TiMBL: Tilburg Memory Based Learner
version 3.0
Reference Guide

ILK Technical Report – ILK 00-01

Walter Daelemans+ Jakub Zavrel+ Ko van der Sloot
Antal van den Bosch

Induction of Linguistic Knowledge
Computational Linguistics
Tilburg University

(+) CNTS - Language Technology Group
University of Antwerp

P.O. Box 90153, NL-5000 LE, Tilburg, The Netherlands
URL: http://ilk.kub.nl[1]

March 8, 2000

[1]This document is available from http://ilk.kub.nl/~ilk/papers/ilk0001.ps.gz.

Contents

Preface

Memory-Based Learning (MBL) has proven to be successful in a large number of tasks in Natural Language Processing (NLP). In our group at Tilburg University we have been working since the end of the 1980's on the development of Memory-Based Learning techniques and algorithms[1]. With the establishment of the ILK (Induction of Linguistic Knowledge) research group in 1997, the need for a well-coded and uniform tool for our main algorithms became more urgent. TiMBL is the result of combining ideas from a number of different MBL implementations, cleaning up the interface, and using a whole bag of tricks to make it more efficient. We think it can make a useful tool for NLP research, and, for that matter, for many other domains where classification tasks are learned from examples.

Memory-Based Learning is a direct descendant of the classical k-Nearest Neighbor (k-NN) approach to classification. In typical NLP learning tasks, however, the focus is on discrete data, very large numbers of examples, and many attributes of differing relevance. Moreover, classification speed is a critical issue in any realistic application of Memory-Based Learning. These constraints, which are different from those of traditional pattern recognition applications with their numerical features, often lead to different data-structures and different speedup optimizations for the algorithms. Our approach has resulted in an architecture which makes extensive use of indexes into the instance memory, rather than the typical flat file organization found in straightforward k-NN implementations. In some cases the internal organization of the memory results in algorithms which are quite different from k-NN, as is the case with IGTREE. We believe that our optimizations make TiMBL one of the fastest discrete k-NN implementations around.

The main effort in the development of this software was done by Ko van der Sloot. The code started as a rewrite of `nibl`, a piece of software developed by Peter Berck from a Common Lisp implementation by Walter Daelemans. Some of the index-optimizations are due to Jakub Zavrel. The code has benefited substantially from trial, error and scrutiny by the other members of the ILK group (Sabine Buchholz, Jorn Veenstra and Bertjan Busser) and CNTS group at

[1] Section 4.6 provides a historical overview of our work on the application of MBL in NLP.

the University of Antwerp (in particular Gert Durieux). We would like to thank Ton Weijters of Eindhoven Technical University and the other members of the CNTS group for their contributions. This software was written in the context of the "Induction of Linguistic Knowledge" research programme, partially supported by the Foundation for Language Speech and Logic (TSL), funded by the Netherlands Organization for Scientific Research (NWO). Last but not least, our thanks go to those users of TiMBL who have contributed to it immensely by giving us feedback and reporting bugs.

The current release (version 3.0) adds a whole number of new features to those available in the previous version (2.0) The most notable changes are:

- Server functionality. Instead of processing test items from a file, TiMBL can now also open a socket and process test items presented through it. This allows one to save the time needed for the initialization phase when small amounts of test material are repeatedly presented. It also opens the possibility of having large numbers of "classification agents" cooperate in real time.
- Leave-one-out testing. To get an estimate of the classification error, one can now test by "leave-one-out", in effect testing on every case once, while training on the rest of the cases, without completely re-initializing the classifier for every test case.
- Support for sparse binary features. For tasks with large numbers of sparse binary features, TiMBL now allows for an input format which lists only the "active" features, avoiding the listing of the many (zero-valued) features for each case.
- Additional feature weighting metrics. We have added chi-squared and shared variance measures as weighting schemes.
- The command line interface has slightly been cleaned up, and re-organized.

A more elaborate description of the changes from version 1.0 up to 3.0 can be found in Chapter 3. Although these new features have been tested for some time in our research group, the software may still contain bugs and inconsistencies in some places. We would appreciate it if you would send bug reports, ideas about enhancements of the software and the manual, and any other comments you might have, to `Timbl@kub.nl`.

This reference guide is structured as follows. In Chapter 1 you can find the terms of the license according to which you are allowed to use TiMBL. The subsequent chapter gives some instructions on how to install the TiMBL package on your computer. Chapter 3 lists the changes that have taken place up to the current version. Readers who are interested in the theoretical and technical details of Memory-Based Learning and of this implementation can proceed to Chapter 4. Those who just want to get started using TiMBL can skip this chapter, and directly proceed either to Chapters 5 and 7, which respectively provide a reference to the file formats and command line options of TiMBL, or to Appendix A, where a short hands-on tutorial is provided on the basis of a case study with a data set from a linguistic domain (prediction of Dutch diminutive suffixes). Chapter 8 gives a specification of the application programmer's interface (API) to the TiMBL library.

Chapter 1

License terms

Downloading and using the TiMBL software implies that you accept the following license terms:

Tilburg University and University of Antwerp (henceforth "Licensers") grant you, the registered user (henceforth "User") the non-exclusive license to download a single copy of the TiMBL program code and related documentation (henceforth jointly referred to as "Software") and to use the copy of the code and documentation solely in accordance with the following terms and conditions:

- The license is only valid when you register as a user. If you have obtained a copy without registration, you must immediately register by sending an e-mail to `Timbl@kub.nl`.
- User may only use the Software for educational or non-commercial research purposes.
- Users may make and use copies of the Software internally for their own use.
- Without executing an applicable commercial license with Licensers, no part of the code may be sold, offered for sale, or made accessible on a computer network external to your own or your organization's in any format; nor may commercial services utilizing the code be sold or offered for sale. No other licenses are granted or implied.
- Licensers have no obligation to support the Software it is providing under this license. To the extent permitted under the applicable law, Licensers are licensing the Software "AS IS", with no express or implied warranties of any kind, including, but not limited to, any implied warranties of merchantability or fitness for any particular purpose or warranties against infringement of any proprietary rights of a third party and will not be liable to you for any consequential, incidental, or special damages or for any claim by any third party.

Prof.dr. Walter Daelemans
CNTS - Language Technology Group
GER / University of Antwerp
Universiteitsplein 1
B-2610 Wilrijk (Antwerp)
Belgium

Chapter 2

Installation

You can get the TiMBL package as a gzipped tar archive from:

`http://ilk.kub.nl/software.html`

Following the links from that page, you will be required to fill in registration information and to accept the license agreement. You can proceed to download the file `Timbl.3.0.tar.gz` This file contains the complete source code (C++) for the TiMBL program, a few sample data sets, the license and the documentation. The installation should be relatively straightforward on most UNIX systems.

To install the package on your computer, unzip the downloaded file:

```
> gunzip Timbl.3.0.tar.gz
```

and unpack the tar archive:

```
> tar -xvf Timbl.3.0.tar
```

This will make a directory `Timbl.3.0` under your current directory. Change directory to this:

```
> cd Timbl.3.0
```

and compile the executable by typing `make`[1]. If the process was completed successfully, you should now have executable files named `Timbl, Client, tse, classify`, and a static library `libTimbl.a`.

The e-mail address for problems with the installation, bug reports, comments and questions is `Timbl@kub.nl`.

[1] We have tested this with `gcc` version 2.8.1, and 2.9.5.1

Chapter 3

Changes

This chapter gives a brief overview of the changes from version 1.0 to 2.0, and from version 2.0 to 3.0 for users already familiar with the program.

3.1 From version 2.0 to 3.0

- Server functionality. Instead of processing test items from a file, you can now specify a portnumber with `-S portnumber` to open a socket and send commands for classification of test patterns or change of parameters to it. A sample client program is included in the distribution. This allows fast response times when small amounts of test material are presented at various intervals. It also opens the possibility of having large numbers of "classification agents" cooperate in real time, or of classication of the same data with different parameters. The syntax of our simple Client/Server protocol is described in Chapter 6.

- Leave-one-out testing. To get an estimate of the classification error, without setting aside part of one's data as a test set, one can now test by "leave-one-out" (`-t leave_one_out`), in effect testing on every case once, while training on the rest of the cases, without completely re-initializing the classifier for every test case.

- Support for sparse binary features. For tasks with large numbers of sparse binary features, TiMBL now allows for an input format which lists only the "active" features, avoiding the listing of the many (zero-valued) features for each case. This format is described in section 5.1.5.

- Additional feature weighting metrics. We have added chi-squared and shared variance measures as weighting schemes. These weighting metrics are sometimes more robust to large numbers of feature values and other forms of data sparseness.

- Different metrics (Overlap, MVDM or Numeric) can be applied to different features.
- The command line interface has slightly been cleaned up, and re-organized:
  - The `-m metricnumber` switch to choose metrics has been replaced by the use of a specification string following `-m`. E.g. you can specify to use MVDM as the default metric, but use Overlap on features 5-7,9, Numeric on feature 1, and ignore feature 10 (`-m M:O5-7,9:N1:I10`).
  - All of the output needed for analysing the matching of nearest neighbors has been moved to the verbosity setting.
  - Verbosity levels and some other options can be switched on `+v` and off `-v`, even between different classification actions.
  - Because of the large amount of verbosity levels, the `+v` option takes mnemonic abbreviations as arguments instead of numeric verbosity levels. Although the old (numeric) format is still supported, it's use is not encouraged as it will disappear in future versions.
- Because of significant optimizations in the nearest neighbor search, the default is no longer to use inverted indexes. These can however still be turned on by using the `+-` switch on the command line.
- You can now choose the output filename or have it generated by TiMBL on the basis of the test filename and the parameters.
- You can use TiMBL in a pipeline of commands by specifying '-' as either input, output or both.
- Several problems with the display of nearest neighbors in the output have been fixed.
- The API has been adapted a bit to allow more practical use of it.

3.2 From version 1.0 to 2.0

- We have added a new algorithm: TRIBL, a hybrid between the fast IGTREE algorithm and real nearest neighbor search (for more details, see 4.5, or [15]). This algorithm is invoked with the `-a 2` switch and requires the specification of a so-called TRIBL-offset, the feature where IGTREE stops and case bases are stored under the leaves of the constructed tree.
- Support for numeric features. Although the package has retained its focus on discrete features, it can now also process numeric features, scale them, and compute feature weights on them. You specify which features are numeric with the `-N` option on the command line.
- The organization of the code is much more object-oriented than in version 1.0. The main benefit of this is that:

- A Memory-Based Learning API is made available. You can define Memory-Based classification objects in your own C++ programs and access all of the functionality of TiMBL by linking to the TiMBL library.

- It has become easier to examine the way decisions are made from nearest neighbors, because several verbosity-levels allow you to dump similarity values (`-D`), distributions (`-v 16`), and nearest neighbor sets (`-v 32`) to the output file. The `-d` option for writing the distributions no longer exists.

- Better support for the manipulation of MVDM matrices. Using the `-U` and `-u` options it is now possible to respectively save and read back value difference matrices (see Section 5.3).

- Both "pre-stored" and "regular" MVDM experiments now generate filenames with "`mvd`" in the suffix. This used to be "`pvd`" and "`mvd`" respectively.

- a number of minor bugs have been fixed.

Chapter 4

Learning algorithms

TiMBL is a program implementing several Memory-Based Learning techniques. All the algorithms have in common that they store some representation of the training set explicitly in memory. During testing, new cases are classified by extrapolation from the most similar stored cases. The main differences between the algorithms incorporated in TiMBL lie in:

- The definition of *similarity*,
- The way the instances are stored in memory, and
- The way the search through memory is conducted.

In this chapter, various choices for these issues are described. We start in section 4.1 with a formal description of the basic Memory-Based Learning algorithm, i.e. a nearest neighbor search. We then introduce different similarity metrics, such as Information Gain weighting, which allows us to deal with features of differing importance, and the Modified Value Difference metric, which allows us to make a graded guess of the match between two different symbolic values. In section 4.2 and 4.3, we give a description of various optimizations for nearest neighbor search. In section 4.4, we describe the fastest optimization, IGTREE, which replaces the exact nearest neighbor search with a very fast heuristic that exploits the difference in importance between features. Finally, in section 4.5, we describe the TRIBL algorithm, which is a hybrid between IGTREE and nearest neighbor search.

