TiMBL: Tilburg Memory-Based Learner

version 6.2

Reference Guide

ILK Technical Report – ILK 09-01

Walter Daelemans* Jakub Zavrel*† Ko van der Sloot
Antal van den Bosch

Induction of Linguistic Knowledge Research Group
Tilburg centre for Creative Computing
Tilburg University

(*) CLiPS - Language Technology Group
Department of Linguistics
University of Antwerp

(†) Textkernel B.V.

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

October 19, 2009

[1]This document is available from http://ilk.uvt.nl/downloads/pub/papers/ilk.0901.pdf. All rights reserved Induction of Linguistic Knowledge, Tilburg University and CLiPS, University of Antwerp.

Contents

Preface

Memory-Based Learning (MBL) is an elegantly simple and robust machine-learning method applicable to a wide range of tasks in Natural Language Processing (NLP). In our research group at Tilburg University, we have been working since the end of the 1980s on the development of Memory-Based Learning techniques and algorithms. The foundations are bundled in **?**). Section 5.7 provides a historical overview of work on the application of MBL in NLP. With the establishment of the ILK (Induction of Linguistic Knowledge) research group in 1997, and with the increasing use of MBL at the CNTS research group of the University of Antwerp, the need for a well-coded and uniform tool for our main algorithms became more urgent. TiMBL was 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 has become a useful tool for NLP research, and, for that matter, for many other domains where classification tasks are learned from examples, so we started to release the software in 1999. With the release of the sixth version of TiMBL we moved to releasing our software under the GPL license, for anyone to use under the conditions stated in the license.

Memory-Based Learning is a direct descendant of the classical k-Nearest Neighbor (k-NN) approach to classification, which has become known as a powerful pattern classification algorithm for numeric data. 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 demand non-trivial data-structures and speedup optimizations for the core k-NN classifier. Our approach has resulted in an architecture which compresses the typical flat file organization found in straightforward k-NN implementations, into a decision-tree structure. While the decision tree can be used to retrieve the exact k-nearest neighbors (as happens in the IB1 algorithm within TiMBL), it can also be deterministically traversed as in a decision-tree classifier (the method adopted by the IGTREE algorithm). We believe that our optimizations make TiMBL one of the fastest discrete k-NN implementations around.

The main effort in the development and maintenance of this software was and continues to be invested 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 of IB1-IG. Some of the index optimizations in TiMBL are due to Jakub Zavrel. The code has benefited substantially from trial, error and scrutiny by all past and present members of the ILK and CLiPS (formerly CNTS) groups in Tilburg and Antwerp. We are furthermore indebted to Ton Weijters of Eindhoven Technical University for his inspirational early work on k-NN and for his involvements in IGTREE.

Our sincere thanks go to the many users of TiMBL who have contributed to it immensely by giving us feedback and reporting bugs, and to the two organisations that have supported and enabled its development: NWO, the Netherlands Organization for Scientific Research, and the Faculty of Humanities of Tilburg University. NWO funding has spanned three subsequent periods. From 1997 until 2001 development was part of the "Induction of Linguistic Knowledge" research programme, partially funded by the Netherlands Organization for Scientific Research (NWO) and Tilburg University. Between 2001 and 2006 it was funded as part of the "Memory Models of Language" research project under the NWO *Vernieuwingsimpuls* programme, and since 2006 it is funded as part of the "Implicit Linguistics" research project under the NWO Vici programme.

The current release (version 6.2) succeeds major release 6.0 and minor update 6.1. It provides several new distance functions, and the internal code has been overhauled significantly. An elaborate description of the changes from version 1.0 up to 6.2 can be found in Chapter 3. Although all new features have been tested for some time in our research groups, 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@uvt.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. Next, Chapter 4 offers a quick-start tutorial for readers who want to get to work with TiMBL right away. The tutorial describes, step-by-step, a case study with a sample data set (included with the software) representing the linguistic domain of predicting the diminutive inflection of Dutch nouns. Readers who are interested in the theoretical and technical details of Memory-Based Learning and of this implementation can refer to Chapter 5. Chapter 6 provides full reference to the command line options of TiMBL, supported file formats, and TiMBL's server interface.

Chapter 1

GNU General Public License

In publication of research that makes use of TiMBL 6.2, a citation should be given of: *"Walter Daelemans, Jakub Zavrel, Ko van der Sloot, and Antal van den Bosch (2009). TiMBL: Tilburg Memory Based Learner, version 6.2, Reference Guide. ILK Technical Report 09-01 Available from* `http://ilk.uvt.nl/downloads/pub/papers/ilk0901.pdf`*"*

For information about commercial licenses for TiMBL 6.2, contact `Timbl@uvt.nl`, or send your request in writing to:

Prof. dr. Walter Daelemans
CLiPS - Language Technology Group
Dept. of Linguistics
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.uvt.nl/timbl`

Following the links from that page, you can download the file `timbl-6.2.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 (> is the command line prompt):

```
> tar xfz timbl-6.2.tar.gz
```

This will make a directory `timbl-6.2` under your current directory.

Alternatively you can do:

```
> gunzip timbl-6.2.tar.gz
```

and unpack the tar archive:

```
> tar xf timbl-6.2.tar
```

Go to the timbl-6.1 directory, and configure the package by typing

```
> cd timbl-6.2
> ./configure --prefix=<location_to_install>
```

If you don't use the `--prefix` option, TiMBL will try to install itself in the directory `/usr/local/`. If you don't have `root` access you can specify a different installation location such as `$HOME/install`

It's not obligatory to install TiMBL, but if you plan to install TiMBL-based extensions such as Mbt[1], Dimbl[2], or Tadpole[3], or you want to build your own extensions using the TiMBL API, installing is the best choice.

After `configure` you can build TiMBL:

```
> make
```

[1]`http://ilk.uvt.nl/mbt`
[2]`http://ilk.uvt.nl/dimbl`
[3]`http://ilk.uvt.nl/tadpole`

and (as recommended) install:

```
> make install
```

If the process was completed successfully, you should now have executable files named `Timbl` and `TimblClient` in the directory `<location_to_install>/bin`, and a static library `libTimbl.a` in the directory `<location_to_install>/lib`. Also some demo programs named `api_test*`, `classify` and `tse` are created in the `./demos` subdirecory of the current directory.

Within the `<location_to_install>` directory a subdirectory is also created: `share/doc/timbl` where the TiMBL 6.2 documentation can be found, and which in turn contains a subdirectory `examples` with example data files. Some of these data sets are used in the Quick Start Section 4 of this document; other data and source files are referred to in the API documentation. The latter, along with a pdf version of this document, can also be found in the `doc` directory. Note that the API documentation is a beta-state document.

TiMBL should now be ready for use. If you want to run the examples and demos from this manual, you should act as follows:

- Be sure to add `<location_to_install>/bin` to your PATH. In many shells something like

  `> export PATH=$PATH:<location_to_install>/bin` will do.

- copy all the files from `<location_to_install>/share/doc/timbl/examples` to some working location. (By default, TiMBL writes its results to the directory where it finds the data.)

- and test:

  `cd` to the working location, and then

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

If you did not install TiMBL, the executable can be found in the `src` directory of the build. The demo files can be found in the `demo` directory.

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

Chapter 3

Changes

This chapter gives a brief overview of the changes from all previously released versions (1.0 up to 6.2) for users already familiar with the program.

3.1 From version 6.1 to 6.2

Version 6.2 differs from 6.1 in a great number of internal changes aimed at making the code better maintainable and extendible, in some minor bug fixes, and in the following more prominent changes:

- A new distance metric, the Dice coefficient, has been added; the metric can be set with `-mDC`. Analogous to the Levenshtein (`-mL`) metric, the Dice coefficient operates at the feature value level; it computes the overlap in character bigrams of two value strings.
- Value difference matrices, as used by the MVDM and Jeffrey divergence distance metrics, can now be written to file, and read into TiMBL, allowing for user-defined value difference metrics to be used, with `--matrixout=<filename>` and `--matrixin=<filename>`.
- The IGTREE algorithm has been optimized beyond the improvements introduced in version 6.0. With very large training sets, IGTREE was reported to be exponentially slower in the later stages of training. Trees are now built in near-linear time.

Finally, besides minor bug fixes, a great number of internal changes were made to make the code better maintainable and extendible.

3.2 From version 6.0 to 6.1

Version 6.1 differs from 6.0 mainly in the changed configuration. It is now based on autotools and is delivered as an installable package. Some bugs have been fixed as well.

3.3 From version 5.1 to 6.0

Version 6.0 differs from 5.1 firstly in terms of internal changes aimed at increasing classification speed and lowering memory usage, of which the most prominent are

- The IGTREE algorithm has been optimized. Learning has been made more memory-lean, while classification has been optimized so that it is now orders of magnitude faster than before on most data sets.
- MVDM matrices are partly prestored; only the MVDM values of pairs of frequent values are precomputed. The threshold frequency n can now be determined with `-c n`. This way memory can be traded for speed, up to a point. The default value 10 remains the recommended one.

Also, two metrics and several verbosity options and other command-line switches are added:

- Two distance metrics are added: `-mC` sets the Cosine metric, and `-mL` sets the Levenshtein metric. The latter metric operates at the feature-value level, and thus offers an alternative to the all-or-nothing Overlap metric for string-valued features. The existing `-L n` option (which sets the value frequency threshold at which the MVDM distance metric backs off to the Overlap metric) is extended to `-L L:n` to denote that MVDM should backoff to the Levenshtein metric rather than to Overlap. [1]
- Class distribution output generated with `+v db` can be normalized so that they add to 1.0, with the additional `-G` option (or `-G0`). As a simple smoothing option, with `-G1:double` all class votes are incremented by `double` before normalization. For example, `-G1:1` (or `-G1` for short) is "add one"-smoothing; `-G1:0.5` adds 0.5 to all class votes.
- With `-Beam=n`, where n is an integer, the `+v db` output is constrained to the n classes receiving the highest votes. This special limit is useful in cases in which the `+v db` output, typically used for further processing, generates far too much output in its default unconstrained setting. [2]
- Class distributions are not stored on non-terminal nodes with IGTREE and TRIBL by default. To revert this default, e.g. to be able to use `+v db` with IGTREE, the setting `+D` can be used.
- With `-T n`, the user can specify that the nth column in the training set of labeled examples contains the label to be predicted, while all other columns represent the input features. By default, the last column is assumed to contain the class labels.
- After classification, TiMBL reports its classification speed at microsecond precision instead of in seconds.
- The verbosity option `+v md` displays the level at which a classification was made by IGTREE (`-a1`), and whether the class label was obtained from a leaf node or an end node.
- With `-X [file]`, TiMBL dumps its internal TiMBL tree into a file containing an XML tree. This option is analogous to `-I [file]`, which prints a TiMBL tree in TiMBL's proprietary format, the difference being that the latter format can be read into TiMBL again.
- Several minor bugs have been resolved.

