Fast Transformation-Based Learning Toolkit

Radu Florian and Grace Ngai
{rflorian,gyn}@cs.jhu.edu
Johns Hopkins University

October 3, 2002

Contents

Chapter 1

Preface

The TBL toolkit presented here is the result of work done by the NLP group at Johns Hopkins University over a period of 2 years. It includes ideas from 3 papers presented by the authors over the 2000-2001 period.

The goal of this toolkit is rather broad: to handle any discrete classification task[1] and to offer a quick and simple way to obtain either baseline results for a particular task, or to obtain a classifier that performs the task at hand. For the toolkit to be able to address the problem, the problem must first be cast as a classification problem, where the goal is to assign classifications to a set of samples, selected from some vectorial space.

1.1 How to Read this Report

The report first introduces the transformation-based learning paradigm in Section 2. Section 3 describes the structure of the system, including the file formats used and the conventions for declaring the rule types. Section 4 quickly describes how the toolkit major applications (the rule list training and application) should be launched, and briefly touches on the interaction between the rules that can appear. Section 5 describes some test cases – for people that are most interested in POS tagging, for instance, this is the section that needs to be read – mainly section 5.2 and subsequent sections. For the people with very little time on their hands in search for POS tagging, the Section 5.2.3 describes the 2 scripts that will do most of the work for this task. Also presented are base NP chunking (Section 5.1) and word sense disambiguation (Section 5.3), the latter being presented as a case where samples are independent. Finally, in the Appendix all the parameters that can be defined in the parameter file are described in Section A.1), and all the scripts that are part of the distribution are also described, together with all their parameters (Section A.2).

If you rather went directly into business, go to the directory *test-cases* of the distribution, pick the appropriate problem for you, read the short README file in the directory (which lists mostly the commands to run for the problem), copy the template files from that directory, adapt the rules to fit your problem and take it from there; come back to this report if you have some trouble with the programs, or want to find some flag that does something or the other...

1.2 Contacting the Authors

If you have problems with the code (compiling, found a bug, cannot make it to work, want to extend the code, etc), you can contact the authors at:

rflorian@cs.jhu.edu
gyn@cs.jhu.edu
fnTBLtk@nlp.cs.jhu.edu

The last address is the mailing list address, so if you send mail there, it will be forwarded to all the people that are subscribed to the mailing list — that might be the fastest way to get your problem fixed. If you want to submit changes to the program, submit a patch file to rflorian@cs.jhu.edu, describing the changes that were made, and we will include the patch (and your name) in the next release of the code.

[1] A task in which the vector components are discrete values rather than continuous values.

Chapter 2

An Introduction to Transformation-Based Learning

In 1992, Eric Brill introduced the formalism of the transformation-based learning algorithm in its current formulation, but probably a more well-known reference to the technique is Eric Brill's article from 1995, [Bri95].

The central idea behind transformation based learning (henceforth TBL) is to start with some simple solution to the problem, and apply transformations – at each step the transformation which results in the largest benefit is selected and applied to the problem. The algorithm stops when the selected transformation does not modify the data in enough places, or there are no more transformations to be selected.

While TBL can be used in a more general setup, the toolkit we are describing here is merely concerned with classification tasks, where the goal is to assign classifications to a set of samples. Although this assumption might be restrictive (not all tasks of interest may be transformed into a classification task – e.g. parsing), most natural language related tasks can be cast as classification tasks. It is not inconceivable, though, that an extension of the current toolkit for such tasks might be build, but some extensive modifications to the code are needed.

The fast TBL system (henceforth fnTBL[1]) implements ideas from 3 articles: generating probabilities using the transformation rule list [FHN00], fast training[NF01] and multi-task classification [FN01].

2.1 The Algorithm Description

To make the presentation clearer, we will introduce some notations that will be used throughout the documentation (the notation is compatible with the above-mentioned articles):

1. X will denote the space of samples. How a sample is represented is dependent on the task. For example, in prepositional phrase attachment (the problem of deciding the point of attachment of a prepositional phrase), a sample can be formed from a sentence by extracting the head verb of the main verb phrase, the head noun of the noun, the preposition and the head noun of the noun phrase following the preposition:

   $$\text{I washed the shirt with the pockets} \Rightarrow (\text{washed}, \text{shirt}, \text{with}, \text{pocket})$$

   In part-of-speech tagging (the task of assigning a label to each word in a sentence, corresponding to its part-of-speech function), a sample might be represented as the word itself

   $$\text{the}_{\text{DT}}\ \text{car}_{\text{NN}}\ [\text{exited}_{\text{VBD}}]\ \text{the}_{\text{DT}}\ \text{road}_{\text{NN}} \Rightarrow \text{exited}$$

   This example is contrasting with the previous one in the following way: in the first one the samples in the training set form a *set* (the samples are independent), while in the second one they form a *sequence* (they are interdependent). TBL (and fnTBL in particular) can handle easily both cases; in fact, in the case when they are interdependent, the TBL shows it's full learning power. In the case where the samples are independent, it might be wise to try also some other method (decision trees, for instance), as the greedy approach of TBL can sometimes lead to sub-optimal results.

[1] fnTBL is to be pronounced as funtible $< f\Lambda nt : bl >$

Still, TBL possesses an advantage over standard machine learning techniques such as decision trees and decision lists, as it's error driven, therefore optimizing directly the error-rate (rather than other presumably correlated metrics, such as entropy or Gini index).