4.1 Memory Based Learning

Memory-based learning is founded on the hypothesis that performance in cognitive tasks is based on reasoning on the basis of similarity of new situations to *stored representations of earlier experiences*, rather than on the application of *mental rules* abstracted from earlier experiences (as in rule induction and rule-based processing). The approach has surfaced in different contexts using a variety of alternative names such as similarity-based, example-based, exemplar-based, analogical, case-based, instance-based, and lazy learning [25, 7, 22, 2, 1]. Historically, memory-based learning algorithms are descendants of the k-nearest neighbor (henceforth k-NN) algorithm [8, 19, 2].

An MBL system, visualized schematically in Figure 4.1, contains two components: a *learning component* which is memory-based (from which MBL borrows its name), and a *performance component* which is similarity-based.

The learning component of MBL is memory-based as it involves adding training instances to memory (the *instance base* or case base); it is sometimes referred to as 'lazy' as memory storage is done without abstraction or restructuring. An instance consists of a fixed-length vector of n feature-value pairs, and an information field containing the classification of that particular feature-value vector.

In the performance component of an MBL system, the product of the learning component is used as a basis for mapping input to output; this usually takes the form of performing classification. During classification, a previously unseen test example is presented to the system. The similarity between the new instance X and all examples Y in memory is computed using a *distance metric* $\Delta(X,Y)$. The extrapolation is done by assigning the most frequent category within the k most similar example(s) as the category of the new test example.

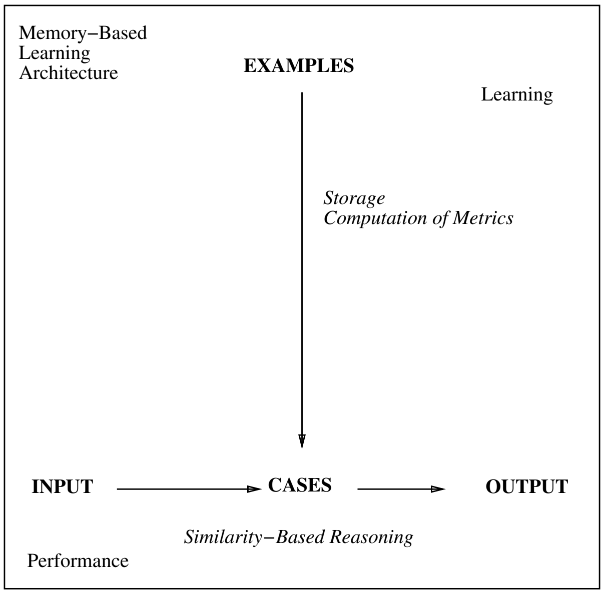

Figure 4.1: General architecture of an MBL system.

4.1.1 Overlap metric

The most basic metric that works for patterns with symbolic features is the **Overlap metric**[1] given in equations 4.1 and 4.2; where $\Delta(X, Y)$ is the distance between patterns X and Y, represented by n features, and δ is the distance per feature. The distance between two patterns is simply the sum of the differences between the features. The k-NN algorithm with this metric is called IB1 [2]. Usually k is set to 1.

$$\Delta(X,Y) = \sum_{i=1}^{n} \delta(x_i, y_i) \tag{4.1}$$

where:

$$\delta(x_i, y_i) = \begin{cases} \frac{x_i - y_i}{max_i - min_i} & \text{if numeric, else} \\ 0 & \text{if } x_i = y_i \\ 1 & \text{if } x_i \neq y_i \end{cases} \tag{4.2}$$

We have made three additions to the original algorithm [2] in our version of IB1. First, in the case of nearest neighbor sets larger than one instance ($k > 1$ or ties), our version of IB1 selects the classification that has the highest frequency in the class distribution of the nearest neighbor set. Second, if a tie cannot be resolved in this way because of equal frequency of classes among the nearest neighbors, the classification is selected with the highest overall occurrence in the training set. Third, in our implementation, the value of k refers to k-nearest distances rather than k-nearest cases.

4.1.2 Information Gain feature weighting

The distance metric in equation 4.2 simply counts the number of (mis)matching feature-values in both patterns. In the absence of information about feature relevance, this is a reasonable choice. Otherwise, we can add domain knowledge bias to weight or select different features (see e.g. [6] for an application of linguistic bias in a language processing task), or look at the behavior of features in the set of examples used for training. We can compute statistics about the relevance of features by looking at which features are good predictors of the class labels. Information Theory gives us a useful tool for measuring feature relevance in this way [23, 24].

Information Gain (IG) weighting looks at each feature in isolation, and measures how much information it contributes to our knowledge of the correct class label. The Information Gain of feature i is measured by computing the difference in uncertainty (i.e. entropy) between the situations without and with knowledge of the value of that feature (equation 4.3).

$$w_i = H(C) - \sum_{v \in V_i} P(v) \times H(C|v) \tag{4.3}$$

[1] This metric is also referred to as Hamming distance, Manhattan metric, city-block distance, or L1 metric.

Where C is the set of class labels, V_i is the set of values for feature i, and $H(C) = -\sum_{c \in C} P(c) \log_2 P(c)$ is the entropy of the class labels. The probabilities are estimated from relative frequencies in the training set. For numeric features, values are first discretized into a number (the default is 20) of equally spaced intervals between the minimum and maximum values of the feature. These groups are then used in the IG computation as if they were discrete values (note that this discretization is not used in the computation of the distance metric).

It is important to realize that the IG weight is really a probability weighted average of the informativity of the different values of the feature. On the one hand, this pre-empts the consideration of values with low frequency but high informativity. Such values "disappear" in the average. On the other hand, this also makes the IG weight very robust to estimation problems. Each parameter (=weight) is estimated on the whole data set.

Information Gain, however, tends to overestimate the relevance of features with large numbers of values. Imagine a data set of hospital patients, where one of the available features is a unique "patient ID number". This feature will have very high Information Gain, but it does not give any generalization to new instances. To normalize Information Gain for features with different numbers of values, Quinlan [24] has introduced a normalized version, called **Gain Ratio**, which is Information Gain divided by $si(i)$ (split info), the entropy of the feature-values (equation 4.5).

$$w_i = \frac{H(C) - \sum_{v \in V_i} P(v) \times H(C|v)}{si(i)} \tag{4.4}$$

$$si(i) = -\sum_{v \in V_i} P(v) \log_2 P(v) \tag{4.5}$$

The resulting Gain Ratio values can then be used as weights w_f in the weighted distance metric (equation 4.6)[2]. The k-NN algorithm with this metric is called IB1-IG [17].

$$\Delta(X,Y) = \sum_{i=1}^{n} w_i \; \delta(x_i, y_i) \tag{4.6}$$

The possibility of automatically determining the relevance of features implies that many different and possibly irrelevant features can be added to the feature set. This is a very convenient methodology if domain knowledge does not constrain the choice enough beforehand, or if we wish to measure the importance of various information sources experimentally. However, because IG values are computed for each feature independently, this is not necessarily the best strategy. Sometimes better results can be obtained by leaving features out than by letting them in with a low weight. Very redundant features can also be

[2] In a generic use IG refers both to Information Gain and to Gain Ratio throughout this manual. In specifying parameters for the software, the distinction between both needs to be made, because they often result in different behavior.

challenging for IB1-IG, because IG will overestimate their joint relevance. Imagine an informative feature which is duplicated. This results in an overestimation of IG weight by a factor two, and can lead to accuracy loss, because the doubled feature will dominate the similarity metric.

4.1.3 Chi-squared feature weighting

Unfortunately, as [33] have shown, the Gain Ratio measure, just as all information based measures, does also have a bias towards features with more values. The reason for this is that the Gain Ratio statistic is not corrected for the number of degrees of freedom of the contingency table of classes and values. [33] proposed a feature selection measure based on the chi-squared statistic, as values of this statistic can be compared across conditions with different numbers of degrees of freedom.

The χ^2statistic is computed from the same contingency table as the Information Gain measure by the following formula (Equation 4.7).

$$\chi^2 = \sum_i \sum_j \frac{(E_{ij} - O_{ij})^2}{E_{ij}} \tag{4.7}$$

where O_{ij} is the observed number of cases with value v_i in class c_j, i.e. $O_{ij} = n_{ij}$, and E_{ij} is the expected number of cases which should be in cell $(v_i,\, c_j)$ in the contingency table, if the null hypothesis (of no predictive association between feature and class) is true (Equation 4.8). Let $n_{.j}$ denote the marginal for class j (i.e. the sum over column j of the table), $n_{i.}$ the marginal for value i, and $n_{..}$ the total number of cases (i.e. the sum of all the cells of the contingency table).

$$E_{ij} = \frac{n_{.j} n_{i.}}{n_{..}} \tag{4.8}$$

The χ^2statistic is well approximated by the chi-square distribution with $\nu = (m-1)(n-1)$ degrees of freedom, where m is the number of values and n is the number of classes. We can then either use the χ^2values as feature weights in Equation 4.6, or we can explicitly correct for the of degrees of freedom by using the **Shared Variance** measure (Equation 4.9).

$$SV_i = \frac{\chi_i^2}{N \times min(|C|, |V|) - 1} \tag{4.9}$$

Where $|C|$ and $|V|$ are the number of classes and the number of values respectively. We will refer to these variations of MBL as IB1-χ^2 and IB1-SV.

One should keep in mind, that the correspondence to the chi-square distribution generally becomes poor if the expected frequencies in the contingency table cells become small. A common recommendation is that the χ^2test cannot be trusted when more than 20% of the expected frequencies are less than 5, or any are less than 1.

4.1.4 Modified Value Difference metric

It should be stressed that the choice of representation for instances in MBL remains the key factor determining the strength of the approach. The features and categories in NLP tasks are usually represented by symbolic labels. The metrics that have been described so far, i.e. Overlap and IG Overlap, are limited to exact match between feature-values. This means that all values of a feature are seen as equally dissimilar. However, if we think of an imaginary task in e.g. the phonetic domain, we might want to use the information that 'b' and 'p' are more similar than 'b' and 'a'. For this purpose a metric was defined by Stanfill & Waltz [25] and further refined by Cost & Salzberg [7]. It is called the (Modified) Value Difference Metric (MVDM; equation 4.10), and it is a method to determine the similarity of the values of a feature by looking at co-occurrence of values with target classes. For the distance between two values V_1, V_2 of a feature, we compute the difference of the conditional distribution of the classes C_i for these values.

$$\delta(V_1, V_2) = \sum_{i=1}^{n} |P(C_i|V_1) - P(C_i|V_2)| \tag{4.10}$$

For computational efficiency, all pairwise $\delta(V_1, V_2)$ values can be pre-computed before the actual nearest neighbor search starts. Note that for numeric features, no MVDM is computed in TiMBL, but a scaled difference (see Equation 4.2) of the actual numeric feature values.

Although the MVDM metric does not explicitly compute feature relevance, an implicit feature weighting effect is present. If features are very informative, their conditional class probabilities will on average be very skewed towards a particular class. This implies that on average the $\delta(V_1, V_2)$ will be large. For uninformative features, on the other hand, the conditional class probabilities will be pretty uniform, so that on average the $\delta(V_1, V_2)$ will be very small.

MVDM differs considerably from Overlap based metrics in its composition of the nearest neighbor sets. Overlap causes an abundance of ties in nearest neighbor position. For example, if the nearest neighbor is at a distance of one mismatch from the test instance, then the nearest neighbor set will contain the entire partition of the training set that matches all the other features but contains *any* value for the mismatching feature (see [34] for a more detailed discussion). With the MVDM metric, however, the nearest neighbor set will either contain patterns which have the value with the lowest $\delta(V_1, V_2)$ in the mismatching position, or MVDM will select a totally different nearest neighbor which has less exactly matching features, but a smaller distance in the mismatching features. In sum, this means that the nearest neighbor set is usually much smaller for MVDM at the same value of k. In NLP tasks we have found it very useful to experiment with values of k larger than one for MVDM, because this re-introduces some of the beneficial smoothing effects associated with large nearest neighbor sets.

