Computational Intelligence, Volume 16, Number 1, pp. 1-28, 2000

AN INTEGRATED INSTANCE-BASED LEARNING ALGORITHM

D. RANDALL WILSON, TONY R. MARTINEZ

Computer Science Department, Brigham Young University, Provo, Utah 84602, USA
World-Wide Web: http://axon.cs.byu.edu
E-mail: randy@axon.cs.byu.edu, martinez@cs.byu.edu

The basic nearest-neighbor rule generalizes well in many domains but has several shortcomings, including inappropriate distance functions, large storage requirements, slow execution time, sensitivity to noise, and an inability to adjust its decision boundaries after storing the training data. This paper proposes methods for overcoming each of these weaknesses and combines these methods into a comprehensive learning system called the Integrated Decremental Instance-Based Learning Algorithm (IDIBL) that seeks to reduce storage, improve execution speed, and increase generalization accuracy, when compared to the basic nearest neighbor algorithm and other learning models. IDIBL tunes its own parameters using a new measure of fitness that combines confidence and cross-validation (CVC) accuracy in order to avoid discretization problems with more traditional leave-one-out cross-validation (LCV). In our experiments IDIBL achieves higher generalization accuracy than other less comprehensive instance-based learning algorithms, while requiring less than one-fourth the storage of the nearest neighbor algorithm and improving execution speed by a corresponding factor. In experiments on 21 datasets, IDIBL also achieves higher generalization accuracy than those reported for 16 major machine learning and neural network models.

Key words: Inductive learning, instance-based learning, classification, pruning, distance fuction, distance-weighting, voting, parameter estimation.

INTRODUCTION

The *Nearest Neighbor* algorithm (Cover & Hart 1967; Dasarathy 1991) is an inductive learning algorithm that stores all of the *n* available training examples (*instances*) from a *training set, T,* during learning. Each instance has an *input vector* ***x***, and an *output class c*. During *generalization*, these systems use a distance function to determine how close a new input vector ***y*** is to each stored instance, and use the nearest instance or instances to predict the output class of ***y*** (*i.e.*, to *classify* ***y***).

The nearest neighbor algorithm is intuitive and easy to understand, it learns quickly, and it provides good generalization accuracy for a variety of real-world classification tasks (*applications*).

However, in its basic form, the nearest neighbor algorithm has several weaknesses:

- Its distance functions are often inappropriate or inadequate for applications with both linear and nominal attributes (Wilson & Martinez 1997a).
- It has large storage requirements, because it stores all of the available training data in the model.
- It is slow during execution, because all of the training instances must be searched in order to classify each new input vector.
- Its accuracy degrades rapidly with the introduction of noise.
- Its accuracy degrades with the introduction of irrelevant attributes.
- It has no ability to adjust its decision boundaries after storing the training data.

Many researchers have developed extensions to the nearest neighbor algorithm, which are commonly called *instance-based* learning algorithms (Aha, Kibler & Albert 1991; Aha 1992; Dasarathy 1991). Similar algorithms include *memory-based reasoning* methods (Stanfill & Waltz 1986; Cost & Salzberg 1993; Rachlin et al. 1994), *exemplar-based generalization* (Salzberg 1991; Wettschereck & Dietterich 1995), and *case-based* classification (Watson & Marir 1994).

Some efforts in instance-based learning and related areas have focused on one or more of the above problems without addressing them all in a comprehensive system. Others have used solutions to some of the problems that were not as robust as those used by others.

The authors have proposed several extensions to instance-based learning algorithms as well (Wilson & Martinez 1996, 1997a,b, 1999), and have purposely focused on only one or only a few of the problems at a time so that the effect of each proposed extension could be evaluated independently, based on its own merits.

This paper proposes a comprehensive learning system, called the *Integrated Decremental Instance-Based Learning* (IDIBL) algorithm, that combines successful extensions from earlier work with some new ideas to overcome many of the weaknesses of the basic nearest neighbor rule mentioned above.

Section 1 discusses the need for a robust heterogeneous distance function and describes the *Interpolated Value Distance Metric*. Section 2 presents a *decremental* pruning algorithm (*i.e.*, one that starts with the entire training set and removes instances that are not needed). This pruning algorithm reduces the number of instances stored in the system and thus decreases storage requirements while improving classification speed. It also reduces the sensitivity of the system to noise. Section 3 presents a distance-weighting scheme and introduces a new method for combining cross-validation and confidence to provide a more flexible concept description and to allow the system to be fine-tuned. Section 4 presents the learning algorithm for IDIBL, and shows how it operates during classification.

Section 5 presents empirical results that indicate how well IDIBL works in practice. IDIBL is first compared to the basic nearest neighbor algorithm and several extensions to it. It is then compared to results reported for 16 well-known machine learning and neural network models on 21 datasets. IDIBL achieves the highest overall average generalization accuracy of any of the models examined in these experiments.

Section 6 presents conclusions and mentions areas of future research, such as adding feature weighting or selection to the system.

1. HETEROGENEOUS DISTANCE FUNCTIONS

There are many learning systems that depend upon a good distance function to be successful, including the instance-based learning algorithms and the related models mentioned in the introduction. In addition, many neural network models also make use of distance functions, including radial basis function networks (Broomhead & Lowe 1988; Renals & Rohwer 1989; Wasserman 1993), counterpropagation networks (Hecht-Nielsen 1987), ART (Carpenter & Grossberg 1987), self-organizing maps (Kohonen 1990) and competitive learning (Rumelhart & McClelland 1986). Distance functions are also used in many fields besides machine learning and neural networks, including statistics (Atkeson, Moore & Schaal 1996), pattern recognition (Diday 1974; Michalski, Stepp & Diday 1981), and cognitive psychology (Tversky 1977; Nosofsky 1986).

1.1. Linear Distance Functions

A variety of distance functions are available for such uses, including the Minkowsky (Batchelor 1978), Mahalanobis (Nadler & Smith 1993), Camberra, Chebychev, Quadratic, Correlation, and Chi-square distance metrics (Michalski, Stepp & Diday 1981; Diday 1974); the Context-Similarity measure (Biberman 1994); the Contrast Model (Tversky 1977); hyperrectangle distance functions (Salzberg 1991; Domingos 1995) and others.

Although there have been many distance functions proposed, by far the most commonly used is the Euclidean Distance function, which is defined as

[1] $$E(\mathbf{x},\mathbf{y}) = \sqrt{\sum_{a=1}^{m}(x_a - y_a)^2}$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are two input vectors (one typically being from a stored instance, and the other an input vector to be classified) and m is the number of input variables (*attributes*) in the application.

None of the above distance functions is designed to handle applications with both linear and nominal attributes. A *nominal* attribute is a discrete attribute whose values are not necessarily in any linear order. For example, a variable representing color might have values such as *red, green, blue, brown, black* and *white*, which could be represented by the integers 1 through 6, respectively. Using a linear distance measurement such as Euclidean distance on such values makes little sense in this case.

Some researchers have used the *overlap* metric for nominal attributes and normalized Euclidean distance for linear attributes (Aha, Kibler & Albert 1991; Aha 1992; Giraud-Carrier & Martinez 1995). The overlap metric uses a distance of 1 between attribute values that are different, and a distance of 0 if the values are the same. This metric loses much of the information that can be found from the nominal attribute values themselves.

1.2. Value Difference Metric for Nominal Attributes

The Value Difference Metric (VDM) was introduced by Stanfill and Waltz (1986) to provide an appropriate distance function for nominal attributes. A simplified version of the VDM (without weighting schemes) defines the distance between two values x and y of an attribute a as

[2] $$vdm_a(x,y) = \sum_{c=1}^{C}\left|\frac{N_{a,x,c}}{N_{a,x}} - \frac{N_{a,y,c}}{N_{a,y}}\right|^q = \sum_{c=1}^{C}\left|P_{a,x,c} - P_{a,y,c}\right|^q$$

where

- $N_{a,x}$ is the number of instances in the training set T that have value x for attribute a;
- $N_{a,x,c}$ is the number of instances in T that have value x for attribute a and output class c;
- C is the number of output classes in the problem domain;
- q is a constant, usually 1 or 2; and
- $P_{a,x,c}$ is the conditional probability that the output class is c given that attribute a has the value x, *i.e.*, $P(c \mid x_a)$. As can be seen from (2), $P_{a,x,c}$ is defined as

[3] $$P_{a,x,c} = \frac{N_{a,x,c}}{N_{a,x}}$$

where $N_{a,x}$ is the sum of $N_{a,x,c}$ over all classes, *i.e.*,

[4] $$N_{a,x} = \sum_{c=1}^{C} N_{a,x,c}$$

and the sum of $P_{a,x,c}$ over all C classes is 1 for a fixed value of a and x.

Using the distance measure $vdm_a(x,y)$, two values are considered to be closer if they have more similar classifications (*i.e.*, more similar correlations with the output classes), regardless of what order the values may be given in. In fact, linear discrete attributes can have their values remapped randomly without changing the resultant distance measurements.

One problem with the formulas presented above is that they do not define what should be done when a value appears in a new input vector that never appeared in the training set. If attribute a never has value x in any instance in the training set, then $N_{a,x,c}$ for all c will be 0, and $N_{a,x}$ (which is the sum of $N_{a,x,c}$ over all classes) will also be 0. In such cases $P_{a,x,c}$ = 0/0, which is undefined. For nominal attributes, there is no way to know what the probability should be for such a value, since there is no inherent ordering to the values. In this paper we assign $P_{a,x,c}$ the default value of 0 in such cases (though it is also possible to let $P_{a,x,c} = 1/C$, where C is the number of output classes, since the sum of $P_{a,x,c}$ for $c = 1..C$ is always 1.0).

If this distance function is used directly on continuous attributes, the values can all potentially be unique, in which case $N_{a,x}$ is 1 for every value x, and $N_{a,x,c}$ is 1 for one value of c and 0 for all others for a given value x. In addition, new vectors are likely to have unique values, resulting in the division by zero problem above. Even if the value of 0 is substituted for 0/0, the resulting distance measurement is nearly useless.

Even if all values are not unique, there are often so many different values that there are not enough examples of each one for a reliable statistical sample, which makes the distance measure untrustworthy. There is also a good chance that previosly unseen values will occur during generalization. Because of these problems, it is inappropriate to use the VDM directly on continuous attributes.

Previous systems such as PEBLS (Cost & Salzberg 1993; Rachlin et al. 1994) that have used VDM or modifications of it have typically relied on *discretization* (Lebowitz 1985) to convert continuous attributes into discrete ones, which can degrade accuracy (Wilson & Martinez 1997a).

1.3. Interpolated Value Difference Metric

Since the Euclidean distance function is inappropriate for nominal attributes, and the VDM function is inappropriate for direct use on continuous attributes, neither is appropriate for applications with both linear and nominal attributes. This section presents the Interpolated Value Difference Metric (IVDM) that allows VDM to be applied directly to continuous attributes. IVDM was first presented in (Wilson & Martinez 1997a), and additional details, examples, and alternative distance functions can be found there. An abbreviated description is included here since IVDM is used in the IDIBL system presented in this paper.