[1] KO: this backoff mechanism is removed in Timbl 6.2

[2] KO: this option is renamed to `--Beam` in Timbl version 6.2

3.4 From version 5.0 to 5.1

Version 5.1 adds speed and memory improvements that are notable with datasets that have very large amounts of examples, features, feature values, or classes (and, especially, combinations of those). Previous versions exhibited exponential slowdown in some worst cases; this has been largely countered. On the outside, TiMBL has been updated in the following aspects:

- TiMBL offers extended performance reporting: next to accuracy it reports on micro and macro-averages of F-score and AUC (area under the ROC-curve) with `+v as`. Optionally, it also shows each individual class' precision, recall (or true positive rate), and false positive rate with `+v cs`.
- TiMBL always uses gain ratio feature weighting as the default case, if not specified by the user, also with the MVDM and Jeffrey Divergence similarity metrics.
- Two additional feature orderings for the internal TiMBL trees are added, `-TGxE` and `-TIxE` (gain ratio $\times$ entropy and information gain $\times$ entropy, respectively) to potentially tackle the problem of unbalanced trees.
- Bugs in leave-one-out testing with numeric features and with exemplar weighting were fixed.

3.5 From version 4.3 to 5.0

Version 5.0 is the conclusion of a number of recodings (mostly involving more generic treatment of variables to improve robustness, but also the removal of inverted indexing on the internal tree representation) that have changed the internals of TiMBL considerably. On the outside, TiMBL displays the following new characteristics:

- Next to the Overlap, MVDM, and Numeric distance functions, TiMBL now features the Jeffrey divergence distance function and the Dot-product distance function.
- The exponential-decay distance weighting function can be set using a second parameter, which can change the shape of the function from normal exponential to bell-shaped.
- In addition to the "binary" format, TiMBL can now read a more generic sparse data format. This format allows instances to be coded by tuples of $<$ feature number, feature value $>$ where the value can be symbolic or numeric rather than only binary.
- Tree files generated by TiMBL versions 1.*, 2.* and 3.* are no longer supported.
- The command line interface has had the following additions, including the ones reflecting the above changes:
  - `-m J` activates the Jeffrey divergence distance metric.
  - `-m D` activates the Dot-product distance metric.
  - `-d ED:<a>:<b>` (without whitespace) sets the α and new β parameters. If unspecified, as in `-d ED:<a>` or the older (deprecated) `-d ED <a>`, β is set to 1.0.
  - `-F Sparse` declares that training and test files are in the sparse $<$ feature number, feature value $>$ tuple-format described in more detail in section 6.1.
  - `+v k` is a new verbosity option that prints all class distributions per k-nearest distance per classified instance in the output file. It works analogous to the `+v n` option, but does not print the neighbors themselves.

3.6 From version 3.0 to 4.3

As the last upgrade of the version 4 strain, version 4.3 added some command line functionality and internal code changes to version 4.2. Minor progressive changes from 4.0 to 4.3 are found at the bottom of this list and are marked as such.

- Distance weighting of the k nearest neighbors. This classical exemplar weighting scheme (**?**) allows closer nearest neighbors in the k to have a more prominent vote in classification. TiMBL incorporates linear, inversed, and exponential distance weighting.
- Incremental edited memory-based learning with IB2 (**?**). This incremental version of IB1 adds instances to memory only when those instances are misclassified by the then-current set of instances in memory.
- Frequency-filtered MVDM distance metric. The option, which is not selected by default, is an add-on of the MVDM metric, that backs off from the MVDM metric to the Overlap distance function whenever one or both in a pair of matched values occurs fewer times in the training material than a user-determined threshold.
- TRIBL2. The TRIBL2 algorithm has been implemented as an additional trade-off between IGTREE and IB1. In contrast to TRIBL, TRIBL2 uses no threshold parameter.
- Exemplar weighting. TiMBL can read additional numeric exemplar weights (generated externally) when reading a data file, and use these weights during neighbor distance computation in k-NN classification.
- Cross-validation testing. Analogous to the leave-one-out testing option, with cross-validation testing it is possible to let TiMBL run systematic tests on different values of parameters, without completely re-initializing the classifier in every fold of the validation experiment.
- The number of concurrent connections to a TiMBL server has been restricted, but can be set to different values.
- The command line interface has had several additions reflecting the above changes, plus one extra verbosity option:
  - the `-d metriccode` option sets the distance weighting metric. Three metrics are available: inverse distance (code ID), inverse linear (IL), and exponential decay (ED, which takes an extra argument a, without whitespace, determining the factor of the exponential function). By default, no distance weighting is used (code Z). See Chapter 5 for descriptions.
  - the `-L n` option sets the frequency threshold in the optional switch (backoff) from MVDM or Jeffrey divergence to Overlap; whenever in an MVDM or Jeffrey divergence distance computation one or both of a pair of values occur fewer than `n` times, Overlap is used rather than the MVDM metric. The default value for `n` is 1 (no switching). [3]
  - the `-a 3` or `-a IB2` switch invokes the IB2 algorithm. This algorithm expects to have the `-b` switch set.
  - the `-b n` option sets the number (n) of lines counting from the top of the training set file, which form the bootstrap set of memorized instances to which IB2 will start adding instances incrementally.
  - the `-a 4` or `-a TRIBL2` switch invokes the TRIBL2 algorithm.
  - the `-C n` switch (default: `n` set to 10) restricts the number of concurrent connections to a TiMBL server (cf. the `-S` switch).

[3] KO: Backoff is removed in 6.2

  - the `+v/-v` option has `cm` as a new optional argument; it returns the confusion matrix, obtained after testing, between predicted and actual classes in the test data.

- The "programmer's reference" or API section has been separated from this manual. This new API, describing the underlying structure of TiMBL, is available as a separate document in the TiMBL software distribution.
- Two bugs relating to a type of sparse data problem have been resolved. The first involved leave-one-out experiments on data sets with features that have values that occur only once in the training data. The second bug occurred with the use of the `-F Binary` option with the same type of data.
- **[4.1]** Exemplar weights are stored in the TiMBL-tree.
- **[4.1]** The core representation of TiMBL-trees has been modified, causing no changes at the surface except that the TRIBL variant uses less memory.
- **[4.2]** Feature value and class information in the internal TiMBL tree is hashed, by default, except with binary features. Hashing can be explicitly set on or off through the flag `+H` or `-H`.
- **[4.2]** The discretization of numeric features, used for computing feature weights, has changed from linear binning between minimum and maximum values, to equal-content binning.
- **[4.2]** Tie resolution between equal class distributions in the nearest neighbors set is resolved by first expanding the k by one value. If the tie persists after the enlargement of the nearest neighbor set, the original tie resolution method is applied.
- **[4.3]** Internal changes in the code (with no effect on learning and classification functionality) have been implemented with respect to namespaces.
- **[4.3]** A progress marker (one dot per 10 seconds) in computationally intensive operations on the internal representation of the instance base (e.g. pruning IGTREEs) is added in TiMBL's screen output.
- A number of bugs have been fixed, notably to handle erroneous input more robustly.

3.7 From version 2.0 to 3.0

- Server functionality. Apart from the standard processing of test items from a file, alternatively 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 6.3.
- 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 6.2.1.

- 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.

3.8 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 5.4, or ?)). 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.
- 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 6.2.3).
- 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

Quick-start Tutorial

This quick-start tutorial is meant to get you started with TiMBL right away. 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 5.

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.

4.1 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 (**?**). 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 4.1). The task we consider here is to predict which suffix form is chosen for previously unseen nouns on the basis of their form.

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

[1]These words were collected form the CELEX lexical database (**?**).

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

Table 4.1: Allomorphic variation in Dutch diminutives.

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 (?). This means that 10 separate experiments are performed, and in each "fold" 90% of the data is used for training and 10% for 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 (?). 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`.

4.2 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 (?). See Section 6.2 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 (">" is the command line prompt):

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

Upon completion, a new file has been created with name `dimin.test.IB1.O.gr.k1.out`, which is 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 6.1 (release) (c) ILK 1998 - 2007.
Tilburg Memory Based Learner
Induction of Linguistic Knowledge Research Group
Tilburg University / University of Antwerp
Sat Feb 17 23:36:06 2007

Examine datafile 'dimin.train' 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 @ Sat Feb 17 23:36:06 2007
Finished:    2999 @ Sat Feb 17 23:36:06 2007
Calculating Entropy         Sat Feb 17 23:36:06 2007
Lines of data     : 2999
DB Entropy        : 1.6178929
Number of Classes : 5

Feats   Vals    InfoGain        GainRatio
    1      3    0.030971064     0.024891536
    2     50    0.060860038     0.027552191
    3     19    0.039562857     0.018676787
    4     37    0.052541227     0.052620750
    5      3    0.074523225     0.047699231
    6     61    0.10604433      0.024471911
    7     20    0.12348668      0.034953203
    8     69    0.097198760     0.043983864
    9      2    0.045752381     0.046816705
   10     64    0.21388759      0.042844587
   11     18    0.66970458      0.18507018
   12     43    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 5). 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 @ Sat Feb 17 23:36:06 2007
Finished:    2999 @ Sat Feb 17 23:36:06 2007

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

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

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.IB1.O.gr.k1.out
Algorithm     : IB1
Global metric : Overlap
Deviant Feature Metrics:(none)
Weighting     : GainRatio
Feature 1        : 0.024891535617620
Feature 2        : 0.027552191321752
Feature 3        : 0.018676787182524
Feature 4        : 0.052620750282779
Feature 5        : 0.047699230752236
Feature 6        : 0.024471910753751
Feature 7        : 0.034953203413051
Feature 8        : 0.043983864437713
Feature 9        : 0.046816704745507
Feature 10       : 0.042844587034556
Feature 11       : 0.185070180760327
Feature 12       : 0.325371814230901

Tested:      1 @ Sat Feb 17 23:36:06 2007
Tested:      2 @ Sat Feb 17 23:36:06 2007
Tested:      3 @ Sat Feb 17 23:36:06 2007
Tested:      4 @ Sat Feb 17 23:36:06 2007
Tested:      5 @ Sat Feb 17 23:36:06 2007
Tested:      6 @ Sat Feb 17 23:36:06 2007
Tested:      7 @ Sat Feb 17 23:36:06 2007
Tested:      8 @ Sat Feb 17 23:36:06 2007
Tested:      9 @ Sat Feb 17 23:36:06 2007
Tested:     10 @ Sat Feb 17 23:36:06 2007
Tested:    100 @ Sat Feb 17 23:36:06 2007
Ready:     950 @ Sat Feb 17 23:36:06 2007
Seconds taken: 0.0539 (17627.52 p/s)
overall accuracy:        0.968421  (920/950), of which 39 exact matches

There were 5 ties of which 5 (100.00%) 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 5.1.2). 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 96.7% of the new words was correctly predicted. Train and test set overlap in 39 items, and the algorithm had to break five ties, all of which were broken correctly.

