Artificial Intelligence Review 11, 407-423, 1997

IGTree: Using Trees for Compression and Classification in Lazy Learning Algorithms

Walter Daelemans[(i)], Antal van den Bosch[(ii)], Ton Weijters[(ii)]

[(i)] Walter.Daelemans@kub.nl
Computational Linguistics
Tilburg University, The Netherlands

[(ii)] {antal,weijters}@cs.rulimburg.nl
MATRIKS
Maastricht University, The Netherlands

Abstract

We describe the IGTree learning algorithm, which compresses an instance base into a tree structure. The concept of information gain is used as a heuristic function for performing this compression. IGTree produces trees that, compared to other lazy learning approaches, reduce storage requirements and the time required to compute classifications. Furthermore, we obtained similar or better generalization accuracy with IGTree when trained on two complex linguistic tasks, viz. letter–phoneme transliteration and part-of-speech-tagging, when compared to alternative lazy learning and decision tree approaches (viz., IB1, information-gain-weighted IB1, and C4.5). A third experiment, with the task of word hyphenation, demonstrates that when the mutual differences in information gain of features is too small, IGTree as well as information-gain-weighted IB1 perform worse than IB1. These results indicate that IGTree is a useful algorithm for problems characterized by the availability of a large number of training instances described by symbolic features with sufficiently differing information gain values.

keywords: lazy learning, eager learning, decision trees, information gain, data compression, instance base indexing

1 Introduction

In previous research, we have applied lazy learning techniques to a variety of problems in *language technology* (e.g., converting spelling to phonetic transcription, stress assignment, predicting morphological suffixes, and assigning syllable structure to words). See Daelemans (1995) for an overview, and Cardie (1993) for a similar case-based approach. This type of linguistic problem can be characterized by the following observations:

1. The problem can be described as finding a mapping from a pattern of *symbolic* (nominal and unordered) features (letters, phonemes, part-of-speech tags, etc.) to a *symbolic* class (phonemes, boundary symbols, affixes, tags, etc.).

2. The problem can be described as classification in context: given a target symbol and its immediate local context, produce one of a finite number of possible classes for that symbol. For example, given a spelling symbol and its three left and three right neighbor letters, decide which phonetic symbol it corresponds to.

3. The instance features display an outspoken variation in their relevance to solving the task, and can be ordered according to this relevance. In general, the further away a feature (representing context) from the target, the less relevant it is.

4. The instance space is reasonably large (e.g., seven features with 27 possible values each, in the spelling-to-phonetic-transcription problem) and, typically, there are also many training instances available (on the order of 100,000 or more).

5. The problem is usually described (in terms of linguistic rules) as noisy and complex, with many subregularities and (pockets of) exceptions. In other words, apart from a core of generalizations, there is a relatively large periphery of irregularities.

In lazy learning (e.g., the IB1 algorithm in Aha, Kibler, and Albert, 1991), similarity of a new instance to stored instances is used to find the nearest neighbors of the new instance. The classes associated with the nearest neighbor instances are then used to predict the class of the new instance. In IB1, all features are assigned the same relevance, which is undesirable for our linguistic problems. We noticed that IB1, when extended with a simple feature weighting similarity function, sometimes outperforms both connectionist approaches and knowledge-based "linguistic–engineering" approaches (Daelemans and Van den Bosch, 1992, 1994; Van den Bosch and Daelemans, 1993). The similarity function we introduced in lazy learning (Daelemans and Van den Bosch, 1992) consisted simply of multiplying, when comparing two instances, the similarity between the values for each feature with the corresponding *information gain* for that feature (information gain is also implemented in C4.5, Quinlan, 1993, to guide decision tree building). We will call this version of lazy learning IB1-IG.

To compute similarity in IB1-IG, the similarity function *sim* in Equation 1 is used, in which X and Y are two instances of which the similarity must be computed, $G(fi)$ is the information gain of the ith feature, and $\sigma(x_i, y_i)$ is the overlap between the values of the ith feature in instances X and Y. Both instances contain n features.

$$sim(X,Y) = \sum_{i=1}^{n} G(f_i)\sigma(x_i, y_i) \tag{1}$$

As we are only investigating the learning of instances with symbolic features, the overlap function proposed by Stanfill and Waltz (1986) is used (Equation 2).

$$\sigma(x_i, y_i) = 1 \ if \ x_i = y_i, \ else \ 0 \tag{2}$$

The main idea of *information gain weighting* is to interpret the training material as an information source capable of generating a number of messages (the classes associated with stored instances) with a certain probability. Data base information entropy is equal to the average number of bits of information needed to know the class given an instance. It is computed by Equation 3, where p_i (the probability of class i) is estimated by its relative frequency in the training set.

$$H(D) = -\sum_{p_i} p_i log_2 p_i \tag{3}$$

For each feature, its relative importance in the data base can be calculated by computing its information gain. To do this, we compute the average information entropy for this feature and subtract it from the information entropy of the data base. To compute the average information entropy for a feature (Equation 4), we take the average information entropy of the database restricted to each possible value for the feature. The expression $D_{[f=v]}$ refers to those instances in the database that have value v for feature f, where V is the set of possible values for feature f. Finally, $|D|$ is the number of instances in data base D.

$$H(D_{[f]}) = \sum_{v_i \in V} H(D_{[f=v_i]}) \frac{|D_{[f=v_i]}|}{|D|} \tag{4}$$

Information gain is then obtained by Equation 5.

$$G(f) = H(D) - H(D_{[f]}) \tag{5}$$

The classification function of IB1-IG computes the similarity between a new instance and all stored instances, and returns the class label of the most similar instance.