2. $\mathcal{C}$ will denote the set of possible classifications of the samples. For example, in the first task presented at point 1. the classification set $\mathcal{C}$ consists of the valid POS tags (NN, NNS, VBP, etc), while in the second case the set $\mathcal{C}$ consists of the labels *s* and *n*, corresponding to the verb attachment and noun attachment, respectively.

3. $\mathcal{S}$ will denote the state space $\mathcal{X} \times \mathcal{C}$ (the cross-product between the sample space and the classification space – each point in this space will be a pair of a sample and a classification)

4. π will usually denote a predicate defined on the space $\mathcal{S}^+$ – basically a predicate on a sequence of states. For instance, in the part-of-speech case, π might be

$$\left(\text{word}_{-1}, \text{tag}_{-1}\right) = (\text{car}, \text{NN}) \text{ and } (\text{word}_0, \text{tag}_0) = (\text{exited}, \text{VBD})$$

or

$$\text{word}_{-1} = \text{the}$$

5. A rule r is defined as a pair (π,c) of a predicate and a target state, and it will be written as $\pi \Rightarrow$ c for simplicity. A rule r=(π,c) is said to apply on a sample s if the predicate π returns true on the sample s. Examples of rules include

$$\text{preposition} = \text{of} \Rightarrow \text{attachment} \leftarrow \text{n}$$

and

$$\text{pos}_{-1} = \text{MD and pos}_0 = \text{VBD} \Rightarrow \text{pos} \leftarrow \text{VBN}$$

Obviously, if the samples are interdependent, then the application of a rule will be context-dependent.

6. Given a state $s = (x, c)$ and a rule $r = (\pi, c')$, the state resulting from applying rule r to state s, $r(s)$, is defined as

$$r(s) = \begin{cases} s & \text{if } \pi(s) = \text{false} \\ (x, c') & \text{if } \pi(s) = \text{true} \end{cases}$$

7. We assume that we have some labeled training data T which can be either a set of samples, or a sequence of samples, depending on whether the samples are interdependent or not; We also assume the existence of some test data $\overline{T}$ on which to compute performance evaluations.

8. The score associated with a rule $r = (\pi, c)$ is usually the difference in performance (on the training data) that results from applying the rule:

$$\text{Score (r)} = \sum_{s \in T} score\left(r(s)\right) - \sum_{s \in T} score\left(s\right)$$

where

$$score\left((x, c)\right) = \begin{cases} 1 & \text{if c} = \text{truth}(x) \\ 0 & \text{if c} \neq \text{truth (x)} \end{cases}$$

The TBL algorithm can be now described as follows:

1. Initialize each sample in the training data with a classification (e.g. for each sample x determine its most likely classification c); let T_0 be the starting training data.

2. Considering all the transformations (rules) r to the training data T_k, select the one with the highest score

$$\text{Score (r)}$$

and apply it to the training data to obtain

$$T_{k+1} = r\left(T_k\right) = \left\{r(s) \,|\, s \in T_k\right\}$$

If there are no more possible transformations, or Score (r) $\leq \theta$, stop[2].

[2] θ is a threshold that needs be set in advance; it can be 0, in which case the algorithm learns until there are no more rules to be learned.

3. $k \leftarrow k + 1$

4. Repeat from step 2.

An easy observation that needs to be made is that the pseudo-code presented above is an algorithm, i.e. it will finish on any data that is given. The argument is simple: let us denote the number of errors in T_k by e_k. Then we have the following recurrence:

$$e_{k+1} = e_k - \text{Score}\,(r) \tag{2.1}$$

Since

$$\text{Score}\,(\text{r}) > \theta \geq 0$$

at each stage where we apply a rule, (2.1) implies that $e_{k+1} < e_k$ for any k. Since $e_k \in \text{N}$, it results that the algorithm will stop at some point in time, therefore avoiding, for instance, garden paths.

2.2 The Speedup

The most expensive step in the TBL computation is the computation of the rules' score. There have been many approaches that try to solve the problem in different ways:

- In Brill's tagger [Bri95] (probably the most used TBL software package), the rules are regenerated at each step: only the rules that correct the errors are generated and the good counts are computed this way; then they are examined in the decreasing order of their good counts and the bad counts are computed by applying the rules to each sample — this approach has the advantage that it can stop early (i.e. not evaluate rules whose score cannot be better than the best rule found so far). The disadvantage is that it slows down considerably as the best rule score decreases.
- Ramshaw & Marcus [RM94] proposed a different approach: for each rule store all the positions in the corpus where it applies and for each sample store the list of rules that apply to it. This trick makes the update very fast, but unfortunately requires very large amounts of memory.
- Samuel [Sam98] presented a Monte-Carlo approach to TBL, where only a part of the rule space is explored to determine the "best" transformation to apply to the data.
- Lager, in his μTBL system [Lag99], implements an efficient Prolog version of TBL, which achieves speed-ups by "lazy" learning.

fnTBL (as described in [NF01], provided with the distribution) achieves its speedup by generating the rules both for the good and bad counts computation. The main idea is to store the rule counts and to recompute them as necessary, when a newly selected rule gets applied to the corpus. The advantage is that only samples in the vicinity of the application of the best rule need to be examined and the rules that apply on them need to have their counts updated — and to identify those rules, we are "generating" them — using the set of templates to identify the rules that apply on those positions and update their counts (good and bad) if necessary. This approach can obtain up to 2 orders of magnitude speed-up relative to the approach present in Brill's tagger.