The meaning of the output file names can be explained now:
`dimin.test.IB1.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`).

4.3 Algorithms and metrics

A precise discussion of the different algorithms and metrics implemented in TiMBL is given in Chapter 5. 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 only slightly worse (96.6%) than IB1 (96.7%) for this train-test partitioning of the data — 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 pairwise differences are computed for each pair of values in each feature (see Section 5.1.4). 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 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
```

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

	no weight (overlap)	gain ratio	information gain	chi squared	inverse linear
IB1, $-k1$	86.4	96.7	96.6	96.6	
IB1, $-k3$	73.1	96.4	96.8	96.9	96.6
MVDM, $-k1$	95.8	96.3	96.1	96.2	
MVDM, $-k5$	**97.8**	97.7	97.7	97.7	96.6

Table 4.2: Some results for diminutive prediction.

Whenever more than one nearest neighbor is taken into account for extrapolation, it may be useful to weigh the influence of the neighbors on the final decision as a function of their distance from the test item. Several possible implementations of this distance function are provided. E.g., the following provides inverse distance:

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

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. In that case, similarity reduces 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 6.2.2 for a description of the format of the weights file).

Table 4.2 shows the effect of algorithm, metric, distance weighting of nearest neighbors, and weighting method choice on generalization accuracy using leave-one-out as experimental method. The results show that no feature weighting may lead to excellent performance, in this case in combination with MVDM and $k = 5$.

```
> 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 *information gain*, *gain ratio* or *chi-square* weighting methods.

The default settings provided in TiMBL were selected on the basis of our experience with a large set of (mostly linguistic) datasets. However, as can be seen from this dataset, they are not guaranteed to be the best choice. It is useful to try out a large set of reasonable combinations of options by cross-validation on the training data to achieve best results with MBL. The option `-t @f` where `f` is the name of a file, allows you to predefine various combinations of options to be tested and test them without having the training stages repeated each time. See Chapter 6.1.

4.4 More options

Several input and output options exist to make life easier while experimenting. See Chapter 6.1 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 with the previously best parameter settings to disregard all but the fourth-last and the last two features. As a result we get an accuracy of 97.0%[3].

```
> Timbl -mM:I1-8,10 -f dimin.data -t leave_one_out -k5 -w3
```

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+cm+di+db+n+k`). 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`), a confusion matrix (`+v cm`), advanced statistics besides accuracy: micro-average and macro-average F-score and AUC (`+v as`), per-class advanced statistics (`+v cs`), the nearest neighbors on which decision are based (`+v n`), just the class distributions per k-nearest distance per classified instance (`+v k`), 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 `dimin.test.IB1.O.gr.k1.out` contains lines like the following.

```
+,t,L,=,-,m,@,=,-,l,I,N,E,E { E 1.00000 }
=,=,=,=,=,=,=,=,+,pr,O,p,J,J { E 3.00000, J 12.0000 }
=,=,=,=,=,=,=,=,+,w,e,t,J,J { J 2.00000 }
=,=,=,=,+,t,L,n,-,h,L,s,J,J { J 1.00000 }
=,=,=,=,=,=,=,=,+,t,L,n,T,T { T 1.00000 }
=,=,=,=,=,=,=,=,+,z,o,m,P,P { P 3.00000 }
+,d,a,=,-,m,@,s,-,kr,A,ns,J,J { J 1.00000 }
=,=,=,=,+,=,a,rd,-,m,A,n,E,E { E 2.00000 }
=,=,=,=,=,=,=,=,+,f,M,n,T,T { T 43.0000, E 20.0000 }
-,d,u,=,-,k,@,=,-,m,A,nt,J,J { J 1.00000 }
```

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 very similar instances (low distance) and less similar 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. It can be used to interpret why particular decisions or errors occur.

[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 +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, -*-
```

A confusion matrix, printed when the `+v cm` option is selected, can bring to light specific errors of the classifier that would not be apparent from the overall accuracy. Applied to the diminutive data, the following confusion matrix is computed and printed:

```
> Timbl +v cm -f dimin.train -t dimin.test

Confusion Matrix:
             T      E      J      P      K
        -----------------------------------
     T |    453      0      2      0      0
     E |      0     87      4      1      8
     J |      1      5    346      0      0
     P |      0      3      0     24      0
     K |      0      7      0      0      9
   -*- |      0      0      0      0      0
```

The confusion matrix associates the class predicted by TiMBL (vertically) with the real class of the test items given (horizontally). All cells outside the diagonal contain errors of one class being mistaken for another. For example, the K class (*-kje*) is mispredicted seven times as class E (*-etje*), which is an often-made mistake that can only be disambiguated properly when the stress of the noun is known: compare, for example, *delinkje* (little division) versus *wandelingetje*.

(The bottom line, labeled with `-*-`, would contain aggregate counts of classes occuring in the test data that did not occur in the training data. In the diminutive data this does not occur.)

In general, a confusion matrix allows a more fine-grained analysis of experimental results and better experimental designs (some parameter settings may work for some classes but not for others, or some may improve recall, and others precision, e.g.). From such a matrix, not only accuracy can be derived, but also a number of additional metrics that have become popular in machine learning, information retrieval, and subsequently also in computational linguistics: *recall*, *precision*, and their harmonic mean *F-score*, as well as *true positive rate*, *false positive rate*, and their joint measure *AUC* in ROC space. The details of these advanced statistics are given in Section 5.6.

They can be reported by TiMBL using the `+v as` and `+v cs` verbosity options:

```
> Timbl +v as+cs -f dimin.train -t dimin.test
```

```
Scores per Value Class:
class TP    FP    TN    FN   precision   recall(TPR) FPR         F-score     AUC
T |   453   1     494   2    0.997797    0.995604    0.002020    0.996700    0.996792
E |   87    15    835   13   0.852941    0.870000    0.017647    0.861386    0.926176
J |   346   6     592   6    0.982955    0.982955    0.010033    0.982955    0.986461
P |   24    1     922   3    0.960000    0.888889    0.001083    0.923077    0.943903
K |   9     8     926   7    0.529412    0.562500    0.008565    0.545455    0.776967

