Superlinear Parallelization of k-Nearest Neighbor Retrieval

Antal van den Bosch Ko van der Sloot

ILK Research Group / Dept. of Communication and Information Sciences,
Tilburg University, P.O. Box 90153, NL-5000 LE Tilburg, The Netherlands

Abstract

With m processors available, the k-nearest neighbor classifier can be straightforwardly parallelized with a linear speed increase of factor m. In this paper we introduce two methods that in principle are able to achieve this aim. The first method splits the test set in m parts, while the other distributes the training set over m sub-classifiers, and merges their m nearest neighbor sets with each classification. For our experiments we use TIMBL, an implementation of the k-NN classifier that uses a decision-tree structure for retrieving nearest neigbors, and that employs feature weighting. While the first method consistently scales linearly, with the second method we observe cases of both superlinear and sublinear scaling. Analysis shows that superlinear scaling can occur with datasets of which the feature weights exhibit a low variance; retrieval of nearest neighbors from the tree structure becomes exponentially slower with more data. Hence, the retrieval of classifications from m subclassifier decision structures based on $1/m$th parts of the training set can be substantially more than m times faster.

1 Introduction

The k-nearest neighbor (k-NN) classifier [5, 8, 1] is a classic machine learning algorithm of which the most distinctive feature is its reliance on the original training examples, rather than an abstracted model based on them, to classify new examples. Yet, this so-called lazy learning approach (no energy is spent on creating an abstract model) has the drawback of being slow in classification. Since every classification involves the comparison of a new example to all examples stored in memory, a naive implementation of the k-NN classifier takes $O(nf)$ to classify, where n is the number of training examples, and f is the number of features.

In this paper we describe two parallelization approaches to k-NN aimed at improving classification speed. The two approaches can run on multiple-processor shared-memory architectures, and are based on TIMBL [7], an implementation of IB1 [1] with feature weighting and other similarity function enhancements. The first parallelization, MUMBL (*multiple* TIMBLs), makes trivial use of the power provided by m processors by cloning one classifier into m classifiers, and running a master process that hands out $1/m$th of the test set in parallel to each clone. MUMBL exhibits near-linear scaling, only hindered by some processing overhead of its master process.

The second parallelization, DIMBL (*distributed* TIMBL), is also aimed to scale linearly with more processors available, by directly diminishing the role of n in $O(nf)$. DIMBL spawns m sub-classifiers each trained on $1/m$th part of the training set (i.e., on n/m examples). These smaller sub-classifiers, running on m processors in parallel, each classify the entire test set, but do so in less time than a classifier trained on the complete data set would; according to the worst-case complexity $O(nf)$, one would expect in $1/m$th of the time. A master classifier gathers the m nearest neighbor sets of the subclassifiers, and merges them to form the final nearest neighbor set to base the classification on.

We describe both methods in Section 2. In Section 3 we describe our experimental setup, and in Section 4 we compare TIMBL to its two parallel versions. We observe near-linear speedups with MUMBL, as expected, but we observe some remarkable superlinear speedups with DIMBL. However, we also observe cases in which DIMBL is not much faster than a single classifier. In Section 5 we discuss these results and formulate an explanation on the different speedup results of DIMBL.

2 Two parallelizations of k-NN

We present two parallelizations of a particular implementation of the classic k-NN classifier, TIMBL [7]. We first describe this implementation, which is in itself faster than a naive implementation of k-NN due to the implementation of fast retrieval heuristics for searching for the k nearest neighbors, and improvements through parallelization cannot be seen but relative to the gain of these heuristics. We then introduce MUMBL and DIMBL.