During experimentation, we noticed that accuracy (generalization performance) decreased considerably when instance memory was pruned in some way (e.g., using IB2, Aha *et al.*, 1991, or by eliminating nontypical instances). Storing all training items by lazy learning (e.g., IB1) seems essential for achieving a high generalization performance in many linguistic tasks we investigated. The observation that the problems exhibit a lot of sub-regularity and exceptions may explain why full memory produces better results than an approach in which not all training items are kept in memory (cf. Aha, 1992).

Unfortunately, as the prediction function in lazy learning has to compare a test instance to all stored instances, and our linguistic data sets typically contain hundreds of thousands of instances, processing of new instances is prohibitively slow. Hardware solutions to this problem have been proposed (e.g. data-level parallelism on massively parallel machines, Stanfill and Waltz, 1986; or wafer-scale integration, Kitano, 1993). We will not discuss these here as we focus on comparing implementations of different algorithms on serial machines. What we needed was an algorithmic variant of IB1-IG in which the instance base is reorganized (by compression rather than by pruning) in such a way that access to relevant instances is faster, and no generalization performance is lost.

We developed an algorithm, IGTree (a first version is described in Van den Bosch and Daelemans, 1993), which uses the differences in information gain of features for ordering the instance base and optimizing access to the instance base. For the type of problem described above, IGTree produces a tree structure which is considerably smaller than the original data base; furthermore, tree retrieval is considerably faster than retrieval in IB1-IG.

In Section 2, we describe the IGTree model and its relationships to k-d trees and decision trees. Section 3 describes comparative experiments with IGTree, IB1, IB1-IG, and C4.5 on learning the linguistic tasks. We present our conclusions in Section 4.

2 IGTree

The positive effect of using information gain weights in the overlap function to define similarity in IB1 for our tasks, prompted us to develop an alternative approach in which the instance memory is reorganized, using Information Gain as a heuristic guide, in such a way that it contains the information essential for retrieval, but is compressed into a decision tree structure. In this Section, we will provide both an intuitive and algorithmic description of IGTree, discuss its relations to k-d trees and top down induced decision trees, and provide some analyses on complexity issues.

2.1 The IGTree model

IGTree combines two algorithms: one for constructing decision trees, and one for retrieving classification information from these trees. During the construction of IGTree decision trees, instances are stored as paths of connected nodes. All nodes contain a test (based on one of the features) and a class label (representing the default class at that node). Nodes are connected via arcs denoting the outcomes for the test (feature values). Information gain is used to determine the order in which instance features are used as tests in the tree. This order is fixed in advance, so the maximal depth of the tree is always equal to the number of features, and at the same level of the tree, all nodes have the same test. The reasoning behind this reorganization (which is in fact a compression) is that when the computation of information gain points to one feature clearly being the most important in classification, search can be restricted to matching a test instance to those stored instances that have the same feature value at that feature. Instead of restricting search to those memory instances that match only on this feature, the instance memory can then be optimized further by examining the second most important feature, followed by the third most important feature, etc. A considerable compression is obtained as similar instances share partial paths.

Instead of converting the instance base to a tree in which all instances are fully represented as paths, storing all feature values, we compress the tree even more by restricting the paths to those input feature values that disambiguate the classification from all other instances in the training material. The idea is that it is not necessary to fully store an instance as a path when only a few feature values of the instance make the instance classification unique. This implies that feature values that do not contribute to the disambiguation of the instance classification (i.e., the values of the features with lower information gain values than the the lowest information gain value of the disambiguating features) are *not* stored in the tree. Although one could opt for storing these features, not storing them does not affect the accuracy of IGTree's generalization performance.

Leaf nodes contain the unique class label corresponding to a path in the tree. Non-terminal nodes contain information about the *most probable* or *default* classification given the path thus far, according to the bookkeeping information on class occurrences maintained by the tree construction algorithm. This extra information is essential when using the tree for classification. Finding the classification of a new instance involves traversing the tree (i.e., matching all feature-values of the test instance with arcs in the order of the overall feature information gain), and either retrieving a classification when a leaf is reached, or using the default classification on the last matching non-terminal node if a feature-value match fails.

