NEAREST PROTOTYPE CLASSIFIER DESIGNS: AN EXPERIMENTAL STUDY

James C. Bezdek[1]
Computer Science Department
University of West Florida
Pensacola, FL, 32514, USA
jbezdek@uwf.edu

Ludmila I. Kuncheva[1]
School of Informatics
University of Wales
LL57 1UT Bangor, UK
mas00a@bangor.ac.uk

Abstract

We compare eleven methods for finding prototypes upon which to base the nearest prototype classifier. Four methods for prototype *selection* are discussed: Wilson+Hart (a condensation + error-editing method), and three types of combinatorial search - random search, a genetic algorithm and tabu search. Seven methods for prototype *extraction* are discussed: unsupervised vector quantization , supervised learning vector quantization (with and without training counters), decision surface mapping, a fuzzy version of vector quantization, c-means clustering and bootstrap editing. These eleven methods can be usefully divided two other ways, by whether they employ pre- or post-supervision; and by whether the number of prototypes found is user-defined or “automatic”. Generalization error rates of the eleven methods are estimated on two synthetic and two real data sets. Offering the usual disclaimer that these are just a limited set of experiments, we feel confident in asserting that pre-supervised, extraction methods offer a better chance for success to the casual user than post-supervised, selection schemes. Finally, our calculations do not suggest that methods which find the "best" number of prototypes "automatically" are superior to methods for which the user simply specifies the number of prototypes.

Keywords

Classifier design
Condensation
Editing
Nearest neighbor
Nearest prototype
Search techniques

[1]Research supported by ONR grant 00014-96-1-0642.

Vector quantization

1. Introduction

All of the methods discussed here begin with a *crisply labeled* set of training data $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \Re^p$ (the *training data*). Our presumption is that X contains at least one point with class label i, 1 ≤ i ≤c. The number of prototypes we are looking for, $|\mathbf{V}| = n_p$, depends on the method used, and may be less than, equal to, or greater than c, the number of classes represented in X.

Let **x** be an unlabeled object that we wish to label as belonging to one of c classes. The standard *nearest prototype* (1-np) classification rule assigns **x** to the class of the "most similar" prototype in a set of labeled prototypes (or *reference set*), say $\mathbf{V} = \{\mathbf{v}_1, \ldots, \mathbf{v}_{np}\}$. Hereafter $E_{np}(X_{tr}; \mathbf{V})$ denotes the resubstitution (training error) committed by the 1-np rule that uses **V** when applied to the training data; $E_{np}(X_{test}; \mathbf{V})$ stands for the generalization (testing) error of the same classifier. Why use a nearest prototype classifier? Because it is intuitive, simple, and often, pretty accurate [1]. Human pattern recognition is almost always based on comparison of the presented input to a catalog of stored examples, both typical and atypical, and this fact is reflected in many case- or instance-based paradigms used in machine learning [2].

Good prototypes have two desirable properties: *minimal cardinality* and *maximum classification accuracy* (or equivalently, minimum $E_{np}(X_{test}; \mathbf{V})$). In other words, we seek the smallest possible **V** with the highest possible (generalization) accuracy of the 1-np rule that uses **V**. You expect these two goals to naturally conflict, and our experiments show that they do. Increasing the number of prototypes up to some experimentally determined upper limit (usually with n_p > c) almost always results in a decreasing trend in the test error rate $E_{np}(X_{test}; \mathbf{V})$, and conversely. One goal of our research is to study this conflict - how to find the smallest set of prototypes that provides an acceptable generalization error for the data at hand.

Various classifiers assign different kinds of labels to unlabeled objects. There are four types of class labels - crisp, fuzzy, probabilistic and possibilistic. Let n be the number of objects and integer c denote the number of classes, 1 ≤ c ≤ n. We define three sets of label vectors in $\Re^c$ as follows:

$$N_{pc} = \left\{ \mathbf{y} \in \Re^c : y_i \in [0,\ 1]\ \ \forall\ \ i,\ y_i > 0\ \ \exists\ \ i \right\} = [0,1]^c - \{\mathbf{0}\} \quad ; \qquad (1)$$

$$N_{fc} = \left\{ \mathbf{y} \in N_{pc} : \sum_{i=1}^{c} y_i = 1 \right\} \quad ; \qquad (2)$$

$$N_{hc} = \left\{ \mathbf{y} \in N_{fc} : y_i \in \{0,1\} \forall\ i \right\} = \left\{ \mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_c \right\} \quad . \qquad (3)$$

In (1) **0** is the *zero vector* in $\Re^c$. N_{hc} is the canonical (unit vector) basis of Euclidean c-space, so the *crisp label* for class i, $1 \le i \le c$, is $\mathbf{e}_i = (0, 0, \ldots, \underbrace{1}_{i}, \ldots, 0)^T$, the i-th vertex of N_{hc}. The set N_{fc}, a piece of a hyperplane, is the convex hull of N_{hc}. The vector $\mathbf{y} = (0.1, 0.6, 0.3)^T$ is a constrained label vector; its entries lie between 0 and 1, and sum to 1. The *centroid* of N_{fc} is the equi-membership vector $\mathbf{1/c} = (1/c, \ldots, 1/c)^T$. If **y** is a label vector for some $\mathbf{x} \in \Re^p$ generated by, say, the fuzzy c-means clustering method, we call **y** a *fuzzy label* for **x**. If **y** came from a method such as maximum likelihood estimation in mixture decomposition, **y** would be a *probabilistic label.* In this case, $\mathbf{1/c}$ is the unique point of equal probabilities for all c classes.

$N_{pc} = [0, 1]^c - \{\mathbf{0}\}$ is the unit hypercube in $\Re^c$, *excluding the origin.* Vectors such as $\mathbf{z} = (0.7, 0.2, 0.7)^T$ with each entry between 0 and 1 that are otherwise unrestricted are *possibilistic labels* in N_{p3}. We interpret possibilistic label values as indicants of the extent to which **z** is *typical* of each of the c classes. Possibilistic labels are produced by possibilistic clustering algorithms [3] and by computational neural networks that have unipolar sigmoidal transfer functions at each of c output nodes [4]. For convenience we call all non-crisp labels *soft* labels. Note that $N_{hc} \subset N_{fc} \subset N_{pc}$.

In Section 2 we discuss prototype extraction and the *generalized nearest prototype classifier* (GNPC) [5, 6]. Section 3 contains a concise description of the four selection methods surveyed in this article, and Section 4 describes the seven replacement methods we chose to compare to the four selection methods. The 4 data sets used and our results - comparisons between and similarities of the eleven methods - are discussed in Section 5. Section 6 offers the conclusions we draw from these experiments, and itemizes some interesting topics for further research.

2. Prototype extraction and the GNPC

When the prototypes used in the GNPC have soft labels, each prototype may "vote" with varying assurance for all c classes. For example, if $\mathbf{v}_i$ has the soft label [0.2, 0.4, 0.7], this prototype is a fairly typical example of class 3 but is also related (less strongly) to classes 1 and 2. A real-life example of soft labeling is diagnosing ischemic heart disease, where the occlusion of the three main coronary arteries can be expressed by such a label, each entry being the degree of occlusion of a particular vessel.

If **x** and the $\mathbf{v}_i$'s are represented by feature vectors in $\Re^p$, prototypical similarity is almost always based on some function of pairwise distances between **x** and the $\mathbf{v}_i$'s. Specifically, let

$\mathbf{x} \in \Re^p$ be an input vector. The *generalized nearest prototype classifier* (GNPC) is defined by the 5-tuple :

1. The set of prototypes $\mathbf{V} = \{\mathbf{v}_1, \ldots, \mathbf{v}_{n_p}\} \subset \Re^p$; (GNPC1)
2. The $c \times n_p$ prototype label matrix $L(\mathbf{V}) = [l(\mathbf{v}_1), \ldots, l(\mathbf{v}_{n_p})] \in \Re^c \times \Re^{n_p}$; (GNPC2)
3. A similarity function $S(\mathbf{x}_k, \mathbf{v}_i) = \Theta(\|\mathbf{x}_k - \mathbf{v}_i\|)$ valued in [0,1] that assesses the similarity between inputs and prototypes (the higher the value, the more similar the input to the prototype). ; (GNPC3)
4. A T-norm (in the fuzzy sets sense) to fuse each prototype label and the similarity between that prototype and **x** ; (GNPC4)
5. An aggregation operator which uses the p fused components to produce an overall soft label for **x** . (GNPC5)

When the output of (GNPC5) is a soft label, a crisp class label can always be assigned to **x** by taking the class with the highest "support" in the soft label vector. This operation is a specific form of the general operation of *hardening* soft labels. A great number of classifier models can be described by different choices of the parameters in (GNPC1-GNPC5), and an even greater set can be designed by varying those parameters. References [5, 6] contain a detailed description and many examples of models that fall into the GNPC framework.

Figure 1 shows three of the many groups of classifiers that belong to the GNPC family. Abbreviations appearing in Figure 1 are: *hard c-means* (HCM), *nearest neighbor* (1-nn), *learning vector quantization* (LVQ) and *radial basis function* (RBF). Many of the classifiers in Figure 1 are simple 1-np classifiers; the major distinction between them is in the way their prototypes are obtained. This paper discusses some cases of the models shown in the module highlighted by a heavy border at the bottom left in Figure 1. Three basic characteristics of prototype extraction methods, itemized here as (C1), (C2) and (C3), are discussed next.

(C1) *Selection* ($\mathbf{V}_s \subseteq X$) versus *replacement* ($\mathbf{V}_r \not\subset X$). When talking about prototypes in general, we use the symbol **V**; when emphasis on selection or replacement is needed, we use subscripts ($\mathbf{V}_s$ for selection, $\mathbf{V}_r$ for replacement). Replacement seeks n_p points in $\Re^p$, so the search space is infinite. Selection is limited to searching within $X \subset \Re^p$, so solutions in this case can be sought by combinatorial optimization. Selection is of course a special case of prototype replacement. Figure 2 illustrates the two styles of prototype extraction.

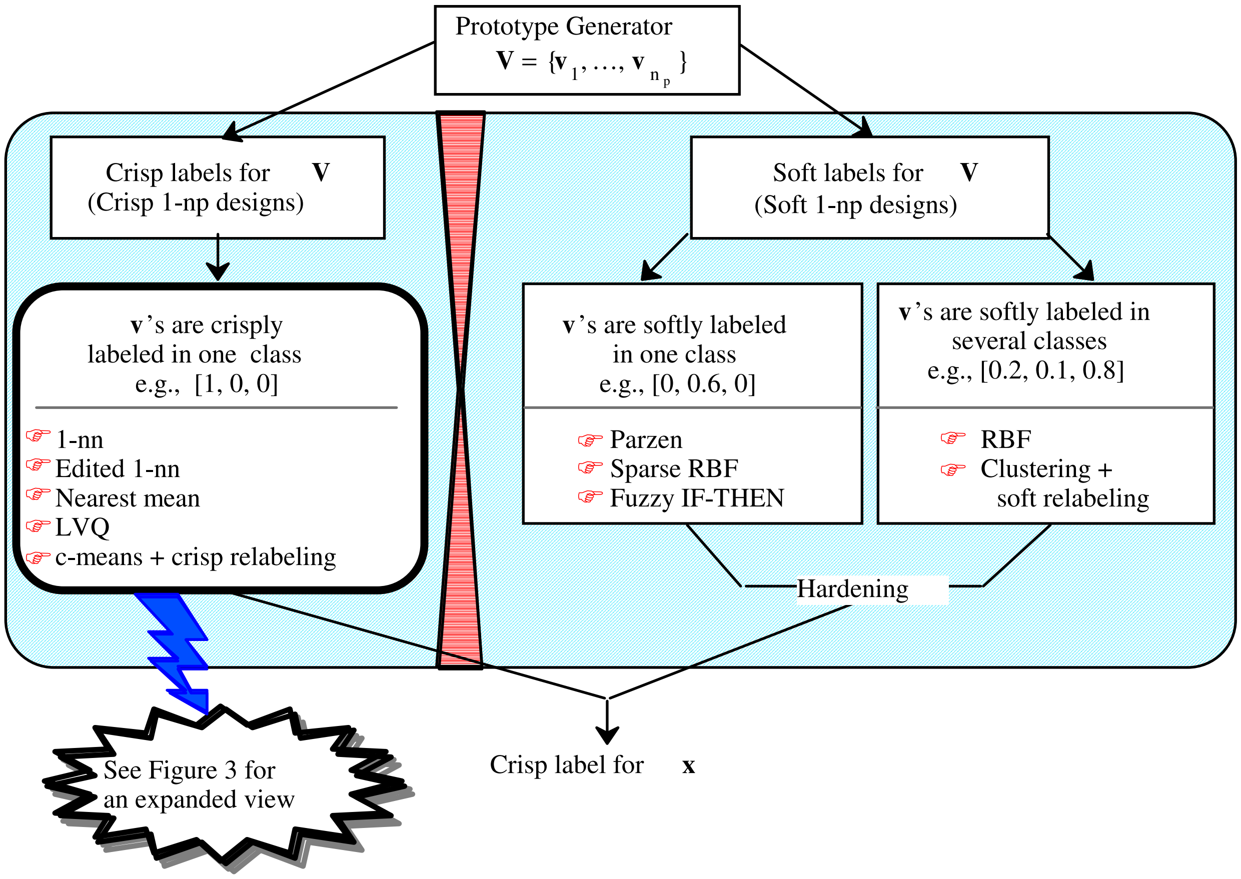
Prototype Generator
$\mathbf{V} = \{\mathbf{v}_1, \ldots, \mathbf{v}_{n_p}\}$
Crisp labels for V
(Crisp 1-np designs)
Soft labels for V
(Soft 1-np designs)
v's are crisply labeled in one class e.g., [1, 0, 0]
1-nn
Edited 1-nn
Nearest mean
LVQ
c-means + crisp relabeling
v's are softly labeled in one class e.g., [0, 0.6, 0]
Parzen
Sparse RBF
Fuzzy IF-THEN
v's are softly labeled in several classes e.g., [0.2, 0.1, 0.8]
RBF
Clustering + soft relabeling
Hardening
Crisp label for x
See Figure 3 for an expanded view

Figure 1. A few models that are generalized nearest prototype classifiers

(C2) *Pre-supervised* versus *post-supervised* designs [7]. Pre-supervised methods use the data *and* the class labels to locate the prototypes. Post-supervised methods first find prototypes without regard to the training data labels, and then assign a class label (*relabel*) to each prototype. Selection methods are naturally pre-supervised because each prototype is a data point and already has its (presumably true) label.