F-Score beta=1, microav: 0.967160
F-Score beta=1, macroav: 0.861914
AUC, microav:            0.980138
AUC, macroav:            0.926060
overall accuracy:        0.967368  (919/950), of which 39 exact matches
```

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.

Chapter 5

Memory-based learning algorithms

TiMBL is a program implementing several memory-based learning algorithms. All implemented 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 among 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 5.1 with a formal description of the basic memory-based learning algorithm, i.e. a nearest neighbor search. We then introduce different distance 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, and describe the standard versus three distance-weighted versions of the class voting mechanism of the nearest neighbor classifier. In Section 5.2, we give a description of various algorithmic optimizations for nearest neighbor search.

Sections 5.3 to 5.5 describe three variants of the standard nearest neighbor classifier implemented within TiMBL, that optimize some intrinsic property of the standard algorithm. First, in Section 5.3, we describe IGTREE, which replaces the exact nearest neighbor search with a very fast heuristic that exploits the difference in importance between features. Second, in Section 5.4, we describe the TRIBL algorithm, which is a hybrid between IGTREE and nearest neighbor search. Third, Section 5.5 describes the IB2 algorithm, which incrementally and selectively adds instances to memory during learning.

The chapter is concluded by Section 5.7, which provides an overview of further reading into theory and applications of memory-based learning to natural language processing tasks.

5.1 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 (**?**; **?**; **?**; **?**; **?**). Historically, memory-based learning algorithms are descendants of the k-nearest neighbor (henceforth k-NN) algorithm (**?**; **?**; **?**).

An MBL system, visualized schematically in Figure 5.1, 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 some *distance metric* $\Delta(X, Y)$. The extrapolation is done by assigning the most frequent category within the found set of most similar example(s) (the k-nearest neighbors) as the category of the new test example. In case of a tie among categories, a tie breaking resolution method is used. This method is described in subsection 5.1.7.

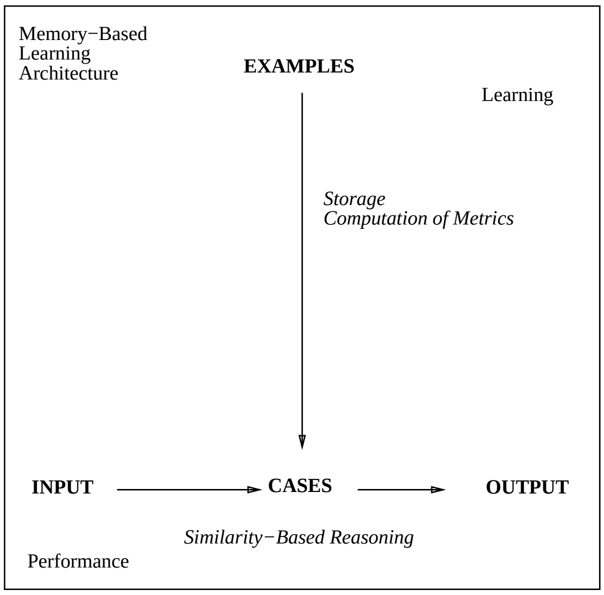

Figure 5.1: General architecture of an MBL system.

5.1.1 Overlap and Levenshtein metrics

The most basic metric that works for patterns with symbolic features is the **Overlap metric**[1] given in Equations 5.1 and 5.2; where $\Delta(X, Y)$ is the distance between instances 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 (**?**).

$$\Delta(X,Y) = \sum_{i=1}^{n} \delta(x_i, y_i) \tag{5.1}$$

where:

$$\delta(x_i, y_i) = \begin{cases} abs(\frac{x_i - y_i}{max_i - min_i}) & \text{if numeric, else} \\ 0 & \text{if } x_i = y_i \\ 1 & \text{if } x_i \neq y_i \end{cases} \tag{5.2}$$

The major difference with the IB1 algorithm originally proposed by (**?**), is that in our version the value of k refers to k-nearest *distances* rather than k-nearest examples. With $k = 1$, for instance, TiMBL's nearest neighbor set can contain several instances that are equally distant to the test instance. Arguably, our k-NN kernel could therefore be called k-nearest distances classification.

Another difference with the original IB1 as well as with other implementations such as k-NN in the WEKA machine learning toolkit (**?**) is the way in which ties are resolved in choosing the majority category among the set of nearest neighbors. Since this method is independent of the distance function we discuss this issue separately in subsection 5.1.7.

The Overlap metric is all-or-nothing. For measuring similarity between numeric or symbolically atomic or arbitrary values this may suffice, but there are cases (especially in NLP) in which string-valued feature values occur that can mismatch with other string values in a meaningfully graded way. For example, the value pair "bathe" and "bathes" only differs in one letter; counting them as more similar than "bathe" and "rumour", for example, may be useful for the classification task at hand. We implemented a classic *edit distance* metric, Levenshtein distance (**?**), which counts the number of insertions, deletions, and substitutions to transform the one string into the other. In our (dynamic programming) implementation the three operations count equally heavily.

5.1.2 Information-gain and gain ratio feature weighting

The distance metric in Equation 5.2 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. **?**) 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 (**?**; **?**).

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 5.3).

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

$$w_i = H(C) - \sum_{v \in V_i} P(v) \times H(C|v) \tag{5.3}$$

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, an intermediate step needs to be taken to apply the symbol-based computation of IG. All real values of a numeric features are temporarily discretized into a number (the default is 20) of intervals. Instances are ranked on their real value, and then spread evenly over the intervals; each interval contains the same number of instances (i.e., by default, $1/20$th of the total amount of instances). Instances in each of these intervals are then used in the IG computation as all having the same unordered, symbolic value per group. Note again that this discretization is only temporary; it 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 (**?**) 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 5.5).

$$w_i = \frac{H(C) - \sum_{v \in V_i} P(v) \times H(C|v)}{si(i)} \tag{5.4}$$

$$si(i) = -\sum_{v \in V_i} P(v) \log_2 P(v) \tag{5.5}$$

The resulting Gain Ratio values can then be used as weights w_f in the weighted distance metric (Equation 5.6)[2]. The k-NN algorithm with this metric is called IB1-IG (**?**).

$$\Delta(X, Y) = \sum_{i=1}^{n} w_i \, \delta(x_i, y_i) \tag{5.6}$$

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 challenging for IB1-IG, because IG will overestimate their joint relevance. Imagine an informative feature which is duplicated. This

[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.

results in an overestimation of IG weight by a factor two, and can lead to accuracy loss, because the doubled feature will dominate the distance metric.

5.1.3 Chi-squared and shared variance feature weighting

Unfortunately, as **?**) have shown, the Gain Ratio measure still has an unwanted 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. **?**) 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 5.7).

$$\chi^2 = \sum_i \sum_j \frac{(E_{ij} - O_{ij})^2}{E_{ij}} \tag{5.7}$$

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 5.8). 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).

$$E_{ij} = \frac{n_{.j} n_{i.}}{n_{..}} \tag{5.8}$$

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 5.6, or we can explicitly correct for the degrees of freedom by using the **Shared Variance** measure (Equation 5.9).

$$SV_i = \frac{\chi_i^2}{N \times (min(|C|, |V_i|) - 1)} \tag{5.9}$$

Where $|C|$ and $|V_i|$ are the number of classes and the number of values of feature i, respectively, and N is the number of instances[3]. 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.

Chi-squared and shared variance weights of *numeric* features are computed via a discretization preprocessing step (also used with computing IG and GR weights). Values are first discretized into a number (20 by default) of equally-spaced intervals between the minimum and maximum values of the feature. These groups are then used as discrete values in computing chi-squared and shared variance weights.

[3]Note that with two classes, the shared variance weights of all features are simply divided by N, and will not be different from χ^2weights.

5.1.4 Modified value difference and Jeffrey divergence metrics

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 **?**) and further refined by **?**). It is called the (Modified) Value Difference Metric (MVDM; Equation 5.10), 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.

$$\delta(v_1, v_2) = \sum_{i=1}^{n} |P(C_i|v_1) - P(C_i|v_2)| \qquad (5.10)$$

For computational efficiency, all pairwise $\delta(v_1, v_2)$ values can be pre-computed before the actual nearest neighbor search starts.

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 **?**) 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 with data sparsity. In many practical applications, we are confronted with a very limited set of examples, with values occuring only a few times or once in the whole data set. If two such values occur with the same class, MVDM will regard them as identical, and if they occur with two different classes their distance will be maximal. In cases of such extreme behaviour on the basis of low-frequency evidence, it may be safer to back off to the Overlap metric, where only an exact value match yields zero distance. TiMBL offers this back-off from MVDM to Overlap through a frequency threshold, that switches from the MVDM to the Overlap metric when one or both of a pair of matched values occurs fewer times in the learning material than this threshold.

Jeffrey divergence, offered as a close neighbor alternative to MVDM, is a statistical dissimilarity metric that can be used to compute the distance between class distributions of two values of the same feature. Functionally it is quite similar to MVDM. It is best known for its application as a distance function in unsupervised vector space models, e.g. in image retrieval, where it is applied to histogram vectors. While MVDM computes a straightforward geometrical distance between two class distribution vectors, Jeffrey divergence introduces a logarithm term, as seen in Equation 5.11. Jeffrey divergence is a symmetric variant of Kullback-Leibner distance; the m term given in Equation 5.12 is used for this purpose.

$$\delta(v_1, v_2) = \sum_{i=1}^{n} (P(C_i|v_1) log \frac{P(C_i|v_1)}{m} + P(C_i|v_2) log \frac{P(C_i|v_2)}{m}) \tag{5.11}$$

$$m = \frac{P(C_i|v_1) + P(Ci|v_2)}{2} \tag{5.12}$$

Compared to MVDM, Jeffrey divergence assigns relatively larger distances to value pairs of which the class distributions are more orthogonal. In other words, it assigns more prominence to zero probabilities, which in the case of sparse data (e.g, with Zipfian distributions of values) are generally better estimations than non-zero probabilities. This makes Jeffrey divergence in principle more robust than MVDM with respect to sparse data.

As with MVDM, TiMBL offers an optional frequency-thresholded back-off from Jeffrey divergence to the Overlap metric to further remedy some negative effects due to data sparseness.

5.1.5 Dot-product and cosine metrics

When features have numeric or binary values, TiMBL can also compute the distance between two instances via the dot product (or inner product) of their feature-value vectors. The dot product (which is higher with better matches) is subsequently inversed to a distance by subtracting it from the maximum dot product attainable, i.e. that on an exact match. In Equation 5.13 this maximal dot product is referred to as dot_{max}.

$$\Delta(X, Y) = dot_{max} - \sum_{i=1}^{n} w_i x_i y_i \tag{5.13}$$

As with the other distance metrics incorporated in TiMBL, we include the feature weight w_i in the metric. When no weighting is set (`-w 0`), all weights are set to 1.0, and equation 5.13 reduces to the normal unweighted dot product.

The dot-product metric is typically used with binary vectors or sparse vectors in general. When either x_i or y_i has a zero value, that value pair is not counted in the dot product. Consequently, the significant deviation from the Overlap metric is that matching values that both have a zero value do not count here, whereas they count as much as any other value match in the Overlap metric.

A commonly used variant of the dot product metric, e.g. in information retrieval, is the cosine metric, which corrects for large differences in the length of the instance vectors. The cosine metric divides the dot product metric by the product of the length of the two vectors. As with the dot product, TiMBL converts the cosine metric similarity to a distance by subtracting it from a cos_{max} term that is larger than the maximal cosine similarity, as given in Equation 5.14. Again, feature weighting is included in the formula:

$$\Delta(X, Y) = cos_{max} - \frac{\sum_{i=1}^{n} w_i x_i y_i}{\sqrt{\sum_{i=1}^{n} w_i x_i^2 \sum_{i=1}^{n} w_i y_i^2}} \tag{5.14}$$

Due to its internal tree structure, TiMBL is not particularly suited to handle feature vectors with thousands or more features. Many features cause very deep and usually very unbalanced trees, from which retrieval can be rather inefficient (especially when there is little variance in the feature weights). Other internal data structures such as inverted indices are typically more suited to these types of vector spaces. For now, inverted indices are not implemented in TiMBL.

5.1.6 Distance-weighted class voting

The most straightforward method for letting the k nearest neighbors vote on the class of a new case is the *majority voting* method, in which the vote of each neighbor receives equal weight, and the class with the highest number of votes is chosen (or in case of a tie, some tie resolution is performed, cf. Subsection 5.1.1).

