Superlinear parallelisation of the k-nearest neighbor classifier

Antal van den Bosch ANTALB@UVT.NL
Ko van der Sloot SLOOT@UVT.NL
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 can 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-based data structure for retrieving nearest neigbors. In a range of experiments the first method consistently scales linearly. With the second method we observe cases of both superlinear and sublinear scaling. A high variance in feature weights dampens the effect of the second type of parallelisation, due to the strong weight-based search heuristics already built into TIMBL. When feature weights exhibit less variance, superlinear scaling can occur, due to relatively faster nearest neighbor retrieval by sub-classifiers on $1/m$th training sets as compared to retrieval by a classifier trained on the full training set.

1. Introduction

The k-nearest neighbor (k-NN) classifier (Fix & Hodges, 1951; Cover & Hart, 1967; Dasarathy, 1991; Aha et al., 1991) 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. This so-called lazy learning approach (no energy is spent on creating an abstract model) has a tremendous drawback, which is that classification tends to be slow. Since every classification takes 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.

This paper introduces two parallelisation approaches to k-NN aimed at improving classification speed. The two approaches can run on multiple-processor shared-memory architectures, and build on TIMBL (Daelemans et al., 2004), an implementation of IB1 (Aha et al., 1991) with feature weighting and other similarity function enhancements. The first parallelisation, MUMBL (*multiple* TIMBL*s*), makes trivial use of the extra power provided by m processors over just one, by cloning one classifier into m classifiers, and running a master process that hands out $1/m$th of the test set to each clone. MUMBL exhibits near-linear scaling, only hindered by some processing overhead of the master classifier.

The second parallelisation, 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. If the latter step is not too costly, classification speedup can be expected to be approximately linear, because the m parallel subclassifiers are all faster than a single classifier trained on the full training set.

We describe both methods in Section 2. In Section 3

Proceedings of the 18th Benelearn
P. Adriaans, M. van Someren, S. Katrenko (eds.)

we describe our experimental setup, and in Section 4 we compare TIMBL to its two parallel versions. We observe near-linear speedups with MUMBL, and we observe some remarkable superlinear speedups with DIMBL. However, we also observe a case in which DIMBL is hardly faster than a single classifier. In Section 5 we discuss these results and come up with an explanation on the different speedup results of DIMBL.

2. Two parallelisations of k-NN

We present two parallelisations of a particular implementation of the classic k-NN classifier, TIMBL (Daelemans et al., 2004). We first describe this implementation, since it is already faster than a naive implementation of k-NN due to the implementation of search heuristics, and improvements through parallelisation cannot be seen but relative to it. We then introduce MUMBL and DIMBL.

2.1. TIMBL: Trie-based k-NN

Finding the k nearest neighbors of a new example of which the classification is sought, implies in the simplest sense a case-by-case comparison of the new example against all n memorized examples. A single comparison implies the computation of a similarity function. The major factor in this function is the number of features, f. Hence k-NN classification has a worst-case complexity of $O(nf)$.

In TIMBL (Daelemans et al., 2004), the similarity function is implemented as a kernel capable of accomodating several distance metrics in various spaces, weighted by information-theoretic, statistical, and 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 (Daelemans & Van den Bosch, 1992), 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:

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

Proper feature weighting 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 there 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 (Knuth, 1973; Daelemans et al., 1997). Classification in a decision tree of that type has a complexity of $O(f\ lg(v))$ (Van den Bosch, 1997), where v is the average branching factor (how many arcs fan out of nodes in the tree, on average). In this order of complexity, the factor n (the number of training examples) has disappeared, making classification more independent of the original size of the training set (note that v will still tend to grow with a higher n in many real datasets).

In reality, feature weights tend to have a variance that is not as extreme as we just sketched, but the variance can be large enough to allow classification to be faster than $O(nf)$, and in fact approach $O(f\ lg(v))$. We enabled this in TIMBL by compressing the training set into a decision tree structure (Daelemans et al., 2004). 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. 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 which will never be able to rise above the partial similarity of the current least similar nearest neighbor. 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.

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 pieces;
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 portion of the test set to each clone;
4. It monitors progress, and gathers 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 $1/m$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

Instead of 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. Computationally speaking, but also in terms of computer i/o, it performs a heavier 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 near-linear scaling may be achieved with DIMBL.

2.4. Related research

Despite the fact that there is some work from the last decade on using parallel processing for machine learning (Waltz, 1995; Chan & Stolfo, 1995; Provost & Aronis, 1996), work in this area has largely focused on running full classifiers in parallel, e.g. in voting ensemble architectures, and in MUMBL. Also, this work was 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. Plaku and Kavraki (2007) 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. Aparício et al. (2006) 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 parallelisation 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.