2.1 TIMBL: Trie-based k-NN

In TIMBL [7], the similarity function is implemented as a kernel capable of accomodating several distance metrics in various spaces, weighted by information-theoretic, statistical, or probabilistic estimations. Of these metrics, the feature weighting metric enables the determination of the nearest neighbor set to be faster than in the worst case. We introduced feature weighting into k-NN classification under the name of IB1-IG [6], as the factor w_i in the distance function $\Delta(X, Y)$ to compute the distance between two instances X and Y, where $\delta(x_i, y_i)$ is the function computing the distance between the values at feature i of both instances:

$$\Delta(X, Y) = \sum_{i=1}^{n} w_i \, \delta(x_i, y_i) \tag{1}$$

An appropriately-chosen feature weighting metric w tells the classifier which features are more important than others, i.e., which features should weigh more heavily in the distance function. A feature can be so important that a nearest neighbor *must* have the same value as the new example has at that feature. If this is the case, it means that, with the help of a simple index, search for a nearest neighbor can be restricted to only those memory examples having the same value at the feature with peak importance. In the extreme case that, when ranked by their feature weight, each higher-ranked feature has a weight larger than the sum of all lower weights (such as in the ordered weight sequence $1, 2, 4, 8, 16, \ldots$), the search for the k nearest neighbors actually reduces to a deterministic top-down traversal down a decision tree, where this tree is a compression in the form of a trie. Classification in a decision tree of that type has a complexity of $O(f)$, i.e. in the order of the number of features (the matching of feature values over branching tests can be done in constant time with a hash function at each node). In this order of complexity, the factor n (the number of training examples) has disappeared, making classification speed essentially independent of the original size of the training set.

In reality, feature weights tend to have a variance that is not as extreme as we just sketched. Then again the variance can be large enough to allow classification to be faster than $O(nf)$, and in fact approach $O(f)$. We enabled this in TIMBL by compressing the training set into a decision tree structure [7]. Training examples are stored in the tree as paths from a root node to a leaf; the arcs of the path are the consecutive feature-values, and the leaf node contains the class label for the original example (or a distribution of classes if the same path leads to more than one class label). From the top down, examples with identical feature values are collapsed into single paths, creating a compressed data structure. The distance computation for the nearest neighbor search can re-use partial results for paths which share prefixes (i.e. matches on the most important features). When search has traversed to a certain level of the tree, the amount of similarity that can still contribute to the overall distance can be computed, on the basis of which entire branches of the tree can be discarded as none of the examples stored there will ever be able to rise above the similarity of the least similar of the k nearest neighbors found so far. By doing this search depth-first, the similarity threshold gets set to high-threshold values quickly, so that large parts of the search space can be ignored early on.

2.2 MUMBL: Multiple TIMBLs

If m processors are available, k-NN classification can always be linearly scaled by simply running clones of the same classifier m times in parallel, and letting each of them classify $1/m$th of the examples to be classified. Individual classifications are performed at the same speed as in a single classifier, but the workload of classifying a test set is distributed. This is what the MUMBL wrapper does. More specifically, the wrapper performs the following procedure:

1. It splits the test set in m equally-sized parts;
2. It creates a tree-based compression of the training set once;

3. It spawns clones of the TIMBL classifier, each equipped with the same tree-based compression of the training set, and hands out a different $1/m$th part of the test set to each clone;

4. It monitors progress, and concatenates the predictions of all clones when they are all ready.

In a multi-processor machine with equally powerful CPUs, all clones will be finished at approximately the same moment. One would expect that the whole process takes about $1/m$th of the time it would take a single classifier, hence, linear scaling could be expected in the number of processors.

2.3 DIMBL: Distributed TIMBL

In contrast with MUMBL's strategy of splitting the test set, DIMBL splits the training set in m parts, and assigns these smaller parts to m TIMBL classifiers. With m processors, a training set with n labeled examples, and a test set, the DIMBL wrapper works its way through the following procedure:

1. It computes the global feature weights of the training set;

2. It splits the training set in m parts;

3. It spawns m TIMBL classifiers, giving them each $1/m$th of the training set (i.e., with n/m examples) and the globally computed feature weights;

4. Each of the sub-classifiers creates a tree-based compressed version of its $1/m$th training set, and activates itself on a processor;

5. Case by case, the wrapper sends all test examples to all of the spawned TIMBL classifiers, and retrieves all of their nearest neighbor sets; it merges these sets into a single set, from which a majority class label is extracted.