Chapter 3

System Description

The fnTBL system has 2 executables:

1. fnTBL-train – that learns the rules that are to be applied
2. fnTBL – that applies the rules to the new test data

Both of them have a set of command line parameters, which will be discussed in the following subsections, and read also from a parameter file, where some parameters that can be changed from problem to problem, are specified.

3.1 The Parameter File

The parameter file is used to store all the parameters that do not change often during the development of a particular task. We found it to be useful, because it reduces significantly the number of command-line parameters that need to be specified, and modifying one parameter does not require the program to be recompiled (as it would if the parameters would be hard-wired in the program).

Its format is as follows:

$$< parameter\_name >< space* >=< space* >< parameter\_value >;$$

where:

- *parameter_name* represents the name of the parameter
- *parameter_value* represents the value of the parameter

The parameter value can also contain previously defined parameters, if they have the dollar sign $ in them. For instance, ${MAIN} is a legal value if the variable MAIN was previously defined. I found this to be useful if one wants to define a main directory, where all the other files reside; when one changes the location of the main directory, only one line needs to be modified.

The following are legal examples of lines in the parameter file:

```
MAIN = /remote/bigram/usr/home/rflorian/workdir/research/data/mtbl-toolkit/text_chunking;
FILE_TEMPLATE = ${MAIN}/file.templ;
RULE_TEMPLATES = ${MAIN}/lexical_predicates.richer.templ;
REASONABLE_SPLIT = 10;
REASONABLE_DT_SPLIT = 5;
EMPTY_LINES_ARE_SEPARATORS = 0;
ELIMINATION_THRESHOLD = 0;
```

The example is actually extracted from a parameter file associated with the fnTBL POS-tagger. A complete set of valid parameters are provided in the Appendix. The most important ones are:

- FILE_TEMPLATE – contains the name of the file with the sample feature names (see 3.2.2);
- RULE_TEMPLATES – contains the name of the file with the rule description (see 3.2.3);
- CONSTRAINTS_FILE – contains the name of the constraints file (see 3.2.5);
- LOG_FILE – contains the name of a file where all the commands will be written – useful to keep track what commands were used, if the parameter is defined;
- EMPTY_LINES_ARE_SEPARATORS – describes if the empty lines are to be considered separators or not. If the samples are interdependent (e.g. POS tagging), then the value should be 1 (they are separators). If the samples are independent, then it should be 0. The reason for its existence is that sometimes even if the samples are independent, one might want to separate the samples into sentences (for instance, in lexical POS tagging, Section 5.2)

3.2 File Formats

This section will briefly describe the file formats used with the fnTBL toolkit. In all the following files, the lines starting with a '#' sign are considered as comments and, therefore, are ignored.

3.2.1 Training and Test File Formats

Both the training file and the test file need to be in a particular format for the fnTBL tools to work properly. The format requires that each sample be on a separate line, with the features separated by white space (spaces or tabular characters). In addition, if the samples are interdependent (like in the POS tagging case), and organized into "blocks" (i.e. sentences), the blocks need to be separated by a blank line.

Here's an example of such a file:

```
Revenue NN NN
rose VBD VBD
5 CD CD
% NN NN
to TO TO
$ $ $
282 NN CD
million CD CD
from IN IN
$ $ $
268.3 NN CD
million CD CD
. . .

The DT DT
drop NN NN
in IN IN
earnings NNS NNS
had VBD VBD
been VBN VBN
anticipated VBN VBN
by IN IN
most JJS JJS
Wall NNP NNP
Street NNP NNP
analysts NNS NNS
, , ,
but CC CC
the DT DT
results NNS NNS
were VBD VBD
reported VBD VBN
after IN IN
the DT DT
market NN NN
```

closed VBD VBD
. . .

There are 3 fields for each word (in this case)[1]:

1. The word itself (e.g. *closed*);
2. The most likely part-of-speech associated with the word (e.g. *VBD*);
3. The true POS of the word[2] (e.g. *VBN*).

As presented in the Section 2.1, the system will start to learn transformations that correct the second field to match, as much as possible, the third field.

Both the training and test data should have this format. Some tools that compute the most likely tag given the word (for instance) are provided with the fnTBL distribution and are described in the Appendix section.

3.2.2 FILE_TEMPLATES

The names of the fields associated with the samples are defined in a file whose name is specified in the parameter file. The name of the parameter is **FILE_TEMPLATES** and the file contained in this parameter should have the following format:

$$\langle feature_1 \rangle \ldots \langle feature_n \rangle \langle class_1 \rangle \ldots \langle class_m \rangle => \langle truth_1 \rangle \ldots \langle truth_m \rangle$$

where

- $\langle feature_i \rangle$ is the name of the i^{th} feature associated with the sample;
- $\langle class_j \rangle$ is the name of the j^{th} "guess" associated with the sample;
- $\langle truth_j \rangle$ is the name of the j^{th} "truth" associated with the sample; there are the same number of $\langle class \rangle$ and $\langle truth \rangle$ features.

To make this clearer, here's an example of a template file, taken from POS tagging:

word pos $=>$ tpos

In this example, the feature is *word*, *pos* is the name of the "guess" of the system at some point in time, and *tpos* is the name of the truth. If one wants to perform a simultaneous classification of POS tagging and text chunking (see [FN01]), then the corresponding file should be:

word pos chunk $=>$ tpos tchunk

where the features are again *word*, the guesses are named *pos* and *chunk* and the truths (there are 2 classifications in this case) are named *tpos* and *tchunk*.