One cautionary note about this metric is connected to data sparsity. In many practical applications, we are confronted with a very limited set of examples. This poses a serious problem for the MVDM metric. Many values occur only once in the whole data set. This means that if two such values occur with the same class the MVDM will regard them as identical, and if they occur with two different classes their distance will be maximal. The latter condition reduces the MVDM to the Overlap metric for many cases, with the addition that some cases will be counted as an exact match or mismatch on the basis of very shaky evidence.

4.2 Tree-based memory

The discussion of the algorithm and the metrics in the section above is based on a naive implementation of nearest neighbor search: a flat array of instances which is searched from beginning to end while computing the similarity of the test instance with each training instance (see the left part of Figure 4.2). Such an implementation, unfortunately, reveals the flip side of the lazy learning coin. Although learning is very cheap: just storing the instances in memory, the computational price of classification can become very high for large data sets. The computational cost is proportional to N, the number of instances in the training set.

In our implementation of IB1 we use a more efficient approach. The first part of this approach is to replace the flat array by a tree-based data structure. Instances 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 a *distribution* of classes, i.e. a count of how many times which class occurs with this pattern of feature-values (see Figure 4.3).

Due to this storage structure, instances with identical feature-values are collapsed into one path, and only their separate class information needs to be stored in the distribution at the leaf node. Many different **tokens** of a particular **instance type** share one path from the root to a leaf node. Moreover, instances which share a prefix of feature-values, also share a partial path. This reduces storage space (although at the cost of some book-keeping overhead) and has two implications for nearest neighbor search efficiency.

In the first place, the tree can be searched top-down very quickly for *exact matches*. Since an exact match ($\Delta(X,Y)=0$) can never be beaten, we choose to omit any further distance computations when one is found with this shortcut[3].

Second, the distance computation for the nearest neighbor search can re-use partial results for paths which share prefixes. This re-use of partial results is in the direction from the root to the leaves of the tree. When we have proceeded to a certain level of the tree, we know how much similarity (equation 4.2) can still contribute to the overall distance (equation 4.1), and discard whole branches of the tree which will never be able to rise above the partial similarity of the current

[3]There is a command line switch (`-x`) which turns the shortcut off in order to get real k-NN results when $k>1$ (i.e. get neighbors at further distances).

ID	size	shape	# holes	class
1	small	compact	1	nut
2	small	long	none	screw
3	small	long	1	key
4	small	compact	1	nut
5	large	long	1	key
6	small	compact	none	screw
7	small	compact	1	nut
8	large	long	none	pen
9	large	long	2	scissors
10	large	long	1	pen
11	large	other	2	scissors
12	small	other	2	key
	large	long	none	pen

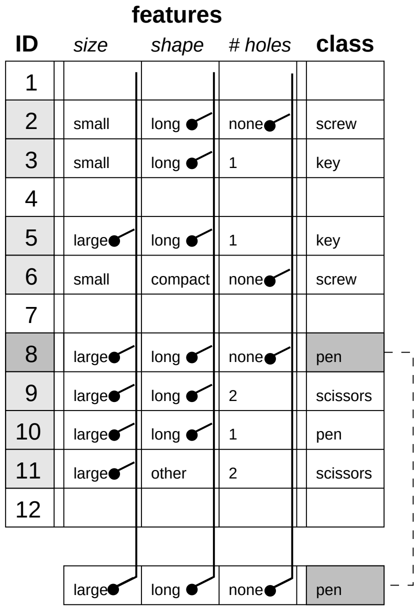

Figure 4.2: The instance base for a small object classification toy problem. The left figure shows a flat array of instances through which sequential nearest neighbor search is performed to find the best match for a test instance (shown below the instance base). In the right part, an inverted index (see text) is used to restrict the search to those instances which share at least one feature value with the test instance.

least similar best neighbor. By doing the search depth first[4], the similarity threshold quickly gets initialized to a good value, so that large parts of the search space can be pruned.

Disregarding this last constraint on search, the number of feature-value comparisons is equal to the number of arcs in the tree. Thus if we can find an ordering of the features which produces more overlap between partial paths, and hence a smaller tree, we can gain both space and time improvements. An ordering which was found to produce small trees for many of our NLP data sets is Gain Ratio divided by the number of feature-values (this is the default setting). Through the `-T` command line switch, however, the user is allowed to experiment with different orderings.

[4]Suggested by Gert Durieux.

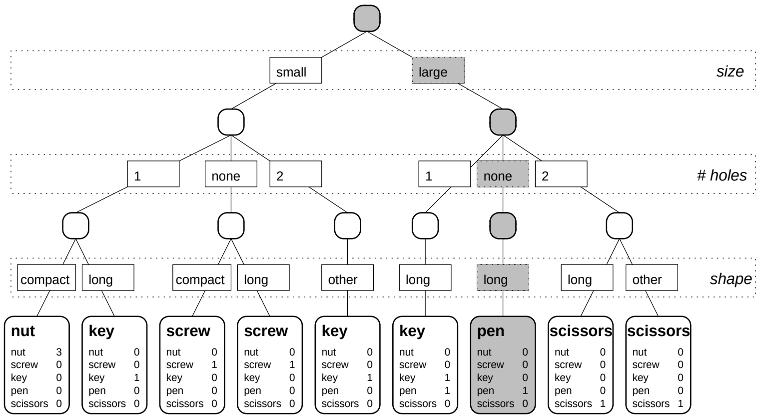

Figure 4.3: A tree-structured storage of the instance base from figure 4.2. An exact match for the test is in this case directly found by a top down traversal of the tree (grey path). If there is no exact match, all paths are interpreted as instances and the distances are computed. The order of the features in this tree is based on Gain Ratio.

4.3 Inverse index

The second part of our approach to efficiency is a speedup optimization based on the following fact. Even in the tree-based structure, the distance is computed between the test instance and *all* instance types. This means that even instance types which do not share a single feature-value with the test instance are considered, although they will surely yield a zero similarity. The use of an **inverted index** excludes these zero similarity patterns. The construction of the inverted index records for all values of each feature a list of instance types (i.e. leaf nodes in the tree described in the previous section) in which they occur. Thus it is an inverse of the instance-base, which records for each instance type which feature-values occur in it[5].

When a test instance is to be classified, we select the lists of instance types for the feature-values that it contains (illustrated in the rightmost part of Figure 4.2). We can now find the nearest neighbor in these lists in a time that is proportional to the number of occurrences of the most frequent feature-value of the test pattern, instead of proportional to the number of instance types.

Although worst case complexity is still proportional to N, the size of the training set, and practical mileage may vary widely depending on the peculiarities of your data, the combination of exact match shortcut, tree-based path re-use, and inverted index has proven in practice (for our NLP datasets) to

[5]Unfortunately this also implies that the storage of both an instance-base and an inverted index takes about twice the amount of memory. Therefore we have chosen to turn it off by default. However, especially for data with very large numbers of low frequent values it can be substantially faster to turn it on.

make the difference between hours and seconds of computation[6].

4.4 IGTree

Using Information Gain rather than unweighted Overlap distance to define similarity in IB1 improves its performance on several NLP tasks [17, 27, 26]. The positive effect of Information Gain on performance prompted us to develop an alternative approach in which the instance memory is restructured in such a way that it contains the same information as before, but in a compressed decision tree structure. We call this algorithm IGTREE [14] (see Figure 4.4 for an illustration). In this structure, similar to the tree-structured instance base described above, instances are stored as paths of connected nodes which contain classification information. Nodes are connected via arcs denoting feature values. Information Gain is used to determine the order in which instance feature-values are added as arcs to the tree. The reasoning behind this 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 memory instances that have the same feature-value as the test instance at that feature. Instead of indexing all memory instances only once 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. Again, considerable compression is obtained as similar instances share partial paths.

Because IGTREE makes a heuristic approximation of nearest neighbor search by a top down traversal of the tree in the order of feature relevance, we no longer need to store all the paths. The idea is that it is not necessary to fully store those feature-values of the instance that have lower Information Gain than those features which already fully disambiguate the instance classification.

Apart from compressing all training instances in the tree structure, the IGTREE algorithm also stores with each non-terminal node information concerning the *most probable* or *default* classification given the path thus far, according to the bookkeeping information maintained by the tree construction algorithm. This extra information is essential when processing unknown test instances. Processing an unknown input 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 (i.e., an exact match was found), or using the default classification on the last matching non-terminal node if an exact match fails.

In sum, it can be said that 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, to obtain considerably simplified and faster processing during classification, as compared to IB1 and IB1-IG.

[6] MVDM and numeric features cannot make use of the inverted index optimization, because it can happen that two cases with not one value in common are still nearest neighbors.

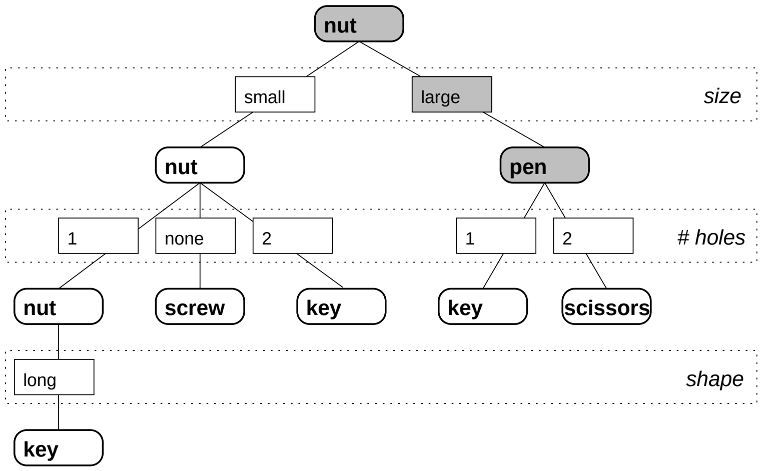

Figure 4.4: A pruned IGTREE for the instance base of Figure 4.2. The classification for the test instance is found by top down search of the tree, and returning the class label (default) of the node after the last matching feature-value (arc). Note that this tree is essentially a compressed version of the tree in Figure 4.3.

The generalization accuracy of IGTREE is usually comparable to that of IB1-IG; most of the time not significantly differing, and occasionally slightly (but statistically significantly) worse, or even better. The two reasons for this surprisingly good accuracy are that (i) most 'unseen' instances contain considerably large parts that fully match stored parts of training instances, and (ii) the probabilistic information stored at non-terminal nodes (i.e., the default classifications) still produces strong 'best guesses' when exact matching fails. The difference between the top-down traversal of the tree and precise nearest neighbor search becomes more pronounced when the differences in informativity between features are small. In such a case a slightly different weighting would have produced a switch in the ordering and a completely different tree. The result can be a considerable change in classification outcomes, and hence also in accuracy. However, we have found in our work on NLP datasets that when the goal is to obtain a very fast classifier for processing large amounts of text, the slight tradeoff between accuracy and speed can be very attractive. Note, also, that by design, IGTREE is not suited for numeric features, as long as it does not use some type of discretization. In TiMBL numbers will simply be treated as literal strings in this case. Moreover, one should realize that the success of IGTREE is determined by a good judgement of feature relevance ordering. Hence IGTREE is not to be used with e.g. "no weights" (`-w 0`). Also, setting the `-k` parameter has no effect on IGTREE performance.

4.5 The TRIBL hybrid

The application of IGTREE on a number of common machine-learning datasets suggested that it is not applicable to problems where the relevance of the predictive features cannot be ordered in a straightforward way, e.g. if the differences in Information Gain are only very small. In those cases, IB1-IG or even IB1 tend to perform significantly better than IGTREE.