The DIMBL wrapper thus has two distinct functions: first, it is the master process which spawns sub-classifiers and talks to these sub-classifiers continuously (it sends test examples, and retrieves nearest-neighbor sets). Second, it performs the final step of the k-NN classifier, by finding the class with the highest vote in the global nearest neighbor set, which it merges on the basis of the m sets submitted by the sub-classifiers. The merging operation is computationally negligible compared to the m nearest neighbor retrieval operations; it consists of (1) merging the nearest neighbor sets, (2) selecting the k nearest distances from the merge, as different sub-classifiers may have found nearest neighbors at different distances, and (3) performing the normal counting operation that determines the most prominent class in the resulting k nearest neighbor set.

DIMBL performs a heavier computational task than the MUMBL wrapper. On the other hand, as each sub-classifier is trained on $1/m$th of the training set, they can be expected to be m times as fast as a classifier trained on all data, hence it is to be expected that linear scaling, or near-linear scaling, may be achieved with DIMBL.

2.4 Related research

We briefly review earlier approaches to fast retrieval of k nearest neigbors, and to parallelization of k-NN classification.

Perhaps the best known fast k-NN retrieval method is kd-trees [3]. Our trie-based approach can be seen as a special variant of kd-trees. One key difference between our approach and default implementations of kd-trees is that the latter do not store class label information, but rather pointers to examples at their nodes. Our approach behaves more like a regular decision-tree classifier, retrieving class label counts immediately from the tree, instead of postponing that to a separate phase. Another deviation from standard kd-trees is that we drive our compression by information-theoretic or statistical feature weighting heuristics, also in the same way that a decision-tree learner operates.

Our approach to retrieving nearest neighbors can also be likened to k-AESA (Approximation and Eliminating Search Algorithm) [19, 10], a depth-first search algorithm that tries to limit the search space efficiently by trying to set the least similar k-nearest distance to a competitive level as quickly as possible. Where k-AESA is defined for metric spaces, our approach applies to symbolic spaces.

Most work on parallel processing for machine learning [20, 4, 12] has focused on running full classifiers in parallel, e.g. in voting ensemble architectures. Most of this work used to be largely driven by the availability of single-machine, shared-memory massive parallel hardware. Recently, MPI (message passing interface) computing and Grid computing in clusters have become more popular and more easily available than single-machine parallel hardware. There is some recent work on using MPI and Grid computing for parallelizing k-NN classification. [11] describe an MPI approach to parallel k-NN classification across a cluster of computers, where the test set is split (like in MUMBL), attaining near-linear speedup figures. [2] describe a complex layered architecture in which they run a large-scale cross-validation experiment with k-NN classifiers to determine an optimal value of k on a large dataset. The architecture uses a combination of threaded parallelization on single machines, MPI computing on clusters, and Grid computing for global job allocation. No clear comparison is provided to enable concluding that they too arrive at near-linear speedup.

The DIMBL idea of distributing the training set over classifiers communicating in a multi-processor shared memory setup does not appear to be as widespread as the MUMBL approach. Yet, a system similar to DIMBL is described by [9], who report more or less linear gains of k-NN on a large numeric three-dimensional classification task, and diminishing gains beyond eight parallel classifiers due to I/O processing costs. As our focus is on non-numerical data that allows compressed storage, our findings turn out to be different from those of [9], as discussed and analyzed in the next sections.

3 Experimental setup

In this study we measure the real (elapsed wall clock) time it takes to classify a test set, on a range of tasks for which we have a single training-test set split, and derive classification speeds (numbers of test instances classified per second) from these measurements. Accuracies are not reported, as we are comparing alternate implementations of functionally the same k-NN classifier, producing the exact same classifications. We perform these experiments on a single machine with eight CPUs[1], and increase the number of parallel TIMBLs from 1 to 7 with each task, with both MUMBL and DIMBL. Not all eight processors are used, to avoid a process collision of the MUMBL and DIMBL wrappers with their spawned subprocesses on one of the processors.