A final compression is obtained by pruning the derived tree. All leaf-node daughters of a mother node that have the same class as that node are removed from the tree, as their class information does not contradict the default class information already present at the mother node. Again, this compression does not affect the accuracy of IGTree's generalization performance.

In sum, in the trade-off between computation during learning and computation during classification, the IGTree approach chooses to invest more time in organizing the instance base using information gain and compression, at the gain of considerably simplified and faster processing during classification, as compared to lazy learning approaches that maintain instances in a flat file rather than using an reorganizing scheme.

A tree produced by the IGTree algorithm is *oblivious* because all nodes at a certain level in the tree test the same feature. The IGTree approach differs in two aspects from other oblivious decision tree (cf. Langley and Sage, 1994) and oblivious decision graph (cf. Kohavi and Li, 1995) approaches. First, in IGTree, information gain of features is used to determine the order in which they are expanded in the decision tree. The second difference is more fundamental, and is also related to the use of information gain as a guiding function in IGTree: in trees generated by IGTree, leaves are not necessarily stored at the same level. During tree building, expansion of the tree is stopped when all instances in the subset indexed by a node are of the same class. At that point, which may be at any level in the tree, all remaining features with a lower information gain value are ignored. Similarly, IGTree classifies a new instance by investigating a variable and often limited number of features, rather than a fixed number of (relevant) features, as in Kohavi and Li (1995).

The recursive algorithms for tree construction and retrieval are given in Figure 1.

2.2 Asymptotic complexity

As far as an asymptotic analysis of the complexity of storage, search and tree-building is concerned, it should be noted that only worst-case results are given. The actual compression (on which complexity of search, building, and storage depend) is completely task-dependent, and should be observed in empirical tests such as those in Section 3.

The worst-case complexity of searching an instance in the tree is proportional to $F * log(V)$, where F is the number of features (equal to the maximal depth of the tree), and V is the average number of values per feature (i.e., the average branching factor in the tree). This complexity presupposes alphabetic sorting of the values so that binary search and storage are possible. Retrieval by search in the tree is independent from the number of training instances, and therefore especially useful for large instance bases. In IB1, search complexity is $O(N * F)$ (with N the number of stored instances). In the grapheme–phoneme transliteration experiment described in Section 3, the average branching factor V is 2.3 (the number of possible values for each feature is 41).

The number of nodes necessary in the worst case to store the instances of the training set is N (maximal number of leaves) + $(N - 1) * (V - 1)$ (number of non-terminal nodes). For each non-terminal node, a default class label and a pointer for each occurring value of the feature denoted by the node should be stored. This makes the storage requirements proportional to N (compare $O(N * F)$ for IB1). In Section 3, we show that trained on the grapheme–phoneme transliteration problem, the IGTree

Procedure **BUILD-IG-TREE**:

Input:

- A training set T of instances with their classes (start value: a full instance base),
- an information-gain-ordered list of features (tests) $F_i ... F_n$ (start value: $F_1 ... F_n$).

Output: A subtree.

1. If T is unambiguous (all instances in T have the same class c), or $i = (n+1)$, create a leaf node with class label c.
2. Otherwise, until $i = n$ (the number of features)
   - Select the first feature (test) F_i in $F_i ... F_n$, and construct a new node N for feature F_i, and as default class c (the class occurring most frequently in T).
   - Partition T into subsets $T_1 ... T_m$ according to the values $v_1 ... v_m$ which occur for F_i in T (instances with the same value for this feature in the same subset).
   - For each $j \epsilon \{1, ..., m\}$:
     if not all instances in T_j map to class c, BUILD-IG-TREE $(T_j, F_{i+1} ... F_n)$,
     connect the root of this subtree to N and label the arc with v_j.

Procedure **SEARCH-IG-TREE**:

Input:

- The root node N of an subtree (start value: top node of a complete IGTree),
- an unlabeled instance I with information-gain-ordered feature values $f_i ... f_n$ (start value: $f_1 ... f_n$).

Output: A class label.

1. If N is a leaf node, output default class c associated with this node.
2. Otherwise, if test F_i of the current node does not originate an arc labeled with f_i, output default class c associated with N.
3. Otherwise,
   - new node M is the end node of the arc originating from N with as label f_i.
   - SEARCH-IG-TREE $(M, f_{i+1} ... f_n)$

Figure 1: Algorithms for building IGTrees ('BUILD-IG-TREE') and searching IGTrees ('SEARCH-IG-TREE')

decision trees use on the average 95% less memory than the IB1 instance bases.

Finally, the cost of building the tree on the basis of a set of instances is proportional to $N * log(V) * F$ in the worst case (compare $O(N)$ for training in IB1).

2.3 Relation to k-d trees and induced decision trees

The IGTree approach has strong similarities to both *decision tree learning* (Top Down Induction of Decision Trees, TDIDT, used for abstraction of knowledge from instances bases or indexing instance bases) and *k-d trees* (used for indexing instance bases).