For this reason we have designed TRIBL, a hybrid generalization of IGTREE and IB1. TRIBL allows you to exploit the trade-off between (i) optimization of search speed (as in IGTREE), and (ii) maximal generalization accuracy. To achieve this, a parameter is set determining the switch from IGTREE to IB1. A heuristic that we have used with some success is based on *average feature information gain*; when the Information Gain of a feature exceeds the sum of the average Information Gain of all features + one standard deviation of the average, then the feature is used for constructing an IGTREE, including the computation of defaults on nodes. When the Information Gain of a feature is below this threshold, and the node is still ambiguous, tree construction halts and the leaf nodes at that point represent case bases containing subsets of the original training set. During search, the normal IGTREE search algorithm is used, until the case-base nodes are reached, in which case regular IB1 nearest neighbor search is used on this sub-case-base. In TiMBL, however, you must specify the switch point from IGTREE to IB1, also referred to as "TRIBL offset", manually.

4.6 NLP applications of TiMBL

This section provides a historical overview of our own work with the application of MBL type algorithms to NLP tasks.

The IB1-IG algorithm was first introduced in [17] in the context of a comparison of memory-based approaches with backprop learning for a hyphenation task. Predecessor versions of IGTREE can be found in [12, 27] where they are applied to grapheme-to-phoneme conversion. See [14] for a detailed description and review of the algorithms. A recent development, now implemented in the TiMBL package is TRIBL [15].

The memory-based algorithms implemented in the TiMBL package have been successfully applied to a large range of Natural Language Processing tasks in our group: hyphenation and syllabification ([17]); assignment of word stress ([11]); grapheme-to-phoneme conversion ([13]); diminutive formation ([16]); morphological analysis ([29, 28]); part of speech tagging ([18, 35]); PP-attachment ([36]); word sense disambiguation([30]); subcategorization ([4]); chunking (partial parsing) ([31]); and shallow parsing ([10, 5]).

Relations to statistical language processing are discussed in [34]. A partial overview paper is [9]. The first dissertation-length study devoted to the approach is [26], in which the approach is compared to alternative learning methods for NLP tasks related to English word pronunciation (stress assignment, syllabification, morphological analysis, alignment, grapheme-to-phoneme conversion). Recently a special issue of the *Journal for Experimental and Theoretical Artificial Intelligence* (Vol. 11(3)) was devoted to Memory-Based Language Processing.

All papers referred to in this section and more recent material are available in electronic form from the ILK homepage: `http://ilk.kub.nl`. We are grateful for any feedback on the algorithms and the way we applied them.

Whereas the work in Tilburg has been oriented primarily towards *language engineering* applications, the CNTS research group of Antwerp University has studied the linguistic and psycholinguistic relevance of memory-based learning for stress assignment in Dutch ([11, 21]), and as a model for *phonological bootstrapping*. A recently started project has as its aim to test predictions from memory-based models for language processing with psycholinguistic experiments.

Chapter 5

File formats

This chapter describes the format of the input and output files used by TiMBL. Where possible, the format is illustrated using the same small toy data set that is shown in Figure 4.2. It consists of 12 instances of 5 different everyday objects (nut, screw, key, pen, scissors), described by 3 discrete features (size, shape, and number of holes).

5.1 Data format

The training and test sets for the learner consist of descriptions of instances in terms of a fixed number of feature-values. TiMBL supports a number of different formats for the instances, but they all have in common that the files should contain one instance per line. The number of instances is determined automatically, and the format of each instance is inferred from the format of the first line in the training set. The last feature of the instance is assumed to be the target category. Should the guess of the format by TiMBL turn out to be wrong, you can force it to interpret the data as a particular format by using the `-F` option. Note that TiMBL, by default, will interpret features as having *symbolic, discrete values.* Unless you specify explicitly that certain features are numeric, using the `-M` option, TiMBL will interpret numbers as just another string of characters. If a feature is numeric, its values will be scaled to the interval [0,1] for purposes of distance computation (see Equation 4.2). The computation of feature weights will be based on a discretization of the feature.

Once TiMBL has determined the input format, it will skip and complain about all lines in the input which do not respect this format (e.g. have a different number of feature-values with respect to that format).

During testing, TiMBL writes the classifications of the test set to an output file. The format of this output file is by default the same as the input format, with the addition of the predicted category being appended after the correct category. If we turn on higher levels of verbosity, the output files will also contain distributions, distances and nearest neighbor sets.

5.1.1 Column format

The **column format** uses white space as the separator between features. White space is defined as a sequence of one or more spaces or tab characters. Every instance of white space is interpreted as a feature separator, so it is not possible to have feature-values containing white space. The column format is auto-detected when an instance of white space is detected on the first line *before a comma has been encountered.* The example data set looks like this in the column format:

```
small compact 1 nut
small long none screw
small long 1 key
small compact 1 nut
large long 1 key
small compact none screw
small compact 1 nut
large long none pen
large long 2 scissors
large long 1 pen
large other 2 scissors
small other 2 key
```

5.1.2 C4.5 format

This format is a derivative of the format that is used by the well-known C4.5 decision tree learning program [24]. The separator between the features is a comma, and the category (viz. the last feature on the line) is followed by a period (although this is not mandatory: TiMBL is robust to missing periods)[1]. White space within the line is taken literally, so the pattern `a, b c,d` will be interpreted as `'a',' b c','d'`. When using this format, especially with linguistic data sets or with data sets containing floating point numbers, one should take special care that commas do not occur in the feature-values and that periods do not occur within the category. Note that TiMBL's C4.5 format does not require a so called *namesfile.* However, TiMBL can produce such a file for C4.5 with the `-n` option. The C4.5 format is auto-detected when a comma is detected on the first line *before any white space has been encountered.* The example data set looks like this in the C4.5 format:

```
small,compact,1,nut.
small,long,none,screw.
small,long,1,key.
small,compact,1,nut.
large,long,1,key.
small,compact,none,screw.
```

[1] The periods after the category are not reproduced in the output

```
small,compact,1,nut.
large,long,none,pen.
large,long,2,scissors.
large,long,1,pen.
large,other,2,scissors.
small,other,2,key.
```

5.1.3 ARFF format

ARFF is a format that is used by the WEKA machine learning workbench [20][2]. Although TiMBL at present does *not* entirely follow the ARFF specification, it still tries to do as well as it can in reading this format. In ARFF the actual data are preceded by a header with various types of information and interspersed with lines of comments (starting with %). The ARFF format is auto-detected when the first line starts with % or @. TiMBL ignores lines with ARFF comments and instructions, and starts reading data from after the `@data` statement until the end of the file. The feature-values are separated by commas, and white space is deleted entirely, so the pattern `a, b c,d` will be interpreted as `'a','bc','d'`. We hope include better support for the ARFF format in future releases.

```
% There are 4 attributes.
% There are 12 instances.
% Attribute information:                          Ints Reals  Enum  Miss
%         'size'                                     0     0    12     0
%         'shape'                                    0     0    12     0
%         'n_holes'                                  9     0     3     0
%         'class.'                                   0     0    12     0
@relation 'example.data'
@attribute 'size' { small, large}
@attribute 'shape' { compact, long, other}
@attribute 'n_holes' { 1, none, 2}
@attribute 'class.' { nut., screw., key., pen., scissors.}
@data
small,compact,1,nut.
small,long,none,screw.
small,long,1,key.
small,compact,1,nut.
large,long,1,key.
small,compact,none,screw.
small,compact,1,nut.
large,long,none,pen.
large,long,2,scissors.
large,long,1,pen.
large,other,2,scissors.
```

[2] WEKA is available from the Waikato University Department of Computer Science, `http://www.cs.waikato.ac.nz/~ml/`.

```
small,other,2,key.
```

5.1.4 Compact format

The compact format is especially useful when dealing with very large data files. Because this format does not use any feature separators, file-size is reduced considerably in some cases. The price of this is that all features and class labels must be of equal length (in characters) and TiMBL needs to know beforehand what this length is. You must tell TiMBL by using the `-l` option. The compact format is auto-detected when neither of the other formats applies. The same example data set might look like this in the column format (with two characters per feature):

```
smco1_nu
smlonosc
smlo1_ke
smco1_nu
lalo1_ke
smconosc
smco1_nu
lalonope
lalo2_sc
lalo1_pe
laot2_sc
smot2_ke
```

5.1.5 Sparse Binary format

The sparse binary format is especially useful when dealing with large numbers of two-valued (binary) features, of which each case only has a very few active ones, such as e.g. in text categorization. Thus instead of representing a case as:

```
1,0,0,0,0,0,0,1,0,0,1,1,0,0,0,0,0,1,small.
```

We can represent it as:

```
1,8,11,12,18,small.
```

This format allows one to specify only the index numbers of the active features (indexes start at one), while implicitly assuming that the value for all the remaining features is zero. Because each case has a different number of active features, we must specify in some other way what the actual number of features is. This must be done using the `-N` option. As the format is very similar to numerical features, there is no auto-detection for it. It must always be "forced", using `-F Binary`. The last feature of a line is always interpreted as being the category string. A case with only zeroes can be represented as either `‘‘‘class’’` or `‘‘,class’’`.

5.2 Weight files

The feature weights that are used for computing similarities and for the internal organization of the memory-base can be saved to a file. A file with weights can be constructed or altered manually and then read back into TiMBL. The format for the weights file is as follows. The weights file may contain comments on lines that start with a # character. The other lines contain the number of the feature followed by its numeric weight. An example of such a file is provided below. The numbering of the weights starts with 1 and follows the same order as in the data file. If features are to be ignored it is advisable not to set them to zero, but give them the value "`Ignored`" or to use the `-s` option.

```
# DB Entropy: 2.29248
# Classes: 5
# Lines of data: 12
# Fea.  Weight
1       0.765709
2       0.614222
3       0.73584
```

5.3 Value difference files

Using the MVDM metric, it can sometimes be interesting to inspect the matrix of conditional class probabilities from Equation 4.10. By using the `-U` option, we can write the computed matrix to a file. This way we can see which values are considered to be similar by the metric. For each feature a row vector is given for each value, of the conditional probabilities of all the classes (columns) given that value.

```
targets A,      B,      C,      D,      E.

feature # 1 Matrix:
small   0.429   0.286   0.286   0.000   0.000
large   0.000   0.000   0.200   0.400   0.400

feature # 2 Matrix:
compact 0.750   0.250   0.000   0.000   0.000
long    0.000   0.167   0.333   0.333   0.167
other   0.000   0.000   0.500   0.000   0.500

feature # 3 Matrix:
1       0.500   0.000   0.333   0.167   0.000
none    0.000   0.667   0.000   0.333   0.000
2       0.000   0.000   0.333   0.000   0.667
```

As long as this format is observed, the file can be modified (manually or by substituting some other vector-based representations for the values), and the new matrix can be read in and used with the MVDM metric.

5.4 Tree files

Although the learning phase in TiMBL is relatively fast, it can sometimes be useful to store the internal representation of the data set for even faster subsequent retrieval. In TiMBL, the data set is stored internally in a tree structure (see Section 4.2). When using IB1, this tree representation contains all the training cases as full paths in the tree. When using IGTREE, unambiguous paths in the tree are pruned before it is used for classification or written to file. In either tree, the arcs represent feature-values and nodes contain class (frequency distribution) information. The features are in the same order throughout the tree. This order is either determined by memory-size considerations in IB1, or by feature relevance in IGTREE. It can explicitly be manipulated using the `-T` option.

We strongly advise to refrain from manually editing the tree file. However, the syntax of the tree file is as follows. After a header consisting of information about the status of the tree, the feature-ordering (the permutation from the order in the data file to the order in the tree), and the presence of numeric features[3] the tree's nodes and arcs are given in non-indented bracket notation.

Starting from the root node, each node is denoted by an opening parenthesis "(", followed by a default class. After this, there is the class distribution list, within curly braces "{ }", containing a non-empty list of categories followed by integer counts. After this comes an optional list of children, within "[ ]" brackets, containing a non-empty list of nodes, separated by comma's[4]. The choice whether distributions are present is maintained throughout the whole tree. Whether children are present is really dependent on whether children *are* present.

