaining prototypes.
  9) Return TG.

- **Adaptive Michigan Particle Swarm Optimization (AMPSO)** [13]

  AMPSO is a PSO algorithm with a particular way of encoding the solution into the system. Its particles encode a prototype, each one being the generated train data represented as the whole particle swarm. Next, PSO rules guide the optimization of the positioning. At each iteration, the position of each prototype is updated with the movement equations, which are neighborhood-based. This method does not have a fixed number of particles. On the contrary, some new operations are defined to allow the PSO search procedure to increase or decrease dynamically the number of particles; thus the number of prototypes generated by the algorithm is not known until the end of its execution. The algorithm employs two different fitness functions: The global fitness function, defined by the standard classification accuracy on a 1-NN classifier, which is used to find the best swarm over the whole PSO procedure; and a local fitness function valued in each particle, defined by using the number of prototypes correctly classified and misclassified by itself. This secondary fitness function is used to evaluate the quality of each particle, in order to judge if it must be erased from the swarm, or if it can be employed as a parent of a new particle. When the whole PSO process has finished, a cleaning process is carried out on the best swarm found, erasing from the prototypes defined by the particles the redundant ones. Then, the set of prototypes generated is taken as the output of the algorithm.

The pseudocode of the AMPSO algorithm is:

1) Initialize Swarm. Dimesion of particles equals to number of attributes.
2) Insert N particles of each class from TR
3) While the Termination Criterion not satified do:
   a) Checck for particle reproduction and deletion.
   b) For each particle do:
      i) Calculate local fitness.
      ii) Calculate its next position.
   c) Move the particles.
   d) Assign classes to the prototypes of TR using the nearest particles.
   e) Evaluate classification accuracy.
4) Delete from the best swarm found, the particles that can be removed without a reduction in the classification accuracy.

- **Prototype Selection Clonal Selection Algorithm (PSCSA)** [14]

  This is based on an artificial immune system [15], using the clonal selection algorithm to find the most appropriate position for a prototype set [14]. This model is composed of an immune memory which stores in its cells the best antigens found in the search process. The Clonar Selection algorithm is initialized by representing the training instances as antigens, and choosing one antigen from each class to fill the immune memory. Then the evolutionary search process starts. The first stage consists of a Hyper-mutation process, where, for each antigen on the training set, the most stimulating antigen in the immune memory is selected. The measure of stimulation is based on how close both antigens are (by means of Hamming or Euclidean Distance). The selected antigen of the memory is used as a parent for the Hyper-mutation process, which generates a given number of descendants. In the next step, a Resource Allocation procedure is called, which balances the total number of clones present in the system by giving half of the resources to the clones of the same class of the current antigen. The other half is equally divided among clones of other classes. While the classification accuracy is improved by the generation of clones, further Mutation processes (and Resource Allocation procedures) are carried out on the surviving clones. This Mutation produces a lower number of clones which depends on the stimulation value of each parent clone. When no improvement of the classification accuracy is achieved, the best clone found is inserted into the immune memory, performing a replacement with the worst antigen present if the immune memory is full. Then a new antigen of the training set is selected to perform the next generation. Finally, when the algorithm meets a global termination criterion, the antigens contained in the immune memory are employed as the training set to classify the test instances, by using the NN classifier.

  The pseudocode of the PSCSA algorithm is:

1) Initialization.
2) While the Termination Criterion not satified do:
   a) Proliferation I (Hyper-mutation).
   b) Resource Allocation.
   c) While resources left do:
      i) Proliferation II (Mutation).
      ii) Resource Allocation.
   d) Insert best antigen into immune memory.

- **Particle Swarm Optimization (PSO)** [16]

  This method defines the particles of the swarm as sets of a fixed number of prototypes, which are modified as the particle is moved in the search space. The usual operators of PSO are employed by this proposal. The representation of each particle consists of a vector of length given by the concatenation of K prototypes (dealing with M-dimensional data). The fitness function employed is defined as the classification error of the set of K prototypes over the training data, by employing the NN rule.