3.2.3 RULE_TEMPLATES

Because the fnTBL is a general-purpose tool, the user needs to specify the rule templates – in other words, to describe the kind of rules the system will try to learn. The description is done by the templates: a template defines which features are checked when a transformation is proposed. For instance, if one wants to implement a bigram look-up, then the template consists of the previous word and the current word.

The predicate of a rule is created as a conjunction of smaller, atomic, predicates that can be as simple as feature identities; other types verify if the word ends/begins with a specific suffix/prefix and yet others verify the identity of one of the previous words. This particular choice of conjunction is selected for:

[1] In general, there should be $|F| + 2 \cdot |C|$ features, where F is the set of features associated with the sample and C is the set of classifications. For POS tagging $F = \{word\}$ and $C = \{pos\}$.

[2] The gold-standard POS, to be more precise.

1. Simplicity – it is easier to represent the rules and templates this way, both internally and externally (as text);
2. Speed-up – the algorithm exploits heavily this choice; if disjunction had been allowed between atomic predicates, the algorithm would have been a lot more complicated and no significant speed-up would have been obtained.

A list of atomic predicates' notation is presented after the rule description.

The format of a rule template is

$$\langle p_1 \rangle \dots \langle p_k \rangle => \langle c_1 \rangle \dots \langle c_l \rangle$$

where:

- $\langle p_i \rangle$ are the atomic predicates, with $\langle feature_1 \rangle \dots \langle feature_n \rangle, \langle class_1 \rangle \dots \langle class_m \rangle$ as arguments
- $\langle c_j \rangle$ are the classifications that are to be changed by the rule, and are chosen from $\langle class_1 \rangle \dots \langle class_m \rangle$

For example

$$\text{word\_} - 1 \text{ word\_}0 => \text{pos}$$

is defining a rule that based on the previous and current word will change the POS feature, while

$$\text{pos\_} - 2 \text{ pos\_} - 1 \text{ pos\_}0 => \text{ pos}$$

will change the POS of a word based on a POS trigram ending on the current position.

Motivated by some of the problems TBL has been applied to, some types of atomic predicates have been implemented. Here is the list of how they can be specified:

1. Feature identity:

   $$\langle feature \rangle$$

   will check the identity of a particular feature.

2. A feature of a neighboring sample (in the case of interdependent samples):

   $$\langle feature \rangle \_ \langle index \rangle$$

   e.g. *word_1*, *pos_-1*, *chunk_0*. $\langle index \rangle$ can be negative as well; there is a restriction that the number has to be an integer belonging to the range $[-128, 127]$;

3. A feature that checks the presence of a particular feature in sequence of samples:

   $$\langle feature \rangle : [\langle index\_start \rangle, \langle index\_end \rangle]$$

   e.g. *word:[1,3]* checks the presence of the particular word in the samples on positions +1,+2 or +3, that is:

   $word : [1, 3] = the$ returns true on a sample $s_i \Leftrightarrow s_{i+1}$ or s_{i+2} or s_{i+3} contains the word the

4. A feature checking that one of the features present in an enumeration has a particular value (useful for independent samples, see the PP attachment case study):

   $$\{\langle feature_1 \rangle, \dots, \langle feature_n \rangle\}$$

   e.g. $\{class_1, class_2\} =$ use%2 : 34 : 01 :: will return true on a given sample s if and only if one of the features $class_1$ or $class_2$ of sample s is use%2 : 34 : 01 ::[3].

One of the big advantages of the fnTBL toolkit is that the predicate structure is modular, and adding a new atomic predicate type to the toolkit should not present a big programming challenge; see the section about the code design for more information.

[3]The presented class is the Wordnet label of one of the senses of the word *use*.

To make life easier, an additional construction is allowed in the rule file format – definitions of a rule placeholder:

$$\langle variable\_name \rangle = \langle rule\_definition \rangle$$

once a variable is defined, it can be recalled as $\$ \langle variable\_name \rangle$ in the rule template file. This construction helps mostly with "set" atomic predicates (the predicate will return true if any one of a set of features has a given value). The following is an rule template example taken from the PP-attachment problem:

$$\begin{array}{l} verb\_parent = \{vp1, vp2, vp3, vp4\} \\ noun\_parent = \{np1, np2, np3, np4\} \\ \ldots \\ \$verb\_parent\, \$noun\_parent \Rightarrow attachment \end{array}$$

In here, a predicate-set placeholder is defined – $verb\_parent$ can replace the predicate $\{vp1, vp2, vp3, vp4\}$ in subsequent rules. The rule template $\$verb\_parent\, \$noun\_parent \Rightarrow attachment$ will try to predict the attachment of a preposition using wordnet parents of the verb and the first noun of the sample.