The IB1 tree that was constructed from our example data set looks as follows:

```
# Status: complete
# Permutation: < 1, 3, 2 >
# Numeric: .
# Version 3
#
( nut { nut 3 screw 2 key 3 pen 2 scissors 2 } [ small ( nut
{ nut 3 screw 2 key 2 } [ 1 ( nut { nut 3 key 1 }
[ compact ( nut { nut 3 } )
, long ( key { key 1 } )
 ]
```

[3] Although in this header each line starts with '#', these lines cannot be seen as comment lines.

[4] This is a minor departure from the format of version 2.0.

```
 )
, none ( screw { screw 2 } [ compact ( screw { screw 1 } )
, long ( screw { screw 1 } )
 ]
 )
, 2 ( key { key 1 } [ other ( key { key 1 } )
 ]
 )
 ]
 )
, large ( pen { key 1 pen 2 scissors 2 } [ 1 ( key { key 1 pen 1 }
[ long ( key { key 1 pen 1 } )
 ]
 )
, none ( pen { pen 1 } [ long ( pen { pen 1 } )
 ]
 )
, 2 ( scissors { scissors 2 } [ long ( scissors { scissors 1 } )
, other ( scissors { scissors 1 } )
 ]
 )
 ]
 )
 ]
 )
```

The corresponding compressed IGTREE version is much smaller.

```
# Status: pruned
# Permutation: < 1, 3, 2 >
# Numeric: .
# Version 3
#
( nut { nut 3 screw 2 key 3 pen 2 scissors 2 }
[ small ( nut { nut 3 screw 2 key 2 } [ 1 ( nut { nut 3 key 1 }
[ long ( key { key 1 } )
 ]
 )
, none ( screw { screw 2 } )
, 2 ( key { key 1 } )
 ]
 )
, large ( pen { key 1 pen 2 scissors 2 } [ 1 ( key { key 1 pen 1 } )
, 2 ( scissors { scissors 2 } )
 ] ) ] )
```

Chapter 6

Server interface

Sometimes it is not practical to put all the test items into one test file. Examples of this include:

- The output of one classification is being reused as a feature of some other classifier.
- The test items come in at arbitrary time intervals.

In such cases it is more practical to load the training patterns once and have TiMBL stand by as a server waiting for new test items to be processed. This can be achieved by starting TiMBL with the `-S portnumber` option. TiMBL will load the training set and do the necessary preparation of statistics and metrics, and then enter an infinite loop, waiting for input on the specified portnumber. When a client connects on this portnumber, the server starts a separate thread for it to process (classification) commands. A sample client program is included in the distribution. The client must communicate with the server using the protocol described below. After accepting the connection, the server first sends a welcome message to the client:
`Welcome to the Timbl server.`
After this, the server waits for client-side requests. The client can now issue four types of commands: classify, set (options), query (status), and exit. The type of command is specified by the the first string of the request line, which can be abbreviated to any prefix of the command, up to one letter (i.e. c,s,q,e). The command is followed by whitespace and the remainder of the command as described below.

`classify testcase`
: testcase is a pattern of features (must have the same number of features as the training set) followed by a category string. E.g.: `small,long,1,??`. Depending on the current settings of the server, it will either return the answer

```
ERROR { explanation }
```

if something's gone wrong, or the answer

```
CATEGORY {category} DISTRIBUTION { category 1 } DISTANCE { 1.000000} NEIGHBORS
ENDNEIGHBORS
```

where the presence of the `DISTRIBUTION`, `DISTANCE` and `NEIGHBORS` parts depends upon the current verbosity setting. Note that if the last string on the answer line is `NEIGHBORS`, the server will proceed to return lines of nearest neighbor information until the keyword `ENDNEIGHBORS`.

`set option`
where option is specified as a string of commandline options (described in detail in Section 7 below). Only the following commandline options are valid in this context: `k, m, q, Q, R, w, v, x, -`. The setting of an option in this client does not affect the behavior of the server towards other clients. The server replies either with `OK` or with `ERROR {explanation}`.

`query`
queries the server for a list of current settings. Returns a number of lines with status information, starting with a line that says `STATUS`, and ending with a line that says `ENDSTATUS`. For example:

```
STATUS
FLENGTH            : 0
MAXBESTS           : 500
TREE_ORDER         : GRO
INPUTFORMAT        : C45
SEED               : -1
PROBALISTIC        : -
VERBOSITY          : F
EXACT_MATCH        : +
USE_INVERTED       : -
GLOBAL_METRIC      : Overlap
METRICS            :
NEIGHBORS          : 1
PROGRESS           : 100000
TRIBL_OFFSET       : 0
WEIGHTING          : GRW
ENDSTATUS
```

`exit`
closes the connection between this client and the server.

Chapter 7

Command line options

The user interacts with TiMBL through the use of command line arguments. When you have installed TiMBL successfully, and you type `Timbl` at the command line without any further arguments, it will print an overview of the most basic command line options.

```
TiMBL 3.0 (c) ILK 1998, 1999, 2000.
Tilburg Memory Based Learner
Induction of Linguistic Knowledge Research Group
Tilburg University / University of Antwerp
Mon Feb 21 16:28:01 2000

usage:  Timbl -f data-file {-t test-file}
or see: Timbl -h
        for all possible options
```

If you are satisfied with all of the default settings, you can proceed with just these basics:

- `-f <datafile>` : supplies the name of the file with the training items.
- `-t <testfile>` : supplies the name of the file with the test items.
- `-h` : prints a glossary of all available command line options.

The presence of a training file will make TiMBL pass through the first two phases of its cycle. In the first phase it examines the contents of the training file, and computes a number of statistics on it (feature weights etc.). In the second phase the instances from the training file are stored in memory. If no test file is specified, the program exits, possibly writing some of the results of learning to files (see below). If there is a test file, the selected classifier, trained on the present training data, is applied to it, and the results are written to a file of which name is a combination of the name of the test file and a code representing the chosen algorithm settings. TiMBL then reports the percentage of correctly classified test items. The default settings for the classification phase are: a Memory-Based Learner, with Gain Ratio feature weighting, with $k = 1$, and with optimizations for speedy search. If you need to change the settings, because you want to use a different type of classifier, or because you need to make a trade-off between speed and memory-use, then you can use the options that are shown using `-h`. The sections below provide a reference to the use of these command line arguments, and they are roughly ordered by the type of action that the option has effect on. Note that some options (listed with "`+/-`") can be turned on (`+`) or off (`-`).

7.1 Algorithm and Metric selection

`-a <n>` : chooses between the standard IB1 (nearest neighbor search) algorithm (n=0, this is the default value), the decision tree-based optimization IGTREE (n=1), and the hybrid of the two: TRIBL (n=2).

`-m <string>` : determines which similarity metrics are used for each feature. The format of this string is as follows:
`GlobalMetric:MetricRange:MetricRange`
Where `GlobalMetric` is used for alle features except for the ones that are assigned other metrics by following `:MetricRange` restrictions. Metric can be one of: O,M,N,I, which stand for respectively Overlap (default), Modified value difference (MVDM), Numeric, or Ignore (which cannot be the global one). A range can be written using comma's for lists, and hyphens for intervals.

`-w <n>` : chooses between feature-weighting possibilities. The weights are used in the metric of IB1 and in the ordering of the IGTREE. Possible values are:

n=0 – No weighting, i.e. all features have the same importance (weight = 1).

n=1 – Gain Ratio weighting (default). See section 4.1.2.

n=2 – Information Gain weighting. See section 4.1.2.

n=3 – Chi-squared (χ^2) weighting. See section 4.1.3.

n=4 – Shared variance weighting. See section 4.1.3.

n=filename – Instead of a number we can supply a filename to the `-w` option. This causes TiMBL to read this file and use its contents as weights. (See section 5.2 for a description of the weights file)

`-k <n>` : Number of nearest neighbors used for extrapolation. Only applicable in conjunction with IB1 (`-a 0`) and TRIBL (`-a 2`). The default is 1. Especially with the MVDM metric it is often useful to determine a good value larger than 1 for this parameter (usually an odd number, to avoid ties). Note that due to ties (instances with exactly the same similarity to the test instance) the number of instances used to extrapolate might in fact be much larger than this parameter.

`-q <n>` : `n` is the TRIBL offset, the index number of the feature where TRIBL should switch from IGTREE to IB1. Does not apply to the other two algorithms.

`-R <n>` : Resolve ties in the classifier randomly, using a random generator with seed n. As a default this is OFF, and ties are resolved in favor of the category which is more frequent in the training set as a whole—remaining ties are resolved on a first come first served basis. For comparison purposes, we have also included the option `-R P <n>`, which causes the classification to be based on a random pick (with seed n) of a category according to the probability distribution in the nearest neighbor set.

`-t <@file>` : If the filename given after `-t` starts with '`@`', TiMBL will read commands for testing from `file`. This file should contain one set of instructions per line. On each line new values can be set for the following command line options: `-e -F -k -m -o -p -q -Q -R -t -u +/-v -w +/-x +/-% +/--`. It is compulsory that each line in `file` contains a `-t <testfile>` argument to specify the name of the test file.

`-t leave_one_out` : No test file is read, but testing is done on each pattern of the training file, by treating each pattern of the training file in turn as a test case (and the whole remainder of the file as training cases).

7.2 Input options

`-F <format>` : Force TiMBL to interpret the training and test file as a specific data format. Possible values for this parameter are: `Compact, C4.5, ARFF, Columns, Binary` (case-insensitive). The default is that TiMBL guesses the format from the contents of the first line of the data file. See section 5.1 for description of the data formats and the guessing rules. The `Compact` format cannot be used with numeric features.

`-l <n>` : Feature length. Only applicable with the Compact data format; `<n>` is the number of characters used for each feature-value and category symbol.

`-N <n>` : (maximum) number of features. Obligatory for Binary format.

`-i <treefile>` : Skip the first two training phases, and instead of processing a training file, read a previously saved (see `-I` option) instance-base or IGTREE from the file `treefile`. See section 5.4 for the format of this file.

`-u <mvdmmatrixfile>` : Replace the computed MVDM matrix with the matrices provided in this file.

-P <path> : Specify a path to read the data files from. This path is ignored if the name of the data file already contains path information.

-S <portnumber> : Starts a TiMBL server listening on the specified port number of the localhost. See Chapter 6 for a description of the communication protocol.

7.3 Output options

-I <treefile> : After phase one and two of learning, save the resulting tree-based representation of the instance-base or IGTREE in a file. This file can later be read back in using the -i option (see above). See section 5.4 for a description of the resulting file's format. This also automatically saves the current weights into treefile.wgt unless this is overridden by W.

-W <file> : Save the currently used feature-weights in a file.

-U <mvdmmatrixfile> : Write the computed MVDM matrix to this file.

-n <file> : Save the feature-value and target category symbols in a C4.5 style "names file" with the name <file>. Take caution of the fact that TiMBL does not mind creating a file with ',' '.' '—' and ':' values in features. C4.5 will choke on this.

-p <n> : Indicate progress during training and testing after every n processed patterns. The default setting is 10000.

-e <n> : During testing, compute and print an estimate on how long it will take to classify n test patterns. This is off by default.

-V : Show the version number.

+/-v <n> : Verbosity Level; determines how much information is written to the output during a run. Unless indicated otherwise, this information is written to standard error. The use of + turns a given verbosity level **on**, whereas - turns it **off** (only useable in non-commandline contexts, such as client/server communication or -t @testcommandfile).. This parameter can take on the following values:

s: work silently (turns off all set verbosity levels).

o: show all options set.

f: show Calculated Feature Weights. (default)

p: show value class probabilities.

e: show exact matches.

di: add distance to output file.

db: add distribution of best matched to output file n: add nearest neigbors to output file (sets `-x` and `--`)

You may combine levels using '+' e.g. `+v p+db` or `-v o+di`.

`+/- %` : Write the percentage of correctly classified test instances to a file with the same name as the output file, but with the suffix "`.%`".

`-o <filename>` : Write the test output to filename. Useful for different runs with the same settings on the same testfile.

`-O <path>` : Write all output to the path given here. The default is to write all output to the directory where the test file is located.

7.4 Internal representation options

`+/- x` : Turns on/off the shortcut search for exact matches in IB1. The default is for this to be ON (which is usually much faster), but when $k > 1$, the shortcut produces different results from a "real" k nearest neighbors search.