In general it can be concluded that most approaches to parallel k-NN classification have attained near-linear speedups in the number of available CPUs, simply by running many individual k-NN classifiers in parallel. The DIMBL approach appears to be newer, and can only be likened to the work by Provost and Aronis (1996) in the sense that they too distribute training material over several classifiers – but not with k-NN.

3. Experimental setup

In this study we measure the real 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

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

Task	Number of training examples	Number of test examples	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

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, despite being a suited method for NLP (Daelemans & Van den Bosch, 2005), 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 one word in its context of neighboring words (a window of three words to the left and to the right, with their part-of-speech tags), and the class encodes the bracketing and labeling of the syntactic phrases. The particular data set stems from the CoNLL-2000 shared task (Tjong Kim Sang & Buchholz, 2000).

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 (most words are not part of a named entity, evidently). The data set stems from the CoNLL-2002 shared task (Tjong Kim Sang, 2002).

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 (Van den Bosch, 2006).

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 (Van den Bosch et al., 2007).

Word prediction (predict), the prediction of words in a context of seven words to the left, and seven to the right, in English text (Van den Bosch, 2006).

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 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 (Van den Bosch, 2004). This procedure estimates a proper value for k, the type of feature weighting, the value similarity metric, and the distance function (Van den Bosch, 2004). Table 2 displays the settings found for the five tasks using this procedure. The value of k estimated to be optimal varies substantially from 3 to 35. As for the other three settings, the differences are less substantial. Gain-ratio (GR) and information-gain (IG)

[1]The hardware used for testing has four Dual Core AMD Opteron 880 2,412 Mhz processors, with 18 Gb of RAM and a SCSI disk subsystem. The machine has 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.

weighting are usually correlated (Quinlan, 1993), as are the modified value difference metric (MVDM) and Jeffrey divergence value difference functions, and the three distance functions (Daelemans et al., 2004).

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

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 more 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. Apparently, splitting the training set in m pieces and assigning TIMBL classifiers to each $1/m$th part, can lead both to strong and weak speed increases compared to a single classifier. Since we expected linear scaling for both wrappers, the deviating results of DIMBL results call for explanations.

5. Discussion

To explain the widely differing results in the scaling of DIMBL we analyse three factors that may explain these deviations: the k parameter, the feature-weighting-based search heuristic already built into TIMBL, and the branching factor of the TIMBL trees.

5.1. The role of the k parameter

Despite the fact that k is neither in $O(nf)$ nor in $O(f\ lg(v))$, the worst-case complexities of standard k-NN classification and classification by deterministic decision-tree traversal, respectively, the k parameter does play a role in the bookkeeping that the k-NN classifier has to perform while gathering the k nearest neighbors from memory, and so it might play some role in running time. Hypothetically, lowering the number of instances (as DIMBL does, by partitioning the training set in m same-sized parts) might alleviate the bookkeeping effort considerably, particularly with high values of k.

From Table 2 we observe that the automatically estimated optimal settings of the k parameter in the k-NN classifier are particularly high for the two tasks that show superlinear scaling, TRANS and PREDICT. The values of $k = 25$ and $k = 35$ suggest that these tasks are best solved by voting over a large set of nearest neighbors. If our hypothesis about bookkeeping costs is right, lowering k might make DIMBL's relative speedup smaller. We performed two comparative experiments with TRANS and PREDICT, setting $k = 1$ (leaving other settings unchanged, except distance weighting, which becomes void with $k = 1$), running the same experiment with a single TIMBL classifier, and DIMBL with seven sub-classifiers. Table 3 compares the original relative speed gains with the new gains.

Table 3. Relative speed gains of DIMBL with seven parallel classifiers on TRANS and PREDICT, with the original parameter settings, and with $k = 1$.

TASK	OPTIMAL VALUE OF k	RELATIVE SPEED GAIN OPTIMAL k	$k = 1$
TRANS	35	12.56	11.34
PREDICT	25	16.68	16.38

As the numbers in Table 3 indicate, reducing k to the lowest possible value hardly changes the relative speed gain attained by DIMBL. We conclude that superlinear scaling is not influenced by the value of k.