3.2.4 The Rule File Format

Both the main programs from the **fnTBL** toolkit (*fnTBL* and *fnTBL-learn)* use rule files files which list the rules learned by the system. These rules have meaningful linguistic information in them, and can be easily read and understood. The format they are presented in is similar to the one of the template files, rules being actually instantiated templates. The format is:

$$\langle pred_1 \rangle = \langle value_1 \rangle \ldots \langle pred_l \rangle = \langle value_l \rangle => \langle class_1 \rangle = \langle cvalue_1 \rangle \ldots \langle class_p \rangle = \langle cvalue_p \rangle$$

where $\langle pred_i \rangle$ is an atomic predicate, $\langle value_k \rangle, \langle cvalue_k \rangle$ are valid values for the respective predicates and $\langle class_p \rangle$ are the classes to be changed. For instance, again for the POS tagging problem, the rule

$$POS_{-1} = DT\ POS_0 = VB => POS = NN$$

will change all the occurrences of infinitive verbs to nouns if the previous part-of-speech is a determiner; similarly, the rule

$$preposition = \text{at} => attachment = \text{v}$$

will change the attachment from noun to verb if the preposition is *at*, in the case of prepositional phrase attachment.

3.2.5 The Constraints File

One less usual parameter for the fnTBL algorithm is the constraints file. The TBL algorithm suffers from one relatively major problem – since it's output is not probabilistic, it has no mechanism of estimating the posterior probability of a class given a sample

$$P(c|s)$$

and therefore it might assign classes to samples that might not make any sense at all. For instance, it might assign to the word *the* the POS tag *VB*, which is, of course, very unlikely. Most probabilistic classifiers do not suffer from this problem, as they have appropriate ways of computing the posterior probability $p(c|s)$. To avoid this problem, the TBL algorithm proposed originally by Eric Brill enforced that no new classes should be assigned to samples: if a sample has been seen in training, then a rule that would change the classification to a classification that has never been seen with that particular sample is not allowed to apply.

We are relaxing here this constraint by allowing the user to specify the set of allowable associations feature-class for a set of examples, by using constraint files. If a particular feature is not present in these constraint files, then a rule that examines that feature can change it as it sees fit. However, if the feature is present in the file, only rules that change it to some specified class are allowed to apply to it. This mechanism allows the user to specify, for instance, that only words which were seen enough times in the data are not allowed to be associated with new classes.

The format of the constraint file is:

$$\langle feature_1 \rangle \ \ldots \ \langle feature_n \rangle \ \langle classification_j \rangle \ \langle file\_name \rangle$$

where $\langle feature_i \rangle_i$ are features on which the conditioning needs to be done, $\langle classification_j \rangle$ is the classification that is constrained and $\langle file\_name \rangle$ is the file containing the constraints – a file containing a list of the pairs that need be constrained. For instance, in the POS tagging example, a constraint in the constraint file could be

$word\ pos$ constraints.dat

and the file *constraint.dat* might contain

saying NN VBG
the DT
their PRP$
them PRP

Any combination of features that does not appear in the constraint file are not constrained at all, and it's possible for those samples to be assigned any classification. Also, it is useful to note that the constraints are stackable; in other words, first a sample is checked against all the constraints – if it passes, then the class change is performed.

Chapter 4

Using The Toolkit

4.1 Creating the Training/Test Files

The first step in using the fnTBL toolkit is to create the training and test files. Even though there are some tools provided with the toolkit that help with the file creation, most preprocessing is left to the user. No tokenization or end-of-sentence detection is performed (even if this may change, as it easy to train a TBL system to perform EOS detection).

Once the corpus is in the required format (one word per line), the tools provided can augment it with the most likely tag given some features (for instance, given the word), and can construct the constraint files. In the POS tagging case, an almost complete solution to creating the initial files is provided (see the POS test case).

In this initial release, the probability model does not work properly, and it should not be used. This problem will be fixed in the next release.

4.2 Training using fnTBL

Once the training file is in place, the rule file can be generated. This is the most time-intensive step of the process, as the main search is performed during training. To train from a corpus, the user should use the command:

fnTBL-train <*train_file*> <*rule_output_file*> [-F <*param_file*>] [-*threshold* <stop_thr>] [-pv] [-t <*tree_prob_file*>] [other options]

where:

- <*train_file*> represents the training file;
- <*rule_output_file*> is the file where the rules will be output;
- <*param_file*> is the file describing the main system parameters (see 3.1);
- <*stop_thr*> sets the stopping threshold – the algorithm stops when a rule with this score is reached – the default is 2;
- -p – turns on the probabilistic classification (the tree generation, as described in [FHN00]);
- -v – turns on some verbose output;
- -V <*verbosity_level*> – defines the verbosity level (5=max) – use with caution  ;
- <*tree_prob_file*> – defines the file in which the probability tree is output into (again, see [FHN00]).

The other options are:

- -allPositiveRules <*number*> – will output all the rules that at the end of the training have a number of good applications greater than the provided number, and 0 bad applications —see section 4.6 for details. The provided number can be of the following form:

- ▷ $n, n > 0$ – in which case it represents the generation threshold;
- ▷ $n\%, n > 0$ – in which case it represents the percentage of samples the rule must apply to;
- ▷ $-n, n > 0$ – in which case the selection threshold is set to $max - n$ where max is the score associated with the best rule selected;

- -minPositiveScore <*number*> — will restrict the rules described at the previous bullet to have at least the given number of goods (in case the number supplied to the -allPositiveRules directive is not a positive integer);

While the program is running, it will output the rules that are selected as it progresses, along with the time it took to compute the current rule. At the end, the total running time is also printed. If you don't want this output, you can redirect the stderr to /dev/null, by using a command like

$$< command >> \&/dev/null$$

if you're using tcsh or

$$< command > 2 > /dev/null$$

if you're using bash/ksh etc.