  The algorithm can be described as:

  1) Initialization: Select a random number of prototypes from TR
  2) While the end condition is reachead:
     a) Evaluate particles.
     b) Update procedure: For each particle, calcule its velocity and update the positioning, according the following equations:

$$v_i = wv_i + c1 * Rand()(p_i - x_i) + c2Rand()(p_g - x_i) \tag{13}$$

$$x_i(t+1) = x_i(t) + v_i \tag{14}$$

     where i varies from 1 to the population size; c1 and c2 are named accelaration constants; p_i is the current particle (a set of prototypes) and p_g is the best positions; Rand() is a random function in the range [0,1]; w is the inertia weight.

- **Iterative Prototype Adjustment based on Differential Evolution (IPADE)** [17]

  The IPADE algorithm follows an iterative prototype adjustment scheme with an incremental approach. At each step, an optimization procedure is used to adjust the position of the prototypes, adding new ones if needed. The aim of this algorithm is to determine the most appropriate number of prototypes per class and adjust their

positioning during the evolutionary process. Specifically, IPADE uses the SFLSDE technique [] as optimizer. At the end of the process, IPADECS returns the best GS found. The outline of the algorithm is the following:

1) GS = Initialization(TR)
2) DE_Optimization(GS, TR)
3) $Accuracy_{Global}$ = Evaluate(GS, TR)
4) registerClass[0..Ω] = $optimizable$
5) WHILE$Accuracy_{Global}$ $<>$ 1.0 or all classes are $non-optimizables$
   a) lessAccuracy = ∞
   b) FOR$i = 1$ to Ω
      i) IFregisterClass[i] == $optimizable$ $AccuracyClass[i]$ = Evaluate (GS, Examples of class i in TR)
      ii) IF$AccuracyClass[i] <$ lessAccuracy lessAccuracy = $AccuracyClass[i]$; $targetClass$ = i;
   c) GS_{test} = $GS \cup$ RandomExampleForClass(TR, $targetClass$)
   d) DE_Optimization(GS_{test}, TR)
   e) $accuracy_{Test}$ = Evaluate(GS_{test}, TR)
   f) IF$accuracy_{Test} > Accuracy_{Global}$ $Accuracy_{Global}$ = $accuracy_{Test}$; GS = GS_{test}
   g) ELSE registerClass[$targetClass$] = $non-optimizable$

- **Hybrid Steady-State Memetic Algorithm for Instance Selection + Scale Factor Local Search in Differential Evolution (SSMA-SFLSDE)** [18]

  Random selection (stratified or not) of prototypes from the TR may not be the most adequate procedure to initialize the GS. The SSMA-SFLSDE approach uses a PS algorithm prior to the adjustment process to initialize a subset of prototypes. Specifically, the SSMA algorithm [] is applied and the resulting SS is inserted as one of the individuals of the population in the DE optimization procedure, that in this case, it is acting as a PG method. In this proposal, each individual of the population encodes a complete GS. The rest of the individuals are initialized with random solutions extracted from the TR, keeping the same structure as the SS selected by the SSMA method.

IV. Class re-labeling

This generation mechanism consists of changing the class labels of samples from TR which could be suspicious of being errors and belonging to other different classes. Its purpose is to cope with all types of imperfections in the training set (mislabeled, noisy and atypical cases). The effect obtained is closely related to the improvement in generalization accuracy of the test data, although the reduction rate is kept fixed.

- **Generalized Editing using Nearest Neighbor (GENN)** [19]

In this work, the kk'-NN rule is proposed. Unless a sample and its k-1 nearest neighbors form a majority group of k' out of k it is edited. Otherwise, it is labeled according to the majority class. For the kk'-NN rule the modification is as follows,

1) Samples of TR are places into groups of k.
2) If there is not a majority of k' of one class the group of k samples is deleted. Otherwise, all samples are labeled as belonging to the majority class.
3) The single NN rule is used on the TG set.