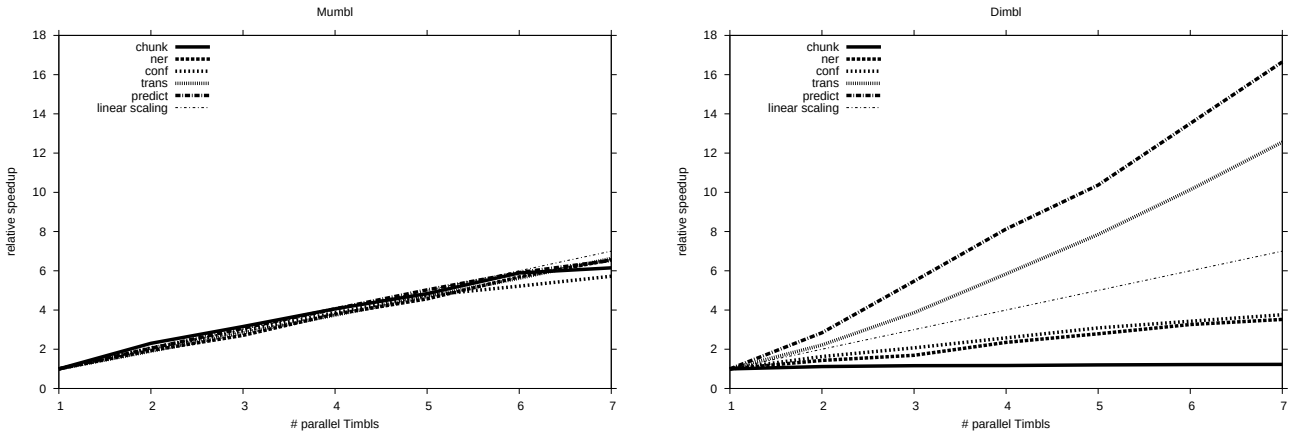

Figure 1. Relative speedup in number of classifications per second of MUMBL (left) and DIMBL (right) on the five NLP tasks, with increasing numbers of parallel TIMBLs. The y axes of the two figures have different scales.

5.2. The role of the feature-weighting heuristic

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 down to 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 4 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. In other words, the feature-weight heuristic appears to overpower the effect of parallelisation, up to the point, with the CHUNK task, that the parallelisation is costing almost as much overhead as it contributes positively. So, the role of the variance in feature weights might well explain the sublinear behavior of DIMBL on CHUNK, NER, and CONF.

On the other hand, it does not explain the occasional superlinear scaling of DIMBL.

Table 4. 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**

5.3. The role of the branching factor in TIMBL trees

The superlinear scaling of DIMBL must be rooted in the implementation of TIMBL; any naive k-NN implementation would have brought only a linear increase in speed. As described earlier, TIMBL looks for nearest neighbors by traversing a decision-tree structure, which is a compression of the original training examples. We observed earlier that with high variance in feature weights, search in the tree overpowers the effect of parallelisation. With low variance, however, the tree structure appears to have a positive effect: classifiers trained on $1/m$th of the original training set are more than m times faster than a single classifier trained on the full training set.

We hypothesize that the root cause is the compression achieved by the TIMBL trees. A straightforward

case-by-case, feature-by-feature comparison of test instances to a $1/m$th training set would simply be m times faster than a comparison to the full training set, but in case of the TIMBL trees, access is not case-by-case. Instead, at the first level of the tree, many training examples are bundled in a lower number of tree nodes (namely, all uniquely occurring values of the most important feature); only at deeper levels of the tree do paths start to represent single training examples. This allows for a faster retrieval of k nearest neighbor sets than a naive implementation of the k-NN classifier would be capable of.

This does not yet explain the superlinear scaling compared to a single TIMBL, which uses the same compressed tree structure. Trees built out of $1/m$th parts of a training set can be searched through more than m times faster than a tree built on the full training set. We hypothesize that this is due to the v factor from $O(f\ lg(v))$, which is the complexity of a single traversal through the tree, from top to bottom. When in a node at any depth, looking at the various arcs that fan out of this node (v), it will take $lg(v)$ time to find a match when matching is done optimally, e.g. through a hash function. Inspecting our experimental results, it turns out that the branching factor in the typical TIMBL tree is highly skewed; at the first level, the root node spawns as many arcs as the most important feature has values, but already at the second level the amount of branching quickly reduces to a handful of arcs, down to single arcs (i.e. path tails that represent individual training examples). For example, the tree constructed on 1/7th of the PREDICT data has 887 arcs fanning out of the root node, while at the second level only 2.2 arcs on average fan out of the 887 nodes; at the third level, this average is down to 1.1.

In other words, it is largely the amount of branching from the root node (i.e. the number of values of the feature with the highest weight) that determines the number of times the TIMBL classifier has to match a test feature value to values stored on arcs in the tree. In the case of the PREDICT task, the most important feature (the word left of the word to be predicted) has 3,478 values in the full training set; in the 1/7th subset the same feature has 887 values, a relative decrease of 74%. In the case of the TRANS task, this relative decrease is also 74%.