(C3) *User-defined* n_p versus *algorithmically defined* n_p. Most methods for prototype generation require n_p to be specified in advance as an explicit parameter of the algorithm (e.g., classical clustering and competitive learning methods). Some models have "adaptive" variants where the initially specified value of n_p can increase or decrease, i.e., prototypes are added or deleted during training under the guidance of some mathematical criterion of prototype "quality". A third group of methods do not specify n_p at all, instead obtaining it as an output at the termination of training. For example, condensation methods which search for a minimal possible consistent set belong to this category. Genetic algorithms and tabu search methods have a trade-off parameter which pits the weight of a misclassification against an increase in the cardinality of **V** by 1. Thus, methods based on these types of search deliver the number of prototypes at termination of training. A method that finds n_p during training will be called an *auto-*n_p method; otherwise, the method is *user-*n_p .

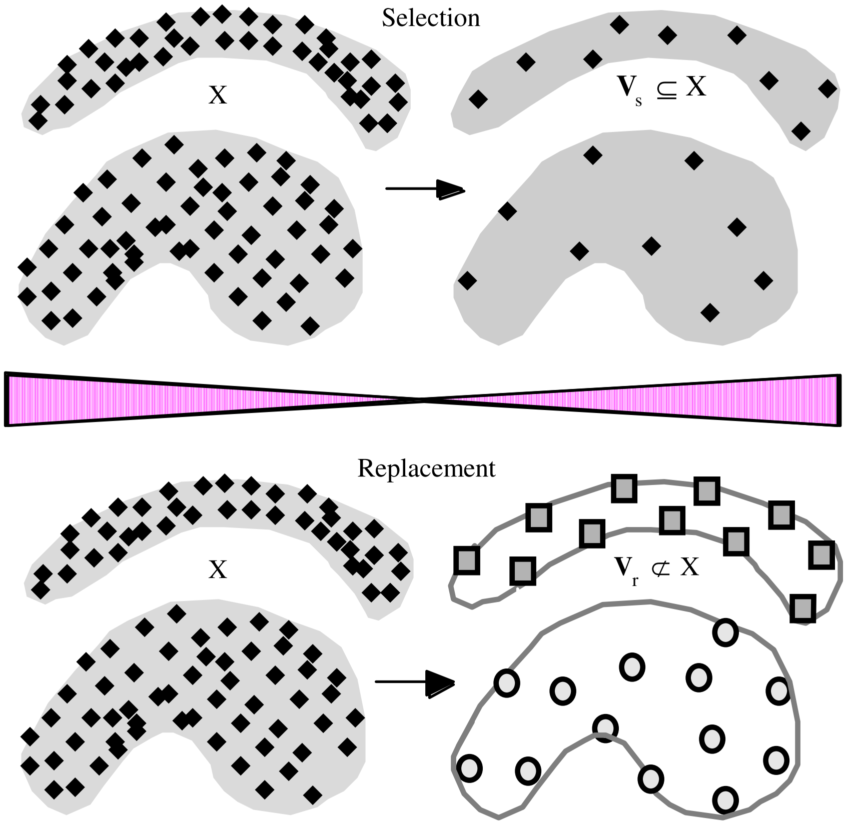

Figure 2. Prototype extraction : selection versus replacement

Issue (C3) is an oft misunderstood aspect of prototype generation for classifier design. The "user-friendliness" of a nearest prototype classifier design is greatly increased by any algorithm that "automatically" finds the best number of prototypes, thereby relieving the designer of making a tradeoff study to select n_p. Do auto-n_p methods really find "better" prototypes than user-n_p methods? Of course not, but there is one less parameter for the user to pick, so auto-n_p methods enjoy popularity on this account (*and*, we can call them "adaptive"!).

An expanded view of the highlighted module in the lower left corner of Figure 1 is shown in Figure 3. New abbreviations appearing in Figure 1 are: *fuzzy c-means* (FCM), *possibilistic c-means* (PCM), condensed *nearest neighbor* (C-nn), *self organizing map* (SOM), *decision surface mapping* (DSM), *minimal consistent subset* (MCS), *LVQ (with) Training Counters* (LVQ-TC) and *genetic algorithm* (GA). The left half of Figure 3 shows prototype selection divided into two main groups : *Condensing* (or condensation) and *Error-editing* (or simply "editing").

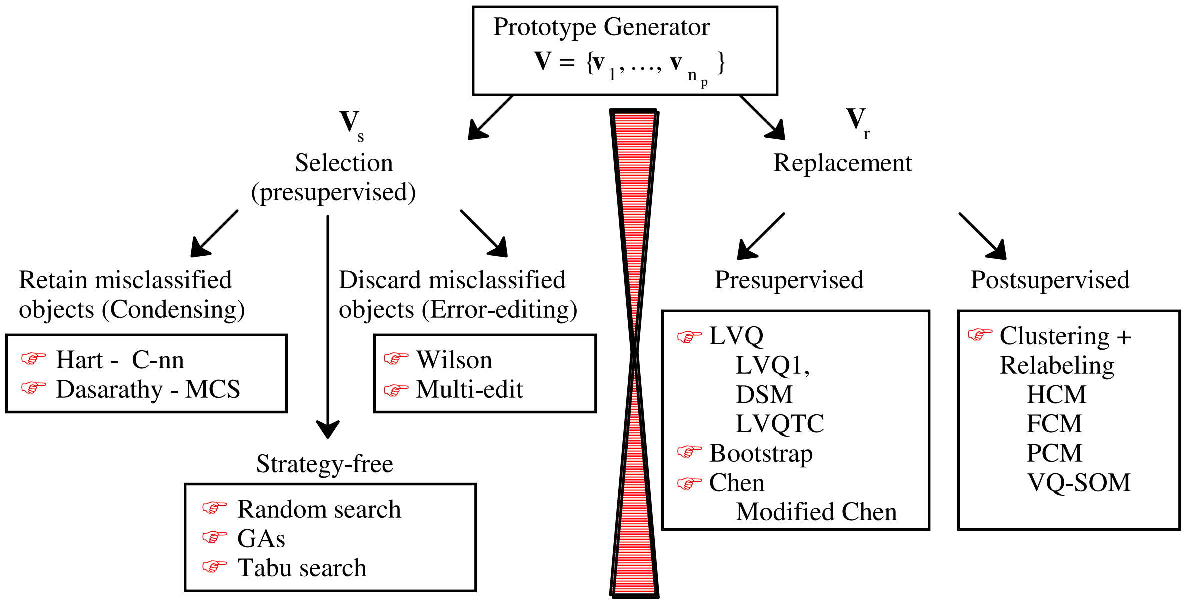

Figure 3. Methods for finding prototypes

Prototype replacement is also divided into two main groups in Figure 3 : *pre-supervised* and *post-supervised* methods. The framework provided by the GNPC model gives us a common platform from which to view many of these methods from the same perspective. For example, we see here both VQ and LVQ, along with some of the many modifications thereof [8-12]. DSM was introduced by Geva and Sitte [13]. Akin to the VQ family is the clustering-and-relabeling approach which itself has many variations, the most popular of which are perhaps the c-means families shown in Figure 3 [1]. Cluster centroids are used as the replacement prototypes $\mathbf{V}_r$, so any clustering method that produces centroids can be usd for this purpose. Having overviewed a piece of the GNPC forest, we turn to a closer look at a few of its trees.

3. Prototype selection

In this section we give brief overviews and the computational protocols for the four selection methods used in our numerical experiments. Following the architecture of Figure 3, we have subsections for condensation, error-edition and search methods. For convenience in Sections 3 and 4, we drop the subscript of X_{tr}, and refer to the training data more simply as X.

Selection by condensation

The objective of condensation is to find a *consistent* reference set, i.e., a set $\mathbf{V}_s \subset X$, such that $E_{np}(X_{tr};\mathbf{V}_s)=0$, i.e., the resubstitution error rate of the 1-np classifier is zero. If there are no identical points with different class labels in X, then at worst X is a consistent set of n prototypes for itself. Condensation aims to find $\mathbf{V}_s$ with the minimal possible cardinality, ideally with n_p<<n. This leads to retaining in $\mathbf{V}_s$ points from X that are close to classification boundaries. Thus, boundaries, together with all the noise that the boundary data points might involve, are modeled as precisely as possible.

Condensation only guarantees zero resubstitution error. The amount of reduction of the reference set and the generalization capability of the nearest prototype classifier built with the selected $\mathbf{V}_s$ are not explicitly involved in condensation selection strategies. However, the general goal of most methods for finding prototypes is to be able to use $\mathbf{V}_s$ with some form of the GNPC to label unknown objects. Then generalization error becomes an important issue. A severe drawback of all condensation methods is that the zero resubstitution error requirement is always enforced, so no trade-off between error rate and the cardinality of $\mathbf{V}_s$ is possible.

Hart's C-nn : The original condensation method is the *condensed nearest neighbor* (C-nn) method of Hart [14]. Hart's C-nn operates with two sets: STORE and GRABBAG. Initially STORE is empty and all elements of X are placed in GRABBAG. The first element of X is moved from GRABBAG into STORE. The next element is classified using the nearest prototype classifier and the current content of STORE. If misclassified, that element is moved from GRABBAG to STORE. When all elements of GRABBAG have been checked, a new iteration starts at the beginning of GRABBAG, using the current content of STORE. The procedure stops when a pass through GRABBAG has not added any element to STORE (or when GRABBAG is empty, and STORE is X). The final content of STORE is taken as $\mathbf{V}_s$. Many modifications of and algorithms similar to Hart's C-nn are known [15].

The output of Hart's C-nn depends on the order of presentation of the elements in X. Different permutations of X can lead to different $\mathbf{V}_s$'s. Cerveron and Ferri [16] suggest running the C-nn multiple times, beginning with different permutations of X. Since C-nn builds STORE gradually, as soon as STORE in the current run reaches the cardinality of the smallest $\mathbf{V}_s$ found so far, the run is terminated and a new run is started from a different permutation of X. This speeds the algorithm towards its destination and seems to produce good sets of consistent prototypes. A *minimal consistent set* algorithm (MCS) for condensation was proposed by

Dasarathy [17]. Dasarathy's MCS decides which elements to retain after a pass through all of X, so unlike C-nn, MCS does not depend on the order in which the elements of X are processed. MCS, however, does not necessarily find $\mathbf{V}_s$ with the true *minimal* cardinality [6].

Selection by editing

Wilson's Method : Error-editing assumes that points from different classes that are close to apparent boundaries between them contain "noise" (i.e., points that "cross" the apparent boundaries between classes, making the labeled clusters seem mixed), and should therefore be discarded. This group of methods include Wilson's method [18] and Multiedit [19, 20]. Both schemes are based on ruling out misclassified objects. In Wilson's method, the k-nn algorithm (Wilson recommends k= 3) is run once on X. All misclassified objects are marked for deletion during the run, and are deleted from X after the run to produce the prototype set $\mathbf{V}_s$.

Multiedit, a theoretically justified version of Wilson's method, is asymptotically Bayes optimal. That is, when the numbers of samples and iterations tends to infinity, the 1-np classifier on the resultant $\mathbf{V}_s$ is equivalent to the Bayes classifier. Multiedit, however, is not suitable for small data sets with overlapping clusters, whereas Wilson's method works well in these cases. Moreover, Wilson's method is appealingly simple.

Error-editing methods have no explicit connection to either the resubstitution or generalization error rate performance of the 1-np classifier based on the resultant $\mathbf{V}_s$. Asymptotic properties do not guarantee good performance when $\mathbf{V}_s$ has to be selected from a finite X. The cardinality of $\mathbf{V}_s$ is not explicitly taken into account. Many early techniques for editing 1-nn training data are summarized in Dasarathy [15]. Our preliminary experiments indicated that the methods of Hart, Wilson, and Randomized Hart were not very effective in terms of either accuracy or data set reduction. In the finalized experiments reported in Section 5 we used Wilson's method followed by Hart's C-nn (called Wilson + Hart).