The original value difference metric (VDM) uses statistics derived from the training set instances to determine a probability $P_{a,x,c}$ that the output class is c given the input value x for attribute a.

When using IVDM, continuous values are discretized into s equal-width intervals (though the continuous values are also retained for later use). We chose s to be C or 5, whichever is greatest, where C is the number of output classes in the problem domain, though our experiments showed little sensitivity to this value. (Equal-frequency intervals can be used instead of equal-width intervals to avoid problems with outliers, though our experiments have not shown a significant difference between the two methods on the datasets we have used).

The width w_a of a discretized interval for attribute a is given by

$$w_a = \frac{|max_a - min_a|}{s} \qquad [5]$$

where max_a and min_a are the maximum and minimum value, respectively, occurring in the training set for attribute a. The discretized value v of a continuous value x for attribute a is an integer from 1 to s, and is given by

[6]
$$v = discretize_a(x) = \begin{cases} x, \text{ if } a \text{ is discrete, else} \\ s, \text{ if } x \geq \max_a, \text{ else} \\ 1, \text{ if } x \leq \min_a, \text{ else} \\ \lfloor (x - \min_a) / w_a \rfloor + 1 \end{cases}$$

After deciding upon s and finding w_a, the discretized values of continuous attributes can be used just like discrete values of nominal attributes in finding $P_{a,x,c}$. Figure 1 lists pseudo-code for how this is done.

```
FindProbabilities(training set T)
    For each attribute a
        For each instance i in T
            Let x be the input value for attribute a of instance i.
            Let v = discretize_a(x) [which is just x if a is discrete]
            Let c be the output class of instance i.
            Increment N_a,v,c by 1.
            Increment N_a,v by 1.
        For each discrete value v (of attribute a)
            For each class c
                If N_a,v=0
                    Then P_a,v,c=0
                    Else P_a,v,c = N_a,v,c / N_a,v
    Return 3-D array P_a,v,c.
```

Figure 1. Pseudo code for finding $P_{a,x,c}$.

The distance function for the Interpolated Value Difference Metric defines the distance between two input vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ as

[7]
$$IVDM(\mathbf{x},\mathbf{y}) = \sum_{a=1}^{m} ivdm_a(x_a, y_a)^2$$

where the distance between two values x and y for an attribute a is defined by $ivdm_a$ as

[8]
$$ivdm_a(x,y) = \begin{cases} vdm_a(x,y) & \text{if } a \text{ is discrete} \\ \sum_{c=1}^{C} \left| p_{a,c}(x) - p_{a,c}(y) \right|^2 & \text{otherwise} \end{cases}$$

Unknown input values (Quinlan 1989) are treated as simply another discrete value, as was done in (Domingos 1995). The formula for determining the interpolated probability value $p_{a,c}(x)$ of a continuous value x for attribute a and class c is

[9] $$p_{a,c}(x) = P_{a,u,c} + \left(\frac{x - mid_{a,u}}{mid_{a,u+1} - mid_{a,u}} \right) * (P_{a,u+1,c} - P_{a,u,c})$$

In this equation, $mid_{a,u}$ and $mid_{a,u+1}$ are midpoints of two consecutive discretized ranges such that $mid_{a,u} \leq x < mid_{a,u+1}$. $P_{a,u,c}$ is the probability value of the discretized range u, which is taken to be the probability value of the midpoint of range u (and similarly for $P_{a,u+1,c}$). The value of u is found by first setting $u = discretize_a(x)$, and then subtracting 1 from u if $x < mid_{a,u}$. The value of $mid_{a,u}$ can then be found as follows.

[10] $$mid_{a,u} = min_a + width_a * (u + .5)$$

Figure 2 shows the values of $p_{a,c}(x)$ for attribute a=1 (*i.e.*, *Sepal Length*) of the *Iris* database (Fisher 1936) for all three output classes (*i.e.* c=1, 2, and 3). Since there are no data points outside the range $min_a..max_a$, the probability value $P_{a,u,c}$ is taken to be 0 when $u < 1$ or $u > s$, which can be seen visually by the diagonal lines sloping toward zero on the outer edges of the graph. Note that the sum of the probabilities for the three output classes sum to 1.0 at every point from the midpoint of range 1 through the midpoint of range 5.

In experiments reported by the authors (Wilson & Martinez 1997a) on 48 datasets from the UCI machine learning databases (Merz & Murphy 1996), a nearest neighbor classifier using IVDM was able to achieve almost 5% higher generalization accuracy on average over a nearest neighbor classifier using either Euclidean distance or the Euclidean-Overlap metric mentioned in Section 1.1. It also achieved higher accuracy than a classifier that used discretization on continuous attributes in order to use the VDM distance function.

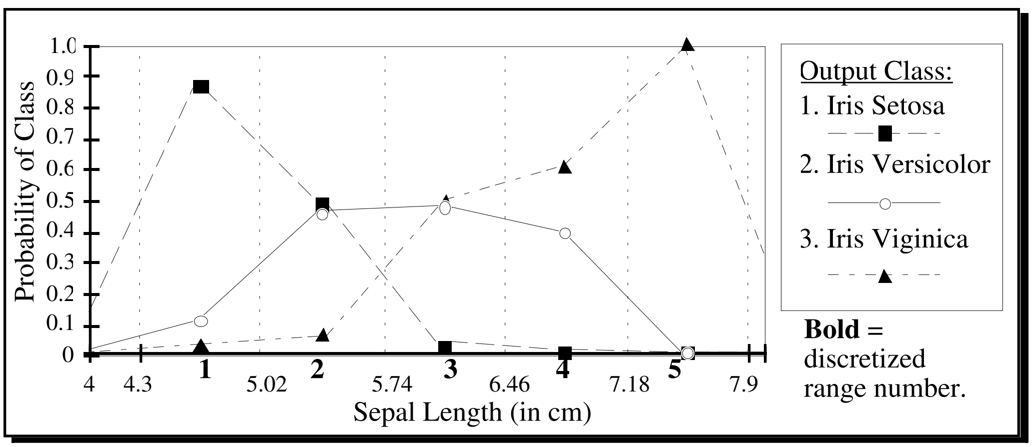

Figure 2. Interpolated probability values for attribute 1 of the *Iris* database.

2. INSTANCE PRUNING TECHNIQUES

One of the main disadvantages of the basic nearest neighbor rule is that it has large storage requirements because it stores all n instances in the training set T in memory. It also has slow execution speed because it must find the distance between a new input vector and each of the n instances in order to find the nearest neighbor(s) of the new input vector, which is necessary for classification. In addition, since it stores every instance in the training set, noisy instances (*i.e.*, those with errors in the input vector or output class, or those not representative of typical cases) are stored as well, which can degrade generalization accuracy.

2.1. Speeding Classification

It is possible to use *k*-dimensional trees ("*k-d* trees") (Wess, Althoff & Derwand 1994; Sproull 1991; Deng & Moore 1995) to find the nearest neighbor in O(log*n*) time in the best case. However, as the dimensionality grows, the search time can degrade to that of the basic nearest neighbor rule (Sproull 1991).

Another technique used to speed the search is *projection* (Papadimitriou & Bentley 1980), where instances are sorted once by each dimension, and new instances are classified by searching outward along each dimension until it can be sure the nearest neighbor has been found. Again, an increase in dimensionality reduces the effectiveness of the search.

Even when these techniques are successful in reducing execution time, they do not reduce storage requirements. Also, they do nothing to reduce the sensitivity of the classifier to noise.

2.2. Reduction Techniques

One of the most straightforward ways to speed classification in a nearest-neighbor system is by removing some of the instances from the instance set. This also addresses another of the main disadvantages of nearest-neighbor classifiers—their large storage requirements. In addition, it is sometimes possible to remove noisy instances from the instance set and actually improve generalization accuracy.

A large number of such reduction techniques have been proposed, including the *Condensed Nearest Neighbor Rule* (Hart 1968), *Selective Nearest Neighbor Rule* (Ritter et. al. 1975), the *Reduced Nearest Neighbor Rule* (Gates 1972), the *Edited Nearest Neighbor* (Wilson 1972), the *All k-NN* method (Tomek 1976), *IB2* and *IB3* (Aha, Kibler & Albert 1991; Aha 1992), the *Typical Instance Based Learning* Zhang (1992), *random mutation hill climbing* (Skalak 1994; Papadimitriou & Steiglitz 1982), and instance selection by *encoding length heuristic* (Cameron-Jones 1995). Other techniques exist that modify the original instances and use some other representation to store exemplars, such as prototypes (Chang 1974); rules, as in RISE 2.0 (Domingos 1995); hyperrectangles, as in EACH (Salzberg 1991); and hybrid models (Dasarathy 1979; Wettschereck 1994).

These and other reduction techniques are surveyed in depth in (Wilson & Martinez 1999), along with several new reduction techniques called *DROP1-DROP5*. Of these, DROP3 and DROP4 had the best performance, and DROP4 is slightly more careful in how it filters noise, so it was selected for use by IDIBL.

2.3. DROP4 Reduction Algorithm

This section presents an instance pruning algorithm called the *Decremental Reduction Optimization Procedure 4* (DROP4) (Wilson & Martinez 1999) that is used by IDIBL to reduce the number of instances that must be stored in the final system and to correspondingly improve classification speed. DROP4 also makes IDIBL more robust in the presence of noise. This procedure is *decremental*, meaning that it begins with the entire training set, and then removes instances that are deemed unneccesary. This is different than the *incremental* approaches that begin with an empty subset *S* and add instances to it, as is done by IB3 (Aha, Kibler & Albert 1991) and several other instance-based algorithms.

In order to avoid repeating lengthy definitions, some notation is introduced here. A training set *T* consists of *n* instances $i = 1..n$. Each instance *i* has *k* nearest neighbors denoted as $i.n_{1..k}$ (ordered from nearest to furthest). Each instance *i* also has a nearest *enemy* which is the nearest instance e to i with a different output class. Those instances that have i as one of their k nearest neighbors are called *associates* of i, and are notated as $i.a_{1..m}$ (sorted from nearest to furthest) where m is the number of associates that i has.

DROP4 uses the following basic rule to decide if it is safe to remove an instance i from the instance set S (where $S = T$ originally).

Remove instance i from S if at least as many of its associates in T would be classified correctly without i.

To see if an instance i can be removed using this rule, each *associate* (*i.e.*, each instance that has i as one of its neighbors) is checked to see what effect the removal of i would have on it.

Removing i causes each associate $i.a_j$ to use its k+1st nearest neighbor ($i.a_j.n_{k+1}$) in S in place of i. If i has the same class as $i.a_j$, and $i.a_j.n_{k+1}$ has a different class than $i.a_j$, this weakens its classification and could cause $i.a_j$ to be misclassified by its neighbors. On the other hand, if i is a different class than $i.a_j$ and $i.a_j.n_{k+1}$ is the same class as $i.a_j$, the removal of i could cause a previously misclassified instance to be classified correctly.