Also, the <*param_file*> parameter can be specified as the value of the shell variable DDINF. So, if you don't want to specify it as a command line parameter, you can set the shell variable to point to the appropriate file, and it will be used just the same.

4.3 Classification using fnTBL

Once the rule file is generated, or if you have the file from other sources, and the test file is in the appropriate format, fnTBL program can be used to apply the rules, by using the command:

fnTBL <*test_file*> <*rule_file*> [-o <*out_file*>] [-F <*param_file*>] [-printRuleTrace]

where:

- <*test_file*> is the file containing the test data, in the column format;
- <*rule_file*> is the rule file generated using **fnTBL-train**;
- <*out_file*> is the optional file where the result will be output (default is standard out);
- <*param_file*> defines the file containing the parameters (also can be specified using the shell variable DDINF);
- -printRuleTrace – after each example is printed the sequence of indices corresponding to the rules that applied on the sample – used for debugging/inspecting purposes.

Important observation: the initial set-up of the test data should be equivalent to the initial set-up of the training data (i.e. using the same most-likely distribution), or the program will not behave properly, as it is to be expected.

A scoring program is also provided, as described in Section A.2.

4.4 Warnings and Errors

One possible warning that can be output by the program has the following form:

$$\text{!! pos\_0} = \text{VB pos\_1} = \text{NN} \Rightarrow \text{pos} = \text{NN does not have } 0 \text{ goods and } 0 \text{ bads } (\text{good} : 2 \text{ bad} : 0)$$

This message warns the user that after applying a rule, its counts are not 0. Normally, after applying a rule to the corpus, the rule does not apply anymore, therefore its good and bad counts should be 0. However, there are cases where this is not true:

consider the case of a "recursive" rule[1] such as the above mentioned one. After applying the rule in the following context:

the	DT	DT
will	VB	NN
reading	VB	NN
session	NN	NN

(on line 3), the rule is applicable again on line 2, while it was not before the application. If the rule is not recursive, then you probably found a bug in the fnTBL code.

A definite error message may appear:

Oups : you found a bug in the TBL code

This message appears if the last rule has been selected 5 times in a row – it usually happens if there is a bug in the updating score of the system. Since the algorithm needs to keep track of the exact counts for each rule at every step of the algorithm, any mistake in count-keeping will result in this error, sooner or later. If you manage to obtain this message, please contact the authors and submit a bug report – see Section 4.8 for details.

4.5 Rule Interaction

As mentioned in Section 2.1, the algorithm will select the rules in the decreasing order of their score. At some point, however, will have to choose between rules that have identical scores. One option would be for it to choose randomly; this is a not-too-desirable behavior, since the results are no longer entirely replicable. Therefore, we made this decision making completely deterministic, as follows: given 2 rules r_1 and r_2 we decide to choose r_1 over r_2 if and only if the following condition holds:

1. $score\,(r_1) > score\,(r_2)$ or
2. $score\,(r_1) = score\,(r_2)$ and
   (a) r_1 has more tokens than r_2 (more atomic predicates) or
   (b) r_1 and r_2 have the same number of atomic predicates and
      i. the template of r_1 was declared before the template of r_2 or
      ii. r_1 and r_2 have the same template and the target of r_1 has a lower index in the vocabulary as the target of r_2.

The procedure is rather complicated, but the choices are based on the experience the authors had in developing the toolkit. Choice 2a may seem a little strange, since is the opposite of Occam's razor, we think it is better to make finer decisions than more general ones, as the finer ones will result in errors less often at the expense of the rule not applying that often – basically, we prefer precision to recall.

To leave more room for experimentation, we have also implemented the rule ordering such that it allows the user to specify the method by which ties are broken. The parameter ORDER_BASED_ON_SIZE (see A), will choose among the following options:

1. The method described above – ORDER_BASED_ON_SIZE=0;
2. The method described above, but with 2a reversed (i.e. choose the rules with less atomic predicates) for ORDER_BASED_ON_SIZE=1;
3. The size is not considered when comparing the predicates – the comparison is based on the order they appear in the rule template file – for ORDER_BASED_ON_SIZE=2.

While the choice 3 is the one that gives the most freedom, it can be a little annoying to think about the relationships between each rule template; if you don't feel like doing it, just select one of the other ones. The default is 0.

[1] A rule whose output classification is part of the decision – such as the example above – the classified class – NN – is part of the predicate.

4.6 Termination Conditions

The training phase finishes when either of the following conditions are met:

- There are no more useful rules (i.e. no of good applications greater than the number of bad applications) can be generated;
- A rule with a score lower than the specified termination threshold is generated.

The transformation-based algorithm suffers from a serious drawback when the training data is small: it does not allow for redundancy. If there are 2 good explanations of a phenomenon (observed through 2 rules that have similar scores), only one will be selected by the process, while the second one will be completely discarded. If the training data is sparse, it can happen that the first rule does not apply to a particular sample, while the second one does. This phenomenon was observed by the authors especially when the training samples are independent; section 5.3 presents such a case.

To help alleviate this problem, the algorithm can be amended as follows:

- After the termination condition is met, select all the transformation rules that have only positive applications and output the ones with highest score (as specified by the -allPositiveRules flag).

This small change has the some advantages:

- It will help correct some of the problems with the non-redundancy of TBL, by selecting some of the alternative explanations ignored by the main algorithm;
- It does not modify the output of the algorithm if run on the training data, because the rules selected in the end do not have any negative application.