- **Depuration Algorithm (Depur)** [20]

  This method is based on the generalized editing in which two parameters have to be defined: k and k', in such a way that $(k+1)/2 <= k' <= k$. The algorithm can be written as follows:

  1) Let $TG = TR$.
  2) For each $x_i \in TR$ do:
     a) Find the k-NN of x_i in $TR - x_i$.
     b) If a class has at least k' representatives among the k neighbors, change the label of xi according to that class. Otherwise, discard x_i from TG

V. Centroid based

These techniques are based on generating artificial prototypes by merging a set of similar examples. The merging process is usually made from the computation of averaged attribute values over a selected set, yielding the so-called centroids. The identification and selection of the set of examples are the main concerns of the algorithms that belong to this category. These methods can obtain a high reduction rate but they are also related to accuracy rate losses.

- **Prototype Nearest Neighbor (PNN)** [21]

  The algorithm of finding Prototypes for Nearest Neighbor classifiers (PNN) can be stated as follows: given a training set TR, the algorithm starts with every point in TR as a prototype. Initially, set A is empty and set B is equal to TR. The algorithm selects an arbitrary point in B and initially assign it to A. After this, the two closest prototypes $\mathbf{x}_p$ in A and $\mathbf{x}_q$ in B of the same class are merged, successively, into a new prototype, $\mathbf{x}_p*$, if the merging will not degrade the classification of the patterns in TR, where $\mathbf{x}_p*$ is the weighted average of $\mathbf{x}_p$ and $\mathbf{x}_q$. Initially, every prototype has an associated weight of unity. The procedure of PNN is sketched below:

  1) Copy TR to B;
  2) For all $\mathbf{x}_q \in$ B, set the weight Wq = 1;
  3) Select a point in B, and move it from B to A;

4) MERGE = 0;
5) While B is not empty do:
   a) Find the closest prototypes $\mathbf{x}_p$ and $\mathbf{x}_q$ from A and B, respectively
   b) If $\mathbf{x}_p$s class is not equal to $\mathbf{x}_q$s class then insert $\mathbf{x}_q$ to A and delete it from B;
   c) Else merge p of weight Wp, and $\mathbf{x}_q$ of weight Wq., to yield $\mathbf{x}_p*$, where $\mathbf{x}_p*$ = (Wp• $\mathbf{x}_p$ + Wq•$\mathbf{x}_q$)/(Wp + Wq). Let the classification error rate of this new set of prototypes be ϵ If the ϵ is increased then insert $\mathbf{x}_q$ to A, and delete it from B; Else delete $\mathbf{x}_p$ and $\mathbf{x}_q$ from A and B, insert $\mathbf{x}_p*$ with weight Wp + Wq to A, and MERGE++;
6) If MERGE is equal to 0 then output A as the set of trained code-book vectors, and the process terminates;
7) Copy A into B and go to 3.

- **Bootstrap Technique for Nearest Neighbor (BTS3)** [22]

  The bootstrapping method is briefly described below:

  1) For every class, i, select a sample $\mathbf{x}_i$ from T_{Ni} in random;
  2) Find the k nearest neighbor samples of the $\mathbf{x}_i$ as $\mathbf{x}_i^1$, $\mathbf{x}_i^2$, ..., $\mathbf{x}_i^k$;
  3) Compute a bootstrap sample (prototype) $\mathbf{y}_i = \frac{1}{k} * \sum_{j=1}^{k} \mathbf{x}_i^j$
  4) Repeat the above three steps $M_i(M_i$ ¡= $N_i)$ times, under a condition that no sample is selected more than once;
  5) Combine all the bootstrap samples $\mathbf{y}_i$ of all classes into one single set of M vectors, and output it as the set of final prototypes.

- **Modified Changs Algorithm (MCA)** [23]

  Bezdek et al. proposed a modification of the PNN. First, instead of using the weighted mean of the PNN to merge prototypes, they utilized the simple arithmetic mean. Secondly, the process for searching for the candidates to be merged was modified by partitioning the distance matrix into submatrices blocked by common labels. This modification eliminated the consideration of candidate pairs with different labels.