In essence, this rule tests to see if removing i would degrade leave-one-out cross-validation accuracy, which is an estimate of the true generalization ability of the resulting classifier. An instance is removed when it results in the same level of estimated generalization with lower storage requirements. By maintaining lists of k+1 neighbors and an average of k+1 associates (and their distances), the leave-one-out cross-validation accuracy can be computed in O(k) time for each instance instead of the usual O(mn) time, where n is the number of instances in the training set and m is the number of input attributes. An O(mn) step is only required once an instance is selected for removal.

The DROP4 algorithm assumes that a list of nearest neighbors for each instance has already been found (as explained below in Section 4), and begins by making sure each neighbor has a list of its associates. Then each instance in S is removed if its removal does not hurt the classification of the instances in T. When an instance i is removed, all of its associates must remove i from their list of nearest neighbors and then must find a new nearest neighbor $a.n_j$ so that they still have k+1 neighbors in their list. When they find a new neighbor $a.n_j$, they also add themselves to $a.n_j$'s list of associates so that at all times every instance has a current list of neighbors and associates.

Each instance i in the original training set T continues to maintain a list of its $k + 1$ nearest neighbors in S, even after i is removed from S. This in turn means that instances in S have associates that are both in and out of S, while instances that have been removed from S have no associates (because they are no longer a neighbor of any instance).

This algorithm removes noisy instances, because a noisy instance i usually has associates that are mostly of a different class, and such associates will be at least as likely to be classified correctly without i. DROP4 also removes an instance i in the center of a cluster because associates there are not near instances of other classes, and thus continue to be classified correctly without i.

Near the border, the removal of some instances can cause others to be classified incorrectly because the majority of their neighbors can become members of other classes. Thus this algorithm tends to keep non-noisy border points. At the limit, there is typically a collection of border instances such that the majority of the k nearest neighbors of each of these instances is the correct class.

The order of removal can be important to the success of a reduction algorithm. DROP4 initially sorts the instances in *S* by the distance to their nearest *enemy*, which is the nearest neighbor of a different class. Instances are then checked for removal beginning at the instance furthest from its nearest enemy. This tends to remove instances furthest from the decision boundary first, which in turn increases the chance of retaining border points.

However, noisy instances are also "border" points, so they can cause the order of removal to be drastically changed. In addition, it is often desirable to remove the noisy instances before any of the others so that the rest of the algorithm is not influenced as heavily by the noise.

DROP4 therefore uses a noise-filtering pass *before* sorting the instances in *S*. This is done using a rule similar to the *Edited Nearest Neighbor* rule (Wilson 1972), which states that any instance misclassified by its *k* nearest neighbors is removed. In DROP4, however, the noise-filtering pass removes each instance only if it is (1) misclassified by its *k* nearest neighbors, *and* (2) it does not hurt the classification of its associates. This noise-filtering pass removes noisy instances, as well as close border points, which can in turn smooth the decision boundary slightly. This helps to avoid "overfitting" the data, *i.e.*, using a decision surface that goes beyond modeling the underlying function and starts to model the data sampling distribution as well.

After removing noisy instances from *S* in this manner, the instances are sorted by distance to their nearest enemy remaining in *S*, and thus points far from the real decision boundary are removed first. This allows points internal to clusters to be removed early in the process, even if there were noisy points nearby.

The pseudo-code in Figure 3 summarizes the operation of the DROP4 pruning algorithm. The procedure *RemoveIfNotHelping(i, S)* applies the *basic rule* introduced at the beginning of this section is satisfied, *i.e.*, it removes instance *i* from *S* if the removal of instance *i* does not hurt the classification of instances in *T*.

```
DROP4(Training set T): Instance set S.
    Let S = T.
    // Initialize the lists of neighbors and associates
    For each instance i in S:
        Make sure we know i.n_1...i.n_k+1, the k+1 nearest neighbors of i in S.
        Add i to each of its neighbors' lists of associates.
    // Do careful noise-filtering pass
    For each instance i in S:
        If i is misclassified by its neighbors
            RemoveIfNotHelping(i, S)
    // Do more aggressive reduction pass
    Sort instances by distance to nearest enemy (furthest ones first).
    For each instance i in S (starting with those furthest from their nearest enemy):
        RemoveIfNotHelping(i, S)
    Return S.

RemoveIfNotHelping(Instance i, Instance set S)
    Let correctWith = # of associates of i in T classified correctly with i as a neighbor.
    Let correctWithout = # of associates of i classified correctly without i.
    If (correctWithout ≥ correctWith)
        Remove i from S.
        For each associate a of i
            Remove i from a's list of nearest neighbors
            Find a new neighbor (i.e., k + 1) for a in S.
            Add a to its new neighbor's list of associates.
    Endif
```

Figure 3: Pseudo-code for DROP4.

In experiments reported by the authors (Wilson & Martinez 1999) on 31 datasets from the UCI machine learning database repository, DROP4 was compared to a *k*NN classifier that used 100% of the training instances for classification. DROP4 was able to achieve an average generalization accuracy that was just 1% below the *k*NN classifier while retaining only 16% of the original instances. Furthermore, when 10% noise was added to the output class in the training set, DROP4 was able achieve higher accuracy than *k*NN while using even less storage than before. DROP4 also compared favorably with the other instance pruning algorithms mentioned in Section 2.2 (Wilson & Martinez 1999).

3. DISTANCE-WEIGHTING AND CONFIDENCE LEVELS

One disadvantage of the basic nearest neighbor classifier is that it does not make adjustments to its decision surface after storing the data. This allows it to learn quickly, but prevents it from generalizing accurately in some cases.

Several researchers have proposed extensions that add more flexibility to instance-based systems. One of the first extensions (Cover & Hart 1967) was the introduction of the parameter *k*, the number of neighbors that vote on the output of an input vector. A variety of other extensions have also been proposed, including various attribute-weighting schemes (Wettschereck, Aha, and Mohri 1995; Aha 1992; Aha & Goldstone 1992; Mohri & Tanaka 1994; Lowe 1995; Wilson & Martinez 1996), exemplar weights (Cost & Salzberg 1993;

Rachlin et al. 1994; Salzberg 1991; Wettschereck & Dietterich 1995), and distance-weighed voting (Dudani 1976; Keller, Gray & Givens 1985).

The value of k and other parameters are often found using *cross-validation* (CV) (Schaffer 1993; Moore & Lee 1993; Kohavi 1995). In *leave-one-out* cross-validation (LCV), each instance i is classified by all of the instances in the training set T other than i itself, so that almost all of the data is available for each classification attempt.

One problem with using CV or LCV to fine-tune a learning system (*e.g.*, when deciding whether to use $k = 1$ or $k = 3$, or when deciding what weight to give to an input attribute) is that it can yield only a fixed number of discrete accuracy estimates. Given n instances in a training set, CV will yield an accuracy estimation of 0 or 1 for each instance, yielding an average estimate that is in the range 0..1, but only in increments of $1/n$. This is equivalent in its usefulness to receiving an integer r in the range 0..n indicating *how many* of the instances are classified correctly using the current parameter settings.

Usually a change in any given parameter will not change r by more than a small value, so the change in r given a change in parameter is usually a small integer such as -3...3. Unfortunately, changes in parameters often have *no* effect on r, in which case CV does not provide helpful information as to which alternative to choose. This problem occurs quite frequently in some systems and limits the extent to which CV can be used to fine-tune an instance-based learning model.

This section proposes a combined Cross-Validation and Confidence measure (CVC) for use in tuning parameters. Section 3.1 describes a distance-weighted scheme that makes the use of confidence possible, and Section 3.2 shows how cross-validation and confidence can be combined to improve the evaluation of parameter settings.

3.1. Distance-Weighted Voting

Let $\boldsymbol{y}$ be the input vector to be classified and let $\boldsymbol{x}_1...\boldsymbol{x}_k$ be the input vectors for the k nearest neighbors of $\boldsymbol{y}$ in a subset S (found via the pruning technique DROP4) of the training set T. Let D_j be the distance from the jth neighbor using the IVDM distance function, i.e., $D_j = \text{IVDM}(\boldsymbol{x}_j, \boldsymbol{y})$.

In the IDIBL algorithm, the voting weight of each of the k nearest neighbors depends on its distance from the input vector $\boldsymbol{y}$. The weight is 1 when the distance is 0 and decreases as the distance grows larger. The way in which the weight decreases as the distance grows depends on which *kernel function* is used. The kernels used in IDIBL are: *majority, linear, gaussian,* and *exponential*.

In majority voting, all k neighbors get an equal vote of 1. With the other three kernels, the weight is 1 when the distance is 0 and drops to the value of a parameter w_k when the distance is D_k, which is the distance to the kth nearest neighbor.

The amount of voting weight w_j for the jth neighbor is

$$[11] \qquad w_j(D_j, D_k, w_k, kernel) = \begin{cases} 1 & \text{if } kernel = \text{majority} \\ w_k + \dfrac{(1-w_k)(D_k - D_j)}{D_k} & \text{if } kernel = \text{linear} \\ w_k^{D_j^2/D_k^2} & \text{if } kernel = \text{gaussian} \\ w_k^{D_j/D_k} & \text{if } kernel = \text{exponential} \end{cases}$$

where w_k is the parameter that determines how much weight the kth neighbor receives; D_j is the distance of the jth nearest neighbor; D_k is the distance to the kth neighbor; and *kernel* is the parameter that determines which vote-weighting function to use.

Note that the majority voting scheme does not require the w_k parameter. Also note that if $k = 1$ or $w_k = 1$, then all four kernels are equivalent. If $k = 1$, then the weight is irrelevant, because only the first nearest neighbor gets any votes. If $wk = 1$, on the other hand, the weight for all four kernels is 1, just as it is in majority voting.

As D_k approaches 0, the weight in Equation (11) approaches 1, regardless of the kernel. Therefore, if the distance D_k is equal to 0, then a weight of 1 is used for the sake of consistency and to avoid dividing by 0. These kernels are illustrated in Figure 4.