We have run our experiments on five classification tasks from the natural language processing (NLP) domain, as this particular domain has a rich offering of large data sets with large numbers of examples, features, feature values, and numbers of classes: Running experiments with these datasets and k-NN classification quickly reveals the limitations of $O(nf)$. The tasks are the following:

Syntactic chunking (CHUNK), the identification of non-recursive syntactical phrases (e.g. verb phrases, noun phrases) in English text. Each example represents a word in its context of three words to the left and to the right, with their part-of-speech tags; the class encodes the bracketing and labeling of the syntactic phrases. The particular data set stems from the CoNLL-2000 shared task [15].

Named-entity recognition (NER), the identification of named entities (e.g. names of people, locations, and organisations) in English text. Each example represents a word in its context, as with the CHUNK task; the class label encodes the bracketing and labeling of named entities. The data set stems from the CoNLL-2002 shared task [14].

Confusible disambiguation (CONF), disambiguating between the two Dutch words *gebeurd* (happened) and *gebeurt* (happens or happening), on the basis of the context of five words to the left and to the right [17].

Translation (TRANS), sentential translation from Dutch to English, based on the Europarl aligned corpus of translated European parliament sessions. One example represents one Dutch word in context (three words to the left and right) translated to an aligned trigram of English words [18].

Word prediction (PREDICT), the prediction of words in a context of seven words to the left, and seven to the right, in English text [17].

Table 1 lists numbers of training and test examples, numbers of features and feature value ranges, and the number and entropy of classes of the five tasks. CHUNK and NER have large numbers of examples; NER

[1]The hardware used for testing has four Dual Core AMD Opteron 880 2,412 Mhz processors, with 32 Gb of RAM and a SCSI RAID-controlled disk subsystem. The involved C++ code was compiled with gcc version 3.4.4. The machine runs Linux kernel 2.6.14.

Table 1: Specifications of the data sets used for the five NLP tasks.

Task	Number of training ex.	Number of test ex.	Number of features	Range of # of feature values	Number of classes	Class entropy
CHUNK	211,727	43,377	14	44 – 2,669	22	2.65
NER	203,621	46,435	14	45 – 23,623	8	0.98
CONF	80,000	19,833	10	1,220 – 13,883	2	0.92
TRANS	20,000	27,805	7	2,746 – 3,305	2,887	8.93
PREDICT	20,000	20,000	14	3,748	3,748	9.52

Table 2: Settings for k, feature weight, value similarity function, and distance function found through wrapped progressive sampling.

Task	k	Weight	Similarity	Distance
CHUNK	15	GR	MVDM	IL
NER	3	GR	JD	ED1
CONF	15	GR	MVDM	ID
TRANS	35	IG	JD	ED1
PREDICT	25	GR	JD	ID

and CONF have large numbers of feature values, and TRANS and PREDICT have many classes. The latter two datasets have been kept artificially small, to allow for the experiments to be managable in time.

For each task we performed an automatic heuristic search for an optimal set of algorithmic parameters using wrapped progressive sampling [16]. This procedure estimates an appropriate value for k, the type of feature weighting, the value similarity metric, and the distance function [16]. Table 2 displays the settings found for the five tasks using this procedure. The value of k estimated to be optimal varies from 3 to 35. As for the other three settings, the differences are less substantial. Gain-ratio (GR) and information-gain (IG) weighting are usually correlated [13], as are the modified value difference metric (MVDM) and Jeffrey divergence value difference functions, and the three distance functions [7].

4 Results

We measured the number of classifications of test instances per second, and made these measurements relative to having a single TIMBL classifier. The relative speedups with 2 to 7 parallel TIMBLs in both the MUMBL and DIMBL wrappers are visualized in Figure 1. As the left part of the figure shows, MUMBL exhibits a faithful linear speedup, with slight signs of network overhead taking its toll with increasing parallelism.

At the right hand side of Figure 1 the relative speedups of DIMBL are displayed, showing rather different results. Three of the five tasks display sublinear speedups, while on the other hand two tasks are performed with superlinear speedups. The three tasks with sublinear speedups are CHUNK, NER, and CONF. Especially the CHUNK task has a remarkably low speedup; with 7 parallel TIMBLs, classification is only a factor of 1.2 times faster than a single TIMBL. The relative speedups of the other two tasks with 7 parallel classifiers are only 3.5 for NER, and 3.7 for CONF. The two tasks exhibiting remarkably superlinear speedups are TRANS (12.6 times faster classification with 7 parallel TIMBLs), and PREDICT, with a relative speedup factor of 16.7 with 7 parallel TIMBLs; more than twice the expected speed gain. Since we expected linear scaling for both wrappers, the deviating results of DIMBL results call for explanations.