`+/- -` : Turn on/off the use of inverted files. Turning this on will sometimes make testing (considerably) faster, but approximately doubles the memory load. The default is off.

`-B <n>`: Set the maximum number of stored nearest neighbors.

Chapter 8

Programmer's reference to the TiMBL API

The TiMBL software is written in C++ in a more or less object-oriented manner. This makes it possible to offer you a set of C++ classes and an Application Programming Interface (API) to access TiMBL classifiers directly from your own programs. For example, if you are building a Part-of-Speech tagger, you want to initialize several classifiers once, and let them classify test items as they come along, rather than in a batch. Or perhaps you want to use the output of one classifier as input to the next, or you want to customize the TiMBL interface to your needs. This is all possible. All you have to do is to include the TiMBL header file in your program:

```
#include "MBLClass.h"
```

In this software distribution you can find two example programs (`tse.cc` and `classify.cc`) that demonstrate the use of TiMBL classes in another C++ program. Have a look at these to get a feel for how this works. To compile your own programs, or these examples, you need to link with the `Timbllib.a` library:

```
g++ -o yourprogram yourprogram.o Timbllib.a
```

Note, however, that *the license does not permit you to redistribute* modified versions of TiMBL or derivative works without prior permission. Also, note that the API is still "work in progress" and we might make changes to it in future releases.

This chapter gives an overview of the TiMBL classes needed to include Memory-Based classifiers in your own programs, and describes the interface through which they can be accessed. We have not attempted to document any of the internal structure of the TiMBL source code here.

8.1 Class hierarchy

The main classes that make up TiMBL are shown in Figure 8.1. A separate class is defined for each type of algorithm (`-a 0/1/2` in terms of command line options). The actual classifiers (`IB1Class`, `IGTREEClass`, and `TRIBLClass`) are derived directly from the virtual base class `MBLClass`.[1]

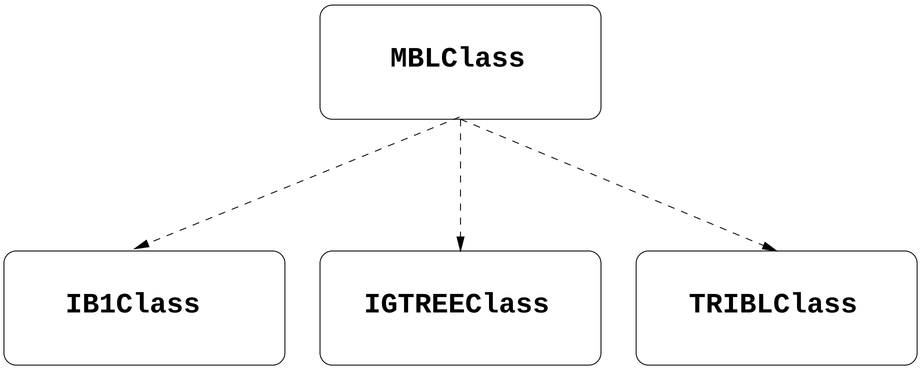

Figure 8.1: The main components of the TiMBL class hierarchy. The arrows denote inheritance.

Since `MBLClass` is a virtual base class, and hence it does not have its own constructor, you will typically want to declare an pointer to an object of this class.

```
MBLClass * TheClassifierPtr = NULL;
```

Such a pointer can then be instantiated to one of the existing classes, depending on your needs.

```
switch(algorithm){
   case MBL:
     TheClassifierPtr = new IB1Class;
     break;
   case IGTREE:
     TheClassifierPtr = new IGTREEClass;
     break;
   case TRIBL:
     TheClassifierPtr = new TRIBLClass;
     break;
}
```

[1] There is also a `TimblExperiment`, which puts the wrappers around `MBLClass` that make up the TiMBL program. This class is not documented here as we do not consider it clean enough to be accessible from the outside.

Or you can directly make an object of the desired class, e.g.:

```
IGTREEClass TheClassifier;
```

8.2 Public member functions

The constructors and destructors are specific for each of the derived classes. The rest of the interface is the same for all classes and is defined at the level of the `MBLClass`. The return type of most functions is `bool`. A return value of 0 means that something went wrong, 1 is a successful return.

8.2.1 The constructors

```
IB1Class::IB1Class(const int N = 1000, const char * = NULL );
IGTREEClass::IGTREEClass(const int N = 1000, const char * = NULL );
TRIBLClass::TRIBLClass(const int N = 1000, const char * = NULL );
```

These constructor functions take optional `const int` and `const char *` arguments. The integer `N` specifies the maximum amount of features for the classifier object (defaults to 1000). By way of the string you can assign a meaningful name to the created object. The function:

```
char * MBLClass::ExpName();
```

will return that name to you at a later point in time.

8.2.2 The destructors

```
IB1Class::~IB1Class();
IGTREEClass::~IGTREEClass();
TRIBLClass::~TRIBLClass();
```

They clean up all the mess.

8.2.3 Member functions

The first two functions below emulate the functionality of the first two phases of TiMBL (Examining and Learning). They will act with default values of all parameters for the type of classifier that they are invoked from. To use different parameter settings, use the `SetOption()` function described below. After invoking `Learn(filename)`, a classifier is available for subsequent classification of test patterns. There are several classification functions available and several additional input/output possibilities. Note that all string argument interfaces are available both using `const char *` and the STL type `const string &`.

`bool MBLClass::PrepareExperiment( char * `*f*` );`
Use the file with name *f* to go through phase 1 of learning (fills the statistics tables for feature values, targets, weights etc.) If needed, this function is automatically called by `Learn()` (see below).

`bool MBLClass::Learn(const char * `*f*` = NULL );`
Build an instance-base from the file *f* (phase 2 of learning), and if needed, call `PrepareExperiment()` first. If *f* `== NULL`, or the argument is omitted, then the same file is used as was used by `PrepareExperiment()` (if any).

`bool * MBLClass::Classify( const char * `*Line*`, char *& `*Result*`, char *& `*Distribution*`, double & `*Distance*`);`

The string *Line* is parsed into features according to the current input format and classified by the present classifier. If classification fails the function returns `false`, otherwise `true`. If succesfull, the string *Result* holds the value of the most likely category, and the string *Distribution* holds the whole distribution of categories in the nearest neighbor set, e.g. "{ `A 12 B 3 C 81`}". You can parse this string and extract the category values from it yourself, so that you can e.g. re-normalize them to form a (conditional) probability distribution. The variable *Distance* will contain the distance of the nearest neighbor used for classification.

`bool MBLClass::GetInstanceBase( const char * `*f*` );`
Read an instance base from file *f*.

`bool MBLClass::WriteInstanceBase( const char * `*f*` );`
Write an instance base to file *f*.

`bool MBLClass::GetWeights( const char * `*f*` );`
Read the weights to be used from file *f*.

`bool MBLClass::SaveWeights( const char * `*f*` );`
Write the currently used weights to file *f*.

`bool MBLClass::GetArrays( const char * `*f*` );`
Read the MVDM matrices from file *f*.

`bool MBLClass::WriteArrays( const char * `*f*` );`
Write the MVDM matrices to file *f*.

`bool MBLClass::WriteNamesFile( const char * `*f*` );`
Create a C4.5 namesfile *f*.

8.2.4 Setting parameters and options

Virtually all of the parameters and options that can be specified in TiMBL using command line options, can also be controlled in the classes derived from `MBLClass`. This is done with the `SetOption()` function:

`bool MBLClass::SetOption( const char * s );`

The string that is given as an argument contains a option setting command of the form:
`‘‘option : value’’`
Case does not matter, so you can achieve the same result by calling
`SetOption( "NEIGHBORS: 2" );`
or
`SetOption( "neighBors: 2" );`
The possible option setting commands, together with their permitted values are given in the table below. Many combinations or sequences of settings are not permissible, because they "don't make sense". The caller of the function is responsible for ruling out impossible combinations. The `SetOption` function will return 0 if an attempt is made to set a non-existing option or to set an existing option to a non-existing value.

The following functions are used to display option information:

`bool MBLCLass::ShowOptions();`

Shows all options with their current and possible values.

`bool MBLCLass::ShowSettings();`

Shows only the current settings of all options.

An overview of the options and their values:

Option	Values

`GLOBALMETRIC`	`Overlap`
	`VD` or `Value Difference`
	`Numeric`
	Select the global metric (only applies to IB1 and TRIBL algorithms).
	example: `SetOption( "Metric: VD" );`

`METRICS`	$i = LocalMetric$, where $LocalMetric$ is one of:
	`Overlap`
	`VD` or `Value Difference`
	`Numeric`
	`Ignore`
	Select the local metric for feature i (only applies to IB1 and TRIBL algorithms).
	example: `SetOption( "Metrics: 4=Ignore" );`

`NEIGHBORS`	n (integer) sets the number of neighbors to use (does not apply to IGTREE classifiers). example: `SetOption( "NEIGHBORS: 2" );`

`WEIGHTING`	`NOW` or `No Weighting`
	`GRW` or `GainRatio`
	`IGW` or `InfoGain`
	`X2W` or `Chi-square`
	`SVW` or `Shared Variance`
	selects the desired feature weighting scheme.
	example: `SetOption( "Weighting: InfoGain" );`

Option	Values
TRIBL_OFFSET	n (integer) only applies to the TRIBL algorithm. Sets the feature at which a transition is made from IGTREE to TRIBL. example: `SetOption( "Tribl_offset:  3" );`

TREE_ORDER	`UDO` or `Data File Ordering`
	`DO` or `Default Ordering`
	`GRO` or `GainRatio`
	`IGO` or `InformationGain`
	`1/V` or `Inverse Values`
	`G/V` or `GainRatio/Values`
	`I/V` or `InformationGain/Values`
	`1/S` or `Inverse SplitInfo`
	Changes the ordering of the features in the instance base tree.
	example: `SetOption( "TREE_ORDER: DO" );`

INPUTFORMAT	`Compact`
	`C45` or `C 4.5`
	`Columns`
	`ARFF`
	`Binary`
	Forces input to be interpreted in this format.
	example: `SetOption( "InputFormat :  ARFF" );`

FLENGTH	n (integer) If `INPUTFORMAT` is `Compact`, you have to specify how wide the feature values are (n number of characters). example: `SetOption( "FLENGTH: 2" );`
SEED	n (integer) sets the seed n for the random generator. example: `SetOption( "SEED: 123" );`
VERBOSITY	`s` – output just the minimal amount of information. `o` – give an overview of the settings. `f` – show the computed feature weights (this is the default) `p` – show value-class probabilities. `e` – show each exact matches. `di` – write the distance of the nearest neighbor to the output file. `db` – write the distribution that was used for extrapolation to the output file. `n` – write the nearest neighbors used for extrapolation to the output file. Setting combinations of levels may be achieved by using a plus sign as separator. example: `SetOption( "VERBOSITY: di+db+n" );`

Option	Values
`EXACT_MATCH`	`true` : prefer exact matches during testing. `false` : return all neighbors regardless of exact matches. example: `SetOption( "exact_match:  true" );`
`USE_INVERTED`	`true` : use inverted files when testing. `false` : don't. example: `SetOption( "use_inverted:  1" );`
`PROGRESS`	n (integer) indicates how often (number of lines) you want to see an update on the experiment's progress. example: `SetOption( "Progress:  10000" );`

Bibliography

[1] D. W. Aha. Lazy learning: Special issue editorial. *Artificial Intelligence Review*, 11:7–10, 1997.

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

[3] R. H. Baayen, R. Piepenbrock, and H. van Rijn. *The CELEX lexical data base on CD-ROM.* Linguistic Data Consortium, Philadelphia, PA, 1993.

[4] Sabine Buchholz. Distinguishing complements from adjuncts using memory-based learning. In *Proceedings of the ESSLLI-98 Workshop on Automated Acquisition of Syntax and Parsing*, 1998.

[5] Sabine Buchholz, Jorn Veenstra, and Walter Daelemans. Cascaded grammatical relation assignment. In *Proceedings of EMNLP/VLC-99, University of Maryland, USA*, pages 239–246, 1999.