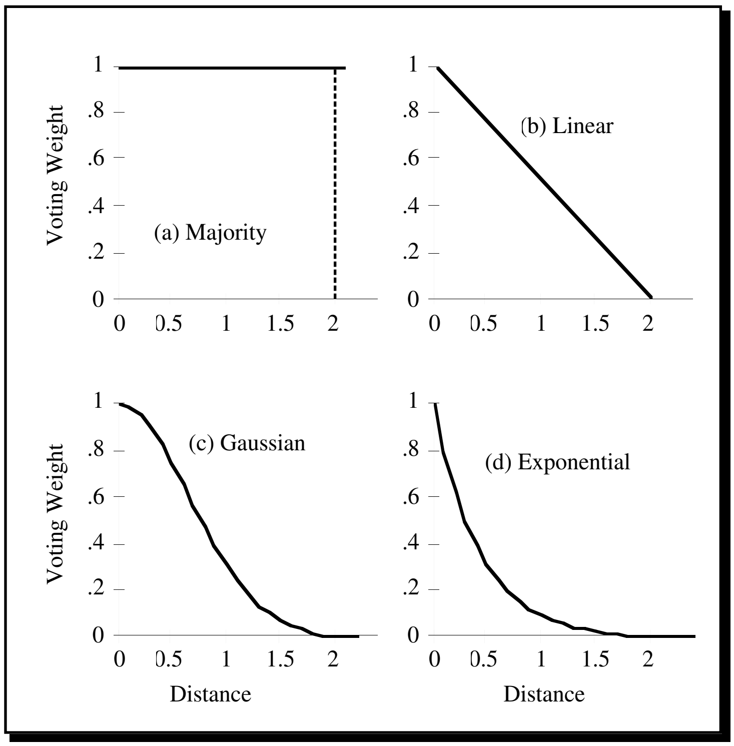

Figure 4. Distance-weighting kernels, shown with $D_k = 2.0$ and $w_k = .01$.

Sometimes it is preferable to use the *average* distance of the k nearest neighbors instead of the distance to the kth neighbor to determine how fast voting weight should drop off. This can be done by computing what the distance D'_k of the kth nearest neighbor would be if the neighbors were distributed evenly. This is accomplished by setting D'_k to

[12] $$D'_k = \frac{2 \cdot \sum_{i=1}^{k} D_i}{k+1}$$

and using D'_k in place of D_k in Equation 11. When $k = 1$, Equation 12 yields $D'_k = 2D_k / 2 = D_k$, as desired. When $k > 1$, this method can be more robust in the presence of changes in the system such as changing parameters or the removal of instances from the classifier. A boolean flag called *avgk* will be used to determine whether to use D'_k instead of D_k, and will be tuned along with the other parameters as described in Section 3.2.

One reason for using distance-weighted voting is to avoid *ties.* For example, if $k = 4$, and the four nearest neighbors of an input vector happen to include two instances from one class and two from another, then the two classes are tied in the number of their votes and the system must either choose one of the classes arbitrarily (resulting in a high probability of error) or use some additional heuristic to break the tie (such as choosing the class that has a nearer neighbor than the other).

The *possibility* of a t-way tie vote depends on k and c, where t is the number of classes tied and c is the number of output classes in the classification task. For example, when $k = 7$ and $c = 4$, it is possible to get a 3-way tie (*i.e.*, $t = 3$) in which three classes each get 2 votes and the remaining class gets 1, but when $k = 4$ and $c = 4$, a 3-way tie is not possible.

The *frequency* of a t-way tie given a particular value of k depends largely on the distribution of data for a classification task. In a sampling of several datasets from the UCI Machine Learning Database repository we found that 2-way ties happened between 2.5% and 36% of the time when $k = 2$, depending on the task, and 3-way ties happened between 0% and 7% of the time when $k = 3$. Ties tend to become less common as k grows larger.

Two-way ties are common because they can occur almost anywhere along a border region between the data for different classes. Three-way ties, however, occur only where two decision boundaries intersect. Four-way and greater ties are even more rare because they occur where three or more decision boundaries meet.

Table 1 shows a typical distribution of tie votes for a basic kNN classifier with values of $k = 1..10$ shown in the respective columns and different values of t shown in the rows. This data was gathered over five datasets from the UCI repository that each had over seven output classes (*Flags*, *LED-Creator*, *LED+17*, *Letter-Recognition*, and *Vowel*), and the average over the five datasets of the percentage of ties is shown for each combination of t and k. Impossible entries are left blank.

Table 1 shows that on this collection of datasets, about 27% of the test instances had 2-way ties when $k = 2$, 6.88% had 3-way ties when $k = 3$, and so on. Interestingly, 6.88% is about 25% of the 27.14% reported for k=t=2. In fact, about one-fourth of the instances are involved in a 2-way tie when $k = 2$, about one-fourth of *those* are also involved in a 3-way tie when $k = 3$, about one-fourth of those are involved in a 4-way tie when $k = 4$, and so on, until the pattern breaks down at $k = 6$ (probably because the data becomes too sparse).

	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	k=9	k=10
1 winner:	100.00	72.86	93.12	90.64	95.99	93.24	96.73	91.32	96.61	95.42
2-way tie:		27.14	0.00	7.60	3.65	6.19	2.64	8.37	2.94	4.50
3-way tie:			6.88	0.00	0.00	0.55	0.63	0.21	0.34	0.04
4-way tie:				1.76	0.00	0.00	0.00	0.09	0.10	0.03
5-way tie:					0.36	0.00	0.00	0.00	0.00	0.00
6-way tie:						0.02	0.00	0.00	0.00	0.00
7-way tie:							0.00	0.00	0.00	0.00

Table 1. Percentage of tie votes with different values of k.

Using distance-weighted voting helps to avoid tie votes between classes when classifying instances. However, a more important reason for using distance-weighted voting in IDIBL is to allow the use of *confidence*. This allows the system to avoid a different kind of ties—ties between fitness scores for alternative parameter settings as explained in Section 3.2.

3.2. Cross-Validation and Confidence (CVC)

Given the distance-weighted voting scheme described in Section 3.1, IDIBL must set the following parameters:

- *k*, the number of neighbors that vote on the class of a new input vector.
- *kernel*, the shape of the distance-weighted voting function.
- *avgk*, the flag determining whether to use the average distance to the *k* nearest neighbors rather than the *k*th distance.
- w_k, the weight of the *k*th neighbor (except in majority voting).

When faced with a choice between two or more sets of parameter values, some method is needed for deciding which is most likely to yield the best generalization. This section describes how these parameters are set automatically, using a combination of leave-one-out cross-validation and confidence levels.

3.2.1. Cross-Validation

With leave-one-out cross-validation (LCV), the generalization accuracy of a model is estimated from the average accuracy attained when classifying each instance *i* using all the instances in *T* except *i* itself. For each instance, the accuracy is 1 if the instance is classified correctly, and 0 if it is misclassified. Thus the average LCV accuracy is r / n, where *r* is the number classified correctly and *n* is the number of instances in *T*. Since *r* is an integer from 0 to *n*, there are only $n + 1$ accuracy values possible with this measure, and often two different sets of parameter values will yield the same accuracy because they will classify the same number of instances correctly. This makes it difficult to tell which parameter values to use.

3.2.2. Confidence

An alternative method for estimating generalization accuracy is to use the *confidence* with which each instance is classified. The average confidence over all *n* instances in the training set can then be used to estimate which set of parameter values will yield better generalization. The confidence for each instance is

$$conf = \frac{votes_{correct}}{\sum_{c=1}^{C} votes_c} \quad [13]$$

where $votes_c$ is the sum of weighted votes received for class *c* and $votes_{correct}$ is the sum of weighted votes received for the correct class.

When majority voting is used, $votes_c$ is simply a count of how many of the *k* nearest neighbors were of class *c*, since the weights are all equal to 1. In this case, the confidence will be an integer in the range 0..*k*, divided by *k*, and thus there will be only $k + 1$ possible confidence values for each instance. This means that there will be $(k + 1)(n + 1)$ possible accuracy estimates using confidence instead of $n + 1$ as with LCV, but it is still possible for small changes in parameters to yield no difference in average confidence.

When distance-weighted voting is used, however, each vote is weighted according to its distance and the current set of parameter values, and $votes_c$ is the sum of the weighted votes for each class. Even a small change in the parameters will affect how much weight each neighbor gets and thus will affect the average confidence.

After learning is complete, the confidence can be used to indicate how confident the classifier is in its generalized output. In this case the confidence is the same as defined in Equation 13, except that $votes_{correct}$ is replaced with $votes_{out}$, which is the amount of voting weight received by the class that is chosen to be the output class by the classifier. This is also equal to the maximum number of votes (or maximum sum of voting weights) received by any class, since the majority class is chosen as the output.

Average confidence has the attractive feature that it provides a continuously valued metric for evaluating a set of parameter values. However, it also has drawbacks that make it inappropriate for direct use on the parameters in IDIBL. Average confidence is increased whenever the ratio of votes for the correct class to total votes is increased. Thus, this metric strongly favors $k = 1$ and $w_k = 0$, regardless of their effect on classification, since these settings give the nearest neighbor more relative weight, and the nearest neighbor is of the same class more often than other neighbors. This metric also tends to favor exponential weighting since it drops voting weight more quickly than the other kernels.

Therefore, using confidence as the sole means of deciding between parameter settings will favor any settings that weight nearer neighbors more heavily, even if accuracy is degraded by doing so.

Schapire, Freund, Bartlett and Lee (1997) define a measure of confidence called a *margin*. They define the classification margin for an instance as "the difference between the weight assigned to the correct label and the maximal weight assigned to any single incorrect label," where the "weight" is simply the confidence value *conf* in Equation 13. They show that improving the margin on the training set improved the upper bound on the generalization error. They also show that one reason that *boosting* (Freund and Shapire 1995) is effective is that it increases this classification margin.

Breiman (1996) pointed out that both boosting and *bootstrap aggregation* (or *bagging*) depend on the base algorithm being *unstable*, meaning that small changes to the training set can cause large changes in the learned classifier. Nearest neighbor classifiers have been shown to be quite stable and thus resistent to the improvements in accuracy often achieved by the use of boosting or bagging, though that does not necessarily preclude the use of the margin as a confidence measure.

However, when applied to this specific problem of fine-tuning parameters in a distance-weighted instance-based learning algorithm, using the margin to measure the fitness of parameter settings would suffer from the same problem as the confidence discussed above, *i.e.*, it would favor the choice of $k = 1$, $w_k = 0$, and exponential voting in most cases, even if this harmed LCV accuracy. Using the margin instead of the confidence measure defined in Equation 13 turned out to make no empirical difference on the datasets used in our experiments with only a few exceptions, and only insignificant differences in those remaining cases. We thus use the more straightforward definition of confidence from Equation 13 in the remainder of this paper.

3.2.3. CROSS-VALIDATION AND CONFIDENCE (CVC)

In order to avoid the problem of always favoring $k = 1$, $w_k = 0$, and *kernel* = *exponential*, IDIBL combines *Cross-Validation and Confidence* into a single metric called *CVC*. Using CVC, the accuracy estimate cvc_i of a single instance i is

$$cvc_i = \frac{n \cdot cv + conf}{n+1} \tag{14}$$

where n is the number of instances in the training set T; *conf* is as defined in Equation 13; and *cv* is 1 if instance i is classified correctly by its neighbors in S, or 0 otherwise.