We can see the voting process of the k-NN classifier as an attempt to make an optimal class decision, given an estimate of the conditional class probabilities in a local region of the data space. The radius of this region is determined by the distance of the k-furthest neighbor.

Sometimes, if k is small, and the data is very sparse, or the class labels are noisy, the "local" estimate is very unreliable. As it turns out in experimental work, using a larger value of k can often lead to higher accuracy. The reason for this is that in densely populated regions, with larger k the local estimates become more reliable, because they are "smoother". However, when the majority voting method is used, smoothing can easily become oversmoothing in sparser regions of the same data set. The reason for this is that the radius of the k-NN region can become extended far beyond the local neighborhood of the query point, but the far neighbors will receive equal influence as the close neighbors. This can result in classification errors that could easily have been avoided if the measure of influence would somehow be correlated with the measure of similarity. To remedy this, we have implemented three types of distance weighted voting functions in TiMBL.

A voting rule in which the votes of different members of the nearest neighbor set are weighted by a function of their distance to the query, was first proposed by Dudani (**?**). In this scheme, henceforth referred to as IL (for inverse-linear), a neighbor with smaller distance is weighted more heavily than one with a greater distance: the nearest neighbor gets a weight of 1, the furthest neighbor a weight of 0 and the other weights are scaled linearly to the interval in between (**?**), Equation 1.).

$$w_j = \begin{cases} \frac{d_k - d_j}{d_k - d_1} & \text{if } d_k \neq d_1 \\ 1 & \text{if } d_k = d_1 \end{cases} \tag{5.15}$$

Where d_j is the distance to the query of the j'th nearest neighbor, d_1 the distance of the nearest neighbor, and d_k of the furthest (k'th) neighbor.

Dudani (**?**), eq.2,3) further proposed the *inverse distance weight* (henceforth ID). In Equation 5.16 a small constant is usually added to the denominator to avoid division by zero (**?**).

$$w_j = \begin{cases} \frac{1}{d_j + \epsilon} \end{cases} \tag{5.16}$$

Another weighting function considered here is based on the work of **?**), who argues for a universal perceptual law which states that the relevance of a previous stimulus for the generalization to a new stimulus is an exponentially decreasing function of its distance in a psychological space (henceforth ED). This gives the weighed voting function of Equation 5.17, where α and β are constants determining the slope and the power of the exponential decay function.

$$w_j = e^{-\alpha d_j^\beta} \tag{5.17}$$

Note that in Equations 5.16 and 5.17 the weight of the nearest and furthest neighbors and the slope between them depend on their absolute distance to the query. This assumes that the relationship between absolute distance and the relevance gradient is fixed over different datasets. This assumption is generally false; even within the same dataset, different feature weighting metrics can cause very different absolute distances.

Figure 5.2 visualises a part of the curves of ID and ED, the latter with a few varied settings of α and β. Generally, both distance weighting functions assign highly differing weights for close neighbors, and less differing weights for more distant neighbors. ID assigns very high votes (distance weights) to nearest neighbors at distances approaching 0.0 - in effect it assigns absolute preference to exact matches. In contrast, all ED variants have a vote of 1.0 for exact matches, and have a shallower curve than the ID curve for higher distances. Higher values of α in the ED function assign relatively higher weights to exact matches. When β is set to larger values than 1.0, the ED curve becomes bell-shaped, effectively assigning relatively less different weights between exact-matching neighbors and near-exact matching instances.

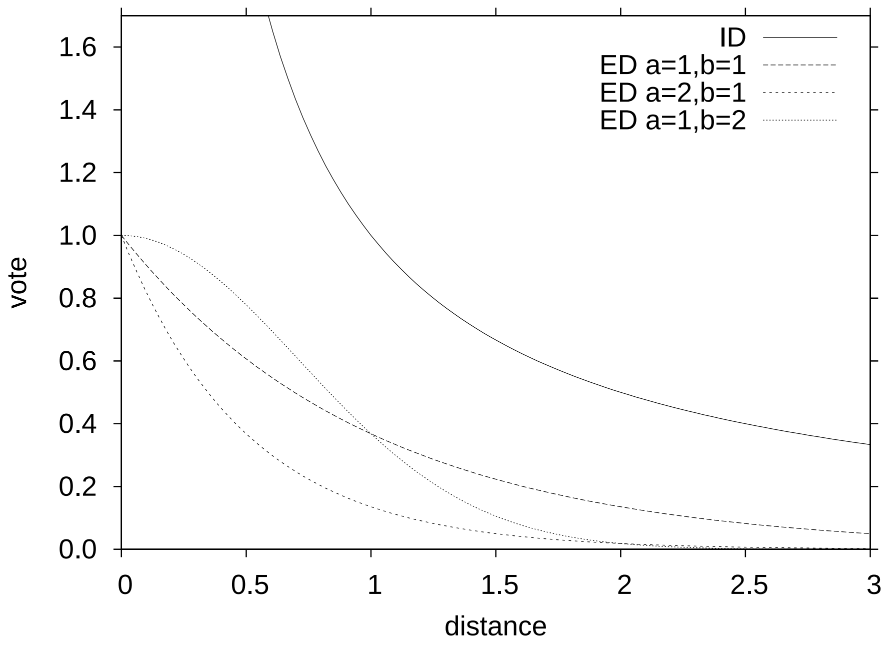

Figure 5.2: Visualisation of the Inverse Distance weighting function (IL) and three variants of the Exponential Decay distance weighting function (ED) varying settings of α (1) and β (b).

Following Dudani's proposal, the benefits of weighted voting for k-NN have been discussed widely, e.g. (**?**; **?**; **?**), but mostly from an analytical perspective. With the popularity of Instance-Based Learning applications, these issues have gained a more practical importance. In his thesis on k-NN classifiers, **?**) cites Dudani, but proceeds to work with Equation 5.16. He tested this function on a large amount of datasets and found weak evidence for performance increase over majority voting. An empirical comparison of the discussed weighted voting methods in (**?**) has shown that weighted voting indeed often outperforms unweighted voting, and that Dudani's original method (Equation 5.15) mostly outperforms the other two methods. From that set of experiments, it also seems that Dudani's method shows its optimal performance at much larger values of k than the other voting methods.

5.1.7 Tie breaking

Thus far we have described the last step of k-NN classification as taking the majority category among the set of nearest neighbors, where their vote is either unweighted or weighted by their distance (subsection 5.1.6). Especially in case of unweighted voting, ties can occur; e.g. of a set of ten nearest neighbors, five vote for class A, and the other five for B. The procedure for breaking this tie in the k-NN classifier in TiMBL is as follows. First, the value of the k parameter is incremented by 1, and the additional nearest neighbors at this new kth distance are added to the current nearest neighbor set (k is subsequently reset to its user-specified value). If the tie in the class distribution persists, then the class label is selected with the highest overall occurrence in the training set. If that is also equal, then the first class is taken that was encountered when reading the training instance file.

Optionally, TiMBL can be set to avoid ties by making a *random* choice of a classification from a class distribution in a nearest-neighbor set, weighted by the distribution of the classes in the set.

5.1.8 Exemplar weighting

Exemplar weighting in memory-based learning captures the intuition that some instances are better (more reliable, more typical, more regular) nearest neighbors than others. Classification accuracy could benefit from giving these instances some priority in the k-NN classification process. This idea has been explored in the context of on the one hand classification (**?**; **?**), and on the other hand editing bad instances from memory (**?**). **?**), as a classic example, uses *class-prediction strength*: the ratio of the number of times the instance type is a nearest neighbor of another instance with the same class and the number of times that the instance type is the nearest neighbor of another instance type regardless of the class. Another example is *typicality* as used by **?**).

Exemplar weights could in principle be used either as weights in the class voting (as distance weights, cf. Subsection 5.1.6), or as weights in the distance metric (eq. 5.6). TiMBL supports only the latter type, and in this respect exemplar weighting is not an intrinsic part of TiMBL. TiMBL does not compute exemplar weighting metrics itself, but only allows users to specify preprocessed exemplar weights with the `-s` input option. Subsequently, when the distance between a test instance and a memory instance is computed, TiMBL uses the memory instance's weight as follows, where $\Delta^E(X,Y)$ is the exemplar-weighted distance between instances X and Y, and ew_X is the exemplar weight of memory instance X:

$$\Delta^E(X,Y) = \frac{\Delta(X,Y)}{ew_X + \epsilon} \tag{5.18}$$

ϵ is the smallest non-zero number, and is used to avoid division by zero. Exemplar weights approaching zero yield very large distances; relatively higher values yield relatively smaller distances.

Note that when a training instance occurs more than once in a training set, TiMBL expects it to have the same example weight with all occurrences; TiMBL cannot handle different example weights for the same instance type. TiMBL produces a warning (*Warning: deviating exemplar weight in line #...*), and uses the first weight found for the instance.

5.2 Indexing optimizations

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. 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, times f, the number of features. In our current implementation of IB1 we use tree-based indexing to alleviate these costs.

5.2.1 Tree-based indexing

The tree-based memory indexing operation replaces the flat array by a tree 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. Due to this storage structure, instances with identical feature-values are collapsed into a single 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.

First, the tree can be searched top-down very quickly for *exact matches*. When $k = 1$, an exact match ($\Delta(X, Y) = 0$) can never be beaten, so then it is possible to omit any further distance computations. The shortcut is built into TiMBL, but by default it is not used with $k > 1$. TiMBL does, however, offer the possibility to use the shortcut at any value of k, with the command line switch (`+x`. Using it can speed up classification radically for some types of data, but with $k > 1$, the shortcut is not guaranteed to give the same performance (for better or for worse) as classification without it.

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 5.2) can still contribute to the overall distance (Equation 5.1), and discard whole branches of the tree which will never be able to rise above the partial similarity of the current least similar nearest 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[5].

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. Note that different orderings may only affect classification speed, not the actual classifications.

[4]Suggested by Gert Durieux.

[5]With the special command line setting `--silly` this tree search shortcut is switched off; as the name of the setting suggests, this is not recommended, except for explicit speed comparisons.

5.3 IGTree

Using Information Gain rather than unweighted Overlap distance to define similarity in IB1 improves its performance on several NLP tasks (?; ?; ?). 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 (?) 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, 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 simplified and faster processing during classification, as compared to IB1 and IB1-IG.

The generalization accuracy of IGTREE is usually comparable to that of IB1-IG; often slightly worse, but sometimes even better. The two causes for IGTREE's surprisingly good accuracies attained with dramatically faster classification are that (i) most 'unseen' instances contain 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 tradeoff between a somewhat lower accuracy against stellar speed increases can be very attractive.

It should be noted that by design, IGTREE is not suited for numeric features, as it does not use some type of discretization. If present in data, numbers will simply be treated as literal strings by IGTREE. 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 obviously has no effect on IGTREE performance.

5.4 The TRIBL and TRIBL2 hybrids

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 (**?**) and TRIBL2 as hybrid combinations of IGTREE and IB1. Both algorithms aim to exploit the trade-off between (i) optimization of search speed (as in IGTREE), and (ii) maximal generalization accuracy. They do that by splitting the classification of new instances into a quick decision-tree (IGTREE) traversal based on the first (most important and most class-disambiguating) features, followed by a slow but relatively accurate k-NN (IB1) classification based on the remaining less important features. The difference between TRIBL and TRIBL2 is that the former algorithm fixes the point in the feature ordering where IGTREE is succeeded by IB1, while TRIBL2 determines this switching point automatically per classification. We briefly describe both variants.

For TRIBL, a parameter (`-q`) determines the switching point in the feature ordering 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 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.

TRIBL2 does not employ a fixed switching point. Rather, during the classification of an instance it continues to use IGTREE as long as it finds matching feature values in the weighting-governed feature ordering. Only when it finds a mismatch it reverts to IB1 classification on all remaining features. The reasoning behind this mismatch-based switching is that it offers a fairly optimal minimalisation of the use of IB1; it is only invoked when mismatching occurs, which is the typical point in which IB1 can improve on decision-tree-style classification, which does not consider the other potentially matching features in the ordering (**?**).

5.5 IB2: Incremental editing