[6] Claire Cardie. Automatic feature set selection for case-based learning of linguistic knowledge. In *Proc. of Conference on Empirical Methods in NLP.* University of Pennsylvania, 1996.

[7] S. Cost and S. Salzberg. A weighted nearest neighbour algorithm for learning with symbolic features. *Machine Learning*, 10:57–78, 1993.

[8] 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.

[9] W. Daelemans. Memory-based lexical acquisition and processing. In P. Steffens, editor, *Machine Translation and the Lexicon*, volume 898 of *Lecture Notes in Artificial Intelligence*, pages 85–98. Springer-Verlag, Berlin, 1995.

[10] W. Daelemans, S. Buchholz, and J. Veenstra. Memory-based shallow parsing. In M. Van Someren and G. Widmer, editors, *Proceedings of CoNLL-99, Bergen, Norway, June 12, 1999*, 1999.

[11] W. Daelemans, S. Gillis, and G. Durieux. The acquisition of stress: a data-oriented approach. *Computational Linguistics*, 20(3):421–451, 1994.

[12] W. Daelemans and A. Van den Bosch. Tabtalk: reusability in data-oriented grapheme-to-phoneme conversion. In *Proceedings of Eurospeech '93*, pages 1459–1466, Berlin, 1993. T.U. Berlin.

[13] W. Daelemans and A. Van den Bosch. Language-independent data-oriented grapheme-to-phoneme conversion. In J. P. H. Van Santen, R. W. Sproat, J. P. Olive, and J. Hirschberg, editors, *Progress in Speech Processing*, pages 77–89. Springer-Verlag, Berlin, 1996.

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

[15] W. Daelemans, A. Van den Bosch, and J. Zavrel. A feature-relevance heuristic for indexing and compressing large case bases. In M. Van Someren and G. Widmer, editors, *Poster Papers of the Ninth European Conference on Machine Learing*, pages 29–38, Prague, Czech Republic, 1997. University of Economics.

[16] Walter Daelemans, Peter Berck, and Steven Gillis. Data mining as a method for linguistic analysis: Dutch diminutives. *Folia Linguistica*, XXXI(1-2), 1997.

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

[18] Walter Daelemans, Jakub Zavrel, Peter Berck, and Steven Gillis. MBT: A memory-based part of speech tagger generator. In E. Ejerhed and I.Dagan, editors, *Proc. of Fourth Workshop on Very Large Corpora*, pages 14–27. ACL SIGDAT, 1996.

[19] P. .A. Devijver and J. Kittler. *Pattern recognition. A statistical approach.* Prentice-Hall, London, UK, 1982.

[20] S.R. Garner. WEKA: The Waikato Environment for Knowledge Analysis. In *Proc. of the New Zealand Computer Science Research Students Conference*, pages 57–64, 1995.

[21] S. Gillis, G. Durieux, and W. Daelemans. A computational model of P&P: Dresher and kaye (1990) revisited. In M. Verrips and F. Wijnen, editors, *Approaches to parameter setting*, volume 4 of *Amsterdam Studies in Child Langage Development*, pages 135–173. xxx, 1995.

[22] J. Kolodner. *Case-based reasoning.* Morgan Kaufmann, San Mateo, CA, 1993.

[23] J.R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1:81–206, 1986.

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

[25] C. Stanfill and D. Waltz. Toward memory-based reasoning. *Communications of the* ACM, 29(12):1213–1228, December 1986.

[26] A. Van den Bosch. *Learning to pronounce written words: A study in inductive language learning*. PhD thesis, Universiteit Maastricht, 1997.

[27] A. Van den Bosch and W. Daelemans. Data-oriented methods for grapheme-to-phoneme conversion. In *Proceedings of the 6th Conference of the EACL*, pages 45–53, 1993.

[28] A. Van den Bosch and W. Daelemans. Memory-based morphological analysis. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics, University of Maryland, USA*, pages 285–292, 1999.

[29] A. Van den Bosch, W. Daelemans, and A. Weijters. Morphological analysis as classification: an inductive-learning approach. In K. Oflazer and H. Somers, editors, *Proceedings of the Second International Conference on New Methods in Natural Language Processing, NeMLaP-2, Ankara, Turkey*, pages 79–89, 1996.

[30] J. Veenstra, A. Van den Bosch, S. Buchholz, W. Daelemans, and J. Zavrel. Memory-based word sense disambiguation. *Computing and the Humanities*, 34(1-2), Special issue on SENSEVAL, 2000.

[31] J.B. Veenstra. Fast NP chunking using Memory-Based Learning techniques. In *Proceedings of Benelearn'98, Wageningen, the Netherlands, available as ILK technical report ILK-98-07*, 1998.

[32] S. Weiss and C. Kulikowski. *Computer systems that learn*. San Mateo, CA: Morgan Kaufmann, 1991.

[33] A.P. White and W.Z. Liu. Bias in information-based measures in decision tree induction. *Machine Learning*, 15(3):321–329, 1994.

[34] J. Zavrel and W. Daelemans. Memory-based learning: Using similarity for smoothing. In *Proc. of 35th annual meeting of the ACL*, Madrid, 1997.

[35] J. Zavrel and W. Daelemans. Recent advances in memory-based part-of-speech tagging. In *VI Simposio Internacional de Comunicacion Social*, pages 590–597, Santiago de Cuba, 1999.

[36] J. Zavrel, W. Daelemans, and J. Veenstra. Resolving PP attachment ambiguities with memory-based learning. In Mark Ellison, editor, *Proc. of the Workshop on Computational Natural Language Learning (CoNLL'97), ACL*, Madrid, 1997.

Appendix A

Tutorial: a case study

This tutorial is meant to get you started with TiMBL quickly. We discuss how to format the data of a task to serve as training examples, which choices can be made during the construction of the classifier, how various choices can be evaluated in terms of their generalization accuracy, and various other practical issues. The reader who is interested in more background information on TiMBL implementation issues and a formal description of Memory-Based Learning, is advised to read Chapter 4.

Memory-Based Learning (MBL) is based on the idea that intelligent behavior can be obtained by analogical reasoning, rather than by the application of abstract *mental rules* as in rule induction and rule-based processing. In particular, MBL is founded in the hypothesis that the extrapolation of behavior from stored representations of earlier experience to new situations, based on the similarity of the old and the new situation, is of key importance.

MBL algorithms take a set of examples (fixed-length patterns of feature-values and their associated class) as input, and produce a *classifier* which can classify new, previously unseen, input patterns. Although TiMBL was designed with linguistic classification tasks in mind, it can in principle be applied to any kind of classification task with symbolic or numeric features and discrete (non-continuous) classes for which training data is available. As an example task for this tutorial we go through the application of TiMBL to the prediction of Dutch diminutive suffixes. The necessary data sets are included in the TiMBL distribution, so you can replicate the examples given below on your own system.

A.1 Data

The operation of TiMBL will be illustrated below by means of a real natural language processing task: prediction of the diminutive suffix form in Dutch [16]. In Dutch, a noun can receive a diminutive suffix to indicate *small size* literally or metaphorically attributed to the referent of the noun; e.g. *mannetje* means *little man*. Diminutives are formed by a productive morphological rule which attaches a form of the Germanic suffix *-tje* to the singular base form of a noun. The suffix shows variation in its form (Table A.1). The task we consider here is to predict which suffix form is chosen for previously unseen nouns on the basis of their form.

Noun	Form	Suffix
huis (house)	huisje	*-je*
man (man)	mannetje	*-etje*
raam (window)	raampje	*-pje*
woning (house)	woninkje	*-kje*
baan (job)	baantje	*-tje*

Table A.1: Allomorphic variation in Dutch diminutives.

For these experiments, we collect a representation of nouns in terms of their syllable structure as training material[1]. For each of the last three syllables of the noun, four different features are collected: whether the syllable is stressed or not (values - or +), the string of consonants before the vocalic part of the syllable (i.e. its onset), its vocalic part (nucleus), and its post-vocalic part (coda). Whenever a feature value is not present (e.g. a syllable does not have an onset, or the noun has less than three syllables), the value '=' is used. The class to be predicted is either E (-etje), T (-tje), J (-je), K (-kje), or P (-pje).

Some examples are given below (the word itself is only provided for convenience and is not used). The values of the syllabic content features are given in phonetic notation.

-	b	i	=	-	z	@	=	+	m	A	nt	J	biezenmand
=	=	=	=	=	=	=	=	+	b	I	x	E	big
=	=	=	=	+	b	K	=	-	b	a	n	T	bijbaan
=	=	=	=	+	b	K	=	-	b	@	l	T	bijbel

Our goal is to use TiMBL in order to train a classifier that can predict the class of new, previously unseen words as correctly as possible, given a set of training examples that are described by the features given above. Because the basis of classification in TiMBL is the storage of all training examples in memory, a test of the classifier's accuracy must be done on a separate test set. We will call these datasets `dimin.train` and `dimin.test`, respectively. The training set `dimin.train` contains 2999 words and the test set contains 950 words, none of which are present in the training set. Although a single train/test partition suffices here for the purposes of explanation, it does not factor out the bias of choosing this particular split. Unless the test set is sufficiently large, a more reliable generalization accuracy measurement is used in real experiments, e.g. 10-fold cross-validation [32]. This means that 10 separate experiments are performed, and in each "fold" 90% of the data is used for training and 10% for

[1] These words were collected form the CELEX lexical database [3]

testing, in such a way that each instance is used as a test item exactly once. Another reliable way of testing the real error of a classifier is leave-one-out [32]. In this approach, every data item in turn is selected once as a test item, and the classifier is trained on all remaining items. Accuracy of the classifier is then the number of data items correctly predicted. With the option `-t leave_one_out`, this testing methodology is used by TiMBL. We will use this option in the tutorial on the file `dimin.data`, the union of `dimin.train` and `dimin.test`.

A.2 Using TiMBL

Different formats are allowed for training and test data files. TiMBL is able to guess the type of format in most cases. We will use comma-separated values here, with the class as the last value. This format is called C4.5 format in TiMBL because it is the same as that used in Quinlan's well-known C4.5 program for induction of decision trees [24]. See Section 5 for more information about this and other file formats.

An experiment is started by executing TiMBL with the two files (`dimin.train` and `dimin.test`) as arguments:

```
Timbl -f dimin.train -t dimin.test
```

Upon completion, a new file has been created with name `dimin.test.mbl.O.gr.k1.out`, which is in essence identical to the input test file, except that an extra comma-separated column is added with the class predicted by TiMBL. The name of the file provides information about the MBL algorithms and metrics used in the experiment (the default values in this case). We will describe these shortly.