This metric weights the cross-validation accuracy more heavily than the confidence by a factor of n. This technique is equivalent to using *numCorrect* + *avgConf* to make decisions, where *numCorrect* is the number of instances in T correctly classified by their neighbors in S and is an integer in the range 0..n, and *avgConf* is the average confidence (from Equation 13) for all instances and is a real value in the range 0..1. Thus, the LCV portion of CVC can be thought of as providing the whole part of the score, with confidence providing the fractional part. Dividing *numCorrect* + *avgConf* by $n + 1$ results in a score in the range 0..1, as would also be obtained by averaging Equation 14 for all instances in T.

This metric gives LCV the ability to make decisions by itself unless multiple parameter settings are tied, in which case the confidence makes the decision. There will still be a bias towards giving the nearest neighbor more weight, but only when LCV cannot determine which parameter settings yield better leave-one-out accuracy.

3.3. Parameter Tuning

This section describes the learning algorithm used by IDIBL to find the parameters k, w_k, *avgk*, and *kernel*, as described in Sections 3.1 and 3.2. The parameter-tuning algorithm assumes that for each instance i in T, the nearest *maxk* neighbors in S have been found. Parameter tuning takes place both before and after pruning is done. S is equal to T prior to pruning, and S is a subset of T after pruning has taken place.

The neighbors of each instance i, notated as $i.n_1 ... i.n_{maxk}$, are stored in a list ordered from nearest to furthest for each instance, so that $i.n_1$ is the nearest neighbor of i and $i.n_k$ is the kth nearest neighbor. The distance $i.D_j$ to each of instance i's neighbor $i.n_j$ is also stored to avoid continuously recomputing this distance.

In our experiments, *maxk* was set to 30 before pruning to find an initial value of k. After pruning, *maxk* was set to this initial value of k since increasing the size of k does not make much sense after instances have been removed. In addition, the pruning process leaves the list of k (but not *maxk*) neighbors intact, so this strategy avoids the lengthy search to find every instance's nearest neighbors again. In our experiments IDIBL rarely if ever chose a value of k greater than 10, but we used *maxk* = 30 to leave a wide margin of error since not much time was required to test each value of k.

CVC is used by IDIBL to automatically find good values for the parameters k, w_k, *kernel*, and *avgk*. Note that none of these parameters affect the distance between neighbors but only the amount of voting weight each neighbor gets. Thus, changes in these parameters can be made without requiring a new search for nearest neighbors or even an update to the stored distance to each neighbor. This allows a set of parameter values to be evaluated in O(kn) time instead of the O(mn^2) time required by a naive application of leave-one-out cross-validation. This efficient method is similar to the method used in RISE (Domingos 1995).

To evaluate a set of parameter values, cvc_i as defined in Equation 13 is computed as follows. For each instance i, the voting weight for each of its k nearest neighbors $i.n_j$ is found according to $w_j(i.D_j, D_k, w_k, kernel)$ defined in Equation 11, where d_k is $i.D_k$ if *avgk* is false, or D'_k as defined in Equation 12 if *avgk* is true. These weights are summed in their separate respective classes, and the confidence of the correct class is found as in Equation 13. If the majority class is the same as the true output class of instance i, cv in Equation 14 is 1. Otherwise, it is 0. The average value of cvc_i over all n instances is used to determine the fitness of the parameter values.

The search for parameter values proceeds in a greedy manner as follows. For each iteration, one of the four parameters is chosen for adjustment, with the restriction that no parameter can be chosen twice in a row, since doing so would simply rediscover the same parameter value. The chosen parameter is set to various values as explained below while the remaining parameters are held constant. For each setting of the chosen parameter, the CVC fitness for the system is calculated, and the value that achieves the highest fitness is chosen as the new value for the parameter.

At that point, another iteration begins, in which a different parameter is chosen at random and the process is repeated until several attempts at tuning parameters does not improve the best CVC fitness found so far. In practice, only a few iterations are required to find good settings, after which improvements cease and the search soon terminates. The set of parameters that yield the best CVC fitness found at any point during the search are used by IDIBL for classification. The four parameters are tuned as follows.

1. Choosing *k*. To pick a value of k, all values from 2 to *maxk* (=30 in our experiments) are tried, and the one that results in maximum CVC fitness is chosen. Using the value $k = 1$ would make all of the other parameters irrelevant, thus preventing the system from tuning them, so only values 2 through 30 are used until all iterations are complete.

2. Choosing a kernel function. Picking a vote-weighting kernel function proceeds in a similar manner. The kernels *linear*, *gaussian*, and *exponential* are tried, and the kernel that yields the highest CVC fitness is chosen. Using *majority* voting would make the parameters *wk* and *avgk* irrelevant, so this setting is not used until all iterations are complete. At that point, majority voting is tried with values of k from 1 to 30 to test both $k = 1$ (for which the kernel function is irrelevant to the voting scheme) and majority voting in general, to see if either can improve upon the tuned set of parameters.

3. Setting *avgk*. Selecting a value for the flag *avgk* consists of simply trying both settings, *i.e.*, using D_k and D'_k and seeing which yields higher CVC fitness.

4. Searching for w_k. Finding a value for w_k is more complicated because it is a real-valued parameter. The search for a good value of w_k begins by dividing the range 0..1 into ten subdivisions and trying all eleven endpoints of these divisions. For example, for the first pass, the values 0, .1, .2, ..., .9, and 1.0 are used. The value that yields the highest CVC fitness is chosen, and the range is narrowed to cover just one division on either side of the chosen value, with the constraint that the range cannot go outside of the range 0..1. For example, if .3 is chosen in the first round, then the new range is from .2 to .4, and this new range would be dividing into 10 subranges with endpoints .20, .21, ..., .40. The process is repeated three times, at which point the effect on classification becomes negligible.

IDIBL tunes each parameter separately in a random order until several attempts at tuning parameters does not improve the best CVC fitness found so far. After each pass, if the CVC fitness is the best found so far, the current parameter settings are saved. The parameters that resulted in the best fitness during the entire search are then used during subsequent classification.

Tuning each parameter takes O(kn) time, so the entire process takes O(knt) time, where t is the number of iterations required before the stopping criterion is met. In practice t is small (e.g., less than 20), since tuning each parameter once or twice is usually sufficient. These time requirements are quite acceptable, especially compared to algorithms that require repeated

O(mn^2) steps.

Pseudo-code for the parameter-finding portion of the learning algorithm is shown in Figure 5. This algorithm assumes that the nearest *maxk* neighbors of each instance *T* have been found and returns the parameters that produce the highest CVC fitness of any tried. Once these parameters have been found, the neighbor lists can be discarded, and only the raw instances and best parameters need to be retained for use during subsequent classification.

```
FindParams(maxAttempts, training set T): bestParams
    Assume that the maxk nearest neighbors have been
        found for each instance i in T.
    Let timeSinceImprovement=0.
    Let bestCVC=0.
    Initialize parameters with k=3, kernel=linear, avgk=FALSE (i.e., D_k), and w_k=0.5.
    While timeSinceImprovement < maxAttempts
        Choose a random parameter p to adjust.
        If (p="k") try k=2..30, and set k to the best value found.
        If (p="kernel") try linear, gaussian, and exponential.
        If (p="avgk") try D_k and D'_k.
        If (p="w_k")
            Let min=0 and max=1
            For iteration=1 to 3
                Let width=(min-max)/10.
                Try w_k=min..max in steps of width.
                Let min=best w_k-width (if min<0, let min=0)
                Let max=best w_k+width (if max>1, let max=1)
            Endfor
        If bestCVC was improved during this iteration,
            then let timeSinceImprovement=0,
            and let bestParams=current parameter settings.
    Endwhile.
    Let kernel=majority, and try k=1..30.
    if bestCVC was improved during this search,
        then let bestParams=current parameter settings.
    Return bestParams.
```

Figure 5. Pseudo-code for parameter-finding algorithm.

In Figure 5, to "try" a parameter value means to set the parameter to that value, find the CVC fitness of the system, and, if the fitness is better than any seen so far, set *bestCVC* to this fitness and remember the current set of parameter values in *bestParams*.

Note that IDIBL still has parameters that are not tuned, such as *maxk*, the maximum value of k that is allowed (30); the number of subdivisions to use in searching for w_k (*i.e.*, 10); and *maxAttempts*, the number of iterations without improvement before terminating the algorithm. While somewhat arbitrary values were used for these paramters, these parameters are *robust*, *i.e.*, they are not nearly as critical to IDIBL's accuracy as are the sensitive parameters that are automatically tuned. For example, the value of k is a sensitive parameter and has a large effect on generalization accuracy, but the best value of k is almost always quite small (e.g., less than 10), so the value used as the maximum is a robust parameter, as long as it is not so small that it overly limits the search space.

4. IDIBL LEARNING ALGORITHM

Sections 1-3 present several independent extensions that can be applied to instance-based learning algorithms. This section shows how these pieces fit together in the *Integrated Decremental Instance-Based Learning* (IDIBL) algorithm. The learning algorithm proceeds according to the following five steps.

Step 1. Find IVDM Probabilities. IDIBL begins by calling *FindProbabilities* as described in Section 1.3 and outlined in Figure 1. This builds the probability values needed by the IVDM distance function. This distance function is used in all subsequent steps. This step takes $O(m(n+v))$ time, where n is the number of instances in the training set, m is the number of input attributes in the domain, and v is the average number of values in each input attribute for the task.

Step 2. Find Neighbors. IDIBL then finds the nearest *maxk* (=30 in our implementation) neighbors of each instance i in the training set T. These neighbors are stored in a list as described in Section 3, along with their distances to i, such that the nearest neighbor is at the head of the list. This step takes $O(mn^2)$ time and is typically the most computationally intensive part of the algorithm.

Step 3. Tune Parameters. Once the neighbors are found, IDIBL initializes the parameters with some default values ($w_k = 0.2$, *kernel* = *linear*, *avgk* = *true*, $k = 3$) and uses the function *FindParameters*, with $S = T$, as described in Section 3.3 and outlined in Figure 5. It actually forces the first iteration to tune the parameter k, since that parameter can have such a large effect on the behavior of the others. It continues until four iterations yield no further improvement in CVC fitness, at which point each of the four parameters has had a fairly good chance of being tuned without yielding improvement. This step takes $O(knt)$ time, where t is the number of parameter-tuning iterations performed.

Step 4. Prune the Instance Set. After the initial parameter tuning, DROP4 is called in order to find a subset S of T to use in subsequent classification. DROP4 uses the best parameters found in the previous step in all of its operations. This step takes $O(mn^2)$ time, though in practice it is several times faster than Step 2, since the $O(mn)$ step must only be done when an instance is pruned, rather than for every instance, and only the instances in S must be searched when the $O(mn)$ step is required.