In memory-based learning it seems sensible to keep any instance in memory that plays a (potentially) positive role in the correct classification of other instances. Alternatively, when it plays no role at all, or when it is disruptive for classification, it may be a good idea to discard, or *edit* it from memory. On top of not harming or even improving generalization performance, the editing of instances from memory could also alleviate the practical processing burden of the k-NN classifier kernel, since it would have less instances to compare new instances to. This potential double pay-off spawned a distinct line of work on editing in the k-NN classifier quite early **?**) and **?**).

TiMBL offers an implementation of one particular editing algorithm called IB2 (**?**), an extension

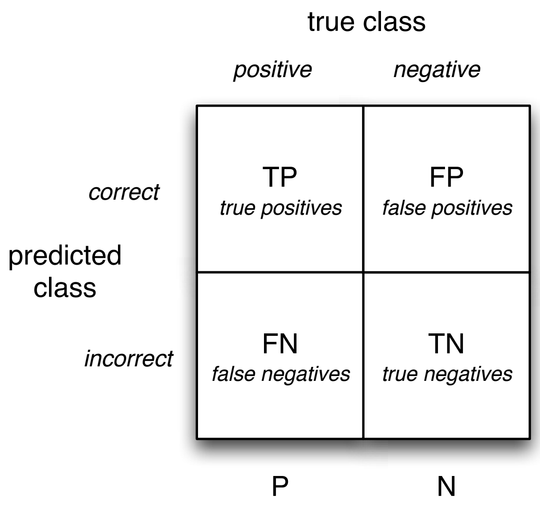

Figure 5.3: Class-specific confusion matrix containing the basic counts used in the advanced performance metrics.

to the basic IB1 algorithm introduced in the same article. IB2 implements an incremental editing strategy. Starting from a seed memory filled with a certain (usually small) number of labeled training instances, IB2 adds instances incrementally to memory only when they are *misclassified* by the k-NN classifier on the basis of the instances in memory at that point. These instances are added, since they are assumed to be representatives of a part of the complete instance space in which they themselves and potentially more nearest-neighbor instances have a particular class different from the class of neigboring instances already in memory. The economical idea behind IB2 is that this way typically only instances on the boundaries of such areas are stored, and not the insides of the areas; the classification of instances that would be positioned well inside such areas is assumed to be safeguarded by the memorized boundary instances surrounding it.

Although the IB2 may optimize storage considerably, its strategy to store all misclassified instances incrementally makes IB2 sensitive to noise (**?**). It is also yet unclear what the effect is of the size of the seed.

5.6 Advanced evaluation metrics

Aside from accuracy (the percentage of correctly classified test instances), TiMBL offers some more evaluation metrics that have become common in information retrieval and machine learning in general, namely precision, recall, and F-score, and ROC-space (with dimensions true positive rate and false positive rate), and AUC. We describe these metrics in more detail here.

Figure 5.3 displays the general confusion matrix for one class C, splitting all classifications on a test set into four cells. The TP or true positives cell contains a count of examples that have class C and are predicted to have this class correctly by the classifier. The FP or false positives cell contains a count of examples of a different class that the classifier incorrectly classified as C. The FN or false negatives cell contains examples of class C for which the classifier predicted a different class label than C. On the basis of these four numbers and the total number of positive examples $P = TP + FN$ and negative examples $N = FP + TN$, we can compute the following performance measures:

Precision $= \frac{TP}{TP+FP}$, or the proportional number of times the classifier has correctly made the decision that some instance has class C.

Recall or True Positive Rate (TPR) $= \frac{TP}{P}$, or the proportional number of times an example with class C in the test data has indeed been classified as class C by the classifier.

False Positive Rate (FPR) $= \frac{FP}{N}$, or the proportional number of times an example with a different class than C in the test data has been classified as class C by the classifier.

F-score $= \frac{2 \times precision \times recall}{precision + recall}$, or the harmonic mean of precision and recall (**?**), is a commonly used metric to summarize precision and recall in one measure. The left part of Figure 5.4 shows F-score isolines in the two-dimensional space of recall (x-axis) and precision (y-axis). The curvature of the isolines is caused by the harmonic aspect of the formula (in contrast, the normal mean has straight isolines orthogonal to the $x = y$ diagonal), which penalizes large differences between precision and recall. The isolines could be likened to height isolines in a map, where the peak of the hill is at the upper right corner of the space.

AUC or *area under the curve* in the so-called ROC or *receiver operator characteristics* space (**?**; **?**), is the surface of the grey area in the right graph of Figure 5.4. The ROC space is defined by the two dimensions FPR (false positive rate, x-axis) and TPR (true positive rate, or recall, y-axis). The difference with F-score is that it does not make use of the statistically unreliable precision metric; rather, it takes into account all cells of the matrix in Figure 5.3 including the TN (true negative) cell (for a more detailed description and arguments for using ROC analysis, cf. (**?**)). Its "peak" is in the upper left corner, at a FPR of zero and a TPR of 1. Rather than using the harmonic mean, it is common to report on the AUC, area under the classifier's TPR-FPR curve, where in the case of a discrete-output classifier such as TIMBL this can be taken to mean the two lines connecting the experiment's TPR and FPR to the $(0, 0)$ coordinate and the $(1, 1)$ coordinate, respectively; the AUC is then the grey area between these points and coordinate $(1, 0)$.

While these advanced statistics can be computed per class, they can also be averaged to produce a single outcome for a full test set. Common methods for averaging F-scores and AUC scores are micro-averaging and macro-averaging. In micro-averaging, each class' F-score or AUC is weighted proportionally to the frequency of the class in the test set. A macro-average adds all the F-scores or AUCs and divides this sum by the number of classes in the training set. In computing these averages, TiMBL bases itself on the classes in the training set. When a class does not re-occur in test material, it can have no recall, but it can have precision, hence it is always incorporated in averages. A class that occurs in test material but not in training material can never be predicted correctly, and is never included in averages.

5.7 NLP applications of TiMBL

This section provides a brief historical overview of work, performed in the Tilburg and Antwerp groups and by others, with the application of MBL-type algorithms to NLP tasks. For more historical background, see (**?**).

The Tilburg and Antwerp groups have published a number of articles containing descriptions of the algorithms and specialised metrics collected in TiMBL, usually demonstrating their functioning using NLP tasks. The IB1-IG algorithm was first introduced in (**?**) in the context of a comparison of memory-based approaches with error-backpropagation learning for a hyphenation task. Predecessor versions of IGTREE can be found in (**?**; **?**) where they are applied to grapheme-to-phoneme conversion. See (**?**) for a description and review of IGTREE and IB1-IG. TRIBL is

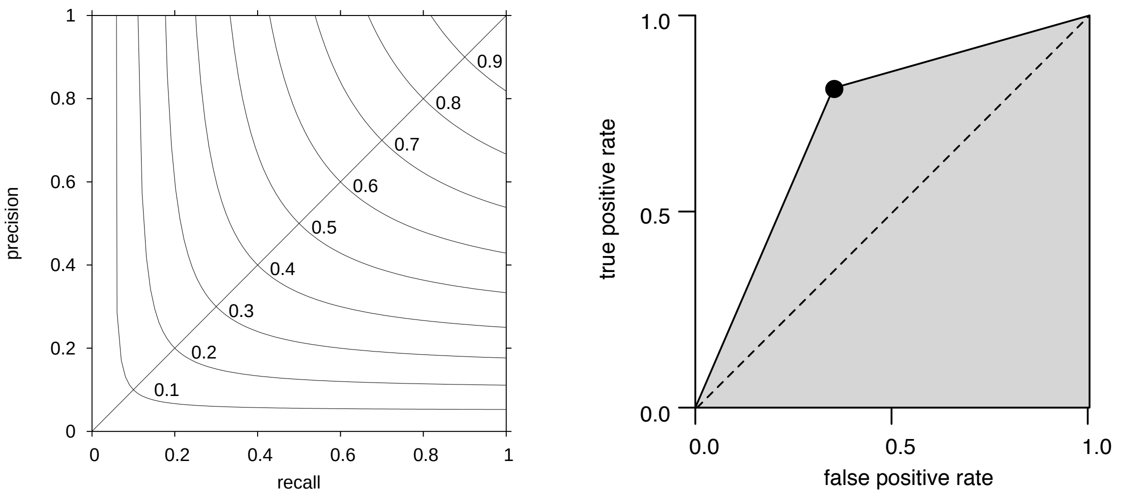

Figure 5.4: Precision–recall space with F-score isolines (left), and ROC space with an experimental outcome marked by the dot, and the outcome's AUC, the shaded surface between the dot and coordinates $(0,0)$, $(1,0)$, and $(1,1)$ (right).

described in (**?**). Experiments with distance-weighted class voting are described in (**?**). Aspects of using binary-valued (unpacked multi-valued) features are discussed in (**?**). Comparisons between memory-based learning and editing variants are reported in (**?**; **?**). A hybrid of TiMBL and the RIPPER rule-induction algorithm (**?**) is described in (**?**; **?**). Using TiMBL as a classifier combination method is discussed in (**?**). **?**) describes an extension of TiMBL with error-correcting output codes. **?**) report on an experiment to import maximum-entropy matrices to replace MVDM matrices (**?**), improving over the maximum-entropy classifier. **?**) presents a search algorithm to find optimal combinations of parameter settings automatically, given a labeled training set of examples, showing large gains of the default settings (also of other machine learning algorithms). Parallelization of TiMBL, through splitting either the training set or the test set in n pieces in shared-memory multi-processor architectures, is explored in (**?**).

The memory-based algorithms implemented in the TiMBL package have been targeted to a large range of Natural Language Processing tasks. Examples of applications in the morpho-phonological and speech areas are hyphenation and syllabification (**?**); classifiying phonemes in speech (**?**); assignment of word stress (**?**); grapheme-to-phoneme conversion, (**?**; **?**; **?**); diminutive formation (**?**); and morphological analysis (**?**; **?**; **?**). Although these examples are applied mostly to Germanic languages (English, Dutch, and German), applications to other languages with more complicated writing systems or morphologies, or with limited resources, have also been presented: for example, letter-phoneme conversion in Scottish Gaelic (**?**), morphological analysis of Arabic (**?**), or diacritic restoration in languages with a diacritic-rich writing system (**?**; **?**).

At the syntactic sentence level TiMBL has been applied to part of speech tagging (**?**; **?**; **?**); PP-attachment (**?**); subcategorization (**?**); phrase chunking (**?**; **?**); dependency parsing (**?**), article generation (**?**); shallow parsing (**?**; **?**; **?**); clause identification (**?**; **?**); sentence-boundary detection (**?**); predicting the order of prenominal adjectives for generation (**?**); and, beynd the sentence level, to anaphora resolution (**?**; **?**; **?**). More recently, memory-based learning has been integrated as a classifier engine in more complicated dependency parsing systems (**?**; **?**; **?**).

Memory-based learning has been applied succesfully to lexical semantics, in particular to word sense disambiguation (**?**; **?**; **?**; **?**; **?**; **?**), but also in other lexical semantic tasks such as determining noun countability (**?**), animacy (**?**), and semantic relations within noun compounds (**?**; **?**).

On the textual level, TiMBL has been used for information extraction (**?**; **?**; **?**), text classification (**?**), question classification (**?**; **?**), spam filtering (**?**; **?**), named-entity recognition (**?**; **?**; **?**; **?**; **?**), and error detection in textual databases (**?**).

In the field of discourse and dialogue modeling, TiMBL has been used for shallow semantic analysis of speech-recognised utterances (**?**; **?**; **?**; **?**; **?**), in disfluency detection in transcribed spontaneous speech (**?**), and in classifying ellipsis in dialogue (**?**).

Relations to statistical language processing, in particular the interesting equivalence relations with back-off smoothing in probabilistic classifiers, are discussed in (**?**). Relations between classification-based word prediction and statistical language modeling are identified in (**?**; **?**).

In machine translation, k-nearest neighbor classification bears a close relation with example-based machine translation (EBMT). A first EBMT-implementation using TiMBL is described in (**?**).

The first dissertation-length study devoted to the approach is (**?**), 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). TiMBL is also central in the Ph.D. theses of **?**), **?**), and **?**). In 1999 a special issue of the *Journal for Experimental and Theoretical Artificial Intelligence* (Vol. 11(3), edited by Walter Daelemans) was devoted to Memory-Based Language Processing. The introduction to this special issue discusses the inspiration sources and alternative developments related to the memory-based approach taken in TiMBL (**?**).

Whereas most work using TiMBL has been oriented towards natural language engineering applications, the linguistic and psycholinguistic relevance of memory-based learning is another focus of research in Antwerp, Tilburg and elsewhere. Work in this area has been done on stress assignment in Dutch (**?**; **?**), reading aloud (**?**), phonological bootstrapping (**?**), the above-mentioned prediction of linking morphemes in Dutch (**?**), morphology (**?**; **?**), and the Dutch plural inflection (**?**). A comparison to other analogical methods for linguistics is provided in (**?**).

All Tilburg/Antwerp papers referred to in this section, as well as more recent papers, are available in electronic form from the ILK *home page:* `http://ilk.uvt.nl` *and the* CLiPS *home page:* `http://www.cnts.ua.ac.be/cnts`.

Chapter 6

Software usage and options

6.1 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 6.2 (release) (c) ILK 1998 - 2007.
Tilburg Memory Based Learner
Induction of Linguistic Knowledge Research Group
Tilburg University / University of Antwerp
Mon Sep 21 14:28:00 2009

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 the name of which 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 (`-`).

6.1.1 Algorithm and metric selection

`-a <n> or <string>` : determines the classification algorithm. Possible values are:

- `0` or `IB1` – the IB1 (k-NN) algorithm (default). See Sections 5.1 and 5.2.
- `1` or `IGTREE` – IGTREE, decision-tree-based optimization. See Section 5.3.
- `2` or `TRIBL` – TRIBL, a hybrid of IB1 and IGTREE. See Section 5.4.
- `3` or `IB2` – IB2, incremental edited memory-based learning. See Section 5.5.