A fundamental difference with decision trees concerns the purpose of IGTrees. The goal of Top Down Induction of Decision Trees, as in the state-of-the-art program C4.5 (Quinlan, 1993), is to *abstract* from the training examples. In contrast, we use decision trees for *lossless* compression of the training examples. Pruning of the resulting tree in order to derive understandable decision trees or rule sets is therefore not an issue in our approach. By *lossless*, we mean that the classifications of the training instances can be completely reconstructed, not that all feature-value information in the original training set can be reconstructed. Generalization is achieved by the defaults at each node, not by pruning. It should be noted here that IGTree decision trees can easily be expanded in such a way that compression is also lossless in terms of feature-value information, when node construction is not halted at the point where classification becomes unambiguous. However, we will refer in this paper only to the variant of IGTree in which features not relevant to classification are not stored.

A simplicification of IGTree as opposed to TDIDT approaches such as C4.5, is that IGTree generates oblivious decision trees, i.e., it computes information gain only once to determine a fixed *feature ordering.* TDIDT approaches, in contrast, recompute information gain (or similar feature selection functions) at each arc of the tree to guide selection of the next test. Finally, in IGTree, defaults are computed at each node of the tree (i.e., defaults are local), whereas in TDIDT, global defaults are used (although in C4.5, a similar local default assignment procedure is used).

In terms of high compression without generalisation performance loss, *C4.5rules* (Quinlan, 1993) appears a strong alternative to IGTree. However, C4.5rules, which extracts compact rule sets from trees generated by C4.5, becomes disproportionally slow when the C4.5-tree is large, as in our experiments: e.g., a C4.5-tree of $> 30,000$

nodes, generated within about a half hour (which is similar to IGTree's processing time), takes several days to be processed by C4.5rules.

K-d trees (Friedman, Bentley, and Finkel 1977) are binary trees that have been proposed for indexing databases of instances (with ordered feature values, e.g., numeric values) for use in k-nearest neighbor approaches. The basic idea is to make use of the observed density of the instance space to structure it for efficient retrieval of the m nearest neighbors of a new (query) instance. To build the k-d tree, the original instance space is partitioned into disjoint subsets by selecting a feature (e.g., the one with the highest inter-quartile-distance) and a threshold value, and creating nodes for each of these subsets. Instances with values for that feature less than or equal to the threshold are stored in one daughter, the others in the other daughter. Nodes therefore represent subsets of the instances. This process is recursively repeated until the number of instances in a node becomes less than a parameter called *bucket size* (maximal allowed number of instances in a leaf node), in which case a leaf node is constructed. The leaf node does not contain class information, as in IGTree, but pointers to the instances in the original instance base that are captured in the bucket. During retrieval of nearest neighbors, given a query instance, the k-d tree is traversed as in decision trees and IGTrees, and at leaf nodes, a queue with the m nearest neighbors is updated. Two tests, based on the similarity of the most similar instance in the queue to the query, are used to determine whether it is necessary to inspect the sister of the current leaf node, and whether all nearest neighbors have been found (if not, backtracking is necessary). Recently, there has been renewed interest in k-d trees and related approaches for efficiently indexing instance bases in lazy learning (Omohundro, 1991; Deng & Moore, 1995; Wess, Althoff, & Derwand, 1994; Wess, 1995).

In contrast to k-d trees, the purpose of IGTrees is classification, not efficient nearest neighbor search. IGTrees cannot be used to find the nearest neighbors because the defaults on the leaf nodes do not contain information about the number nor the identity of instances on which they were based. Instances sharing the same subset of feature-value pairs and having the same class in the training set, are not differentiated. Another difference between k-d trees and IGTrees is that the former are restricted to ordered feature values, while the latter are restricted to unordered symbolic features.

Efforts are under way (Wess *et al.*, 1994; Wess, 1995) to extend k-d trees with symbolic values. However, extending the test determining whether backtracking is needed for the case of symbolic features significantly increases the computational cost of executing this test, and the test can perform poorly under certain circumstances. There is no gain in retrieval time because the test has to verify each dimension of the attribute space, which can be high for unordered symbolic attributes (Althoff, personal communication). No empirical studies addressing this issue have been published yet.

Although we developed IGTree to deal with the nominal, unordered features with which we describe our linguistic instances, IGTree can be extended to handle continuous features by means of discretization techniques (cf. Dougherty, Kohavi, and Sahami, 1995).

Figure 2 graphically shows the differences between k-d trees, IGTrees and C4.5 decision trees on a small symbolic dataset. On the basis of size, shape, and number of holes, an object is to be classified as nut, screw, key, pen, or scissors. The instance base contains 12 instances. It should be noted that (i) instances 5 and 10 are ambiguous (i.e., they have the same feature values but map to different classes); (ii) the information gain, computed over the full set of instances, of feature 'size' is 0.75, of 'shape' is 0.90, and of 'number of holes' is 1.10; (iii) in the case of k-d trees, 'size' and 'shape' are not treated as numeric features as their values in the instance base are not numeric; in Figure 2 the situation is shown of a k-d tree algorithm which tests a symbolic feature by expanding the tree for every occurring value of that feature. As can be seen in Figure 2, the tree generated by IGTree differs from the tree generated by C4.5 in the number of tests (i.e., IGTree performs less tests than C4.5), and in the number of nodes and leaves (e.g., the sum amount of nodes and leaves is smaller in the case of IGTree than in the case of C4.5). The difference between IGTree and k-d tree is that the buckets in the k-d tree point to instances in the instance base, whereas the nodes and leaves in IGTree do not denote instances, but classifications.