Step 5. Retune Parameters. Finally, IDIBL calls *FindParameters* one more time, except that this time all of the instances in T have neighbors only in the pruned subset S. Also, *maxk* is set to the value of k found in step 4 instead of the original larger value. *FindParameters* continues until eight iterations yield no further improvement in CVC fitness. Step 3 is used to get the parameters in a generally good area so that pruning will work properly, but Step 5 tunes the parameters for more iterations before giving up in order to find the best set of parameters reasonably possible.

```
TrainIDIBL(training set T):S, bestParams, P_{a,v,c}
    P_{a,v,c} = FindProbabilities(T)
    For each instance i in T
        Find nearest maxk (=30) neighbors of instance i.
    bestParams = FindParameters(4,T)
    S = DROP4(T)
    Let maxk = k from bestParams
    bestParams = FindParameters(8,T)
    Return S, bestParams, and P_{a,v,c}
```

Figure 6. Pseudo-code for the IDIBL learning algorithm.

High-level pseudo-code for the IDIBL learning algorithm is given in Figure 6, using pseudo-code for *FindProbabilities* from Figure 1, *DROP4* from Figure 3, and *FindParameters* from Figure 5.

At this point, IDIBL is ready to classify new input vectors it has not seen before. The lists of neighbors maintained by each instance can be disposed of, as can all pruned instances (unless later updates to the model are anticipated). Only the subset S of instances, the four tuned parameters, and the probability values $P_{a,v,c}$ required by IVDM need be retained.

When a new input vector $\boldsymbol{y}$ is presented to IDIBL for classification, IDIBL finds the distance between $\boldsymbol{y}$ and each instance in S using IVDM. The nearest k neighbors (using the best value of k found in Step 5) vote using the other tuned parameters in the distance-weighted voting scheme. For each class c, a sum $vote_c$ of the weighted votes thus computed, and the confidence of each class c is given by dividing the votes for each class by the sum of votes for all the classes, as shown in Equation 15.

[15] $$classConf_c = \frac{votes_c}{\sum_{i=1}^{C} votes_c}$$

The confidence computed in the manner indicates how confident IDIBL is in its classification. The output class that has the highest confidence is used as the output class of $\boldsymbol{y}$.

The learning algorithm is dominated by the $O(mn^2)$ step required to build the list of nearest neighbors for each instance, where n is the number of instances in T and m is the number of input attributes. Classifying an input vector takes only $O(rm)$ time per instance, where r is the number of instances in the reduced set S, compared to the $O(nm)$ time per instance required by the basic nearest neighbor algorithm. This learning step is done just once, while subsequent classification is done indefinitely, so the time complexity of the algorithm is often less than that of the nearest neighbor rule in the long run.

5. EMPIRICAL RESULTS

IDIBL was implemented and tested on 30 datasets from the UCI machine learning database repository (Merz & Murphy 1996). Each test consisted of ten trials, each using one of ten partitions of the data randomly selected from the data sets, *i.e.*, 10-fold cross-validation. For each trial, 90% of the available training instances were used for the training set T, and the remaining 10% of the instances were classified using only the instances remaining in the training set *T* or, when pruning was used, in the subset *S*.

5.1. Comparison of IDIBL with *k*NN (and Intermediate Algorithms)

In order to see how the extensions in IDIBL affect generalization accuracy, results for several instance-based learning algorithms are given for comparison. The average generalization accuracy over the ten trials is reported for each test in Table 2. The *k*NN algorithm is a basic *k*-nearest neighbor algorithm that uses $k = 3$ and majority voting. The *k*NN algorithm uses normalized Euclidean distance for linear attributes and the overlap metric for nominal attributes.

Table 2. Generalization accuracy of a basic *k*NN classifier, one enhanced with IVDM, IVDM enhanced with DROP4, IDIBL using LCV for fitness, and the full IDIBL system using CVC.

Dataset	kNN	IVDM	DROP4	(size)	LCV	(size)	IDIBL	(size)
Annealing	94.61	96.11	94.49	9.30	96.12	7.53	**95.96**	7.67
Australian	81.16	80.58	**85.37**	7.59	85.36	9.50	85.36	11.60
Breast Cancer (WI)	95.28	95.57	96.28	3.85	96.71	4.80	**97.00**	5.63
Bridges	53.73	60.55	59.55	24.00	**64.27**	28.93	63.18	34.89
Credit Screening	81.01	80.14	**85.94**	6.96	85.22	8.42	85.35	11.29
Flag	48.84	57.66	**61.29**	21.82	**61.29**	26.70	57.66	32.07
Glass	70.52	70.54	69.59	25.49	68.66	28.40	**70.56**	38.68
Heart Disease	75.56	81.85	**83.71**	15.43	82.59	20.33	83.34	24.28
Heart (Cleveland)	74.96	78.90	80.81	15.44	81.48	18.23	**83.83**	29.37
Heart (Hungarian)	74.47	80.98	80.90	12.96	**83.99**	13.19	83.29	18.06
Heart (Long-Beach-VA)	71.00	66.00	72.00	7.11	**76.00**	13.61	74.50	14.78
Heart (More)	71.90	73.33	76.57	14.56	78.26	17.34	**78.39**	20.46
Heart (Swiss)	91.86	87.88	**93.46**	2.44	**93.46**	4.33	**93.46**	5.78
Hepatitis	77.50	82.58	79.29	12.83	81.21	13.91	**81.88**	18.43
Horse-Colic	60.82	**76.78**	76.46	19.79	76.12	21.15	73.80	26.73
Image Segmentation	93.57	92.86	93.81	10.87	**94.29**	12.33	**94.29**	15.50
Ionosphere	86.33	**91.17**	90.88	6.52	88.31	7.85	87.76	21.18
Iris	95.33	94.67	95.33	8.30	94.67	8.89	**96.00**	10.15
LED+17 noise	42.90	60.70	70.50	15.20	70.90	23.52	**73.60**	40.09
LED	57.20	56.40	72.10	12.83	72.80	17.26	**74.88**	43.89
Liver Disorders	**63.47**	58.23	63.27	27.44	60.58	37.13	62.93	43.92
Pima Indians Diabetes	70.31	69.28	72.40	18.29	75.26	22.00	**75.79**	29.28
Promoters	82.09	**92.36**	85.91	18.34	87.73	18.56	88.64	21.80
Sonar	**86.60**	84.17	81.64	23.56	78.81	23.50	84.12	50.10
Soybean (Large)	89.20	**92.18**	86.29	27.32	85.35	22.00	87.60	31.03
Vehicle	70.22	69.27	68.57	25.86	68.22	31.56	**72.62**	37.31
Voting	93.12	95.17	**96.08**	6.05	95.62	5.98	95.62	11.34
Vowel	**98.86**	97.53	86.92	41.65	90.53	30.70	90.53	33.57
Wine	95.46	**97.78**	95.46	9.92	94.90	9.30	93.76	8.74
Zoo	94.44	**98.89**	92.22	21.61	92.22	19.26	92.22	22.22
Average:	**78.08**	**80.67**	**81.57**	15.78	**82.03**	17.54	**82.60**	23.99
Wilcoxon:	**99.50**	**96.51**	**99.42**	*n/a*	**95.18**	*n/a*	*n/a*	*n/a*

The *IVDM* column gives results for a *k*NN classifier that uses IVDM as the distance function instead of the Euclidean/overlap metric. The *DROP4* column and the following "(size)" column show the accuracy and storage requirements when the *k*NN classifier using the IVDM distance function is pruned using the DROP4 reduction technique. This and other "(size)" columns show what percent of the training set *T* is retained in the subset *S* and subsequently used for actual classification of the test set. For the *k*NN and IVDM columns, 100% of the instances in the training set are used for classification.

The column labeled *LCV* adds the distance-weighted voting and parameter-tuning steps, but uses the leave-one-out cross-validation (LCV) metric instead of the cross-validation and confidence (CVC) metric.

Finally, the column labeled *IDIBL* uses the same distance-weighting and parameter-tuning steps as in the LCV column, but uses CVC as the fitness metric, *i.e.*, the full IDIBL system as described in Section 4. The highest accuracy achieved for each dataset by any of the algorithms is shown in bold type.

As can be seen from Table 2, each enhancement raises the average generalization accuracy on these datasets, including DROP4, which reduces storage from 100% to about 16%. Though these datasets are not particularly noisy, in experiments where noise was added to the output class in the training set, DROP4 had higher accuracy than the unpruned system due to its noise-filtering pass (Wilson & Martinez 1999).

A one-tailed Wilcoxon signed ranks test (Conover 1971; DeGroot 1986) was used to determine whether the average accuracy of IDIBL was significantly higher than the other methods on these datasets. As can be seen from the bottom row of Table 2, IDIBL had significantly higher generalization accuracy than each of the other methods at over a 95% confidence level.

It is interesting to note that the confidence level is higher when comparing IDIBL to the DROP4 results than to the IVDM results, indicating that although the average accuracy of DROP4 happened to be higher than IVDM (which retains all 100% of the training instances), it was statistically not quite as good as IVDM when compared to IDIBL. This is consistent with our experience in using DROP4 in other situations, where it tends to maintain or slightly reduce generalization accuracy in exchange for a substantial reduction in storage requirements and a corresponding improvement in subsequent classification speed (Wilson & Martinez 1999).

The IDIBL system does sacrifice some degree of storage reduction when compared to the DROP4 column. When distance-weighted voting is used and parameters are tuned to improve CVC fitness, a more precise decision boundary is found that may not allow for quite as many border instances to be removed. In addition, the parameter-tuning step can choose values for k larger than the default value of $k = 3$ used by the other algorithms, which can also prevent DROP4 in IDIBL from removing as many instances.

The results also indicate that using CVC fitness in IDIBL achieved significantly higher accuracy than using LCV fitness, with very little increase in the complexity of the algorithm.

5.2. Comparison of IDIBL with Other Machine Learning and Neural Network Algorithms

In order to see how IDIBL compares with other popular machine learning algorithms, the results of running IDIBL on 21 datasets were compared with results reported by Zarndt (1995). We exclude results for several of the datasets that appear in Table 2 for which the results are not directly comparable. Zarndt's results are also based on 10-fold cross-validation, though it was not possible to use the same partitions in our experiments as those used in his. He reported results for 16 learning algorithms, from which we have selected one representative learning algorithm from each general class of algorithms. Where more than one algorithm was available in a class (e.g., several decision tree models were available), the one that achieved the highest results in Zarndt's experiments is reported here.

Results for IDIBL are compared to those achieved by the following algorithms:

- C4.5 (Quinlan 1993), an inductive decision tree algorithm. Zarndt also reported results for ID3 (Quinlan 1986), C4, C4.5 using induced rules (Quinlan 1993), Cart (Breiman et al. 1984), and two decision tree algorithms using minimum message length (Buntine 1992).
- CN2 (using ordered lists) (Clark & Niblett 1989), which combines aspects of the AQ rule-inducing algorithm (Michalski 1969) and the ID3 decision tree algorithm (Quinlan 1986). Zarndt also reported results for CN2 using unordered lists (Clark & Niblett 1989).
- a "naive" Bayesian classifier (Langley, Iba & Thompson 1992; Michie, Spiegelhalter & Taylor 1994) (*Bayes*).
- the Perceptron (Rosenblatt 1959) single-layer neural network (*Per*).
- the Backpropagation (Rumelhart & McClelland 1986) neural network (*BP*).
- IB1-4 (Aha, Kibler & Albert 1991; Aha 1992), four instance-based learning algorithms.