- `4` or `TRIBL2` – TRIBL2, a non-parameteric version of TRIBL. See Section 5.4.

`-m <string>` : determines which distance 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 the restrictions given by `:MetricRange`. A range can be written using comma's for lists, and hyphens for intervals. The metric code can be one of the following five:

1. `O` – Overlap (default; see Subsection 5.1.1)
2. `L` – Levenshtein (see Subsection 5.1.1)
3. `M` – Modified value difference (MVDM; see Subsection 5.1.4)
4. `J` – Jeffrey divergence (see Subsection 5.1.4)
5. `D` – Dot product (see Subsection 5.1.5)
6. `C` – Cosine (see Subsection 5.1.5)
7. `N` – Numeric (all features are numeric. see Subsection 5.1.1)
8. `I` – Ignore (ignore specified features)

For example, `-mO:N3:I2,5-7` sets the global metric to overlap, declares the third feature to be numeric, and ignores features 2 and 5, 6, and 7.

Ignore *can* be the global metric; it must be followed by a `MetricRange` string with metric `O`, `M`, `J`, `D`, or `N` specifying in the range which features are *not* ignored.

`-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 5.1.2.
- n=2 – Information Gain weighting. See section 5.1.2.
- n=3 – Chi-squared (χ^2) weighting. See section 5.1.3.
- n=4 – Shared variance weighting. See section 5.1.3.
- n=<filename>:<number> or n=<filename> – Instead of the five weight settings above we can supply a filename to the `-w` option. This causes TiMBL to read this file and use its contents as weights. If only <filename> is given as an argument, the file is supposed to contain one list of feature weights for all features. The <filename>:<number> option assumes that a weights file generated by TiMBL with the `-W` option (and possibly edited by the user) is read back in; the number refers to one of the five numbers above. See section 6.2.2 for a description of the format of weights files.

`-k <n>` : number of nearest neighbors used for extrapolation. Only applicable in conjunction with IB1 (`-a 0`), TRIBL (`-a 2`), TRIBL2 (`-a 4`) and IB2 (`-a 3`). 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.

`-d <val>` : The type of class voting weights that are used for extrapolation from the nearest neighbor set. `val` can be one of:

- `Z` : normal majority voting; all neighbors have equal weight (default).
- `ID`: Inverse Distance weighting. See Section 5.1.6, Equation 5.15.
- `IL`: Inverse Linear weighting. See Section 5.1.6, Equation 5.16.
- `ED:<a>:<b>`: Exponential Decay weighting with decay parameters `a` (α) and `b` (β). No spaces are allowed in the string. Parameter `b` can be left unspecified: `ED:<a>` assumes $\beta = 1$. The syntax used in previous TiMBL versions (`ED<a>`) is still supported but deprecated. See Section 5.1.6, Equation 5.17.

`-L <n>` : frequency threshold for switching from the MVDM or Jeffrey Divergence to the Overlap distance metric. The default is 1 (never switch). When in a pair of matched values one or both values occur less frequently than `n` times in the learning material, TiMBL switches from MVDM or Jeffrey Divergence to Overlap. Higher values of `n` force TiMBL to use the Overlap metric more.

Instead of backing off to the Overlap metric, it is also possible to back-off to the Levenshtein metric with `-L L:<n>`. [1]

Only applicable in conjunction with the MVDM (`-mM`) and Jeffrey divergence (`-mJ`) distance metrics.

`-b <n>` : determines n (≥ 1), the number of instances, to be taken from the top of the training file, to act as the bootstrap set of memorized instances before IB2 starts adding new instances. Only applicable in conjunction with IB2 (`-a 3`).

`-q <n>` : `n` is the TRIBL offset, the index number of the feature (counting from 1) after which TRIBL should switch from IGTREE to IB1. Only applicable in conjunction with TRIBL (`-a 2`).

`-R <n>` : Resolve ties in the classifier randomly, using a random generator with seed n. `-R <n>` 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. By default, `-R` is not used, but rather the deterministic tie resolution scheme described in Subsection 5.1.1.

`-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: `-B -d -e -k -L -M -o -p -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 <testfile>` : the string `<testfile>` is the literal name of the file with the test items.

`-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). Only applicable in conjunction with IB1 (`-a0`).

[1] KO: niet meer in 6.2

`-t cross_validate` : An n-fold cross-validation experiment is performed on the basis of n files (e.g. $1/n$ partitionings of an original data file). The names of these n files need to be in a text file (one name per line) which is given as argument of `-f`. In each fold $f = 1 \ldots n$, file number f is taken as test set, and the remaining $n - 1$ files are concatenated to form the training set. Only applicable in conjunction with IB1 (`-a0`).

6.1.2 Input options

`-f <datafile>` : the string `<datafile>` is the literal name of the file with the training items, or (in conjunction with `-t cross_validate`, the file containing the names of the cross-validation files.

`-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, Sparse, Binary` (case-insensitive). The default is that TiMBL guesses the format from the contents of the first line of the data file. ARFF is not automatically detected. See section 6.2.1 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.

`-i <treefile>` : Skip the first two training phases: instead of processing a training file, read a previously saved (see `-I` option) instance-base or IGTREE from the file `treefile`. See section 6.2.4 for the format of this file.

`--matrixin=<filename>` : Retrieve distance metrics from file `filename`.

`-u <valueclassprobfile>` : Replace the automatically computed value-class probability 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`: Use the whitespace-delimited exemplar weights, given after each training instance in the training file `<datafile>`, during classification. `<testfile>` may contain exemplar weights, but they are not used in classification. If the test file does not have an exemplar weights column, you must specify `-s1`. Exemplar weights can also be ignored (in both training and test files) by specifying `-s0`.

`-S <portnumber>` : Starts a TiMBL server listening on the specified port number of the localhost. See Chapter 6.3 for a description of the communication protocol.

6.1.3 Output options

`-I <treefile>` : After phase 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). For IGTREE this also automatically saves the current weights into `treefile.wgt` unless this is overridden by `-W`. See section 6.2.4 for a description of the resulting file's format.

`--matrixout=<filename>` : Store calculated MVDM or Jeffrey divergence distance metrics in file `filename`.

`-X <xmlfile>` : instead of the proprietary file format written with the `-I` switch, `-X` writes the TiMBL tree into an XML tree in `<xmlfile>`. This XML file cannot be read back into TiMBL.

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

`-U <valueclassprobfile>` : Write the automatically computed value-class probability 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. Off by default.

`+/-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 (case-insensitive):

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

`o`: show all options set.

`f`: show calculated feature weights. (default)

`p`: show MVDM matrices.

`e`: show exact matches.

`as`: show overall advanced statistics (micro and macro averages of F-score and AUC).

`cm`: show confusion matrix between actual and predicted classes.

`cs`: show per-class statistics (precision, recall, true positive rate, false positive rate, F-score, AUC).

`di`: add the distance of the nearest neighbor to the output file.

`db`: add class distribution in the nearest neighbor set to the output file.

`md`: add matching depth and node type (N for non-ending node, L for leaf) to output file (-a1 only).

`k`: add a summary of class distribution information of all nearest neighbors to the output file (sets `-x`)

`n`: add nearest neigbors to the output file (sets `-x`)

`b`: provide branching statistics of the internal tree, overall and per level.

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

`-G <n>`: Normalize class distributions generated by `+v db`.

`0`: Normalize distributions so that they add up to 1.0

`1:<f>`: Smooth by adding floating-point f to all class votes (e.g. `-G1:1` performs add-one smoothing).

`--Beam=<n>`: Limit the number of returned classes and class votes returned by `+v db` to n. Default is infinity (no limit).

`+/- %` : Write the percentage of correctly classified test instances, the number of correctly classified instances, and the total number of classified instances (one number per line, three lines in total) 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, where the default output file name would normally be the same.

`-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.

`-V` : Show the TiMBL version number.

6.1.4 Internal representation options

`-N <n>` : (maximum) number of features. Obligatory for Sparse and Binary formats. When larger than a pre-defined constant (default 2500), N needs to be set explicitly for all algorithms.

`+/- x` : turns the shortcut search for exact matches on or off in IB1 (and IB2, TRIBL, and TRIBL2). The default is to be off (`-x`). Turning it on makes IB1 generally faster, but with $k > 1$ the shortcut produces different results from a genuine k nearest neighbors search, since absolute preference is given to the exact match.

`-M <n>` : Set the maximum number of nearest neighbors printed using the `+vn` verbosity option. By default this is set to 500, but when you are interested in the contents of really large nearest neighbor sets (which is possible with large k or large data sets with few features), `n` can be increased up to 100,000.

`+/- H` : Turn on/off hashing of feature values and class labels in TiMBL trees. Hashing is done by default, but with short (e.g. one-character) feature values and/or classes less memory is used when hashing is set off.

`-B <n>` : Number of bins used for discretizing numeric data (only used for computing feature weights).

`-c <n>` : Clipping (threshold) frequency for prestoring MVDM matrices. Cells in the matrix are only stored if both feature values occur more than `<n>` times.

`-T <string>` : Set the ordering of the TiMBL tree (with IB1 and IB2), i.e., rank the features according to the metric identified by `<string>`. The default ordering is `G/V` (according to gain ratio divided by the number of values), but some orderings may produce faster classification. Note that different orderings do *not* change the classification behavior of IB1 and IB2. `<string>` can take the following values:

`DO`: no ordering (the ordering of the features in the data file is taken)

`GRO`: gain ratio (eq. 5.4)

`IGO`: information gain (eq. 5.3)

`1/V`: $1/V$, where V is the number of values

`G/V`: gain ratio divided by the number of values

`I/V`: information gain divided by the number of values

`X2O`: χ^2 (eq. 5.7)

`X/V`: χ^2 divided by the number of values

`SVO`: shared variance (eq. 5.9)

`S/V`: shared variance divided by the number of values

`GxE`: gain ratio $\times si$, where si is the split info of the feature (eq. 5.5)

`IxE`: information gain $\times si$

`1/S`: $1/si$

6.1.5 Hidden options

TiMBL offers an undisclosed number of hidden options that have been built in over time for particular reasons. Some have survived over time, and although their use is not for the faint-hearted, some may offer interesting functionalities. A small list of disclosed hidden options follows.

`--sloppy=`{`true|false`}: in combination with leave-one-out (LOO) testing, this option switches off all weight recomputation. By default, leaving out one training example out causes feature weights, value-class matrices, and derived metrics such as MVDM to be recomputed, because strictly the example-specific statistics should be absent when it is held out and classified. `--sloppy` skips this, causing a significant speedup, and usually slightly better LOO scores. Use only if your experimental method allows it. Default value is `false`.

`--silly=`{`true|false`}: set to `true`, switches off the optimized nearest-neighbor search in IB1 and TRIBL. This causes TiMBL to fall back to comparing all feature values of a test instance to full paths in the TiMBL tree. This causes TiMBL to slow down dramatically on most datasets. Setting is available to enable testing the effect of optimized search. Default value is `false`.

`--Diversify`: modifies all features weights by subtracting the smallest weight (plus ϵ) from all weights. The smallest weight thus becomes ϵ. This modification "diversifies" the feature weights, and was introduced to enhance the effect of DIMBL, the multi-CPU variant of TiMBL[2].

6.2 File formats

This section describes the format of the input and output files used by TiMBL. Where possible, the format is illustrated using the classical "objects" data set, which 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).

6.2.1 Data files

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,

[2]For DIMBL, see `http://ilk.uvt.nl/dimbl`

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 5.2). 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.

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
```