Section 3 describes experiments illustrating the comparative performance (i.e., generalization accuracy and storage requirements) of IGTree, IB1, IB1-IG, and C4.5, for several linguistic tasks.

3 Experiments

In this Section we describe in detail empirical results achieved with IGTree on the letter–phoneme transliteration problem for Dutch. We compare the performance of

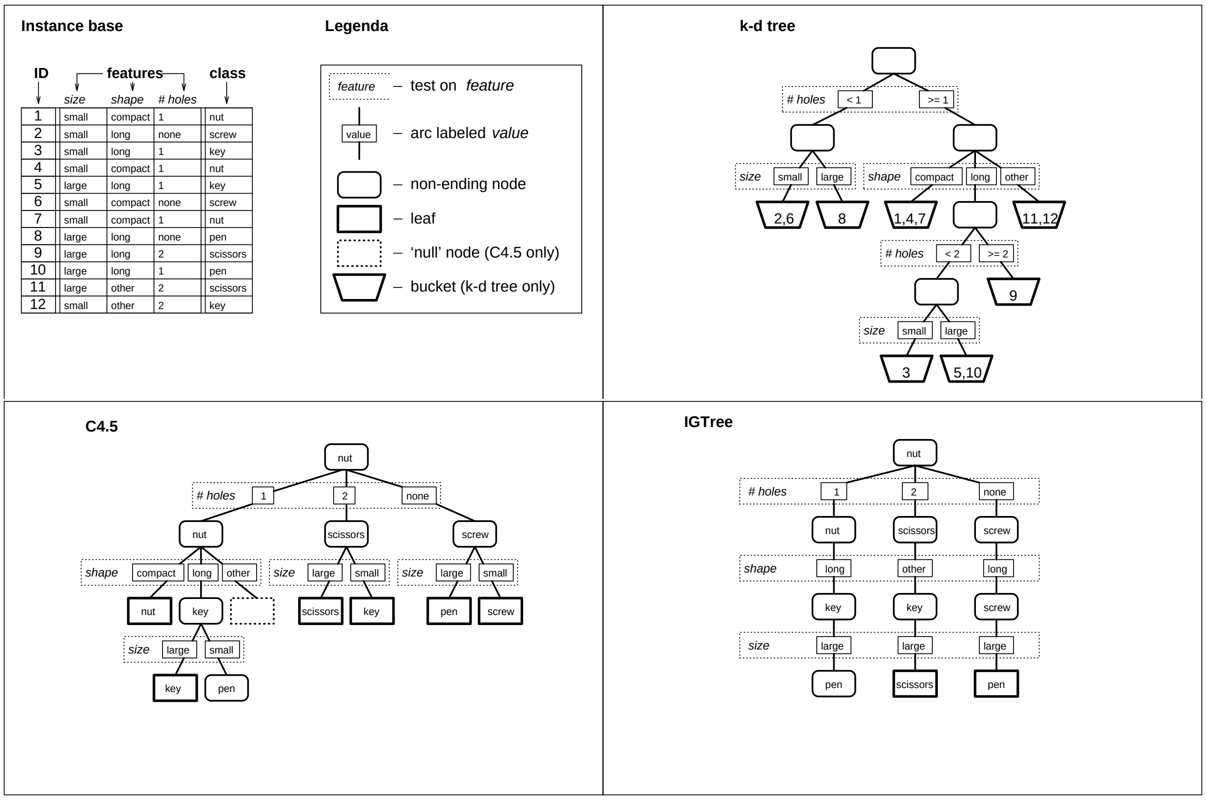

Figure 2: K-d tree, C4.5 decision tree, and IGTree decision tree constructed on the basis of a small instance base.

IGTree in terms of generalization accuracy and storage to IB1, IB1-IG, and C4.5. We provide similar but less detailed results on two other tasks: part-of-speech tagging and hyphenation.

3.1 Letter–Phoneme Transliteration

Letter–phoneme transliteration is a well-known benchmark problem, first discussed in the context of Machine Learning by Sejnowski and Rosenberg (1987). They report on several experiments with the standard connectionist Back-propagation algorithm (Rumelhart *et al.*, 1986) on the NETtalk data. In our experiments, we employ the same encoding scheme that Sejnowski and Rosenberg used to generate their instances (i.e., by moving a fixed-length window over a spelling word, and generating an instance by taking a snapshot of the word visible in the window). Each instance contains a target letter (in the middle) surrounded by left and right context letters. The class associated with the spelling input window is, in our case, the phonemic mapping of the target

Table 1: Example generation of fixed-length window instances (3 left context characters, 3 right context characters) from the word-pronunciation pair <boek> - /buk/. Underscores depict word boundaries. The '–' is the phonemic null mapping to the <e>.