All of the instance-based models reported by Zarndt were included since they are most similar to IDIBL. IB1 is a simple nearest neighbor classifier with $k = 1$. IB2 prunes the training set, and IB3 extends IB2 to be more robust in the presence of noise. IB4 extends IB3 to handle irrelevant attributes. All four use the Euclidean/overlap metric, and use *incremental* pruning algorithms, *i.e.*, they make decisions on which instances to prune before examining all of the available training data.

The results of these comparisons are presented in Table 3. The highest accuracy achieved for each dataset is shown in bold type. The average over all datasets is shown near the bottom of the table.

As can be seen from the results in the table, no algorithm had the highest accuracy on all of the datasets, due to the *selective superiority* (Brodley 1993) of each algorithm, *i.e.*, the degree to which each bias (Mitchell 1980) is appropriately matched for each dataset (Dietterich 1989; Wolpert 1993; Schaffer 1994; Wilson & Martinez 1997c). However, IDIBL had the highest accuracy of any of the algorithms for more of the datasets than any of the other learning models. It also had the highest overall average generalization accuracy.

In addition, a one-tailed Wilcoxon signed ranks test was used to verify whether the average accuracy on this set of classification tasks was significantly higher than each of the others. The bottom row of Table 3 gives the confidence level at which IDIBL is significantly higher than each of the other classification systems. As can be seen from the table, the average accuracy for IDIBL over this set of tasks was significantly higher than all of the other algorithms except for Backpropagation at over a 99% confidence level.

The accuracy for each of these datasets is shown in Table 2 for *k*NN, IVDM, DROP4 and LCV, but for the purpose of comparison, the average accuracy on the smaller set of 21 applications in Table 3 is given here as follows: *k*NN, 76.5%; IVDM, 80.1%; DROP4, 81.3%; LCV, 81.4%; and IDIBL, 81.9% (as shown in Table 2). The *k*NN and IB1 algorithms differ mainly in their use of $k = 3$ and $k = 1$, respectively, and their average accuracies are quite similar (77.7% vs. 76.5%). This indicates that the results are comparable, and that the enhancements offered by IDIBL do appear to improve generalization accuracy, at least on this set of classification tasks.

The accuracy for some algorithms might be improved by a more careful tuning of system parameters, but one of the advantages of IDIBL is that parameters do not need to be hand-tuned. The results presented above are theoretically limited to this set of applications, but the results indicate that IDIBL is a robust learning system that can be successfully applied to a variety of real-world problems.

Table 3. Generalization accuracy of IDIBL and several well-known machine learning models.

Dataset	C4.5	CN2	Bayes	Per	BP	IB1	IB2	IB3	IB4	IDIBL
Annealing	94.5	98.6	92.1	96.3	**99.3**	95.1	96.9	93.4	84.0	96.0
Australian	**85.4**	82.0	83.1	84.9	84.5	81.0	74.2	83.2	84.5	**85.4**
Breast Cancer	94.7	95.2	93.6	93.0	96.3	96.3	91.0	95.0	94.1	**97.0**
Bridges	56.5	58.2	66.1	64.0	**67.6**	60.6	51.1	56.8	55.8	63.2
Credit Screening	83.5	83.0	82.2	83.6	85.1	81.3	74.1	82.5	84.2	**85.4**
Flag	56.2	51.6	52.5	45.3	**58.2**	56.6	52.5	53.1	53.7	57.7
Glass	65.8	59.8	**71.8**	56.4	68.7	71.1	67.7	61.8	64.0	70.6
Heart Disease	73.4	78.2	75.6	80.8	82.6	79.6	73.7	72.6	75.9	**83.3**
Hepatitis	55.7	63.3	57.5	67.3	68.5	66.6	65.8	63.5	54.1	**81.9**
Horse-Colic	70.0	65.1	68.6	60.1	66.9	64.8	59.9	56.1	62.0	**73.8**
Ionosphere	90.9	82.6	85.5	82.0	**92.0**	86.3	84.9	85.8	89.5	87.8
Iris	94.0	92.7	94.7	95.3	96.0	95.3	92.7	95.3	**96.6**	96.0
LED+17 noise	66.5	61.0	64.5	60.5	62.0	43.5	39.5	39.5	64.0	**73.6**
LED	70.0	68.5	68.5	70.0	69.0	68.5	63.5	68.5	68.0	**74.9**
Liver Disorders	62.6	58.0	64.6	66.4	**69.0**	62.3	61.7	53.6	61.4	62.9
Pima Diabetes	72.7	65.1	72.2	74.6	**75.8**	70.4	63.9	71.7	70.6	**75.8**
Promoters	77.3	87.8	78.2	75.9	87.9	82.1	72.3	77.2	79.2	**88.6**
Sonar	73.0	55.4	73.1	73.2	76.4	**86.5**	85.0	71.1	71.1	84.1
Voting	**96.8**	93.8	95.9	94.5	95.0	92.4	91.2	90.6	92.4	95.6
Wine	93.3	90.9	94.4	**98.3**	98.3	94.9	93.2	91.5	92.7	93.8
Zoo	93.3	96.7	**97.8**	96.7	95.6	96.7	95.6	94.5	91.1	92.2
Average:	**77.4**	**75.6**	**77.7**	**77.1**	**80.7**	**77.7**	**73.8**	**74.2**	**75.7**	**81.9**
Wilcoxon:	99.5	99.5	99.5	99.5	81.8	99.5	99.5	99.5	99.5	*n/a*

6. CONCLUSIONS

The basic nearest neighbor algorithm has had success in some domains but suffers from inadequate distance functions, large storage requirements, slow execution speed, a sensitivity to noise, and an inability to fine-tune its concept description.

The *Integrated Decremental Instance-Based Learning* (IDIBL) algorithm combines solutions to each of these problems into a comprehensive learning system. IDIBL uses the *Interpolated Value Difference Metric* (IVDM) to provide an appropriate distance measure between input vectors that can have both linear and nominal attributes. It uses the DROP4 reduction technique to reduce storage requirements, improve classification speed, and reduce sensitivity to noise. It also uses a distance-weighted voting scheme with parameters that are tuned using a combination of cross-validation accuracy and confidence (CVC) in order to provide a more flexible concept description.

In experiments on 30 datasets, IDIBL significantly improved upon the generalization accuracy of similar algorithms that did not include all of the enhancements. When compared with results reported for other popular learning algorithms, IDIBL achieved significantly higher average generalization accuracy than any of the others (with the exception of the backpropagation network algorithm, where IDIBL was higher but not by a significant amount).

The basic nearest neighbor algorithm is also sensitive to irrelevant and redundant attributes. The IDIBL algorithm presented in this paper does not address this shortcoming directly. Some attempts at attribute weighting were made during the development of the IDIBL algorithm, but the improvements were not significant enough to report here, so this remains an area of future research.

Since each algorithm is better suited for some problems than others, another key area of future research is to understand under what conditions each algorithm—including IDIBL—is successful, so that an appropriate algorithm can be chosen for particular applications, thus increasing the chance of achieving high generalization accuracy in practice.

REFERENCES

Aha, David W., and Robert L. Goldstone. 1992. Concept Learning and Flexible Weighting. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, Bloomington, IN: Lawrence Erlbaum, pp. 534-539.

Aha, David W. 1992. Tolerating noisy, irrelevant and novel attributes in instance-based learning algorithms. *International Journal of Man-Machine Studies*, vol. 36, pp. 267-287.

Aha, David W., Dennis Kibler, Marc K. Albert. 1991. Instance-Based Learning Algorithms. *Machine Learning*, vol. 6, pp. 37-66.

Atkeson, Chris. 1989. Using local models to control movement. In D. S. Touretzky (Ed.), *Advances in Neural Information Processing Systems 2*. San Mateo, CA: Morgan Kaufmann.

Atkeson, Chris, Andrew Moore, and Stefan Schaal. 1997. Locally weighted learning. *Artificial Intelligence Review*, vol. 11, pp. 11-73.

Batchelor, Bruce G. 1978. *Pattern Recognition: Ideas in Practice*. New York: Plenum Press.

Biberman, Yoram. 1994. A Context Similarity Measure. *In Proceedings of the European Conference on Machine Learning (ECML-94)*. Catalina, Italy: Springer Verlag, pp. 49-63.

Breiman, Leo, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. 1984. *Classification and Regression Trees*, Wadsworth International Group, Belmont, CA.

Breiman, Leo. 1996. Bagging Predictors. *Machine Learning*, vol. 24, pp. 123-140.

Brodley, Carla E. 1993. Addressing the Selective Superiority Problem: Automatic Algorithm/Model Class Selection. *Proceedings of the Tenth International Machine Learning Conference,* Amherst, MA, pp. 17-24.

Broomhead, D. S., and D. Lowe 1988. Multi-variable functional interpolation and adaptive networks. *Complex Systems*, vol. 2, pp. 321-355.

Buntine, Wray. 1992. Learning Classification Trees. *Statistics and Computing*, vol. 2, pp. 63-73.

Cameron-Jones, R. M. 1995. Instance Selection by Encoding Length Heuristic with Random Mutation Hill Climbing. *In Proceedings of the Eighth Australian Joint Conference on Artificial Intelligence*, pp. 99-106.

Carpenter, Gail A., and Stephen Grossberg. 1987. A Massively Parallel Architecture for a Self-Organizing Neural Pattern Recognition Machine. *Computer Vision, Graphics, and Image Processing*, vol. 37, pp. 54-115.

Chang, Chin-Liang. 1974. Finding Prototypes for Nearest Neighbor Classifiers. *IEEE Transactions on Computers*, vol. 23, no. 11, November 1974, pp. 1179-1184.

Clark, Peter, and Tim Niblett. 1989. The CN2 Induction Algorithm. *Machine Learning*, vol. 3, pp. 261-283.

Conover, W. J. 1971. *Practical Nonparametric Statistics*. New York: John Wiley, pp. 206-209, 383.

Cost, Scott, and Steven Salzberg. 1993. A Weighted Nearest Neighbor Algorithm for Learning with Symbolic Features. *Machine Learning*, vol. 10, pp. 57-78.

Cover, T. M., and P. E. Hart. 1967. Nearest Neighbor Pattern Classification. *Institute of Electrical and Electronics Engineers Transactions on Information Theory*, vol. 13, no. 1, January 1967, pp. 21-27.

Dasarathy, Belur V., and Belur V. Sheela. 1979. A Composite Classifier System Design: Concepts and Methodology. *Proceedings of the IEEE*, vol. 67, no. 5, May 1979, pp. 708-713.

Dasarathy, Belur V. 1991. Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques, Los Alamitos, CA: IEEE Computer Society Press.

DeGroot, M. H. 1986. *Probability and Statistics* (Second Edition). Reading, MA: Addison-Wesley.