C4.5 format

This format is a derivative of the format that is used by the well-known C4.5 decision tree learning program (?). 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)[3]. White space within the line is taken literally, so the pattern `a, b c,d` will be interpreted as `'a',' b c','d'`. An exception is the class label, which should not contain any whitespace. 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 as 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.
```

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

```
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.
```

ARFF format

ARFF is a format that is used by the WEKA machine learning workbench (**?**; **?**)[4]. 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. The ARFF format is *not* autodetected, and needs to be specified on the commanline with `-F ARFF`.

In ARFF data, the actual data are preceded by a information on feature types, feature names, and names of values in case of symbolic features. TiMBL ignores all of these lines, and starts reading data from after the `@data` statement until the end of the file. Feature-values are supposed to be separated by commas; white space is deleted entirely, so the pattern `a, b c,d` will be interpreted as `'a','bc','d'`. There should be no whitespace in class labels. We hope to 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.
small,other,2,key.
```

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

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

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
```

Sparse format

The sparse format is relevant for data with features of which a significant portion of the values is 0.0 (numeric), 0 (binary), or some "null" symbolic value. Storing only the non-null values typically takes less space on disk.

Consider, for example, a data set in text classification with 10,000 features each representing the tf*idf weight of a term. It would be uneconomical to store instances as long lines of

```
0.02, 0.0, 0.0, 0.0, 0.54, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ... , 0.01,sports
```

The Sparse format allows to store such an instance as

```
(1,0.02)(5,0.54)...(10000,0.01)sports
```

That is, a sequence of ($< index >, < value >$) expressions between parentheses each indicating that the feature number $index$ has value $value$, with the class label at the end, directly following the last parenthesis. The feature index is assumed to start at 1. In case of symbolic values, whitespace included in the parentheses are considered significant (i.e., part of the values). A case with only null values can be represented as either `'class` or `,class`.

This option must be specified by the user (`-F Sparse`); it is not guessed from the data. It must also be accompanied by a user declaration of the number of features (`-N <number>`).

Sparse Binary format

The sparse binary format, a simplified version of the Sparse 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 numeric features, it must always be user-declared 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`.

6.2.2 Weight files

The feature weights used for computing similarities and for the internal organization of the memory base can be saved to a file `-W`. These files can be read back into TiMBL with `-w <filename>:<weight number>`, where the weight number refers to the five options in TiMBL. It is also possible to change these files manually before reading them in – and additionally it is also possible to write a file from scratch and read this into TiMBL. This allows the experimenter to handcraft feature weights.

The generic 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. In this example, the data set has three features.

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

Weight files written by TiMBL are of the same format, but write all weights in a concatenation, separated by # lines that carry the abbreviated name of the weight (nw, gr, ig, x2, sv). The following example illustrates this format (which can be edited manually, as long as the same number of lines is kept):

```
# DB Entropy: 1.61789
# Classes: 5
# Lines of data: 2999
# nw
# Fea.  Weight
1       1
2       1
3       1
#
# gr
# Fea.  Weight
1       0.0428445870345557
2       0.185070180760327
3       0.325371814230901
#
# ig
# Fea.  Weight
1       0.213887591411729
2       0.669704582861074
```

```
3       1.27807624584789
#
# sv
# Fea.  Weight
1       0.0762436694064095
2       0.233998145488354
3       0.596896311429044
#
# x2
# Fea.  Weight
1       914.619058199289
2       2807.0417532783
3       7160.36815190281
#
```

6.2.3 Value difference files

Using the MVDM metric, it can sometimes be interesting to inspect the matrix of conditional class probabilities from Equation 5.10. 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.

6.2.4 Tree files

Although the learning phase in TiMBL is relatively fast, it can be useful to store the internal representation of the data set both for later usage and for faster subsequent learning. In TiMBL, the data set is stored internally in a tree structure (see Section 5.2). 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; on the same data, IGTREE trees are usually considerably smaller than IB1 trees. In either tree type, the arcs represent feature values and nodes contain class 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. First 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 is provided[5]. Subsequently, unless hashing has been set off explicitly (`-H`), a legenda is given of numeric hash codes for the class names (one unique integer per class) and feature value names (one unique integer per value). Subsequently, the tree's nodes and arcs are given in a proprietary non-indented bracket notation.

Starting from the root node, each node is denoted by an opening parenthesis "(", followed by an integer coding the default class. After this, there is the class distribution list, within curly braces "{ }", containing a non-empty list of category codes followed by integer counts. After this comes an optional comma-separated list of arcs to child nodes, within "[ ]" brackets. An arc is labeled with a coded feature value. The node that the arc leads to again has a class distribution, and any number of child nodes pointed to by arcs.

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

```
# Status: complete
# Permutation: < 1, 3, 2 >
# Numeric: .
# Version 4 (Hashed)
#
Classes
1       nut
2       screw
3       key
4       pen
5       scissors
Features
1       small
2       compact
3       1
4       long
5       none
6       large
7       2
8       other

(1{ 1 3, 2 2, 3 3, 4 2, 5 2 }[1(1[3(1[2(1{ 1 3 })
,4(3{ 3 1 })
]
)
,5(2[2(2{ 2 1 })
,4(2{ 2 1 })
]
)
,7(3[8(3{ 3 1 })
]
)
]
)
,6(4[3(3[4(3{ 3 1, 4 1 })
]
)
,5(4[4(4{ 4 1 })
]
)
,7(5[4(5{ 5 1 })
,8(5{ 5 1 })
]
)
]
```

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

```
)
]
)
```

The corresponding compressed IGTREE version is considerably smaller.

```
# Status: pruned
# Permutation: < 1, 3, 2 >
# Numeric: .
# Version 4 (Hashed)
#
Classes
1       nut
2       screw
3       key
4       pen
5       scissors
Features
1       small
2       compact
3       1
4       long
5       none
6       large
7       2
8       other

(1{ 1 3, 2 2, 3 3, 4 2, 5 2 }[1(1{ 1 3, 2 2, 3 2 }[3(1{ 1 3, 3 1 }[4(3{ 3 1 })
]
)
,5(2{ 2 2 })
,7(3{ 3 1 })
]
)
,6(4{ 3 1, 4 2, 5 2 }[3(3{ 3 1, 4 1 })
,7(5{ 5 2 })
]
)
]
)
```

TiMBL tree files generated by versions 1.0 to 3.0 of TiMBL, which do not contain hashed class and value names, are no longer recognized in current TiMBL versions. Backward compatibility to trees generated by versions 1.0 to 3.0 is preserved in TiMBL version 4 up to release 4.3.1.

6.3 Server interface

It is not always practical or possible to have all test items in one static test file. Example cases include:

- The output of one classifier is being reused as a feature of some other classifier instantly, hence it is not available until it is processed.
- 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, processing any given commands, such as to classify a new example. 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 Chapter 6.1 below). Only the following commandline options are valid in this context: `k, m, d, B, L, Q, 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
NULL_VALUE           :
TREE_ORDER           : G/V
DECAY                : Z
INPUTFORMAT          : Column
```

```
    SEED                 : -1
    DECAYPARAM           : 1.000000
    SAMPLE_WEIGHTS       : -
    IGNORE_SAMPLES       : +
    PROBALISTIC          : -
    VERBOSITY            : F
    EXACT_MATCH          : -
    USE_INVERTED         : -
    GLOBAL_METRIC        : Overlap
    METRICS              :
    NEIGHBORS            : 1
    PROGRESS             : 100000
    TRIBL_OFFSET         : 0
    IB2_OFFSET           : 0
    WEIGHTING            : GRW
    ENDSTATUS
```

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