One important observation: if one wants to use this feature, one should have defined rule templates that do not depend on the classification. For instance, beside having the rule

$$\text{word\_} - 1 \text{ pos\_}0 \Rightarrow \text{pos}$$

you should also have the rule

$$\text{word\_} - 1 \Rightarrow \text{pos}$$

While this might seem as a strange condition, it is made necessary by the fact that if all the rule templates are dependent on the current classification, then the "positive" rules generated at the end will either have a score lower than the best rule (otherwise they would have been the best rule) or will have a non-useful form (e.g. word_-1 pos_0=NN$\Rightarrow$ pos=NN).

Finally, the rules are output in the direct order of their good counts, such that better rules get the final decision in deciding the classification of a sample.

4.7 Data Representation Size

The fnTBL toolkit tries to keep the amount of memory used to a minimum. To do this, it is set-up by default to use the following data types:

- For the feature values (e.g. words, POS, chunk tags, etc) – **unsigned int** (32 bits on most compilers; 4294967296 max) – the corresponding type is **wordType;**
- For the indices to feature templates (e.g. words, POS, chunk tags, etc) – **unsigned char** (8 bits on most compilers; 256 max vals);
- For the feature positions (e.g. feature indices) – **unsigned char** (8 bits, 256 max);
- For feature differences (e.g. how many features before/after the current one) – **signed char** (8 bits, $[-128, 127]$, 256 max).

The user has the possibility of adjusting the sizes of these types, making the approach usable when the data requires it, as follows:

- Changing the feature value representation size (e.g. your data vocabulary size is greater than 64k) – edit the Makefile, and replace the definition

  $$\text{WORD\_TYPE} = \text{'unsigned short'}$$

  with (for instance)

  $$\text{WORD\_TYPE} = \text{'unsigned int'}$$

  and recompile the program. You may need to run "make clean" before recompilation, to be sure that all the sources are remade properly. If you have more than 4294967296 samples in the corpus (you really have that many?) then you should set it to "unsigned long long".

- Changing the feature position size (e.g. you have more than 256 features per example) – edit the compilation variable POSITION_TYPE to the next available type; for instance from

  $$\text{POSITION\_TYPE} =' \text{ unsigned char}'$$

  to

  $$\text{POSITION\_TYPE} =' \text{ unsigned short}'$$

4.8 Bug Reports

The fnTBL toolkit is still in its infancy and it is possible that there are still bugs in the code. We haven't observed any for a few months now, but the toolkit was constructed by us, and it is possible that we never did something to break it. If you manage to break it in any way (that includes the 2 executables and the scripts that came with the toolkit), you are invited to report the bug to the authors, either by e-mail at one of the following addresses:

rflorian@cs.jhu.edu
gyn@cs.jhu.edu
fnTBLtk@nlp.cs.jhu.edu

or using one of the links from the main fnTBL toolkit web page:

http://nlp.cs.jhu.edu/~rflorian/fntbl

When submitting a bug report, we are asking you to submit the following information:

1. A brief description of how the bug was obtained;
2. The configuration files used when you obtained the bug;
3. The training data file and/or the rule files that were used, if possible.

The data files will make the debugging process a lot easier; since we can imagine that giving the authors access to your data files might make you uncomfortable, we promise not to use the data in any other way than for debugging the process and also to delete it as soon as the bug was fixed; we are even willing to sign NDAs, if required.

Chapter 5

Test Cases

5.1 Base Noun Phrase Chunking

We will start the example section with the task of Base NP chunking – where the goal is to identify the basic, non-recursive, noun phrases in a sentence. This task can be converted to a classification task, where each word is to be assigned a class corresponding to its relative position in a noun phrase: inside a noun phrase, beginning a noun phrase or outside a noun phrase. In Figure 5.1, we show an actual sentence from the Wall Street Journal corpus, together with its part-of-speech labeling and the chunk labels. There are many ways to map from base-NP brackets to base NP chunks (as described in[SV99]); in this particular example we will use B for words that start a base NP, I for words that are inside or end one and O for words that are outside any noun phrase.

The samples in this task are 2 dimensional and are of the form

$$(\text{word}, \text{pos})$$

The POS associated with the word is in this case fixed, and is obtained by tagging the data with your favorite tagger (in our case, the data is the one provided by Ramshaw and Marcus, therefore the tagger is Brill's tagger).

Figure 5.1(a) displays an example of parameter file for the baseNP task. The constraints file (*constraints.chunk.templ*) can contain constraints of the following form:

- word – chunk constraints (of the form *oak I B*);
- pos – chunk constraints (of the form *NN I B*).

An example of rule templates one could use are listed in Figure 5.1. A much larger list, consisting of the templates that seemed useful to us, is provided in the directory *test-cases/baseNP* of the main distribution. Experimentation with different templates is usually needed for good results, so feel free to adjust and create new rule templates as you see fit.

5.1.1 Creating the Training and Test Data

We start by assuming that you have the data in a 3 column format – a sequence of samples

$$word\, pos\, tchunk$$

where *word* is the word, *pos* is the part-of-speech associated with it (this can be the true POS or one assigned by a POS tagger) and *tchunk* is the true chunk tag associated with the word (*I*,*O* or *B*). The sentences are presumed to be separated by one blank line.