Deng, Kan, and Andrew W. Moore. 1995. Multiresolution Instance-Based Learning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'95).*

Diday, Edwin. 1974. Recent Progress in Distance and Similarity Measures in Pattern Recognition. *Second International Joint Conference on Pattern Recognition*, pp. 534-539.

Dietterich, Thomas G. 1989. Limitations on Inductive Learning. In *Proceedings of the Sixth International Conference on Machine Learning*. San Mateo, CA: Morgan Kaufmann, pp. 124-128.

Domingos, Pedro 1995. Rule Induction and Instance-Based Learning: A Unified Approach. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, Canada: Morgan Kaufmann, pp. 1226-1232.

Dudani, Sahibsingh A. 1976. The Distance-Weighted *k*-Nearest-Neighbor Rule. *IEEE Transactions on Systems, Man and Cybernetics*, vol. 6, no. 4, April 1976, pp. 325-327.

Fisher, R. S. 1936. The Use of Multiple Measurements in Taxonomic Problems. *Annual Eugenics*, vol. 7, part II, pp. 179-188.

Gates, G. W. 1972. The Reduced Nearest Neighbor Rule. *IEEE Transactions on Information Theory*, vol. IT-18, no. 3, pp. 431-433.

Giraud-Carrier, Christophe, and Tony Martinez. 1995. An Efficient Metric for Heterogeneous Inductive Learning Applications in the Attribute-Value Language. *Intelligent Systems*, pp. 341-350.

Hart, P. E. 1968. The Condensed Nearest Neighbor Rule. Institute of Electrical and Electronics Engineers Transactions on Information Theory, vol. 14, pp. 515-516.

Hecht-Nielsen, R. 1987. Counterpropagation Networks. *Applied Optics*, vol. 26, no. 23, pp. 4979-4984.

Keller, James M., Michael R. Gray, and James A. Givens, Jr. 1985. A Fuzzy *K*-Nearest Neighbor Algorithm. *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 15, no. 4, July/August 1985, pp. 580-585.

Kohavi, Ron. 1995. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'95).*

Kohonen, Teuvo. 1990. The Self-Organizing Map. *In Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464-1480.

Langley, Pat, Wayne Iba, and Kevin Thompson. 1992. An Analysis of Bayesian Classifiers. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, AAAI Press/MIT Press, Cambridge, Massachusetts, pp. 223-228.

Lebowitz, Michael. 1985. Categorizing Numeric Information for Generalization. Cognitive Science, vol. 9, pp. 285-308.

Lowe, David G. 1995. Similarity Metric Learning for a Variable-Kernel Classifier. *Neural Computation.*, *7-1*, pp. 72-85.

Merz, C. J., and P. M. Murphy. 1996. *UCI Repository of Machine Learning Databases*. Irvine, CA: University of California Irvine, Department of Information and Computer Science. Internet: http://www.ics.uci.edu/~mlearn/ MLRepository.html.

Michalski, Ryszard S. 1969. On the quasi-minimal solution of the general covering problem. *Proceedings of the Fifth International Symposium on Information Processing*, Bled, Yugoslavia, pp. 12-128.

Michalski, Ryszard S., Robert E. Stepp, and Edwin Diday. 1981. A Recent Advance in Data Analysis: Clustering Objects into Classes Characterized by Conjunctive Concepts. *Progress in Pattern Recognition*, vol. 1, Laveen N. Kanal and Azriel Rosenfeld (Eds.). New York: North-Holland, pp. 33-56.

Michie, D., D. Spiegelhalter, and C. Taylor. 1994. *Machine Learning, Neural and Statistical Classification*, Ellis Horwood, Hertfordshire, England. Book 19.

Mohri, Takao, and Hidehiko Tanaka, "An Optimal Weighting Criterion of Case Indexing for Both Numeric and Symbolic Attributes. In D. W. Aha (Ed.), Case-Based Reasoning: Papers from the 1994 Workshop, Technical Report WS-94-01. Menlo Park, CA: AIII Press, pp. 123-127.

Moore, Andrew W., and Mary S. Lee. 1993. Efficient Algorithms for Minimizing Cross Validation Error. In *Machine Learning: Proceedings of the Eleventh International Conference*, Morgan Kaufmann.

Nadler, Morton, and Eric P. Smith. 1993. *Pattern Recognition Engineering*. New York: Wiley.

Nosofsky, Robert M. 1986. Attention, Similarity, and the Identification-Categorization Relationship. *Journal of Experimental Psychology: General*, vol. 115, no. 1, pp. 39-57.

Papadimitriou, C. H., and Steiglitz, K. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ.

Papadimitriou, Christos H., and Jon Louis Bentley. 1980. A Worst-Case Analysis of Nearest Neighbor Searching by Projection. *Lecture Notes in Computer Science*, vol. 85, Automata Languages and Programming, pp. 470-482.

Quinlan, J. R. 1986. Induction of Decision Trees", *Machine Learning*, vol. 1, pp. 81-106.

Quinlan, J. R. 1989. Unknown Attribute Values in Induction. *Proceedings of the 6th International Workshop on Machine Learning*, San Mateo, CA: Morgan Kaufmann, pp. 164-168.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*, San Mateo, CA: Morgan Kaufmann.

Rachlin, John, Simon Kasif, Steven Salzberg, David W. Aha. 1994. Towards a Better Understanding of Memory-Based and Bayesian Classifiers. In *Proceedings of the Eleventh International Machine Learning Conference,* New Brunswick, NJ: Morgan Kaufmann, pp. 242-250.

Renals, Steve, and Richard Rohwer. 1989. Phoneme Classification Experiments Using Radial Basis Functions. *Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN'89)*, vol. 1, pp. 461-467.

Ritter, G. L., H. B. Woodruff, S. R. Lowry, and T. L. Isenhour. 1975. An Algorithm for a Selective Nearest Neighbor Decision Rule. *IEEE Transactions on Information Theory*, vol. 21, no. 6, November 1975, pp. 665-669.

Rosenblatt, Frank. 1959. *Principles of Neurodynamics*, New York, Spartan Books.

Rumelhart, D. E., and J. L. McClelland, *Parallel Distributed Processing*, MIT Press, 1986.

Salzberg, Steven. 1991. A Nearest Hyperrectangle Learning Method. *Machine Learning*, vol. 6, pp. 277-309.

Schaffer, Cullen. 1994. A Conservation Law for Generalization Performance. In *Proceedings of the Eleventh International Conference on Machine Learning (ML'94)*, Morgan Kaufmann, 1994.

Schaffer, Cullen. 1993. Selecting a Classification Method by Cross-Validation. *Machine Learning*, vol. 13, no. 1.

Schapire, Robert E., Yoav Freund, Peter Bartlett, and Wee Sun Lee. 1997. Boosting the Margin: A New Explanation for the Effectiveness of Voting Methods. In Fisher, D., ed., *Machine Learning: Proceedings of the Fourteenth International Conference (ICML'97)*, Morgan Kaufmann Publishers, San Francisco, CA, pp. 322-330.

Skalak, David B. 1994. Prototype and Feature Selection by Sampling and Random Mutation Hill Climbing Algorithms. *Proceedings of the Eleventh International Conference on Machine Learning (ML'94)*.

Sproull, Robert F. 1991. Refinements to Nearest-Neighbor Searching in *k*-Dimensional Trees. *Algorithmica*, vol. 6, pp. 579-589.

Stanfill, C., and D. Waltz. 1986. Toward memory-based reasoning. *Communications of the ACM*, vol. 29, December 1986, pp. 1213-1228.

Tomek, Ivan. 1976. An Experiment with the Edited Nearest-Neighbor Rule. *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 6, no. 6, June 1976, pp. 448-452.

Tversky, Amos. 1977. Features of Similarity. *Psychological Review*, vol. 84, no. 4, pp. 327-352.

Wasserman, Philip D. 1993. *Advanced Methods in Neural Computing*, New York, NY: Van Nostrand Reinhold.

Watson, I., and F. Marir. 1994. Case-Based Reasoning: A Review. *The Knowledge Engineering Review*, vol. 9, no. 4.

Wess, Stefan, Klaus-Dieter Althoff and Guido Derwand. 1994. Using *k*-d Trees to Improve the Retrieval Step in Case-Based Reasoning. Stefan Wess, Klaus-Dieter Althoff, & M. M. Richter (Eds.), *Topics in Case-Based Reasoning*. Berlin: Springer-Verlag, pp. 167-181.

Wettschereck, Dietrich 1994. A Hybrid Nearest-Neighbor and Nearest-Hyperrectangle Algorithm, In *Proceedings of the 7th European Conference on Machine Learning*, LNAI-784, F. Bergadano and L. de Raedt (editors), pp. 323-335.

Wettschereck, Dietrich, and Thomas G. Dietterich. 1995. An Experimental Comparison of Nearest-Neighbor and Nearest-Hyperrectangle Algorithms. *Machine Learning*, vol. 19, no. 1, pp. 5-28.

Wettschereck, Dietrich, David W. Aha, and Takao Mohri. 1995. A Review and Comparative Evaluation of Feature Weighting Methods for Lazy Learning Algorithms. Technical Report AIC-95-012, Washington, D.C.: Naval Research Laboratory, Navy Center for Applied Research in Artificial Intelligence.

Wilson, D. Randall, and Tony R. Martinez. 1996. Instance-Based Learning with Genetically Derived Attribute Weights. *International Conference on Artificial Intelligence, Expert Systems and Neural Networks (AIE'96),* pp. 11-14.

Wilson, D. Randall, and Tony R. Martinez. 1997a. Improved Heterogeneous Distance Functions. *Journal of Artificial Intelligence Research*, vol. 6, no. 1, pp. 1-34.

Wilson, D. Randall, and Tony R. Martinez 1997b. Instance Pruning Techniques. In Fisher, D., ed., *Machine Learning: Proceedings of the Fourteenth International Conference (ICML'97)*, Morgan Kaufmann Publishers, San Francisco, CA, pp. 403-411.

Wilson, D. Randall, and Tony R. Martinez. 1997c. Bias and the Probability of Generalization. In *Proceedings of the 1997 International Conference on Intelligent Information Systems* (*IIS'97*), pp. 108-114, 1997.

Wilson, D. Randall, and Tony R. Martinez. 1999. Reduction Techniques for Exemplar-Based Learning Algorithms. To appear in *Machine Learning Journal.*

Wilson, Dennis L. 1972. Asymptotic Properties of Nearest Neighbor Rules Using Edited Data. *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 2, no. 3, July 1972, pp. 408-421.

Wolpert, David H. 1993. On Overfitting Avoidance as Bias. Technical Report SFI TR 92-03-5001. Santa Fe, NM: The Santa Fe Institute.

Zarndt, Frederick. 1995. *A Comprehensive Case Study: An Examination of Connectionist and Machine Learning Algorithms*, Master's Thesis, Brigham Young University.

Zhang, Jianping. 1992. Selecting Typical Instances in Instance-Based Learning. *Proceedings of the Ninth International Conference on Machine Learning*.