Hybrids of condensation and error-editing are also known. Editing techniques are often followed by condensing techniques. Editing "cleans up" the input data, yielding a $\mathbf{V}_{s,\text{initial}}$ that supposedly contains only "easy" points in it. Then a condensation method reduces $\mathbf{V}_{s,\text{initial}}$ to a possibly smaller number of relevant final prototypes, say $\mathbf{V}_{s,\text{final}}$. Ferri [21] proposes a third step : Multiedit is used for phase 1 "clean up" ; Hart's C-nn for phase 2 condensation; $\mathbf{V}_{s,\text{final}}$ is then used to reclassify all the original points in X, and the newly labeled data set, say X', is used with the DSM method to further refine the classification boundary.

Interestingly, condensation and error-editing rely on different ideas and implement different strategies (retain or discard boundary objects) aimed at the same general goal: finding the smallest possible $\mathbf{V}_s$ with the highest possible 1-np accuracy. There is a third group of methods for prototype selection that attempt to achieve the same goal through criterion-driven combinatorial optimization. We call this group of methods *"boundary indifferent"*, because only the mathematical goal is specified, so an algorithm " decides" whether or not to retain boundary prototypes. An instant benefit from this third approach is that the cardinality of $\mathbf{V}_s$ and the classification accuracy can be balanced through the criterion used. The basic combinatorial optimization problem to be solved by this approach (and the one used by our genetic algorithm search method in the numerical experiments of Section 5) is:

$$\underbrace{\max}_{\mathbf{V}_s \in P(X)} \left\{ J(\mathbf{V}_s) \right\} = \underbrace{\max}_{\mathbf{V}_s \in P(X)} \left\{ \left(1 - E_{np}(X;\mathbf{V}_s)\right) - \alpha \frac{|\mathbf{V}_s|}{|\mathbf{X}|} \right\} , \qquad (4)$$

where P(X) is the power set of X, $(1 - E_{np}(X; \mathbf{V}_s))$ is the resubstitution classification accuracy achieved by the 1-np design based on $\mathbf{V}_s$, and α is a positive coefficient which essentially determines the trade-off between accuracy and cardinality. The function J in (4) is a typical fitness function for data editing; see [6, 16] for other useful functional forms. Next, we briefly review three methods from this third group that all make use of (4) to evaluate potential solutions to the selection problem: random selection, GA-based search, and Tabu search.

Random selection (RS) : The desired cardinality n_p of $\mathbf{V}_s$ and the number of trials T are specified in advance. Then T random subsets of X of cardinality n_p are generated, and the one with the smallest error rate is retained as $\mathbf{V}_s$. Skalak calls this method a Monte Carlo simulation [22]. In both [6] and [22] random search is reported to work unexpectedly well.

Genetic algorithms (GAs) : Editing the training data with GAs has been discussed by Chang and Lippmann [23]; Kuncheva, [24, 25]; and Kuncheva and Bezdek [6]. Selection of a subset can be naturally encoded in terms of GAs (and more generally, the evolutionary search paradigm). A binary-valued chromosome C represents a subset S of the labeled data set X. The i-th bit in C corresponds to $\mathbf{x}_i$ from X and has value 1 if $\mathbf{x}_i$ is included in S, and 0, otherwise. An example of this type of encoding is shown in Figure 4.

$$X = \{\mathbf{x}_1,\ldots,\mathbf{x}_8\} \quad (+) \quad C = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ \hline\end{array} \quad \Rightarrow \quad S = \{\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_6\}$$

Figure 4. Binary encoding of a subset of X

Our GA approach for the numerical experiments of the next section assesses a *population* of chromosomes in each generation. An initial population of n_{pop} subsets of X like S in Figure 4 is randomly generated. For each S, each bit is zero or one with probability P_{ini} , which regulates the cardinality of the initial population. Various members of the initial population can have different cardinalities. Each chromosome is evaluated by the fitness function J in (4). The *whole* population set is used to choose $n_{pop}/2$ parent couples and each couple produces two children by uniform crossover, i.e., each pair of corresponding bits of the parent chromosomes are swapped with probability $P_c = 0.5$. Higher or lower values for P_c will increase the resemblance between the children and their parents. The set of offspring is then *mutated*, i.e., each bit of each offspring chromosome is altered with a (small) predefined probability P_m (this probability was fixed at $P_m = 0.1$). New chromosomes are assessed by the fitness function J, pooled with the parent population, and the n_{pop} chromosomes from the pool with the highest J values survive as the next generation (*elitist strategy*). The algorithm terminates after T generations, where T is picked in advance (T = 100 in our experiments). The best chromosome from the last generation (the chromosome with the highest J) defines the prototype set $\mathbf{V}_s$, and the number of prototypes is thus determined "automatically", in the auto-n_p sense as used here. You should realize, however, that there are other parameters the user does have to choose with a method such as this one, and $|\mathbf{V}_s| = n_p$ implicitly depends on your choices for these other parameters.

Our GA model with the selected parameter values is close to random selection. The reason for these parameter choices was threefold: (1) RS works reasonably well; (2) our GA model is simpler than “classical” ones which use more sophisticated selection schemes, e.g., the roulette wheel strategy; and (3) due to the enhanced random component, our GA is less likely to depend heavily on the initialization than classical schemes. But the price we pay for this is that our GA is also less likely to terminate quickly at local extrema of J. Our computational experience is that a few runs of this simper scheme can lead to a reasonably good solution.

An even simpler evolutionary algorithm for data editing called “random mutation hill climbing” was proposed by Skalak [22]. Instead of evolving a population of chromosomes simultaneously, only one chromosome evolves (should we call it *survival of the only*?), and only the mutation operation is performed on it. The best set in T mutations is returned as $\mathbf{V}_s$. The evolutionary schemes given in [6] and [22] are both heuristic; it is debatable which is the better of the two. GA conducts a larger search by keeping different subsets of candidates in its early stages. On the other hand, the random mutation method is *really* simple, and, like the GA discussed in [6], is shown in [22] to outperform RS.

Tabu search (TS) : Tabu search is an interesting alternative to the heavily randomized methods in the boundary-indifferent group [16]. In this scheme the number of iterations T is fixed but the cardinality n_p is not. Similar to random mutation hill climbing, TS operates on one subset only, called the current solution S, which is represented as in Figure 4. A tabu vector of length $|X|$ is set up with all of its entries initially set to zero. An entry of 0 in the k-th place in the Tabu vector indicates that $\mathbf{x}_k$ can be added or deleted from S, while an entry that is greater than 0 prohibits a change in the status of $\mathbf{x}_k$. A parameter called *tabu tenure* (T_t) is specified, defining the number of iterations before a change of any previously altered bit is allowed. An initial subset is picked as S, stored as the initial approximation of $\mathbf{V}_s$, and evaluated by $J(\mathbf{V}_s)$. The TS algorithm operates as follows.

First, all neighboring subsets to S are evaluated by the criterion function J. A neighbor to S in the subset selection problem is any subset of X that differs from S by just one element. For example, the eight neighbor solutions to S in Figure 4 are obtained by altering one bit of the chromosome at a time: $\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_6\}$, $\{\mathbf{x}_3, \mathbf{x}_6\}$, $\{\mathbf{x}_2, \mathbf{x}_6\}$, $\{\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_6\}$, $\{\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_5, \mathbf{x}_6\}$, $\{\mathbf{x}_2, \mathbf{x}_3\}$, $\{\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_6, \mathbf{x}_7\}$ and $\{\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_6, \mathbf{x}_7\}$. The neighbor subset $\hat{S}$ that yields the highest value of J is called the *winning neighbor*. If $J(\mathbf{V}_{\hat{s}}) > J(\mathbf{V}_s)$, $\hat{S}$ replaces S, regardless of the tabu vector, and $\mathbf{V}_s$ and $J(\mathbf{V}_s)$ are updated. If $J(\mathbf{V}_{\hat{s}}) \le J(\mathbf{V}_s)$, the tabu vector is checked. If the move from S to $\hat{S}$ is allowed, (the tabu value T_t = 0), the move is made anyway, and the corresponding slot of the tabu vector is set to T_t . Thus, tabu search does not necessarily have the ascent property. All other non-zero values in the tabu vector are then decreased by one. Different criteria can be applied for terminating the algorithm. Four methods for picking the initial solution are proposed in [16].

1. *Maximal.* The initial set is all of X.
2. *Minimal.* One prototype from each class is randomly picked from X.
3. *Condensed.* Hart's C-nn is run on X, and the resultant consistent set $\mathbf{V}_s$ is used as the initial solution.
4. *Constructive.* Starting from a minimal solution (second option), deletion of elements is banned until a consistent set is obtained. Then normal tabu search is resumed.

The authors' of [16] favor the *constructive* initialization. In the experiments of Section 5 we used their constructive initialization but did not wait until a consistent set was obtained. The initial incremental phase was terminated after a prespecified number of iterations (20 in all of our experiments).

4. Prototype replacement

This section contains brief overviews of the seven methods used in our numerical experiments. Following the architecture of Figure 3, we have subsections for pre-supervised and post-supervised methods.

Pre-supervised designs

Learning vector quantization (LVQ) : The LVQ family comprises a large spectrum of competitive learning schemes [9]. One of the basic designs that can be used for prototype generation is the LVQ1 algorithm. An initial set of *labeled* prototypes is picked by first specifying $n_p \geq c$. Then n_p elements are randomly selected from X to be the initial prototypes, so each class is represented by at least one prototype. LVQ1 has three additional parameters specified by the user: the *learning rate* $\alpha_k \in (0,1)$, a constant $\eta \in (0,1)$ and the terminal number of iterations T. The standard competitive learning update equation is then used to alter the prototype set. If the closest prototype for input $\mathbf{x}_k$ is the vector $\mathbf{v}_{i,\text{old}}$, the LVQ1 update strategy is:

$$\mathbf{v}_{i,\text{new}} = \mathbf{v}_{i,\text{old}} + \alpha_k(\mathbf{x}_k - \mathbf{v}_{i,\text{old}}) \quad \text{when } l(\mathbf{v}_{i,\text{old}}) = l(\mathbf{x}_k) \quad ; \text{or} \tag{5a}$$

$$\mathbf{v}_{i,\text{new}} = \mathbf{v}_{i,\text{old}} - \alpha_k(\mathbf{x}_k - \mathbf{v}_{i,\text{old}}) \quad \text{when } l(\mathbf{v}_{i,\text{old}}) \neq l(\mathbf{x}_k) \quad . \tag{5b}$$

Equation (5a) rewards the winning prototype for getting the correct label by letting it migrate towards the input vector, while (5b) punishes the winning prototype for not labeling the current input correctly by repelling it away from the input vector . In our experiments the learning rate was updated after each presentation of a new vector using the formula $\alpha_{k+1} = \eta\alpha_k$, k= 1,..., n-1; and was restored to the initial, user specified value of α_1 at the end of each pass through X. Before each new pass through LVQ1, X is randomly permuted to avoid dependence of the extracted prototypes on the order of inputs. LVQ1 terminates when either (i) there are no misclassifications in a whole pass through X (and hence, the extracted prototypes are a consistent reference set); or (ii) the prespecified terminal number of iterations is reached. The set of prototypes at termination of LVQ1 is $\mathbf{V}_r$.

Decision surface mapping (DSM) : Geva and Sitte's DSM [13] is a variant of LVQ which they assert better approximates classification boundaries of the training data than LVQ does. These authors say the price for better classification rates is that the DSM is somewhat less stable than standard LVQ's. The only difference between LVQ1 and DSM is that a punishment-reward step is taken *only* in case of misclassification of the current element of X. In LVQ1 the winning prototype is either punished or rewarded, depending on the outcome of the 1-np label match to the input. But in DSM, equations (5) are both implemented only if misclassification occurs. Thus, when the 1-np rule produces the correct label, no update is made, but when misclassification occurs, both (5a) and (5b) are implemented. In this case the winner (from the wrong class) is punished by (5b) and the nearest prototype from the same class as the current input is identified and rewarded by (5a).

Both LVQ1 and DSM operate with a fixed number of prototypes chosen by the user, so are user-n_p methods. An auto-n_p modification of LVQ that can prune and relabel prototypes was proposed by Odorico [12], who called it LVQTC.