- **Generalized Modified Changs Algorithm (GMCA)** [24]

  The GMCA approach is founded on hierarchical clustering. Mollineda et al. claims that PNN and GMCA algorithms have two kinds of drawbacks. First, they have a restricted strategy for building prototypes based on pairwise merging and, consequently, they may provide condensing results which are far from the optimal ones, both from the point of view of their size and their representativity. And second, they employ a considerable amount of computation to check consistency exhaustively for any possible merging.

An algorithmic description of this methods is:

1) TG = TR
2) Compute L as the set of all candidate pairs from TG of the same class.
3) Repeat until a complete pass through L has produced no merge:
   a) Let (p,q) be the pair from L whose distance is minimum and remove it from L.
   b) Let p^* be the result of merging p and q and let $TG*$ = $(TG \cup p^*) - p, q$.
   c) if $TG*$ is consistent, $TG = TG*$ and recompute the set of candidate pairs L

They add a consistency checking procedure based on clustering.

- **Integrated Concept Prototype Learner (ICPL)** [25]

  In this work, we are going to describe the ICPL2 variant which showed to be the best performing ICPL technique. The ICPL2 approach integrates filtering and abstraction/generation techniques by maintaining a balance different kinds of concept prototypes according to instance locality. The abstraction component, based on typicality, employed in the ICPL2 framework is specially designed for concept integration. As filtering techniques, ICPL2 could use every edition method. Concretely, ICPL2 presents a good behavior with the well-known edited nearest neighbor.

  The algorithm is described as:

  1) C_1 = Abstraction Phase (TR).
  2) C_2 = Filtering phase TR).
  3) S = C_1
  4) For each prototype $x_p \in C_2$
     a) Tmp = $S \cup x_p$
     b) Good = Number of instances in TR correctly classified by x_p in Tmp.
     c) Bad = Number of instances in TR incorrectly classified by x_p in Tmp.
     d) if (Good $>$ Bad) Set $S = S \cup x_p$
  5) For each prototype $x_p \in C_1$
     a) Tmp = $S \setminus x_p$
     b) With = Number of instances in TR correctly classified by S.
     c) Without = Number of instances in TR incorrectly classified by Tmp.
     d) if (Without $>=$ With) Set $S = S \setminus x_p$
  6) $TG = S$

- **Adaptive Condensing Algorithm Based on Mixtures of Gaussians (MGauss)** [26]

  This is an adaptive PG method considered in the framework of mixture modeling by Gaussian distributions,

while assuming a statistical independence of features. The prototypes are chosen as the mean vectors of the optimized Gaussians, whose mixtures are fit to model each of the classes [26]. The main steps of this algorithm are:

1) Firstly, in the initialization step M components are located on the class means, and shifted away by the addition of some random disturbances.
2) Secondly, the expectation-maximization optimization is applied to each class independently;
3) When an iteration of the expectation-maximization loop is finished for every class, the error is compared to the error from the previous repetition of the loop in order to recognize if a class is moving its nodes faster than others. If accuracy have been decreased for any of the classes, it is forced to wait until the rest of the classes iterates on the expectation-maximization loop once more.
4) If no class improves its classification accuracy ratio, the balance position have already been reached and the stopping criterion forces the process to stop.

The process can be viewed as an adaptive condensing scheme, in which the prototypes of the resulting set will correspond to the centers of the final Gaussians.

- **Self-generating Prototypes (SGP)** [27]

The main idea of this method is to form a number of groups, each of which contains some patterns of the same class, and each group's mean is used as a prototype for the group. The algorithm can be expressed as:

1) Set G_i with all the prototypes of class i, $i = 1, 2, ..., numberofClasses$ extracted from TR
2) Compute the initial prototypes x_i of TG as the centroids of each class.
3) Set i = 1, M = number of Classes.
4) Compute $djs = ||y_q - x_s||$ for each $y_q \in G_i$ and, $G_{i,s} = 1, .., M$
5) Determine the index of the closest prototype to each pattern y_q as i_j* = arg mins ($d_j s$).
6) If i_j*=k, $\forall y_q \in G_i$ go to step 10.
7) If Class of (x_{i_j*}) ! = Class of (x_k), $\forall y_q \in G_i$, set $M = M + 1$, split group G_i into two subgroups G_i and G_M, and update their means: x_k = mean(G_k), x_M = mean(G_M), Class of $(x_M) = C(x_k)$, go to step 4.
8) If Class of (x_{i_j*}) == Class of (x_k), $x_{i_j*}! = x_{i_j*}$, for some $x_j \in G_k$, remove these patterns from G_k and include them in group Gi_j*.
9) If Class of (x_{i_j*}) == Class of (x_k), for some $x_j \in G_k$, set $M = M + 1$, remove these patterns from G_k and create a new group GM containing these patterns. Update the means.
10) If k = M and no change is made to the groups or the prototypes, then STOP.
11) If k = M, then set k = k + 1 and go to step 4.
12) If k = M, then set k = 1 and go to step 4.

VI. SPACE SPLITTING

This set includes the techniques based on different heuristics for partitioning the feature space, along with several mechanisms to define new prototypes. The idea consists of dividing TR into some regions which will be replaced with representative examples establishing the decision boundaries associated with the original TR. This mechanism works on a space level, due to the fact that the partitions are found in order to discriminate, as well as possible, a set of examples from others, whereas centroid based approaches work on the data level, mainly focusing on the optimal selection of only a set of examples to be treated. The reduction capabilities of these techniques usually depend on the number of regions that are needed to represent TR.

- **Chen Algorithm (Chen)** [28]

  The Chen algorithm can be written as follows:

  1) Select the desired size of TG.
  2) Let b_c = 1 (b_c is the current number of subsets in TR), and i = 1.
  3) Let $B = TR$.
  4) Find the two farthest points, p_1 and p_2 , in B.
  5) Divide the set B into two subsets B_1 and B_2 , where:
  $$\begin{cases} B_1 = p \in B : d(p, p_1) <= d(p, p_2) \\ B_2 = p \in B : d(p, p_2) < d(p, p_1) \end{cases} \tag{15}$$
  6) Let b_c = b_c + 1, C(i) = B_1 , and C(b_c) = B_2 .
  7) Let I_1 = i : C(i) contains instances from two different classes at least, and let I_2 = i : $i <= b_c$ I_1.
  8) Let I =I_1 if I_1 != $\emptyset$ else I = $I2$.
  9) Find the two farthest points, q_1 (i) and q_2 (i), in each C(i) for i $\in$ I .
  10) Find the set C(j) with the largest diameter j.
  11) Let B = C(j), p_1 = q_1 (j), and p_2 = q_2 (j).
  12) If $b_c <$ b (b is the number of final subsets) then go to 4.
  13) Find the centroids c(l,i) for each class l in subset C(i), i = 1,2...size(TG).
  14) Assign to each c(l,i) the class that is most heavily represented in C(i), ties break by largest class and further randomly. Put all c(l,i) in the resulting set TG.

- **Reduction by space partitioning 3 (RSP3)** [29]

  This technique is based on Chen's algorithm [28]. The main difference between them is that in the Chen's algorithm any subset containing a mixture of instances belonging to different classes can be chosen to be divided. By contrast, in RSP3 [29], the subset with the highest overlapping degree is the one picked to be

split. This process tries to avoid drastic changes in the form of decision boundaries associated with TR which are the main shortcomings of Chen's algorithm.