Letter in Context							Phoneme
_	_	_	b	o	e	k	b
_	_	b	o	e	k	_	u
_	b	o	e	k	_	_	–
b	o	e	k	_	_	_	k

letter. As with Sejnowski and Rosenberg (1987), a class may be any of the phonemes in the phonemic alphabet, or a *phonemic null* inserted at points where a cluster of two or more spelling letters maps to one phoneme. An example of the generation of instances from a word-pronunciation pair, <boek> (book) - /buk/, is shown in Table 1.

Automatic learning of letter–phoneme transliteration of English (NETtalk, Sejnowski & Rosenberg, 1987) has been claimed as a success story for Back-propagation (but see Stanfill and Waltz, 1986, Wolpert, 1990, Weijters, 1991, and van den Bosch & Daelemans, 1993 for examples of k-nn algorithms outperforming Back-propagation). The connectionist approach was replicated for Dutch in NetSpraak (Weijters & Hoppenbrouwers, 1990).

From CELEX, a lexical data base of English, German, and Dutch, we derived a data base consisting of 20,000 Dutch word-pronunciation pairs. Words and phonemic transcriptions were made of equal length by inserting phonemic nulls ('–') in the phonemic transcriptions (by an alignment algorithm described in Daelemans and Van den Bosch, 1994). By using the *windowing* technique described above, the 20,000 word set was converted into a data base containing 218,594 instances. Each instance thus contains seven feature values (each of which is one out of 41 values: the alphabet including letters with diacritics, and the space that occurs before and after words), and is associated with one out of 55 possible phonemes. In our experiments, we used a 10-fold CV setup: i.e., we trained and tested each algorithm on ten different partitions (90% training material, 10% testing material) of the full data base. All performance results reported below are averaged over these experiments.

In Table 2, the performance scores on correctly classified test instances (i.e., correctly transliterated phonemes) and their standard deviations are displayed. We report the scores of IG-Tree, IB1 with the overlap similarity function, IB1 with the information-gain-weighted similarity function (IB1-IG), C4.5 without pruning, and C4.5 with pruning (C4.5-p). It should be noted that C4.5 was run with (i) the information-gain-criterion rather than the gain-ratio-criterion (as our data does not contain value variance anomalies which would be handled by the gain-ratio-criterion, and as this is the same function as used in IG-Tree), (ii) when pruning, the default pruning confidence level of 10% is used, and (iii) a minimum number of instances on either side of a test is set at 1, which is similar to IGTree, rather than the default value of 2. Furthermore, Table 2 reports the average number of bytes needed to store the instance base or decision tree in memory. Given the fact that in our problem feature-values and classes are represented by one byte, the memory allocated by IB1 (and IB1-IG) can be computed by multiplying the number of instances with the number of features plus one (the class). In our implementation of IGTree, seven bytes per node are used: a 4-byte pointer, a feature-value, a (default) class, and one byte indicating the number of daughter nodes. If the same compact memory storage method would be implemented in C4.5, this algorithm would need eight bytes per node (including one byte to denote the feature number). However, in C4.5 (version 7), each feature is expanded for all its possible values, leading to a very large number of 'empty' end nodes that contain no feature-values. Needing only seven bytes per node in this case, the results in Table 2 are based on the numbers of nodes reported by C4.5.

Table 3 indicates the significance levels of the differences between the generalization accuracy scores reported in Table 2.

The performance results in Table 2 and Table 3 indicate that IB1 with the overlap distance similarity function and C4.5 with pruning are at a significant disadvantage as compared to IGTree, IB1-IG, and C4.5 without pruning. IB1-IG outperforms, with a slight but significant margin, both IGTree and C4.5.

The average memory usage displayed in Table 2 demonstrates the considerable compression (95.1%) obtained with IGTree as compared to IB1-IG, without losing much generalization performance. In comparison, with pruning, C4.5 obtains 82.3% compression.

Table 2: Average generalization performance in terms of correctly transliterated phonemes of unseen Dutch word-pronunciation pairs, with standard deviation, and average memory usage in bytes needed to store the instance base or decision tree in memory, for IGTree, IB1, IB1 with an IG-weighted similarity function (IB1-IG), C4.5 without pruning, and C4.5 with pruning (C4.5-p).