LVQ with training counters (LVQTC) : Odorico's LVQTC algorithm operates similarly to LVQ1. The amount that the winning prototype $\mathbf{v}_{i,\mathrm{old}}$ is updated depends on the distance to input $\mathbf{x}_k$ and the history of the prototype. A prototype's historical importance is determined by the number of times it has been the winner. The procedure is initialized as is LVQ1, with the added twist that an empty training counter with c slots (one for each class) is attached to each prototype. When $\mathbf{x}_k$ is from class i, the i-th entry of the training counter of the nearest prototype will be incremented, regardless of the true label of that prototype. If the winning prototype is from class i (shares the same label as the input vector), then a slight variation of the reward equation (5a) is applied to the winning prototype $\mathbf{v}_{i,\mathrm{old}}$:

$$\mathbf{v}_{i,\mathrm{new}} = \mathbf{v}_{i,\mathrm{old}} + \frac{\alpha_k}{q_i}\left(\mathbf{x}_k - \mathbf{v}_{i,\mathrm{old}}\right) \qquad \text{when } l(\mathbf{v}_{i,\mathrm{old}}) = l(\mathbf{x}_k) \qquad , \tag{6}$$

where q_i is the total number of times that $\mathbf{v}_i$ has won the competition (i.e., the sum of all entries in its training counter vector). In (6) α_k is updated as in (5), but is not restored to its initial value at the end of each pass through X. The rationale for this treatment of the learning rate is that prototypes which have been modified many times have already found a good place in the feature space and should be less affected by subsequent inputs. This strategy can be thought of as a "decreasing reward" - not as strong as punishing the prototype for being wrong as in (5b) by sending it away - but more like making it less welcome the more times it shows up at your doorstep. Equation (6) is the same as the update equation for one of the earliest competitive learning models, viz., *sequential hard c-means* [26], for the particular choice $\alpha_k \equiv 1$. Odorico may or may not have recognized this, but in any case added some novel heuristics to the original algorithm which seem both justifiable and useful.

At the end of each pass through X the prototype counters $\{q_i\}$ are compared to a user-specified *retention threshold* Q. If $q_i < Q$ (indicating that $\mathbf{v}_i$ has not won the competition "often enough" to merit retention), $\mathbf{v}_i$ is removed from $\mathbf{V}$. The prototypes which survive the retention test are checked to see if relabeling is needed. If the number of "calls" to a prototype from a wrong class exceeds Q, the prototype is relabeled to that class, and is placed at the centroid of all the vectors from that class which called it. Although this algorithm has many heuristics and depends critically on the choice of Q, it has the advantage that the final prototypes $\mathbf{V}_r$ provide *soft class labels* that are easily defined by the training counters. These soft labels can be used to guide classification decisions.

Bootstrap editing . Four simple and intuitive bootstrap editing methods are proposed in Hamamoto et al. [27], all of which are pre-supervised, R-prototype extraction models. For the one that we use in Section 5 (which we will identify as BTS(3)), three parameters are user-specified: the number of nearest neighbors k, the desired number of prototypes n_p and the number of random trials T. A random sample of size n_p is then drawn from X. Each data point is replaced by the mean of its k-nearest neighbors from X with *the same class label.* The 1-np classifier is run on X using the new set of labeled prototypes. The best set from T runs is returned as the final $\mathbf{V}_r$. In our experience, Hamamoto et al.'s method gives pretty nice results. This bootstrapping scheme is a simple, fast, and unashamedly random way to get pre-supervised R-prototypes that seem to provide a low resubstitution error rate.

Post-supervised designs

Methods in this category disregard the class labels during training, and use X as if it were unlabeled to find a set $\mathbf{V}_r$ of algorithmically labeled prototypes. The prototypes are then *relabeled* using the training data labels. Methods in this group can be implemented in the pre-supervised mode by using the data from one class at a time, since this obviates the necessity for the relabeling step. When working with the data class by class, the prototypes that are found for each labeled class already have the assigned physical labels. The c subsets of prototypes are then pooled to form $\mathbf{V}_r$. We focus here on post-supervised designs. Your intuition probably tells you that pre-supervised prototype extraction ($\mathbf{V}_s$ as opposed to $\mathbf{V}_r$) will usually produce better 1-np classifiers than post-supervised models. We'll keep an eye on this supposition as we conduct numerical experiments in the next section (cf. Kuncheva and Bezdek [6], Kohonen [8], and Diamantini and Spalvieri [10]).

Vector Quantization (VQ) : Vector quantization has been used to find prototypes for many years. For example, Xie et al. [28] used it (in the pre-supervised mode) to support a version of k-nearest neighbor and Parzen classifier designs. In this study we adhere to the basic algorithm and apply it to the whole data set in the post-supervised mode. VQ starts with the user specifying n_p, and randomly selecting an initial set of n_p *unlabeled* prototypes from X. The closest prototype is always rewarded according to the update equation

$$\mathbf{v}_{i,\text{new}} = \mathbf{v}_{i,\text{old}} + \alpha_t(\mathbf{x}_k - \mathbf{v}_{i,\text{old}}) \quad . \tag{7}$$

The learning rate in (7) is indexed on t instead of k, t being the iteration counter (one iteration is one pass through X). The update rule we used is $\alpha_{t+1} = \left(1 - \frac{t}{T}\right)\alpha_t, t = 1, \ldots, T$, where T is a prespecified maximum number of passes through X. Termination is achieved by reaching T or by satisfying a measure of closeness of successive iterates to a user-defined termination threshold, i.e., $\|\mathbf{V}_{k+1} - \mathbf{V}_k\| \le \varepsilon$. To assign physical labels to the prototypes, the 1-np rule is applied to X using the extracted prototypes. The number of winners for each prototype from all c classes are counted. Finally, the most represented class label is assigned to each prototype. Let n_{ij} be the number of elements from class j associated with prototype $\mathbf{v}_i$ via the 1-np rule. Class label k is assigned to prototype $\mathbf{v}_i$ when it holds the majority of votes:

$$n_{ik} = \underbrace{\max}_{1 \le j \le c}\{n_{ij}\} \Rightarrow l(\mathbf{v}_i) = \mathbf{e}_k \quad . \tag{8}$$

Ties in (8) are broken randomly. It is not difficult to show that this relabeling strategy guarantees the smallest number of misclassifications of the resultant 1-np classifier on X. This relabeling scheme is used in all of our other post-supervised designs.

Generalized Learning Vector Quantization - Fuzzy GLVQ-F : GLVQ-F is an unsupervised method for finding prototypes which is similar to VQ but at each step *all* prototypes are updated. The update formula for the special case of weighting exponent m = 2 as given in [29] is

$$\mathbf{v}_{i,\text{new}} = \mathbf{v}_{i,\text{old}} + u_i\alpha_t(\mathbf{x}_k - \mathbf{v}_{i,\text{old}}) \quad \text{, where} \tag{9a}$$

$$u_i = \sum_{j=1}^{c}\left(\frac{\|\mathbf{x} - \mathbf{v}_i\|^2}{\|\mathbf{x} - \mathbf{v}_j\|^2}\right) \quad , 1 \le i \le c \quad . \tag{9b}$$

The rest of the GLVQ-F algorithm is the same as our specification of VQ. See [29] for limit analysis of GLVQ-F, which reduces to VQ under certain conditions.

Clustering and relabeling : Instead of finding prototypes by sequential competitive learning, we can cluster X with any batch clustering algorithm that generates cluster centers, and take the centroids as $\mathbf{V}_r$. This approach can be useful if the centroids are "good" representatives of the clusters in the data (of course, we won't be able to ascertain this for data that have more than p=3 features). Good candidates include the c-means methods [1, 30]. X is clustered disregarding the class labels. The centroids are taken as $\mathbf{V}_r$. The relabeling step assigns a class label to each cluster according to (8). Our experiments use only classical hard c-means [1].

Summarizing, the eleven classifier methods we chose for prototype extraction are listed in Table 1. The notation in the last three columns in Table 1 is used in Figures 9-12: selection = **(S)**, replacement = **(R)**, pre-supervised = **[P]**, post-supervised = **(p)**, auto-n_p = **(A)** and user-n_p = **(U)**. The notation **(A)** ↓ in Table 1 means that the algorithm can only *decrease* the initially specified number of prototypes. LVQTC prunes prototypes after each pass through X, and the HCM implementation [Duin, 31] can return a number of clusters that is smaller than the number of classes in the labeled data.

Table 1. Eleven (of zillions!) methods for finding prototypes

#	Acronym	S or R	Pre/Post	n_p
c1	W+H	**(S)**	**[P]**	**(A)**
c2	GA	**(S)**	**[P]**	**(A)**
c3	RND	**(S)**	**[P]**	**(U)**
c4	Tabu	**(S)**	**[P]**	**(A)**
c5	LVQ1	**(R)**	**[P]**	**(U)**
c6	DSM	**(R)**	**[P]**	**(U)**
c7	LVQTC	**(R)**	**[P]**	**(A)** ↓
c8	BTS(3)	**(R)**	**[P]**	**(U)**
c9	VQ	**(R)**	**(p)**	**(U)**
c10	GLVQ-F	**(R)**	**(p)**	**(U)**
c11	HCM	**(R)**	**(p)**	**(A)** ↓

Why these eleven methods? Our choice was guided by two criteria: (i) the methods should be simple and effective, with a relatively small number of parameters to specify; and (ii), as much of the methodological spectrum shown in Figure 3 as possible should be represented.

5. The Data Sets and Numerical Experiments

We use two small, artificially generated data sets, and two data sets from real application domains. Characteristics of the four data sets are shown in Table 2.

Table 2. Characteristics of the four data sets used in our computational experiments

Name	p # of features	c # of classes	$\lvert X_{tr} \rvert$	$\lvert X_{test} \rvert$	Electronic Access
Cone-Torus	2	3	400	400	http://www.bangor.ac.uk/~mas00a/Z.txt
Normal Mixture	2	2	250	1000	http://www.stats.ox.ac.uk/~ripley/PRNN/
Phoneme	5	2	500	4904	ftp.dice.ucl.ac.be, directory pub/neural-nets/ELENA/
Satimage	36 (4 used)	6	500	5935	ftp.dice.ucl.ac.be, directory pub/neural-nets/ELENA/

A. Cone-torus data. This is a c = 3 class dataset with n = 400 training points (X_{tr}) in $\Re^2$ generated from three differently shaped distributions: a cone (92 training data), half a torus (99 training data), and a normal distribution (209 training data) with prior probabilities for the corresponding classes of 0.25, 0.25 and 0.5. A separate test data set (X_{test}) with 400 more points generated from the same distributions is also provided.

B. Normal mixtures data. This dataset is used by Ripley [31] to illustrate various classification techniques. The training data consists of 125 two-dimensional samples drawn from a mixture of two 2-variate normal distributions with the same covariance matrix. A labeled test set containing 1000 more points drawn from the same mixture distribution is also provided. Ripley asserts that the class distributions have been chosen so that the best possible error rate on the test data is about 8 % .

C. Phoneme data. The Phoneme dataset consists of 5404 5-dimensional vectors characterizing two classes of phonemes: nasals (70.65 %) and orals (29.35 %). A series of classification results based on the Phoneme data are presented in Holmstrom et al. [32], who report classification accuracy that varies between 11%- 25 %, with the first half of the data set used for training and the second half for testing. In our experiments we used the first 500 data points for training and the remaining 4904 for testing.

D. Satimage data. This dataset, generated from the Landsat Multi-Spectral Scanner image data, consists of 6435 pixels, each of which has 36 intensities attached to it. The 36 features comprise 4 spectral bands, and each center pixel provides 9 intensities from its 3×3 neighborhood. Pixels are crisply labeled into one of c = 6 classes, and are presented in random order in the database. The classes are: red soil (23.82 %), cotton crop (10.92 %), grey soil (21.10 %), damp grey soil (9.73 %), soil with vegetation stubble (10.99 %), and very damp grey soil (23.43 %). Our experiments are based on using the 4 features 17, 18, 19 and 20, which are the features recommended by the designers of the database. The first 500 points in this feature subset of the Satimage data were used for training, and the remaining 5935 points in these four features were used for testing.

Our experiments were not configured to find the best classification accuracy that might be possible for a given data set. Rather, our objective here is to use the four data sets as a means for *comparing* various aspects of the 11 prototype extraction methods when the prototypes are used as the basis of the 1-np classifier. Various combinations of initialization parameters were tried in preliminary experiments and the most successful ones were picked for the reported comparisons. You can undoubtedly find better combinations of parameters for the various methods if time is not an obstacle. HCM was implemented using the version available in the PRTOOLS toolbox of Duin [33]. The number of clusters returned by Duin's HCM can be smaller than the initial number $n_{p,ini}$ because some of the clusters may turn out to be degenerate (i.e., empty). Table 3 shows the experimental protocols that we used for each of the 11 methods.