Although the relative superlinear speed increase of DIMBL on the two tasks deviate more than these two 74% decreases may suggest (79% on TRANS, 137% on PREDICT), it is quite obvious that the individual TIMBL subclassifiers in a DIMBL experiment can profit substantially from the fact that the branching in their smaller trees allows for faster retrieval of nearest neighbors than a TIMBL classifier trained on the full training set would be capable of. In other words, smaller trees allow for faster retrieval. This is why DIMBL gets more than a speedup of factor m when distributing the training set over m parallel sub-classifiers.

6. Conclusions

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 simultaenously 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 asking for 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 was expected to yield linear scaling, it turned out to produce both sublinear and superlinear scaling with different data sets. We attribute the sublinear scaling to the fact that the base classifier, TIMBL, already has a strong search heuristic implemented, based on feature weights, that overpowers the effect of parallelisation.

Superlinear scaling with k-NN classification turns out to be possible with datasets that have features with little variance in their weights (where TIMBL's strong search heuristic has little influence). We attribute the superlinear scaling to the fact that the base classifier TIMBL compresses the training set into a decision-tree structure, and smaller trees (i.e. those derived from $1/m$th parts of the training set) allow for faster retrieval than factor m. 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 broaden our experiments to MPI and Grid clusters. Besides being able to work with much larger numbers of processors, expanding to clusters will introduce more networking and i/o overhead, which we have attempted to keep to a minimum in the current study. More detailed profiling and complexity analyses will be necessary to get a more de-

tailed understanding of the causes for speed improvements and delays.

References

Aha, D. W., Kibler, D., & Albert, M. (1991). Instance-based learning algorithms. *Machine Learning*, *6*, 37–66.

Aparício, G., Blanquer, I., & Hernández, V. (2006). A parallel implementation of the k nearest neighbours classifier in three levels: Threads, mpi processes and the grid. *Proceedings of the 7th Meeting of the International Meeting on High Performance Computing for Computational Science*. Porto, Portugal.

Chan, P. K., & Stolfo, S. J. (1995). A comparative evaluation of voting and meta-learning of partitioned data. *Proceedings of the Twelfth International Conference on Machine Learning* (pp. 90–98).

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

Daelemans, W., & Van den Bosch, A. (1992). Generalisation performance of backpropagation learning on a syllabification task. *Proceedings of TWLT3: Connectionism and Natural Language Processing* (pp. 27–37). Enschede.

Daelemans, W., & Van den Bosch, A. (2005). *Memory-based language processing*. Cambridge, UK: Cambridge University Press.

Daelemans, W., Van den Bosch, A., & Weijters, A. (1997). IGTree: using trees for compression and classification in lazy learning algorithms. *Artificial Intelligence Review*, *11*, 407–423.

Daelemans, W., Zavrel, J., Van der Sloot, K., & Van den Bosch, A. (2004). *TiMBL: Tilburg Memory Based Learner, version 5.1.0, reference guide* (Technical Report ILK 04-02). ILK Research Group, Tilburg University.

Dasarathy, B. V. (1991). *Nearest neighbor (NN) norms: NN pattern classification techniques*. Los Alamitos, CA: IEEE Computer Society Press.

Fix, E., & Hodges, J. L. (1951). *Discriminatory analysis—nonparametric discrimination; consistency properties* (Technical Report Project 21-49-004, Report No. 4). USAF School of Aviation Medicine.

Knuth, D. E. (1973). *The art of computer programming*, vol. 3: Sorting and searching. Reading, MA: Addison-Wesley.

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

Provost, F. J., & Aronis, J. M. (1996). Scaling up inductive learning with massive parallelism. *Machine Learning*, *23*, 33.

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

Tjong Kim Sang, E. (2002). Introduction to the conll-2002 shared task: Language-independent named entity recognition. *Proceedings of CoNLL-2002* (pp. 155–158). Taipei, Taiwan.

Tjong Kim Sang, E., & Buchholz, S. (2000). Introduction to the CoNLL-2000 shared task: Chunking. *Proceedings of CoNLL-2000 and LLL-2000* (pp. 127–132).

Van den Bosch, A. (1997). *Learning to pronounce written words: A study in inductive language learning*. Doctoral dissertation, Universiteit Maastricht.

Van den Bosch, A. (2004). Wrapped progressive sampling search for optimizing learning algorithm parameters. *Proceedings of the Sixteenth Belgian-Dutch Conference on Artificial Intelligence* (pp. 219–226). Groningen, The Netherlands.

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

Van den Bosch, A., Stroppa, N., & Way, A. (2007). A memory-based classification approach to marker-based ebmt. *Proceedings of the METIS-II Workshop on New Approaches to Machine Translation* (pp. 63–72). Leuven, Belgium.

Waltz, D. (1995). Massively parallel ai. *International Journal of High Speed Computing*, *5*, 491–501.