Apart from the result file, information about the operation of the algorithm is also sent to the standard output. It is therefore advisable to redirect the output to a file in order to make a log of the results.

```
Timbl -f dimin.train -t dimin.test > dimin-exp1
```

The defaults used in this case work reasonably well for most problems. We will now provide a point by point explanation of what goes on in the output.

```
TiMBL 3.0 (c) ILK 1998, 1999, 2000.
Tilburg Memory Based Learner
Induction of Linguistic Knowledge Research Group
Tilburg University / University of Antwerp
Tue Feb 29 15:23:16 2000

Examine datafile gave the following results:
Number of Features: 12
InputFormat       : C4.5
```

TiMBL has detected 12 features and the C4.5 input format (comma-separated features, class at the end).

```
Phase 1: Reading Datafile: dimin.train
Start:          0 @ Tue Feb 29 15:23:16 2000
Finished:    2999 @ Tue Feb 29 15:23:16 2000
Calculating Entropy         Tue Feb 29 15:23:16 2000
Lines of data     : 2999
DB Entropy        : 1.6178929
Number of Classes : 5

Feats   Vals    X-square        Variance        InfoGain        GainRatio
    1      3    128.41828       0.021410184     0.030971064     0.024891536
    2     50    364.75812       0.030406645     0.060860038     0.027552191
    3     19    212.29804       0.017697402     0.039562857     0.018676787
    4     37    449.83823       0.037499019     0.052541227     0.052620750
    5      3    288.87218       0.048161417     0.074523225     0.047699231
    6     61    415.64113       0.034648310     0.10604433      0.024471911
    7     20    501.33465       0.041791818     0.12348668      0.034953203
    8     69    367.66021       0.030648567     0.097198760     0.043983864
    9      2    169.36962       0.056475363     0.045752381     0.046816705
   10     64    914.61906       0.076243669     0.21388759      0.042844587
   11     18    2807.0418       0.23399815      0.66970458      0.18507018
   12     43    7160.3682       0.59689631      1.2780762       0.32537181

Feature Permutation based on GainRatio/Values :
< 9, 5, 11, 1, 12, 7, 4, 3, 10, 8, 2, 6 >
```

Phase 1 is the training data analysis phase. Time stamps for start and end of analysis are provided. Some preliminary analysis of the training data is done: number of training items, number of classes, entropy of the training data. For each feature, the number of values, and four variants of an information-theoretic measure of feature relevance are given. These are used both for memory organization during training and for feature relevance weighting during testing (see Chapter 4). Finally, an ordering (permutation) of the features is given. This ordering is used for building the tree-index to the case-base.

```
Phase 2: Learning from Datafile: dimin.train
Start:          0 @ Tue Feb 29 15:23:16 2000
Finished:    2999 @ Tue Feb 29 15:23:16 2000

Size of InstanceBase = 19231 Nodes, (384620 bytes), 49.77 % compression
```

Phase 2 is the learning phase; all training items are stored in an efficient way in memory for use during testing. Again timing information (real time) is provided, as well as information about the size of the data structure representing the stored examples and the amount of compression achieved.

```
Starting to test, Testfile: dimin.test
Writing output in:          dimin.test.mbl.O.gr.k1.out
Algorithm     : IB1
Global metric : Overlap (Using Instance Base, prefering exact matches)
Deviant Feature Metrics:
Weighting     : GainRatio

Tested:      1 @ Tue Feb 29 15:23:16 2000
Tested:      2 @ Tue Feb 29 15:23:16 2000
Tested:      3 @ Tue Feb 29 15:23:16 2000
Tested:      4 @ Tue Feb 29 15:23:16 2000
Tested:      5 @ Tue Feb 29 15:23:16 2000
Tested:      6 @ Tue Feb 29 15:23:16 2000
Tested:      7 @ Tue Feb 29 15:23:16 2000
Tested:      8 @ Tue Feb 29 15:23:16 2000
Tested:      9 @ Tue Feb 29 15:23:16 2000
Tested:     10 @ Tue Feb 29 15:23:16 2000
Tested:    100 @ Tue Feb 29 15:23:16 2000
Ready:     950 @ Tue Feb 29 15:23:16 2000
Seconds taken: 1 (950.00 p/s)
918/950 (0.966316), of which 39 exact matches

There were 5 ties of which 3 (60.00%) were correctly resolved
```

In Phase 3, the trained classifier is applied to the test set. Because we have not specified which algorithm to use, the default settings are used (IB1 with information theoretic feature weighting). This algorithm computes the similarity between a test item and each training item in terms of *weighted overlap*: the total difference between two patterns is the sum of the relevance weights of those features which are not equal. The class for the test item is decided on the basis of the least distant item(s) in memory. To compute relevance, Gain Ratio is used (an information-theoretic measure, see Section 4.1.2). Time stamps indicate the progress of the testing phase. Finally, accuracy on the test set is logged, and the number of exact matches[2] and ties (two or more classes are equally frequent in the nearest neighbor set). In this experiment, the diminutive suffix form of 96.6% of the new words was correctly predicted. Train and test set overlap in 39 items, and the algorithm had to break 5 ties, of which 3 correctly.

The meaning of the output file names can be explained now: `dimin.test.mbl.O.gr.k1.out` means output file (`.out`) for `dimin.test` with algorithm MBL (=IB1), similarity computed as *weighted overlap* (`.O`), relevance weights computed with *gain ratio* (`.gr`), and number of most similar memory patterns on which the output class was based equal to 1 (`.k1`).

A.3 Algorithms and Metrics

A precise discussion of the different algorithms and metrics implemented in TiMBL is given in Chapter 4. We will discuss the effect of the most important ones on our data set.

A first choice in algorithms is between using IB1 and IGTREE. In the trade-off between generalization accuracy and efficiency, IB1 usually, but not always, leads to more accuracy at the cost of more memory and slower computation, whereas IGTREE is a fast heuristic approximation of IB1, but sometimes less accurate. The IGTREE algorithm is used when `-a 1` is given on the command line, whereas the IB1 algorithm used above (the default) would have been specified explicitly by `-a 0`.

```
Timbl -a 1 -f dimin.train -t dimin.test
```

We see that IGTREE performs slightly worse than IB1 for this task (it uses less memory and is faster, however).

When using the IB1 algorithm, there is a choice of metrics for influencing the definition of similarity. With *weighted overlap*, each feature is assigned a weight, determining its relevance in solving the task. With the *modified value difference metric* (MVDM), each pair of values of a particular feature is assigned a value difference. The intuition here is that in our diminutive problem, for example, the codas n and m should be regarded as being more similar than n and p. These pair-wise differences are computed for each pair of values in each feature (see Section 4.1.4). Selection between weighted overlap and MVDM is done by means of the `-mM` parameter. The following selects MVDM, whereas `-mO` (*weighted overlap*) is the default.

```
Timbl -mM -f dimin.train -t dimin.test
```

Especially when using MVDM, but also in other cases, it may be useful to extrapolate not just from the most similar example in memory, which is the

[2]An exact match in this experiment can occur when two different nouns have the same feature-value representation.

	gain ratio	inform. gain	overlap	X2
IB1, $-k1$	96.0	96.0	87.2	95.9
IB1, $-k3$	94.8			
MVDM, $-k1$	95.9		95.3	95.9
MVDM, $-k5$	96.4		96.1	96.4

Table A.2: Some results for diminutive prediction.

default, but from several. This can be achieved by using the $-k$ parameter followed by the wanted number of nearest neighbors. E.g., the following applies IB1 with the MVDM metric, with extrapolation from the 5 nearest neighbors.

```
Timbl -mO -k 5 -f dimin.train -t dimin.test
```

Within the IB1 *weighted overlap* option, the default feature weighting method is Gain Ratio. Other feature relevance weighting methods are available as well. By setting the parameter `-w` to 0, an *overlap* definition of similarity is created where each feature is considered equally relevant. Similarity reduces in that case to the number of equal values in the same position in the two patterns being compared. As an alternative weighting, users can provide their own weights by using the `-w` parameter with a filename in which the feature weights are stored (see Section 5.2 for a description of the format of the weights file).

Table A.2 shows the effect of algorithm, metric, and weighting method choice on generalization accuracy using leave-one-out as experimental method.

```
Timbl -t leave_one_out -f dimin.data
```

When comparing MVDM and IB1, we see that the overall best results are achieved with MVDM, but only with a higher value for k, the number of memory items (actually distances) on which the extrapolation is based. Increasing the value of k for (weighted) Overlap metrics decreased performance. Within the feature weighting approaches, overlap (i.e. no weighting) performs markedly worse than the default *information gain* or *gain ratio* weighting methods.

A.4 More Options

Several input and output options exist to make life easier while experimenting. See Chapter 7 for a detailed description of these options. One especially useful option for testing linguistic hypotheses is the ignore option, which allows you to skip certain features when computing similarity. E.g. if we want to test the hypothesis that only the rime (nucleus and coda) and the stress of the last syllable are actually relevant in determining the form of the diminutive suffix, we can execute the following to disregard all but the fourth-last and the last two features. As a result we get an accuracy of 97.2%[3].

[3] It should be kept in mind that the amount of overlap in training and test set has significantly increased, so that generalization is based on retrieval more than on similarity computation.

```
Timbl -mM:I1-8,10 -f dimin.data -t leave_one_out -k5 -w1
```

The +/-V (verbosity) option allows you to control the amount of information that is generated in the output, ranging from nothing much (+V S) to a lot (+V O+P+E+DI+DB+N). Specific verbosity settings exist for dumping option settings (+V O), feature relevance weights (default), value-class conditional probabilities (+V P), exact matches (+V E), distributions (+V DB), and the nearest neighbors on which decision are based (+V N) or the distances to the nearest neighbor (+V DI). E.g. the following command results in an output file with distributions.

```
Timbl +v db -f dimin.train -t dimin.test
```

The resulting output file contains lines like the following.

```
=,=,=,=,+,=,K,=,-,s,A,k,J,J { J 1 }
-,r,i,=,+,j,o,=,-,d,@,=,T,T { T 4 E 1 J 1 }
+,vr,a,=,-,G,@,=,-,l,K,st,J,J { J 1 }
=,=,=,=,=,=,=,=,+,b,o,m,P,P { P 3 }
=,=,=,=,+,v,E,s,-,t,I,N,K,E { E 1 }
-,t,@,=,-,G,@,=,-,z,I,xt,J,J { J 1 }
=,=,=,=,=,=,=,=,+,b,E,l,E,E { E 10 }
+,k,o,=,-,l,i,=,-,br,i,=,T,T { T 1 }
=,=,=,=,+,sx,I,=,-,m,@,l,T,T { T 12 }
=,=,=,=,=,=,=,=,+,kl,M,n,T,T { T 43 E 20 }
```

This information can e.g. be used to assign a certainty to a decision of the classifier, or to make available a second-best back-off option.

```
Timbl +v di -f dimin.train -t dimin.test
```

```
+,l,a,=,-,d,@,=,-,k,A,st,J,J        0.070701
-,s,i,=,-,f,E,r,-,st,O,k,J,J        0.000000
=,=,=,=,=,=,=,=,+,sp,a,n,T,T        0.042845
=,=,=,=,=,=,=,=,+,st,o,t,J,J        0.042845
=,=,=,=,+,sp,a,r,-,b,u,k,J,J        0.024472
+,h,I,N,-,k,@,l,-,bl,O,k,J,J        0.147489
-,m,e,=,-,d,A,l,+,j,O,n,E,E        0.182421
-,sn,u,=,-,p,@,=,+,r,K,=,T,T        0.046229
=,=,=,=,=,=,=,=,+,sp,A,N,E,E        0.042845
+,k,a,=,-,k,@,=,-,n,E,st,J,J        0.114685
```

This can be used to study how specific instances (low distance) and more general patterns (higher distance) are used in the process of generalization.

The listing of nearest neighbors is useful for the analysis of the behavior of a classifier.

```
Timbl +v n -f dimin.train -t dimin.test
```

```
-,t,@,=,-,l,|,=,-,G,@,n,T,T
# k=1, 1 Neighbor(s) at distance: 0.0997233
#       -,x,@,=,+,h,|,=,-,G,@,n, -*-
-,=,I,n,-,str,y,=,+,m,E,nt,J,J
# k=1, 1 Neighbor(s) at distance: 0.123322
#       -,m,o,=,-,n,y,=,+,m,E,nt, -*-
=,=,=,=,=,=,=,=,+,br,L,t,J,J
# k=1, 4 Neighbor(s) at distance: 0.0428446
#       =,=,=,=,=,=,=,=,+,r,L,t, -*-
#       =,=,=,=,=,=,=,=,+,kr,L,t, -*-
#       =,=,=,=,=,=,=,=,+,sx,L,t, -*-
#       =,=,=,=,=,=,=,=,+,fl,L,t, -*-
=,=,=,=,+,zw,A,=,-,m,@,r,T,T
# k=1, 5 Neighbor(s) at distance: 0.0594251
#       =,=,=,=,+,fl,e,=,-,m,@,r, -*-
#       =,=,=,=,+,=,E,=,-,m,@,r, -*-
#       =,=,=,=,+,l,E,=,-,m,@,r, -*-
#       =,=,=,=,+,k,a,=,-,m,@,r, -*-
#       =,=,=,=,+,h,O,=,-,m,@,r, -*-
```

We hope that this tutorial has made it clear that, once you have coded your data in fixed-length feature-value patterns, it should be relatively straightforward to get the first results using TiMBL. You can then experiment with different metrics and algorithms to try and further improve your results.