Table 3. Experimental protocols

#	Method	Initialization parameters				
c1	W+H	$k = 3$				
c2	GA	$ps = 10$	$T = 100$	$P_m = 0.1$	$P_{ini} = 0.1$	$\alpha = 0.5$
c3	RS	$T = 500$	$n_p = \{10, 20\}$			
c4	Tabu	$T = 100$ (+20 preliminary iterations) T_t (tabu tenure) = 3 % of the chromosome length, $\alpha = 0.05$				
c5	LVQ1	$\alpha_{ini} = 0.3$	$\eta = 0.8$	$T = 100$	$n_p = 20$	
c6	DSM	$\alpha_{ini} = 0.3$	$\eta = 0.8$	$T = 100$	$n_p = 20$	
c7	LVQTC	$\alpha_{ini} = 0.3$ $n_{p,ini} = 20$	$\eta = 0.8$	$T = 100$	$Q = 20$	
c8	BTS(3)	$k = 3$	$T = 100$			
c9	VQ	$\alpha_{ini} = 0.3$	$T = 100$	$n_p = 20$		
c10	GLVQ-F	$\alpha_{ini} = 0.3$	$T = 100$	$n_p = 20$		
c11	HCM	$n_{p,ini} = 20$				

Table 4. Experimental results (best case) for the four data sets

#	Method	Training error [%]	Testing error [%]	n_p	% reduction
Cone-torus data: (cols. 4 and 5 plotted in Figure 5)					
c1	W+H	10.75	13.00	36	91
c2	GA	10.00	14.50	77	80.75
c3	RND	15.75	14.50	20	95
c4	Tabu	**9.5**	14.75	21	94.75
c5	LVQ1	15.50	14.50	20	95
c6	DSM	22.25	20.75	20	95
c7	LVQTC	17.00	16.00	**13**	96.75
c8	BTS(3)	16.00	**12.25**	20	95
c9	VQ	14.75	15.75	20	95
c10	GLVQ-F	17.00	18.25	20	95
c11	HCM	14.25	16.00	19	95.25
Normal Mixture data: (cols. 4 and 5 plotted in Figure 6)					
c1	W+H	10.4	14.9	23	90.8
c2	GA	8.8	11.3	42	83.2
c3	RND	10.8	10.65	**10**	96
c4	Tabu	**8.4**	12.9	**10**	96
c5	LVQ1	10.0	**8.6**	20	94
c6	DSM	12.4	9.0	20	92
c7	LVQTC	10.0	10.5	15	94
c8	BTS(3)	10.4	10.2	20	92
c9	VQ	12.0	11.80	20	92
c10	GLVQ-F	12.0	9.2	20	92
c11	HCM	11.2	9.5	19	92.4
Phoeneme data: (cols. 4 and 5 plotted in Figure 7)					
c1	W+H	10.80	20.00	47	90.6
c2	GA	11.80	20.84	98	80.4
c3	RND	18.40	22.14	20	96
c4	Tabu	**9.80**	**18.80**	20	96
c5	LVQ1	15.4	21.53	20	96
c6	DSM	23.2	27.55	20	96
c7	LVQTC	17.6	21.02	**10**	98
c8	BTS(3)	17.00	22.29	20	96
c9	VQ	19.6	24.04	20	96
c10	GLVQ-F	19.40	23.45	20	96
c11	HCM	18.2	20.84	19	96.2
Satimage data: (cols. 4 and 5 plotted in Figure 8)					
c1	W+H	11.4	16.16	32	93.6
c2	GA	11.0	17.83	115	77
c3	RND	17.4	19.51	20	96
c4	Tabu	**10.4**	16.41	24	95.2
c5	LVQ1	15.2	16.68	20	96
c6	DSM	19.6	20.35	20	96
c7	LVQTC	16.0	17.51	**16**	96.8
c8	BTS(3)	16.6	17.17	20	96
c9	VQ	15.2	**15.86**	20	96
c10	GLVQ-F	15.6	16.34	20	96
c11	HCM	14.4	17.07	**16**	96.8

The training and testing errors and the terminal number of prototypes $|\mathbf{V}| = n_p$ for the eleven 1-np classifier methods and four data sets are shown in Table 4. For GA, Tabu search, LVQ1, DSM, LVQTC, VQ, GLVQ-F and HCM, the best of 5 independent runs is reported. Wherever the best training error over different runs was tied, the average of the test errors over the tied runs is given. The best results among the 11 competing schemes for training error, testing error, and number of prototypes are highlighted by bold font in Table 4.

The last column in Table 4 shows the % reduction in $|X_{tr}|$ on replacement of X_{tr} by $\mathbf{V}_s$ or $\mathbf{V}_r$. For example, classifier c1 in Table 4 selects 36 of the 400 training points in the cone-torus data as prototypes, so the percent reduction = 100* (400-36)/400=91%. Observe that $|\mathbf{V}| > c$ for all eleven methods. That is, all of these classifiers are multiple prototype classifiers, in the sense that there is at least one class in the training data which is represented by more than one prototype. This is consistent with both our intuition and earlier studies (cf. [30]).

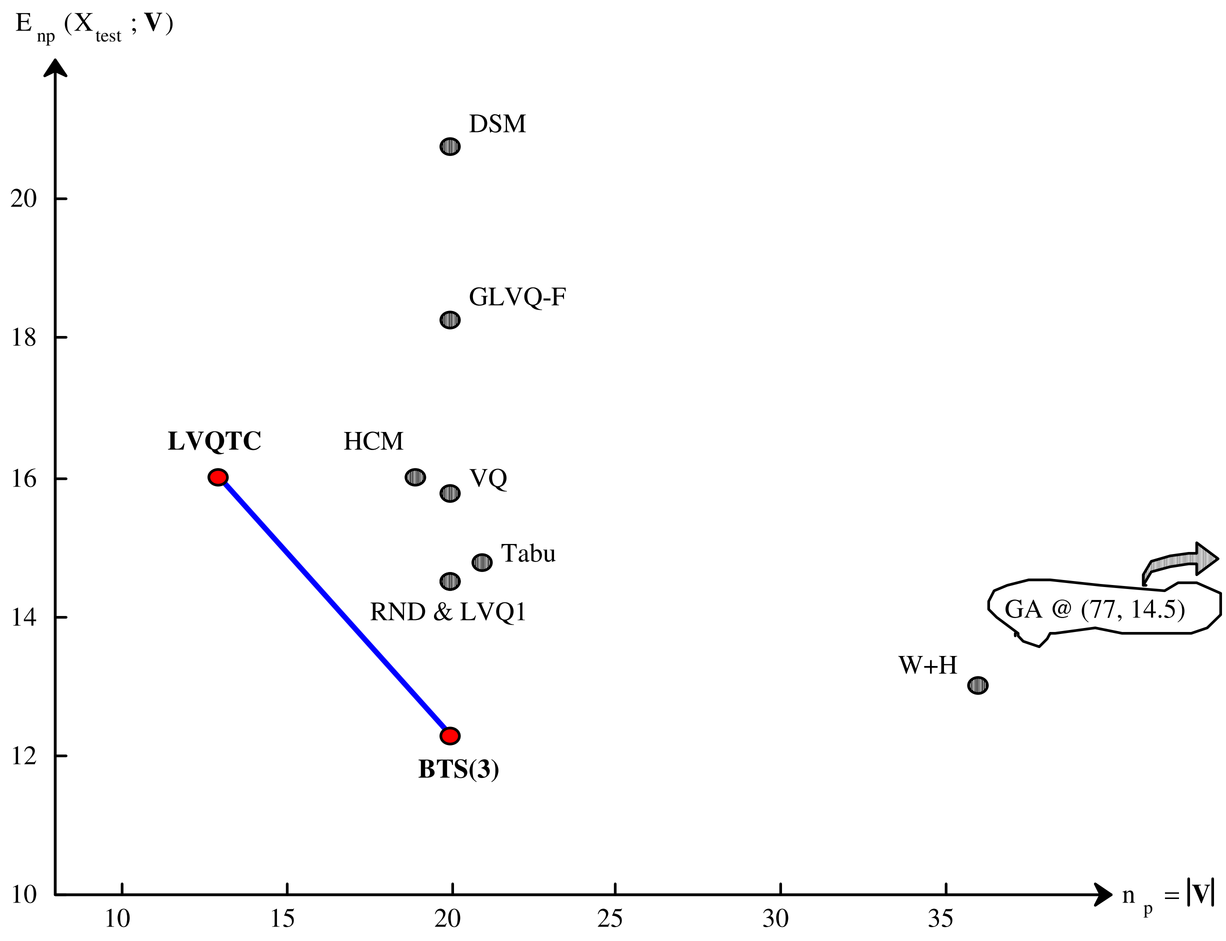

Figure 5. $n_p = |\mathbf{V}|$ **versus** $E_{np}(X_{test};\mathbf{V})$ **for the Cone-Torus data**

Figures 5-8 plot the test error rate (column 5 in Table 4) $E_{np} = E_{np}(X_{test}; \mathbf{V})$ versus the number of terminal prototypes $|\mathbf{V}| = n_p$ (prespecified or found, column 6 in Table 4) for the eleven methods and four data sets. Each method produces a point in the two dimensional space whose coordinate axes are $\left(n_p, E_{np}\right)$. The closer a point is to the origin, the better the 1-np classifier, because such a classifier will have a small number of prototypes and also a small test error rate.

Figure 5 has coordinates for 10 of the 11 methods. The GA approach resulted in 77 prototypes for the cone-torus data, so we decided not to plot this point, to keep the scale in Figure 5 to a size where the other 10 methods can be seen more clearly. The same thing occurs with the other three data sets; the number of prototypes chosen by GA is much larger than those found by the other 10 methods, so we again left the GA coordinates off-scale. This also occurs for the (W+H) classifier in Figures 7 and 8.

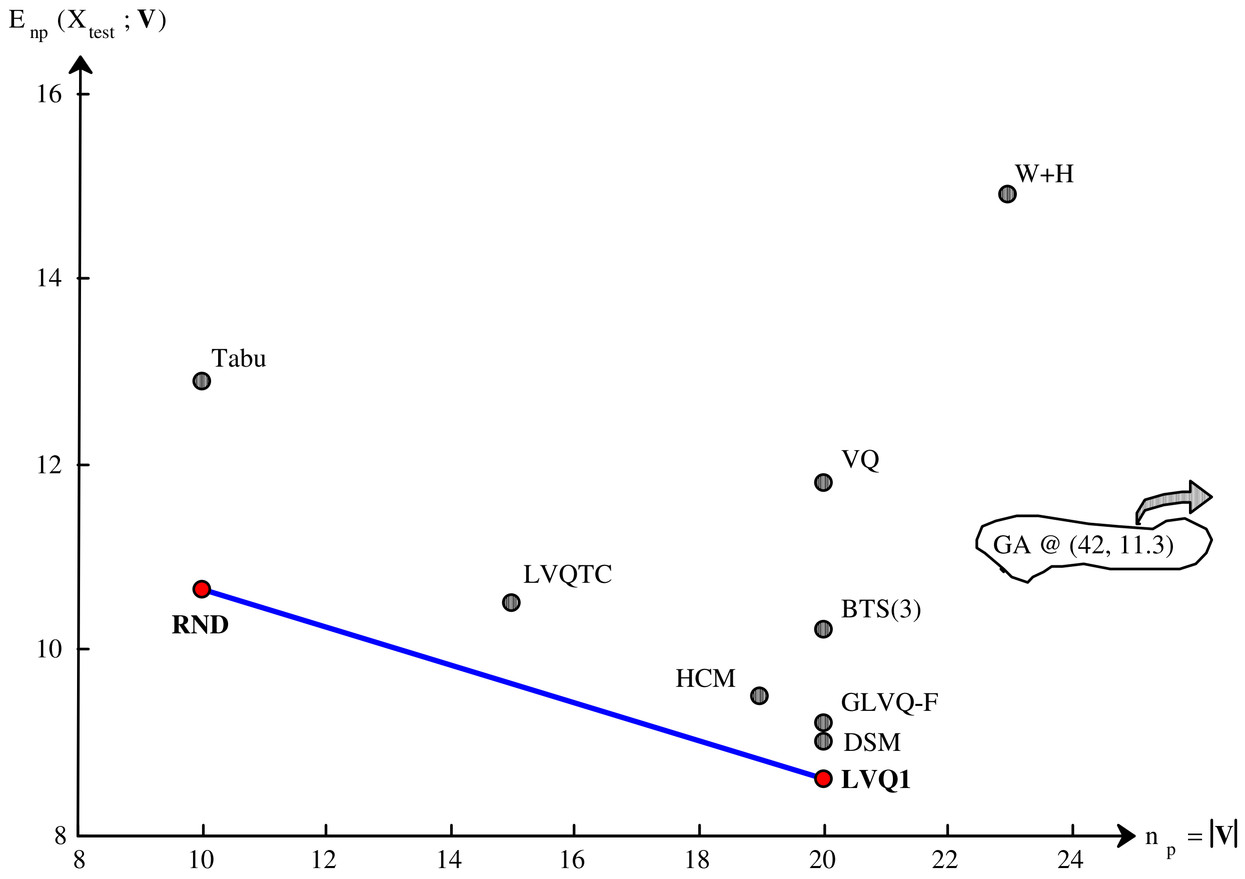

Figure 6. $n_p = |\mathbf{V}|$ **versus** $E_{np}(X_{test}; \mathbf{V})$ **for the Normal Mixtures data**

Figure 6 plots the same 10 methods as Figure 5 for training and testing with the normal mixture. The best classifiers in Figure 6 are RND and DSM, neither of which "wins" the best classifier contest depicted by Figure 5. This illustrates emphatically just how data dependent classifier design can be, and how ludicrous it is to look for a "best all-around" classifier design that generalizes well across a variety of different data regimes.

For the Phoneme data, the LVQTC and Tabu based 1-np designs produce the best results. Comparing Figure 7 to Figures 5 and 6 shows that over the first three data sets used for training and testing, our first repeat winner is the LVQTC based 1-np design. LVQTC occupies a place of distinction in both cases because it uses the minimum number of prototypes, but notice that this is achieved , for example, by sacrificing its testing error, which is worse for LVQTC in Figure 5 than for all but two of the competing 1-np designs.