The RSP algorithm can be written as follows:

1) Let b_c = 1 (b_c is the current number of subsets in TR), and i = 1.
2) Let $B = TR$.
3) Find the two farthest points, p_1 and p_2 , in B.
4) Divide the set B into two subsets B_1 and B_2 , where:

$$\begin{cases} B_1 = p \in B : d(p, p_1) <= d(p, p_2); B_2 = p \in B : d(p, p_2) < d(p, p_1) : \end{cases} \tag{16}$$

5) Let b_c = b_c + 1, C(i) = B_1 , and C(b_c) = B_2 .
6) Let I_1 = i : C(i) contains instances from two different classes at least, and let I_2 = i : $i <= b_c$ I_1.
7) Let I =I_1 if I_1 != $\emptyset$ else I = $I2$.
8) Find the two farthest points, q_1 (i) and q_2 (i), in each C(i) for i $\in$ I .
9) Find the set C(j) with the largest diameter j.
10) Let B = C(j), p_1 = q_1 (j), and p_2 = q_2 (j).
11) If $b_c <$ b (b is the number of final subsets) then go to 4.
12) Find the centroids c(l,i) for each class l in subset C(i), i = 1,2...b.
13) Put all c(l,i) in the resulting set TG.

The third reduction heuristic (RSP3) consists of performing partitions until all subsets are class homogeneous. A set X is said to be class homogeneous if it does not contain a mixture of instances that belong to different classes.

RSP3 can employ both split criteria defined previously: divide the subset with the largest diameter or that with the highest overlapping degree. In fact, the partition criterion is not important here because all heterogeneous subsets have to be finally divided.

- **Pairwise Opposite Class Nearest Neighbor (POC-NN)** [30]

The POC-NN algorithm follows a divide and conquer approach in order to find regions in the search space. Concretely, this method selects border instances. The selection process in POC-NN computes the mean of the instances in each class. For finding a border instance p_b1 in the class C1, POC-NN computes the mean m_2 of the instances in the opposite class C2 and then p_b1 is the instance belonging to class C1 which is the nearest to m_2 . The border instances in the class C2 are found in a similar way; finally, only the border instances are selected.

After performing the prototype selection by POC-NN algorithm on a training set, this training set is separated into many regions of correctly classified classes. The POC-NN method for prototype generation is applied by replacing all POC-NN prototypes in each region with the mean of patterns in its region.

REFERENCES

[1] I. Triguero, J. Derrac, S. García, and F. Herrera, "A taxonomy and experimental study on prototype generation for nearest neighbor classification," *IEEE Transactions on Systems, Man, and Cybernetics–Part C: Applications and Reviews*, 2011, , in press, doi: 10.1109/TSMCC.2010.2103939.

[2] T. Kohonen, "The self organizing map," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464–1480, 1990.

[3] S. Geva and J. Sitte, "Adaptive nearest neighbor pattern classification," *IEEE Transaction On Neural Networks*, vol. 2, no. 2, pp. 318–322, 1991.

[4] Q. Xie, C. A. Laszlo, and R. K. Ward, "Vector quantization technique for nonparametric classifier design," *IEEE Transactions On Pattern Analysis and Machine Intelligence*, vol. 15, no. 12, pp. 1326–1330, 1993.

[5] R. Odorico, "Learning vector quantization with training count (LVQTC)," *Neural Networks*, vol. 10, no. 6, pp. 1083–1088, 1997.

[6] C. Decaestecker, "Finding prototypes for neares neghbour classification by means of gradient descent and deterministic annealing," *Pattern Recognition*, vol. 30, no. 2, pp. 281–288, 1997.

[7] T. V. de Merckt, "NFDT: a system that learns flexible concepts based on decision trees for numerical attributes, in machine learning," in *Proceedings of the 9th International Workshop*, 1992, pp. 322–331.

[8] S. W. Kim and J. Oomenn, "Enhancing prototype reduction schemes with LVQ3-type algorithms," *Pattern Recognition*, vol. 36, pp. 1083–1093, 2003.

[9] F. Fernández and P. Isasi, "Evolutionary design of nearest prototype classifiers," *Journal of Heuristics*, vol. 10, no. 4, pp. 431–454, 2004.

[10] C.-W. Yen, C.-N. Young, and M. L. Nagurka, "A vector quantization method for nearest neighbor classifier design," *Pattern Recognition Letters*, vol. 25, no. 6, pp. 725–731, 2004.

[11] Y. Linde, A. Buzo, and R. Gray, "An algorithm for vector quantizer design," *IEEE Trans. Commun.*, vol. 28, pp. 84–95, 1980.