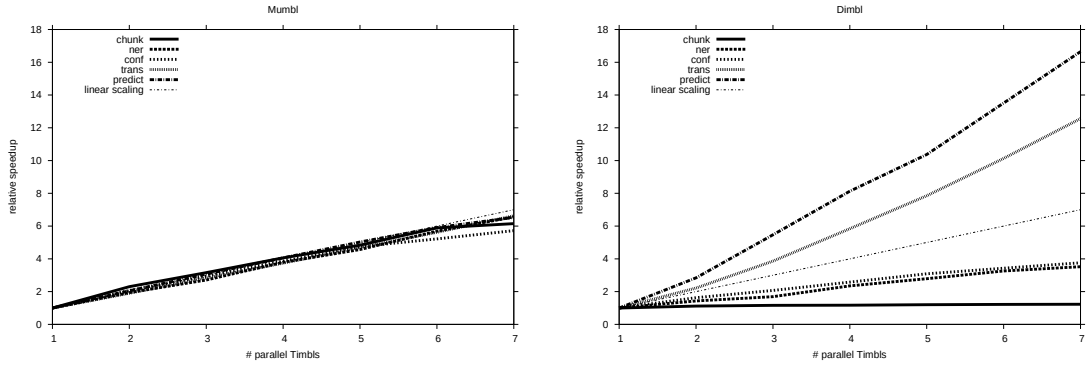

Figure 1: Relative speedup in number of classifications per second of MUMBL (left) and DIMBL (right) on the five NLP tasks, with increasing numbers of parallel TIMBLs.

Table 3: The standard deviation of feature weights (where the mean is normalized to 1.0) and the relative speed gain of DIMBL with seven sub-classifiers, for each of the five tasks. Superlinear gains (larger than 7) are displayed in bold.

Task	Std. deviation of weights	Relative speed gain
CHUNK	0.91	1.2
NER	0.80	3.5
CONF	0.47	3.8
TRANS	0.26	**12.6**
PREDICT	0.03	**16.7**

5 Discussion

As detailed earlier in Section 2, TIMBL's implementation of k-NN includes a heuristic search shortcut in the k-NN searching procedure that can safely ignore vast subareas of the memory (compressed in a decision-tree structure), provided that there is a clear difference between features with high weights and with low weights. In other words, when there is a subset of important features, TIMBL can focus on looking among nearest neighbor candidates that have the same values on these features as the instance to be classified, and ignore the rest, while still returning the exact k nearest neighbors that a naive implementation (with complexity $O(nf)$) would produce.

A sensible estimation of the spread in feature weights is their standard deviation. Table 3 lists the standard deviation of each of the five tasks' feature weights, with the mean normalized to 1.0. The table also lists the relative speedup attained on each task by DIMBL with seven parallel TIMBLs in the third column ("weights"). Although it should be noted that the list constitutes only a limited number of measurements, the Pearson correlation coefficient is strongly negative, at $r = -0.939$ ($t = -4.7, p < 0.05$), suggesting an inverse relation between a high variance in feature weights, and the effect of DIMBL.

All reported results were obtained with single fixed-size training–test splits (cf. Table /refspecs). Both extreme outcomes, the near-zero gain of parallelization with CHUNK, and the superlinear gain with PREDICT, raise the question whether these increases are accidental for the particular splits investigated. Figure 5 displays the speed curves of DIMBL and TIMBL on both tasks, where one speed curve plots the number of seconds it takes to classify a fixed test set of 20 thousand instances (in each of the tasks), and where DIMBL works with seven processors. With CHUNK, classification times increase very modestly, if at all; from five thousand training examples onwards, it takes nearly constant time to classify the same 20 thousand test examples. In other words, classification seems to be almost independent of n; the decision tree optimum of $O(f)$ appears to be approached due to the large differences in feature weights. This explains the relative lack of effect of parallelisation; reducing the size of the training set from m to $1/m$ actually hardly causes a speed gain.