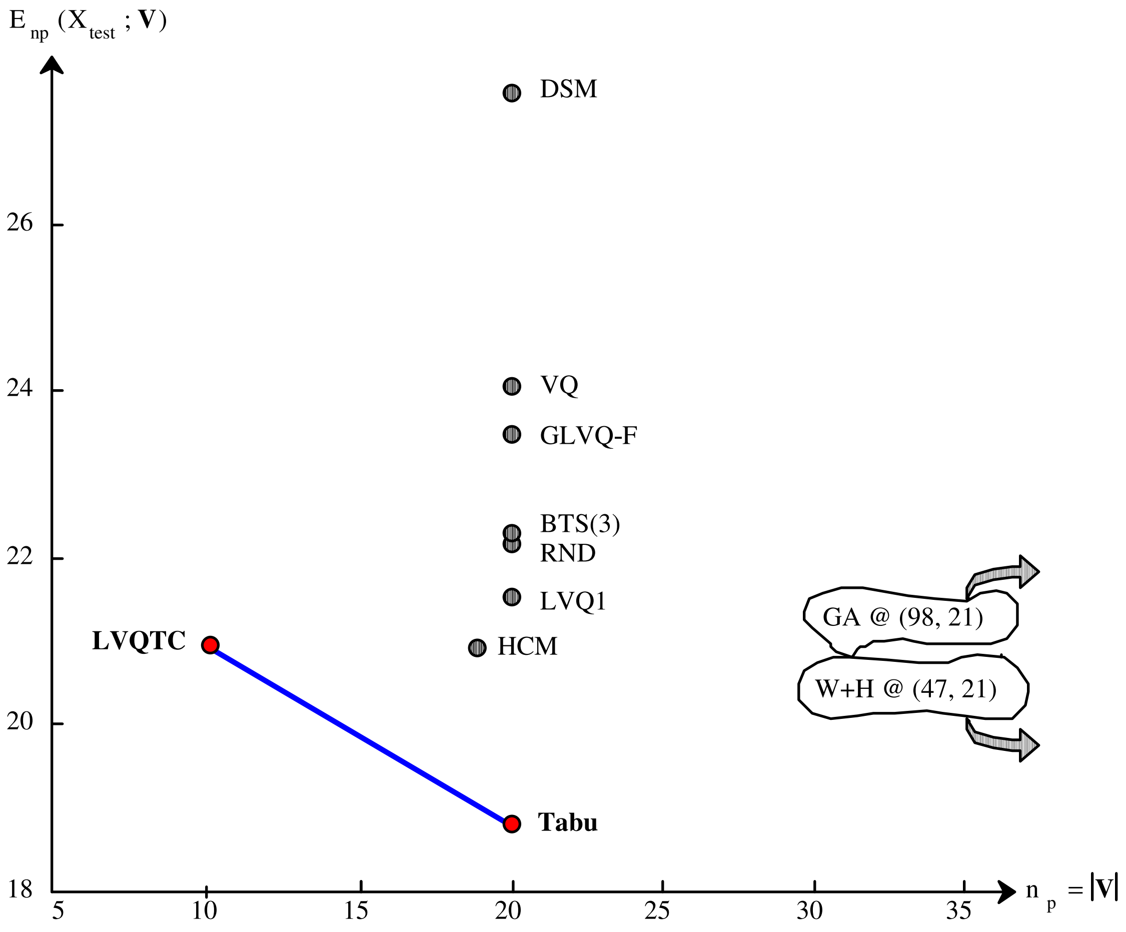

Figure 7. $n_p = |\mathbf{V}|$ **versus** $E_{np}(X_{test}; \mathbf{V})$ **for the Phoneme data**

Yet another pair of classifiers emerge as winners for the Satimage data, represented in Figure 8, where HCM and VQ based 1-np designs perform better in one coordinate or the other than the other nine classifier designs.

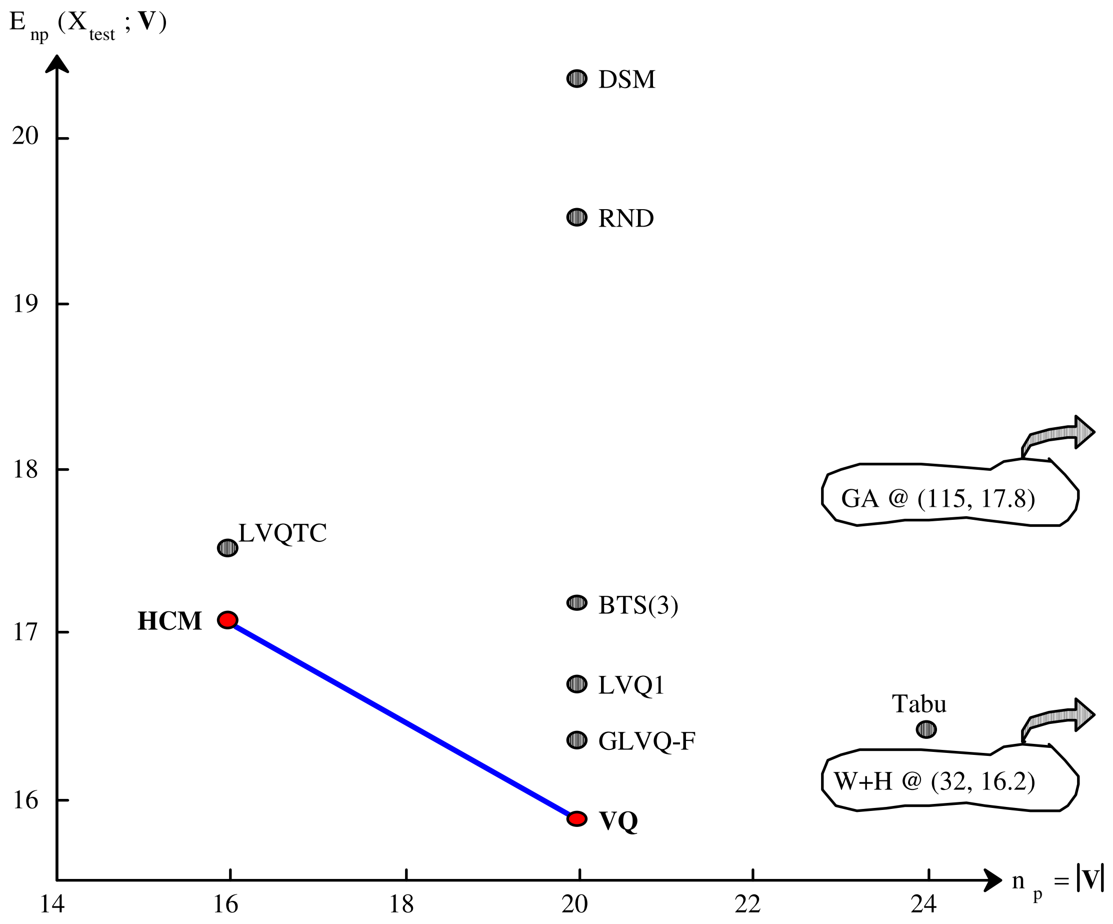

Figure 8. $n_p = |\mathbf{V}|$ **versus** $E_{np}(X_{test}; \mathbf{V})$ **for the Satimage data**

In some sense the best 1-np classifier *overall* will be the one closest to the origin in Figures 5-8, since this design will have the minimum $\left[|\mathbf{V}|^2 + E_{np}^2\right]$; the most *efficient representation* of the training data is perhaps the design that minimums $n_p = |\mathbf{V}|$; and the best design in terms of *generalization error* rate minimizes E_{np} alone. The tradeoff between minimal representation and maximal classification accuracy is evident from the fact that none of the classifiers studied have the smallest coordinates in both dimensions in Figures 5-8. Winning designs for each of the four data sets are connected by lines in Figures 5-8 and summarized in Table 5.

Table 5. Winning classifiers for the four data sets

Data Set	$\lvert X_{tr} \rvert$	$\lvert X_{test} \rvert$	p	c	min $n_p = \lvert V \rvert$	min E_{np}	min $\lvert V \rvert^2 + E_{np}^2$
Cone-Torus	400	400	2	3	13 : LVQTC	BTS(3)	LVQTC
Normal Mixture	250	1000	2	2	10 : RND	LVQ1	RND
Phoneme	500	4904	5	2	10 : LVQTC	Tabu	LVQTC
Satimage	500	5935	4	6	16 : HCM	VQ	HCM

The two ties in Table 4, n_p = 10 with RND and Tabu; and n_p = 16 with LVQTC and HCM, were resolved by choosing the classifier for Tables 5 and 6 that was closer to the origin in Figures 6 and 8, respectively. While this gives us a way to declare a "winner" for these two data sets, please notice that there is hardly any difference between these two designs and the two that do not appear in Table 5 because of our tie-breaking strategy.

Now we return to the three characteristics of 1-np designs listed in Section 1 : (C1) *Selection* **(S)** ($V_s \subseteq X$) versus *replacement* **(R)** ($V_r \not\subset X$); (C2) *Pre-supervised* **[P]** versus *post-supervised* **(p)** designs; and (C3)*User-defined* n_p **(U)** (*user-n_p*) versus *algorithmically defined* n_p **(A)** (*auto-n_p*). One way to evaluate the overall effect of (C1), (C2) and (C3) on the classifiers is to combine the information in Tables 1 and 5. Towards this end, column 1 of Table 6 shows the winning 1-np classifier designs from Table 5, while each row in Table 6 has a set of 3 check (✓) marks corresponding to the three characteristics of the method from Table 1.

Table 6. A summary of the winning designs for the four data sets

Method	Wins in	(S)	(R)	[P]	(p)	(U)	(A)
LVQTC	Fig. 5		✓	✓			✓
BTS(3)	Fig. 5		✓	✓		✓	
LVQ1	Fig. 6		✓	✓		✓	
RND	Fig. 6	✓		✓		✓	
Tabu	Fig. 7	✓		✓			✓
LVQTC	Fig. 7		✓	✓			✓
HCM	Fig. 8		✓		✓		✓
VQ	Fig. 8		✓		✓	✓	
Σ(✓)		2	6	6	2	4	4

The last row in Table 6 shows the number of checks in each column, and the paired "scores" for each of (C1) - (C3) give a very crude indication of relative efficacy of each coordinate in the pair.

Since there are 8 optimal points in Figures 5-8, each pair of coordinates sums to 8: e.g., (S, R) = (2, 6) in the last row of Table 6, so replacement methods achieved results at least as good or better than selection algorithms in 6 of the 8 of the trials represented for the 4 data sets. As you think about this statement, bear in mind that the coordinate pairs on which it is based are in (n_p, E_{np}) space, and only one pair of coordinates per method appears in each of Figures 5-8. It is incorrect to infer from columns 3 and 4 of Table 6, for example, that R methods achieve lower test error rates than S methods in 75% of the trials. For instance, HCM posts a "win" in Figure 8, but there are 5 other classifiers that yield a lower testing error than HCM on the Satimage data. HCM appears in Table 6 because none of its competitors can do better with as few prototypes. A better interpretation of the information in Tables 5 and 6 bears in mind that Figures 5-8 all have an underlying (implicit, but very real) tradeoff going on between n_p and E_{np}.

Figures 9-11 try to address the relevance of (C1)-(C3) to 1-np classifier design in a slightly different way. In these three figures the horizontal axis is the *ratio* of n_p, the number of prototypes found or used, to the cardinality of the training set X_{tr}, averaged over the four data sets A, B, C, D, say $\bar{n}_p = \sum_{i=1}^{4} \left(|\mathbf{V}_i| / 4 \cdot |X_{tr,i}| \right)$, where index i runs over the four data sets in question. The vertical axis in Figures 9-11 is the training error rate averaged over the four data sets, i.e., $\bar{E}_{np} = \sum_{i=1}^{4} E_{np}(X_{tr,i};\mathbf{V}_i)/4$. The GA-based 1-np classifier does not appear in these three figures because its horizontal coordinate is much larger than any of the other classifiers, and we wanted to again show the comparisons at a scale which enables you to see differences amongst the classifiers close to the origin. The 10 points in Figures 9-11 have the same coordinates in all three figures - what changes from Figure 9 to Figure 10 to Figure 11 is the *label* attached to each of the points. In Figure 9, we compare selection to replacement; in Figure 10 pre-supervised designs are compared to post-supervised designs; and in Figure 11, you can see how user-n_p methods stack up to auto-n_p methods. Classifiers which are closer to the origin in Figures 9-11 are again better than ones which are farther away from it.

Figure 9 compares 1-np designs based on prototypes *in* the training data (selection, **(S)**) to those based on prototypes built *from* the training data (replacement, **(R)**). A replacement method (LVQTC) assumes the minimum for the average ratio of $|\mathbf{V}_i| / |X_{tr,i}|$, while a selection method (BTS(3)) yields, albeit slightly, the smallest average test error rate. The "cluster of 6" methods (shown within the dashed boundary) closest to the origin in Figure 9 contains three from each category, implying that it is equally likely that you can find a decent set of prototypes for 1-np classifier design using either strategy.