Algorithm	Generalization accuracy on test phonemes	Standard deviation	Memory usage (bytes)
IGTree	97.07	0.11	77,749
IB1	92.11	0.15	1,573,885
IB1-IG	97.17	0.13	1,573,885
C4.5	97.03	0.14	992,047
C4.5-p	96.21	0.15	278,537

As a second illustration of accuracy, we mention the results of a comparison between IGTree trained on a set of 70,000 Dutch word-pronunciation pairs, and Morpa-cum-Morphon (Nunn and Van Heuven, 1993), a state-of-the-art "linguistic–engineering" system for Dutch. Tested on an identical test set (provided by the developers of Morpa-cum-Morphon), IGTree produced 89.5% correctly transliterated words, whereas Morpa-cum-Morphon only converted 85.3% words correctly (Van den Bosch and Daelemans, 1993).

3.2 Hyphenation and Part-of-Speech Tagging

In order to obtain a better insight into the properties of IGTree, we provide some additional results obtained with IGTree on different datasets.

Hyphenation

The problem of hyphenation (assigning syllable structure to the spelling of a word) is defined as a classification problem by using windowing as in grapheme–phoneme transliteration. For each target symbol (with a context of letters to the left and to the right of it), the class is either yes (start of a syllable at that position) or no (no start of syllable at that position). The experiment was based on 10-fold cross-validation on

Table 3: Significance levels of the differences between the generalization performances of IGTree, IB1, IB1-IG, C4.5, and C4.5-p. One or two asterisks ('*') in a cell in this Table indicate that the algorithm in the row is significantly better than the algorithm in the column. '**' indicates a probability of a Type-I error of 0.001 (t>3.61); '*' indicates a Type I-error probability of 0.05 (t>1.73). A blank cell indicates that the difference is not significant.

	IB1	C4.5-p	C4.5	IGTree
IB1-IG	**	**	*	*
IGTree	**	**		–
C4.5	**	**	–	–
C4.5-p	**	–	–	–

a dataset derived from 20,000 hyphenated English words.

The performance results indicate that IGTree (94.53%) performs significantly better than C4.5 (94.38%) and C4.5 with pruning (92.68%), but performs significantly worse than IB1 (95.30%) and IB1-IG (95.21%). Interestingly, there is no significant difference between IB1 and IB-IG. The information gain weights (reflecting feature accuracy) are insufficiently different in this case to make a difference.

The average memory usage again demonstrates the considerable compression (91.1%) obtained with IGTree as compared to IB1 and IB1-IG. In comparison, with pruning, C4.5 obtains 72.5% compression.

Part-of-Speech Tagging

In part-of-speech tagging, the task is to disambiguate the syntactic category of a word on the basis of preceding and following context. Again a windowing approach can be used to translate a corpus of tagged sentences into an instance base. The experiment was based on a single partitioning of a dataset into a training set of 100,000 instances, and a test set of 10,000 instances.

The performance results indicate that IGTree (95.1%) performs significantly better than IB1 (85.7%) and IB1-IG (94.7%). In this experiment we used Quinlan's (1993) gain ratio criterion rather than information gain, as not all features have an equal number of values in this problem. Memory compression with IGTree was 91.9% compared

Table 4: Information gain values of the seven input features (the focus letter F surrounded by context letters) of the grapheme-phoneme-transliteration task and the hyphenation task, and the gain ratio values of the four input features of the part-of-speech-tagging task.

Task	F-3	F-2	F-1	F	F+1	F+2	F+3
Grapheme-phoneme transliteration	0.185	0.280	0.711	3.059	0.857	0.381	0.218
Hyphenation	0.040	0.093	0.083	0.047	0.081	0.035	0.013
Part-of-speech Tagging	–	0.06	0.23	0.69	0.21	–	–

to IB1.

In Table 4, the information gain values of the seven input features of the grapheme-phoneme-transliteration task (cf. Section 3.1) and the hyphenation task (cf. Section 3.2) are displayed, as well as the gain ratio values of the four input features of the part-of-speech-tagging task (cf. Section 3.2).

4 Conclusions

We have shown that for two tasks, which are typical for a large class of real-world problems in natural language processing (cf. the characterisation of these problems in Section 1), IGTree performs only slightly worse or better in terms of generalization performance than IB1 augmented with an information-gain-weighted similarity function (IB1-IG), gaining considerably in memory resources needed for storage (91.9% and 95.1% compression in our experiments), and in search complexity ($O(F*log(V))$ rather than $O(F*N)$, which becomes especially favorable when N, the number of instances, is very large). Comparing IGTree with C4.5, which is aimed more at abstracting from training examples, we note that the current implementation of C4.5 (with pruning) generalizes less accurately than IGTree and IB1-IG, and uses more memory than IGTree. For a third task, viz. word hyphenation, in which there is no outspoken variation in the information gain values of the features, we have shown that both IGTree and IB1-IG generalise worse than IB1 with the overlap similarity function.

IGTree's tree building procedure is not aimed at indexing individual cases, as with k-d trees, but is aimed at compressed storage of those parts of training instances relevant to classification. Retrieval of class information from IGTree decision trees is speedy and deterministic, and does not involve backtracking. As the generalization accuracy of k-d trees (with symbolic values) can be assumed equal to that of IB1-IG, and since IB1-IG does not significantly perform better than IGTree, we conclude that, for the type of problem we investigated, it is not necessary to put extra processing effort in finding the absolute nearest neighbor to a new instance.