In contrast, the speed curve of TIMBL on the PREDICT task shows a sharp exponential upward trend. It

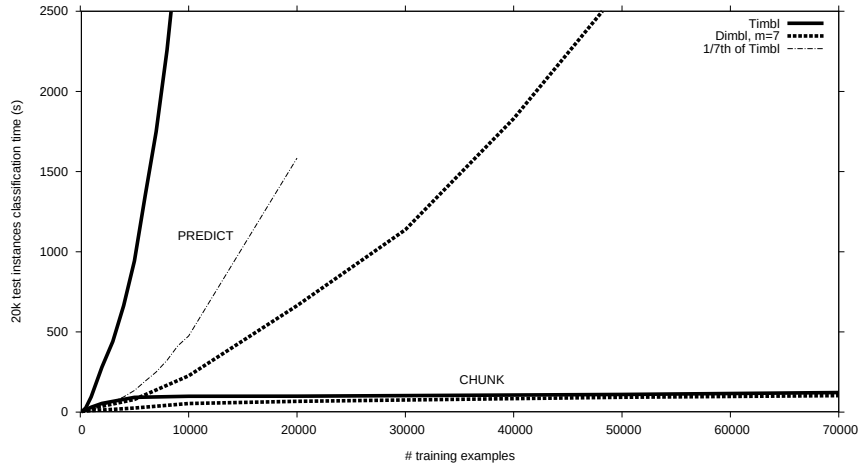

Figure 2: Speed curves of DIMBL and TIMBL on PREDICT and CHUNK, in terms of the number of seconds taken in classifying 20 thousand examples of both tasks, using 7 processors for DIMBL. For reference, the dashed-dotted line represents $1/7$th of TIMBL's speed on the PREDICT task.

becomes very inefficient to classify test examples even when trained on a few thousand examples; a doubling of the amount of training data leads to far more than double the amount of classification time. The speed curve of DIMBL shows essentially the same upward trend, but at a considerably slower rate. For comparison, Figure 5 displays a reference line (dashed-dotted) representing $1/7$th of the speed of TIMBL on PREDICT; it is clear that DIMBL operates faster than that, and the advantage (the vertical distance between the curves) also increases with more training examples. Thus, with m training examples, DIMBL can be beyond m times faster than TIMBL. On the other hand, although it postpones the exponential effect, also DIMBL falls prey to it in the end.

6 Conclusions

It has been observed earlier that when parallelizing the k-NN classifier, linear scaling in the number of parallel threads can be attained by splitting the test set in m parts, and simultaneously running m clones of the k-NN classifier with the full training set in memory. We tested this relatively trivial procedure in the form of the MUMBL wrapper, and indeed attained linear scaling on five NLP tasks, comparing to a single TIMBL classifier.

As an alternative, we proposed the DIMBL wrapper, which splits the training set in m parts, instead of the test set. The wrapper spawns m TIMBL classifiers, each trained on $1/m$th part of the original training set. DIMBL classifies a new instance by requesting the nearest neighbor sets of that instance from each of the spawned classifiers, and merges the nearest neighbor sets to one, from which the final classification is derived.

Although DIMBL is expected to yield linear scaling, as shown earlier in a similar approach to k-NN classification in numeric feature spaces [9], it turns out to produce both sublinear and superlinear scaling with different data sets. The sublinear scaling can be attributed to the fact that the base classifier, TIMBL, already has a strong search heuristic implemented, based on feature weights and related to kd-trees and k-AESA, that neutralizes the effect of parallelization.

Superlinear scaling with k-NN classification, on the other hand, turns out to be possible with datasets that have features with little variance in their weights. We attribute the superlinear scaling to the fact that the base classifier TIMBL becomes exponentially slower when trained on linear increases of examples. Hence, lowering the amount of training data from m to $1/m$ cases per sub-classifier, speeds up each classifier more than m times. The two best observed speedup factors were 12.6 on a translation task, and 16.7 on a word prediction task, both performed with only 7 parallel classifiers.