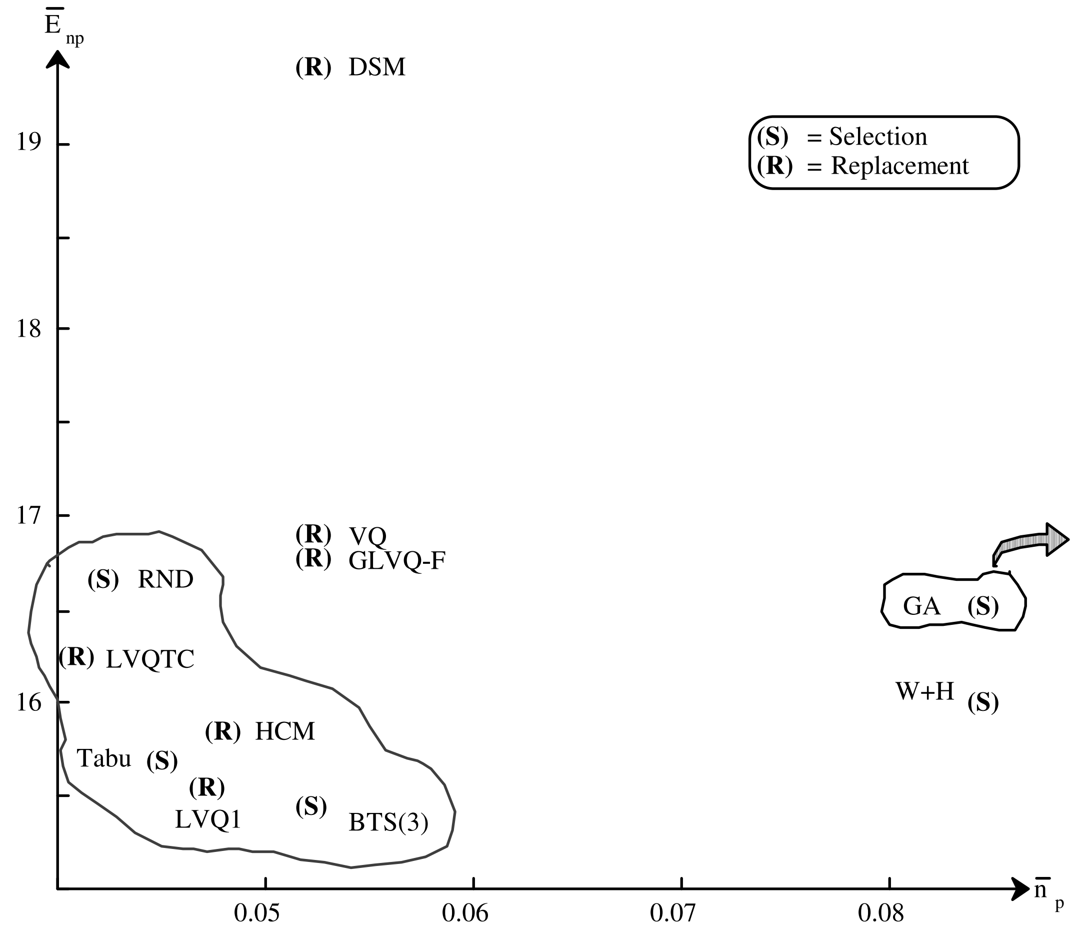

Figure 9. Selection vs. replacement : averages over four data sets

Figure 10 compares pre- to post-supervision. The "cluster of 6" in Figure 10 contains 5 **[P]**'s and only 1 **(p)**: this seems to confirm what your intuition suspects, that using the labels to derive **V** is usually more effective than using them to label **V** after finding it. On the other hand, the two pre-supervised "outlier" designs in Figure 10 (W+H, GA) are all much worse than the three post-supervised methods (HCM, VQ, GLVQ-F), so it is not the case that presupervision is *always* better than post-supervision. Don't forget that Figure 10 is based on computations with four pretty small data sets, and we would not be too shocked if you found four other data sets where the conclusions suggested by your calculations were very different than these.

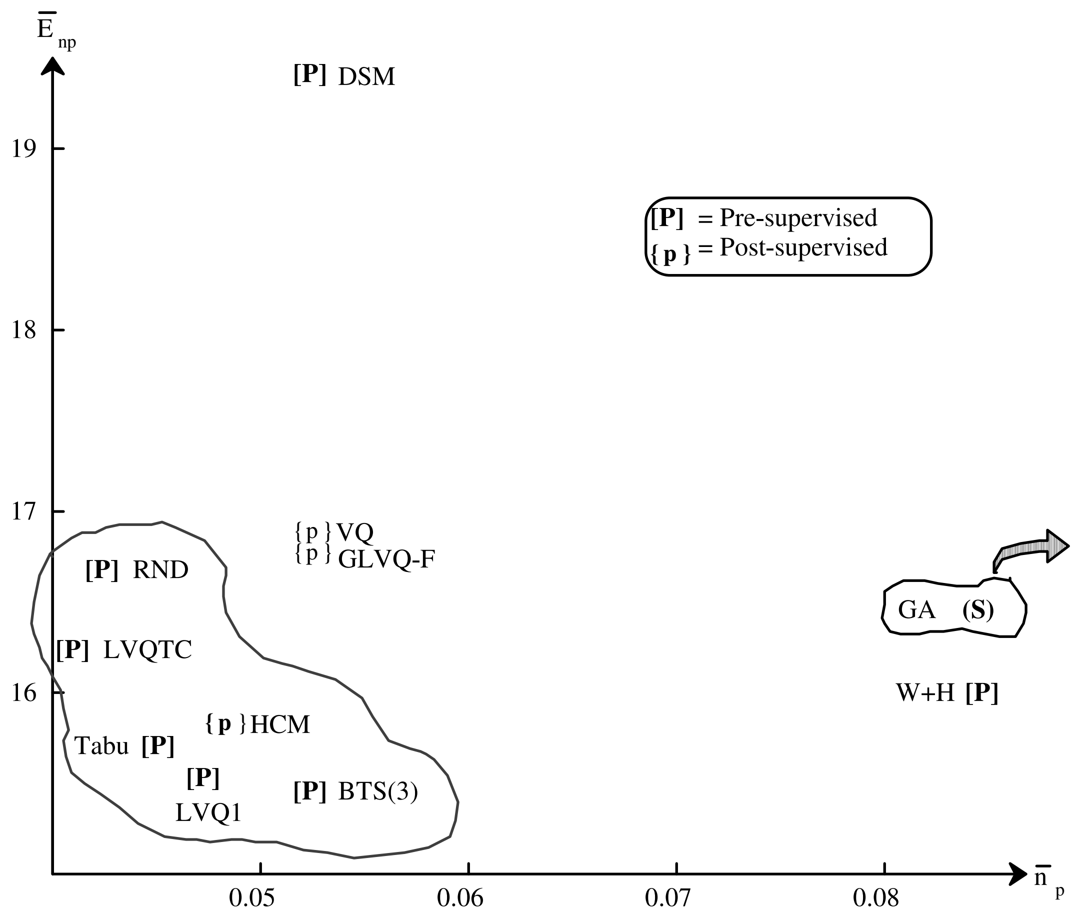

Figure 10. Pre-supervised versus post-supervised designs : averages over four data sets

Figure 11 compares methods that find the number of prototypes automatically (Auto-n_p, **(A)**) to methods in which the user has to tell the algorithm how many to look for (user-n_p, **(U)**). Thus, we see that the minimum number of prototypes is discovered automatically by LVQTC, while the minimum average error rate is produced by BTS(3), a method in which the user selects the number of prototypes, presumably with trial and error comparisons. Figure 11 doesn't provide conclusive evidence for either style of prototype generation, since the "cluster of 6" in Figure 11 contains 3 auto-n_p and 3 user-n_p methods.

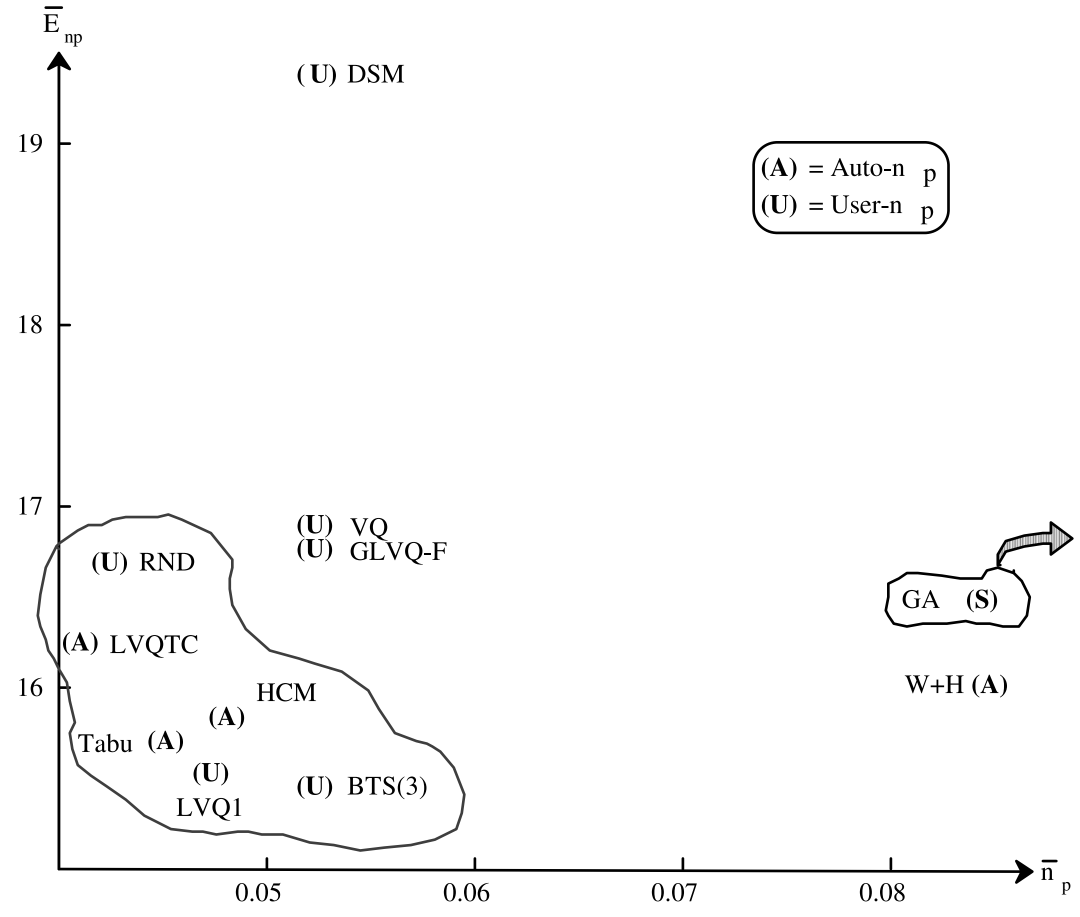

Figure 11. User-p vs. Auto-p : averages over four data sets

Figure 12 combines Figures 9-11 on one graph, where each of the 10 plotted 1-np classifiers is associated with one of the three coordinate triples (C1, C2, C3) from Table 1. As in Figures 5-11, the "best" designs are the ones closest to the origin, and the four Pareto-optimal designs [36] for averages over the four data sets are captured by the shaded region in Figure 12. The coordinates of these four designs in (C1, C2, C3) space show ratios of: 3:1 for replacement vs. selection, 4:0 for pre-supervised designs vs. post-supervised designs, and 2:2 for auto-n_p vs. user-n_p selection of the number of prototypes. This indicates that - at least for these data sets and trials - pre-supervised, replacement prototypes are the more desirable combination of (C1) and (C2), while finding the best *number* of prototypes is done equally well by user specification or "automatically".