Given the fact that for many tasks large numbers of instances are available (several orders of magnitude more than the 1,000 instances of typical benchmark problems), the IGTree approach appears interesting and useful, especially for the type of problem characterized in this paper.

Acknowledgements

We gratefully recognize Jakub Zavrel, Stephen Omohundro, Klaus-Dieter Althoff, Andrew Moore, David Aha and the AI Review referees for valuable comments, discussions and analyses, and thank Fred Wan for his help with the significance tests.

References

Aha, D. W., Kibler, D., & Albert, M. (1991). Instance-based learning algorithms. *Machine Learning*, 7, 37–66.

Aha, D. W. (1992). Generalizing from case studies: A case study. In *Proceedings of the Ninth International Conference on Machine Learning*, 1–10. Aberdeen, Scotland: Morgan Kaufmann.

Cardie, C. (1993). A case-based approach to knowledge acquisition for domain-specific sentence analysis. In *AAAI-93*, 798–803.

Daelemans, W. (1995). Memory-based lexical acquisition and processing. In Steffens, P., editor, *Machine Translation and the Lexicon*, Lecture Notes in Artificial Intelligence 898. Berlin: Springer.

Daelemans, W., Van den Bosch, A. (1992). Generalisation performance of backpropagation learning on a syllabification task. In M. Drossaers & A. Nijholt (Eds.), *TWLT3: Connectionism and Natural Language Processing.* Enschede: Twente University.

Daelemans, W., Van den Bosch, A. (1994). A language-independent, data-oriented architecture for grapheme-to-phoneme conversion. In *Proceedings of ESCA-IEEE Speech Synthesis Conference '94*, New York.

Deng, K. & Moore, A. W. (1995). Multiresolution Instance-Based Learning. In *Proceedings of the Fourteenth International Joint Conference on Artifical Intelligence.* Montreal: Morgan Kaufmann.

Dougherty, J., Kohavi, R. & Sahami, M. (1995). Supervised and unsupervised discretization of continuous features. In *Proceedings of the International Conference on Machine Learning '95.*

Friedman, J., Bentley, J., and Ari Finkel, R. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3).

Kitano, H. (1993). Challenges of massive parallelism. In *IJCAI*, 813–834.

Kohavi, R., & Li, C-H. (1995). Oblivious decision trees, graphs, and top-down pruning. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 1071–1077. Montreal: Morgan Kaufmann.

Langley, P., & Sage, S. (1994). Oblivious decision trees and abstract cases. In D. W. Aha (Ed.), *Case-Based Reasoning: Papers from the 1994 Workshop* (Technical Report WS-94-01). Menlo Park, CA: AAAI Press.

Nunn, A. & van Heuven, V. J. (1993). Morphon, lexicon-based text-to-phoneme conversion and phonological rules. In V. J. van Heuven & L. C. Pols (Eds.), *Analysis and Synthesis of Speech: Strategic Research Towards High-Quality Text-to-Speech Generation.* Berlin: Mouton de Gruyter.

Omohundro, S. M. (1991). Bumptrees for Efficient Function, Constraint, and Classification Learning. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky (eds.) *Advances in Neural Information Processing Systems 3.* San Mateo, CA: Morgan Kaufmann.

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

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1: Foundations. Cambridge, MA: The MIT Press.

Sejnowski, T. J., Rosenberg, C. S. (1987). Parallel networks that learn to pronounce English text. *Complex Systems*, 1, 145–168.

Stanfill, C. and Waltz, D. (1986). Toward memory-based reasoning. *Communications of the ACM*, 29, 1212–1228.

Van den Bosch, A., Daelemans, W. (1993). Data-oriented methods for grapheme-to-phoneme conversion. In *Proceedings of the 6th Conference of the EACL*, 45–53. Utrecht: OTS.

Weijters, A., & Hoppenbrouwers, G. (1990). NetSpraak: een neuraal netwerk voor grafeem-foneem-omzetting. *Tabu*, 20:1, 1-25.

Weijters, A. (1991). A simple look-up procedure superior to NETtalk? In *Proceedings of the International Conference on Artificial Neural Networks*, Espoo, Finland.

Wess, S., Althoff, K. D., and Derwand, G. (1994). Using k-d trees to improve the retrieval step in case-based reasoning. In S. Wess, K. D. Althoff, and M. M. Richter (Eds.), *Topics in Case-Based Reasoning.* Berlin: Springer Verlag.

Wess, S. (1995). *Fallbasiertes Problemlösen in wissensbasierten Systemen zur Entscheidungsunterstützung und Diagnostik.* Doctoral Dissertation, University of Kaiserslautern.

Wolpert, D. H. (1990). Constructing a generalizer superior to NETtalk via a mathematical theory of generalization. *Neural Networks*, 3, 445–452.