[12] J. Li, M. T. Manry, C. Yu, and D. R. Wilson, "Prototype classifier design with pruning." *International Journal on Artificial Intelligence Tools*, vol. 14, no. 1-2, pp. 261–280, 2005.

[13] A. Cervantes, I. M. Galván, and P. Isasi, "AMPSO: A new particle swarm method for nearest neighborhood classification," *IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics*, vol. 39, no. 5, pp. 1082–1091, 2009.

[14] U. Garain, "Prototype reduction using an artificial immune model," *Pattern Analysis and Applications*, vol. 11, no. 3-4, pp. 353–363, 2008.

[15] L. N. de Castro and J. Timmis, *Artificial Immune Systems: A New Computational Intelligence Approach*. Springer, 2002.

[16] L. Nanni and A. Lumini, "Particle swarm optimization for prototype reduction," *Neurocomputing*, vol. 72, no. 4-6, pp. 1092–1097, 2008.

[17] I. Triguero, S. García, and F. Herrera, "IPADE: Iterative prototype adjustment for nearest neighbor classification," *IEEE Transactions on Neural Networks*, vol. 21, no. 12, pp. 1984–1990, 2010.

[18] ——, "Differential evolution for optimizing the positioning of prototypes in nearest neighbor classification," *Pattern Recognition*, vol. 44, no. 4, pp. 901–916, 2011.

[19] J. Koplowitz and T. Brown, "On the relation of performance to editing in nearest neighbor rules," *Pattern Recognition*, vol. 13, pp. 251–255, 1981.

[20] J. S. Sánchez, R. Barandela, A. I. Marqués, R. Alejo, and J. Badenas, "Analysis of new techniques to obtain quality training sets," *Pattern Recognition Letters*, vol. 24, no. 7, pp. 1015–1022, 2003.

[21] C.-L. Chang, "Finding prototypes for nearest neighbor classifiers," *IEEE Transactions on Computers*, vol. 23, no. 11, pp. 1179–1184, 1974.

[22] Y. Hamamoto, S. Uchimura, and S. Tomita, "A bootstrap technique for nearest neighbor classifier design," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 1, pp. 73–79, 1997.

[23] T. Bezdek, J.C .and Reichherzer, G. Lim, and Y. Attikiouzel, "Multiple prototype classifier design," *IEEE Transactions on Systems, Man and Cybernetics C*, vol. 28, no. 1, p. 6779, 1998.

[24] R. Mollineda, F. Ferri, and E. Vidal, "A merge-based condensing strategy for multiple prototype classifiers," *IEEE Transactions on Systems, Man and Cybernetics B*, vol. 32, no. 5, pp. 662–668, 2002.

[25] W. Lam, C. K. Keung, and D. Liu, "Discovering useful concept prototypes for classification based on filtering and abstraction." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 8, pp. 1075–1090, 2002.

[26] M. Lozano, J. M. Sotoca, J. S. Sánchez, F. Pla, E. Pekalska, and R. P. W. Duin, "Experimental study on prototype optimisation algorithms for prototype-based classification in vector spaces," *Pattern Recognition*, vol. 39, no. 10, pp. 1827–1838, 2006.

[27] H. A. Fayed, S. R. Hashem, and A. F. Atiya, "Self-generating prototypes for pattern classification," *Pattern Recognition*, vol. 40, no. 5, pp. 1498–1509, 2007.

[28] C. H. Chen and A. Jóźwik, "A sample set condensation algorithm for the class sensitive artificial neural network," *Pattern Recognition Letters*, vol. 17, no. 8, pp. 819–823, 1996.

[29] J. S. Sánchez, "High training set size reduction by space partitioning and prototype abstraction," *Pattern Recognition*, vol. 37, no. 7, pp. 1561–1564, 2004.

[30] T. Raicharoen and C. Lursinsap, "A divide-and-conquer approach to the pairwise opposite class-nearest neighbor (POC-NN) algorithm," *Pattern Recogniton Letters*, vol. 26, pp. 1554–1567, 2005.