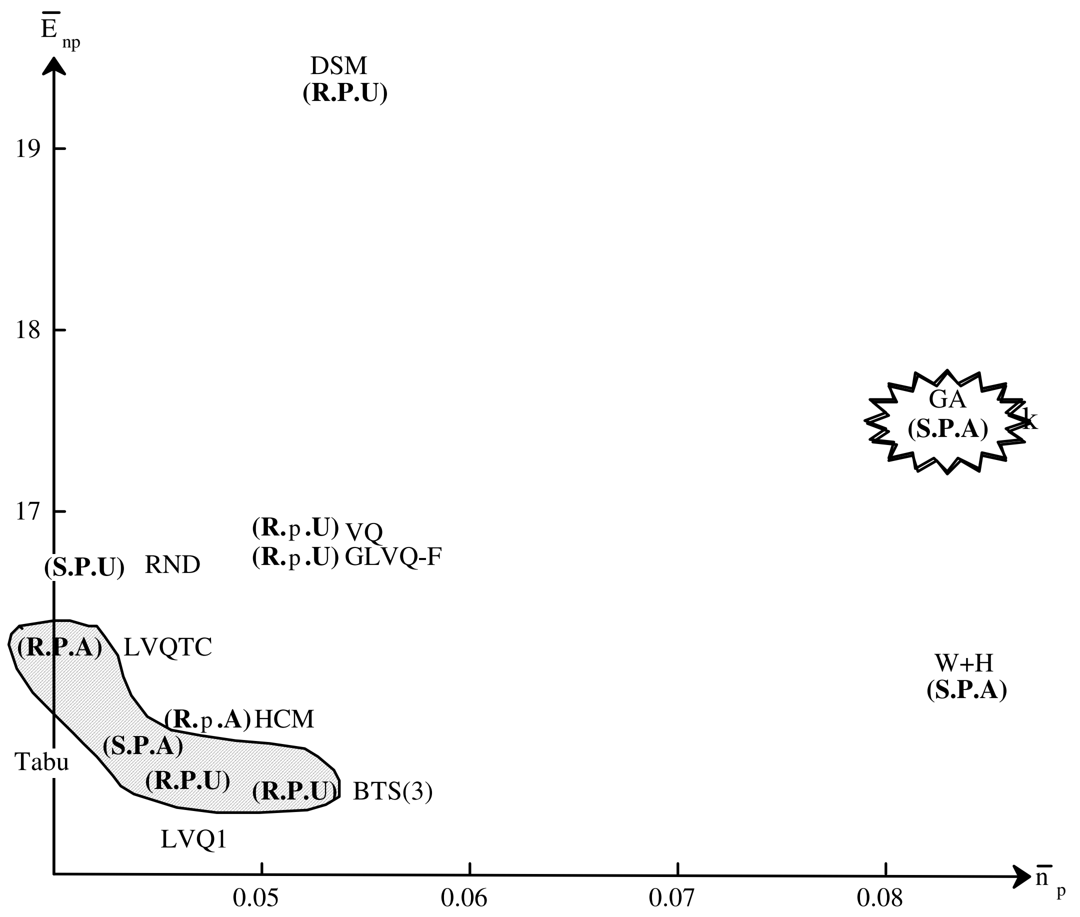

Figure 12. (C1, C2, C3) triples : averages over four data sets

6. Conclusions

We discussed 11 methods for 1-np classifier design, all of which fall under the umbrella of the generalized nearest prototype classifier model [5]. Our study explored the interaction of three characteristics of 1-np designs : (C1) *Selection* ($\mathbf{V}_s \subseteq X$) versus *replacement* ($\mathbf{V}_r \not\subset X$); (C2) *Pre-supervised* versus *post-supervised* designs; and (C3)*User-defined* n_p (*user-n_p*) versus *algorithmically defined* n_p (*auto-n_p*). The error rate on test data and the cardinality of the prototype set were the two criteria for comparison. Not surprisingly, different designs were better for different data sets (see Figures 5-8). Moreover, no clear tendency emerged with respect to (C1), (C2) or (C3). Overall, Tabu search and LVQTC appeared to be the most successful designs. We believe that part of the success of Tabu search is due to the "constructive" initialization scheme of Cerveron and Ferri {16]. In our experiments Tabu search found, usually within the first 20 iterations, an excellent initial guess which was almost never changed later in the algorithm, when the "do-not-delete" restriction is not in force. However, our study did not attempt to find a "best" 1-np design, but rather, to explore the importance of the three characteristics of all 11 designs. What can we say about each of these?

Our general belief about (C1) is that $\mathbf{V}_r \not\subset X$ can be better positioned (for 1-np classification) but more difficult to find than $\mathbf{V}_s \subseteq X$. Figure 12 shows that on average over 4 sets of data, 3 of the 4 best methods are replacement methods. These three methods (LVQ1, LVQTC, BTS(3)) share the property that they build replacement prototypes by optimizing local criteria instead of a global criterion such as that used by batch algorithms like HCM.

For (C2), intuition suggests that pre-supervised methods are better than post-supervised schemes because they use the additional information possessed by the labels of the training data used to find **V**. Figures 11 and 12 make it clear that pre-supervised methods were superior for these 4 data sets and 11 methods. This finding is consistent with that of an earlier study, where all 5 Pareto-optimal designs amongst 16 1-np classifiers that were compared with resubstitution errors committed in the Iris data were pre-supervised [7]. We are confident enough about this point to assert that *some* pre-supervised method will almost always produce labeled prototypes for 1-np classifier design that are superior to those found by post-supervised methods. You noticed our use of the words "some" and "almost always"? *Some*, because in pattern recognition the trick is always to find the right classifier; *almost* always, because there *is* always a data set out there which provides a glaring counter-example to anything you claim is always true. For an example, just notice that DSM, W+H, and GA (which falls outside the frame in Figure 12) are also pre-supervised designs, but they were all inferior to all of the post-supervised designs.

For (C3), auto-n_p methods will certainly be more popular than user-n_p designs, because they seem simpler to use. However, auto-n_p methods simply hide the trial and error procedure for finding the "best" number of prototypes from the user, and instead rely on some mathematical criterion that selects n_p. If the criterion is test error E_{np} , this relieves the user from conducting the trials and committing the errors (but necessitates the use of a validation test set; otherwise, there is no reason to believe that auto-n_p methods will be superior to user-n_p designs, and Figure 12 confirms this.

For each of the 4 data sets there is a fairly large group of designs with similar performance (those near the origin in Figures 5-12). The difference in testing errors or in the number of prototypes retained is small, and from our viewpoint these designs are all more or less equivalent. The GA design is missing from Figures 5-12 because the number of prototypes found by it was so high that if we put GA on the scatterplot, the scale would obscure differences between some of the other designs. In our previous study [Kuncheva and Bezdek, 6], GAs exhibited very good performance. Do we have an explanation for this inconsistency with earlier performance? Nope. Would other choices for α, *ps*, T, the initialization and mutation probabilities and the fitness function lead to better results? Sure - but then, you can improve any of these designs if you have enough time and patience.

What can we conclude about the trade-off between test error rate and the number of prototypes used? Nothing new, but these new experiments do support our convictions about this point more strongly. This trade-off exists, and the general trend you would expect - that plots of n_p vs. E_{np} take the general shape of the graph of y = 1/x - holds throughout Figures 5-12. Perhaps the most interesting thing we can observe about this aspect of the experiments is that none of the auto-n_p methods decided that n_p as small as c, or n_p as large as $|X_{tr}|$, provided the best solution. From this we conclude that you will *almost always* do better with $n_p > c$, i.e., more prototypes than there are classes in the training data, but should almost certainly stop short of using as many prototypes as there are training data (in this latter case, if the prototypes are being selected, you arrive at the k-nearest neighbor rule, which is probably a better choice than 1-np classifiers anyway - but that's another story).

References

[1] J. C. Bezdek, J. Keller, R. Krishnapuram and N. R. Pal, *Fuzzy Models and Algorithms for Pattern Recognition and Image Processing*, Kluwer, Norwell, MA, 1999.

[2] P. Domingos, Context-sensitive feature selection for lazy learners*, Artificial Intelligence Review*, 11, 1997, 227-235.

[3] R. Krishnapuram and J. M. Keller, (1993). A possibilistic approach to clustering, *IEEE Trans. Fuzzy Systems*, 1(2), 98-110.

[4] Zurada, J. (1992). *Introduction to Artificial Neural Systems,* West Publ. Co., St. Paul, MN.

[5] L. I. Kuncheva and J.C. Bezdek, An integrated framework for generalized nearest prototype classifier design, *International Journal of Uncertainty, Fuzziness and Knowledge-based Systems*, 6 (5), 1998, 437-457.

[6] L.I. Kuncheva and J.C. Bezdek, On prototype selection: Genetic algorithms or random search?, *IEEE Trans. on Systems, Man, and Cybernetics*, C28 (1), 1998, 160-164.

[7] L.I. Kuncheva and J.C. Bezdek, Pre-supervised and post-supervised prototype classifier design, *IEEE Trans. on Neural Networks*, 10(5),1999, 1142-1152.

[8] T. Kohonen, Improved versions of learning vector quantization*, Proc. Int. Joint Conference on Neural networks*, San Diego, CA, 1990, I-545-550.

[9] T. Kohonen, *Self-Organizing Maps*, Springer, Germany, 1995.

[10] C. Diamantini and A. Spalvieri, Quantizing for minimum average misclassification risk, *IEEE Trans. on Neural Networks*, **9** (1), 1998,174-182.

[11] C. Diamantini and A. Spalvieri, Certain facts about Kohonen's LVQ1 algorithm, *IEEE Trans CAS- 1: Fundamental Theory and Applications*, 43 (5), 1996, 425-427.

[12] R. Odorico, Learning vector quantization with training counters (LVQTC*), Neural Networks*, **10** (6), 1997, 1083-1088.

[13] S. Geva and J. Sitte, Adaptive nearest neighbor pattern classifier, *IEEE Trans. on Neural Networks*, **2** (2), 1991, 318--322.

[14] P. E. Hart, The condensed nearest neighbor rule, *IEEE Trans. on Information Theory*, IT-14, 1968, 515-516.

[15] B.V. Dasarathy, *Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques,* Los Alamitos, California: IEEE Computer Society Press, 1991.

[16] V. Cerveron and F.J. Ferri, Another move towards the minimum consistent subset: A tabu search approach to the condensed nearest neighbor rule, *IEEE Trans. on Systems, Man and Cybernetics, Part B:Cybernetics*, 31(3), 2001.

[17] B.V. Dasarathy, Minimal consistent set (MCS) identification for optimal nearest neighbor decision systems design, *IEEE Trans. on Systems, Man, and Cybernetics*, 24, 1994, 511-517.

[18] D. L. Wilson, Asymptotic properties of nearest neighbor rules using edited data, *IEEE Trans. on Systems Man and Cybernetics*, SMC-2, 1972, 408-421.

[19] P.A. Devijver and J. Kittler, On the edited nearest neighbor rule, *Proc. 5th International Conference on Pattern Recognition,* IEEE Computer Society Press, Los Alamitos, Calif.,1980, 72-80.

[20] P.A. Devijver and J. Kittler, *Pattern Recognition: A Statistical Approach*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1982.

[21] F. J. Ferri, Combining adaptive vector quantization and prototype selection techniques to improve nearest neighbor classifiers, *Kybernetika*, 34 (4), 1998, 405-410.

[22] D. B. Skalak, Prototype and feature selection by sampling and random mutation hill climbimg algorithms, *Proc. 11th International Conference on Machine Learning, New Brunswick, N.J.*, Morgan Kaufmann, Los Alamitos, CA, 1994, 293-301.

[23] E. I. Chang and R.P. Lippmann, Using genetic algorithms to improve pattern classification performance, in *Advances in Neural Information Processing Systems*, **3**, R.P. Lippmann, J.E. Moody and D.S. Touretzky, Eds., San Mateo, CA: Morgan Kaufmann, 1991, 797-803.

[24] L.I. Kuncheva, Editing for the k-nearest neighbors rule by a genetic algorithm, *Pattern Recognition Letters, Special Issue on Genetic Algorithms*, 16, 1995, 809-814.

[25] L.I. Kuncheva,Fitness functions in editing k-NN reference set by genetic algorithms, *Pattern Recognition*, 30, 1997, 1041-1049.

[26] J. MacQueen, J. Some methods for classification and analysis of multivariate observations,*Proc. Berkeley Symp. Math. Stat. and Prob.*, 1, eds. L. M. LeCam and J. Neyman, Univ. of California Press, Berkeley, 281-297.

[27] Y. Hamamoto, S. Uchimura and S. Tomita}, A bootstrap technique for nearest neighbor classifierdesign}, *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 19 (1), 1997, 73-79.

[28] Q. Xie, C.A. Laszlo and R.K. Ward, Vector quantization technique for nonparametric classifier design, *IEEE Trans. on Pattern Analysis and Machine Intelligence*, **15** (12), 1993, 1326-1330.

[29] N.B. Karayiannis, J.C. Bezdek, N.R. Pal, R.J. Hathaway, and P.-I. Pai, Repairs to GLVQ: A new family of competitive learning schemes, *IEEE Trans. on Neural Networks*, **7**, 1996, 1062-1071.

[30] J.C. Bezdek, T. R. Reichherzer, G. S. Lim, and Y. Attikiouzel, Multiple prototype classifier design, *IEEE Trans. on Systems, Man, and Cybernetics*, **C28** (1), 1998, 67-79.

[31] B.D. Ripley, *Pattern Recognition and Neural Networks*, University Press, Cambridge, 1996.

[32] L. Holmstrom, P. Koistinen, J. Laaksonen and E. Oja, Neural and statistical classifiers - taxonomy and two case studies, *IEEE Trans. on Neural Networks*, **8** (1), 1997, 5-17.

[33] R.P.W. Duin, PRTOOLS, Version 2. A Matlab toolbox for pattern recognition, Pattern Recognition Group, Delft University of Technology, June, 1997.

[34] R. O. Duda and P.E. Hart, *Pattern Classification and Scene Analysis*, John Wiley & Sons", N.Y., 1973.

[35] J.A. Anderson, Logistic discrimination, in P.R. Krishnaiah and L.N. Kanal, Eds., *Classification, Pattern Recognition of Dimensionality*, North Holland, Handbook of Statistics Series, **2**, 1982, 169-191.

[36] W. L Winston, Operations Research, Applications and Algorithms, 3rd ed., 1994, Duxbury Press, Belmont, CA.