In future work we intend to perform large-scale experiments with more data and more processors in supercomputers, to affirm empirically our suspicion that the superlinear scaling of DIMBL can in fact be higher with larger training sets than the current highest speedups, while there must also be a negative effect

of the merge operation with increasing numbers of sub-classifiers.

Acknowledgments

This research was funded by NWO, the Netherlands Organisation for Scientific Research, as part of the *Implicit Linguistics* NWO Vici project. We thank Walter Daelemans, Gert Durieux, and Jakub Zavrel for their key contributions to TIMBL.

References

[1] D. W. Aha, D. Kibler, and M. Albert. Instance-based learning algorithms. *Machine Learning*, 6:37–66, 1991.

[2] G. Aparício, I. Blanquer, and V. Hernández. A parallel implementation of the k nearest neighbours classifier in three levels: Threads, MPI processes and the Grid. In *Proceedings of the 7th Meeting of the International Meeting on High Performance Computing for Computational Science*, Porto, Portugal, 2006.

[3] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1979.

[4] P. K. Chan and S. J. Stolfo. A comparative evaluation of voting and meta-learning of partitioned data. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 90–98, 1995.

[5] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *Institute of Electrical and Electronics Engineers Transactions on Information Theory*, 13:21–27, 1967.

[6] W. Daelemans and A. Van den Bosch. Generalisation performance of backpropagation learning on a syllabification task. In M. F. J. Drossaers and A. Nijholt, editors, *Proceedings of TWLT3: Connectionism and Natural Language Processing*, pages 27–37, Enschede, 1992. Twente University.

[7] W. Daelemans, J. Zavrel, K. Van der Sloot, and A. Van den Bosch. TiMBL: Tilburg Memory Based Learner, version 6.0, reference guide. Technical Report ILK 07-03, ILK Research Group, Tilburg University, 2007.

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

[9] R. Jin and G. Agrawal. A middleware for developing parallel data mining applications. In *Proceedings of the First SIAM Conference on Data Mining*, Chicago, IL, 2001.

[10] A. Juan, E. Vidal, and P. Aibar. Fast k-nearest-neighbours searching through extended versions of the approximating and eliminating search algorithm (aesa). In *Proceedings of the 14th International Conference on Pattern Recognition (ICPR-98)*, volume 1, pages 828–830, Brisbane, Australia, 1998. IEEE Computer Society Press.

[11] E. Plaku and L. Kavraki. Distributed computation of the k-NN graph for large high-dimensional point sets. *Journal of Parallel and Distributed Computing*, 67(3):346–359, 2007.

[12] F. J. Provost and J. M. Aronis. Scaling up inductive learning with massive parallelism. *Machine Learning*, 23(1):33, 1996.

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

[14] E. Tjong Kim Sang. Introduction to the CoNLL-2002 shared task: Language-independent named entity recognition. In *Proceedings of CoNLL-2002*, pages 155–158. Taipei, Taiwan, 2002.

[15] E. Tjong Kim Sang and S. Buchholz. Introduction to the CoNLL-2000 shared task: Chunking. In *Proceedings of CoNLL-2000 and LLL-2000*, pages 127–132, 2000.

[16] A. Van den Bosch. Wrapped progressive sampling search for optimizing learning algorithm parameters. In R. Verbrugge, N. Taatgen, and L. Schomaker, editors, *Proceedings of the Sixteenth Belgian-Dutch Conference on Artificial Intelligence*, pages 219–226, Groningen, The Netherlands, 2004.

[17] A. Van den Bosch. Scalable classification-based word prediction and confusible correction. *Traitement Automatique des Langues*, 46(2):39–63, 2006.

[18] A. Van den Bosch, N. Stroppa, and A. Way. A memory-based classification approach to marker-based EBMT. In F. Van Eynde, V. Vandeghinste, and I. Schuurman, editors, *Proceedings of the METIS-II Workshop on New Approaches to Machine Translation*, pages 63–72, Leuven, Belgium, 2007.

[19] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.

[20] D.L. Waltz. Massively parallel AI. *International Journal of High Speed Computing*, 5(3):491–501, 1995.