[	A.P.	Green	]	currently	has	[	2,664,098	shares	]	outstanding	.
	NNP	NNP		RB	VBZ		CD	NNS		JJ	.
	B	I		O	O		B	I		O	O

Base NP transformation to a classification task

```
MAIN = <your_directory>;
FILE_TEMPLATE = ${MAIN}/file1.templ;
RULE_TEMPLATES = ${MAIN}/rule.chunk.templ;
CONSTRAINTS_FILE = ${MAIN}/constraints.chunk.templ;
LOGFILE = ${MAIN}/log_file;
```

(a) Parameter file for baseNP chunking

```
word pos chunk => tchunk
```

(b) File template for baseNP chunking

Sample files for base NP chunking

chunk_-1 chunk_0 ⇒ chunk	pos_0 chunk_0 ⇒ chunk
chunk_0 chunk_1 ⇒ chunk	pos_-1 pos_0 ⇒ chunk
word_-1 word_0 ⇒ chunk	pos_0 pos_1 ⇒ chunk
word_0 word_1 ⇒ chunk	word_0 pos_0 pos_1 ⇒ chunk
chunk_-1 chunk_0 word_0 ⇒ chunk	pos_-1 pos_0 chunk_-1 chunk_0 ⇒ chunk
chunk_0 chunk_1 word_0 ⇒ chunk	pos:[-3,-1] ⇒ chunk
word_-1 word_0 chunk_0 ⇒ chunk	pos:[1,3] ⇒ chunk
word_0 word_1 chunk_0 ⇒ chunk	pos_-2 pos_-1 pos_0 chunk_0 ⇒ chunk
word:[-3,-1] ⇒ chunk	pos_-1 pos_0 pos_1 chunk_0 ⇒ chunk
word:[1,3] ⇒ chunk	pos_0 pos_1 pos_2 chunk_0 ⇒ chunk
chunk:[-3,-1] ⇒ chunk	chunk_-2 chunk_-1 chunk_0 ⇒ chunk
chunk:[1,3] ⇒ chunk	chunk_-1 chunk_0 chunk_1 ⇒ chunk
	chunk_0 chunk_1 chunk_2 ⇒ chunk

Table 5.1: Sample of lexical rule template file

The first step is to determine the most likely chunk tag for each sample; basically one needs to compute either $ML\,(chunk|word)$ or $ML\,(chunk|pos)$. We found that the second one (most likely given the POS) to be more informative, since base noun phrases are very dependent on the part of speech. The following sequence of commands will assume that you want to create the lexicon based on *pos*; if the one on *word* is desired, change the commands accordingly:

- Create the pos lexicon (a list of chunk tags associated with each *pos* value):

```
mcreate_lexicon.prl -d '1=>2' myfile > pos-chunk.lexicon
```

- Add an additional field to the sample (corresponding to the "current" chunk tag) and assign the most likely value (based on pos) to it:[1]

```
perl -ape '$_=''$F[0] $F[1] -- $F[2]\n'' ' myfile | \\
most_likely_tag.prl -p '1=>2' -t O,O -l pos-chunk.lexicon -- > myfile.init
```

The same processing must be performed on the test data as well.

Additionally, you might want to create constraints, which will enforce that the rules not assign priorly unseen chunk tags to the samples. The pos-chunk.lexicon file can be used as constraints on *pos* and *tchunk*. If you want to create constraints also on words (for instance, never assign to *the* the chunk *O*, or to the word '.' the chunk *I*), you could run the command:

```
mcreate_lexicon.prl -d '0=>2' [-n <threshold>] myfile > word-chunk.lexicon
```

where threshold is the minimum count for a word to enter the constraints[2]. This will create a list of chunk tags for each word with count above the threshold, where the tags are sorted in the order of their cooccurrence count. Then you should create a constraint template, for instance :

```
word tchunk word-chunk.lexicon
pos tchunk pos-chunk.lexicon
```

and update the field *CONSTRAINTS_FILE* in the parameter file to point to the correct file.

[1] The argument to the -t parameter of the most_likely_tag.prl script is capital o (O) and not the number 0.

[2] One might want to not impose constraints on low occurrence words, because the statistics on them might not be reliable.

5.1.2 Training the baseNP Chunker

The training can be performed using the command:

```
fnTBL-train myfile.init chunker.rls -F <param_file> [-threshold <threshold>]
```

The threshold you choose can have a big impact on both the training time and the performance of the program. If you have small data, then a low threshold (0) is useful, as learning all the possible transformations can help. If your data is quite large, then using a threshold of 0 can make the program run for a long time (as there are a lot of rules with a count of 1); a threshold of 1 or 2 can prove more useful, as the rules that have a score of 1 can actually hurt performance, especially if the training and test data are not very similar.

One useful property of TBL is that you can restart the training from where you stopped the last time. The only thing needed is to apply the rules you learned so far to the training data, and you can restart the training from the moment you stopped. Also, if you learned all the rules that have a score of 1, for instance, you can eliminate them from the rule list (as they can be easily identified, because the score is present in those files), and retest the performance of the system.

In conclusion, if you use the entire WSJ data to train the baseNP chunker, use a threshold of 1 if your machine is not super fast, otherwise use a threshold of 0, with the option of eliminating the rules with a score of 1 if they hurt performance. If you train on the smaller WSJ data (sections 15-18), then use a threshold of 0, as the training time is relatively short.

5.1.3 Testing using fnTBL

To test, you need to put the test data in the same 4 column mode file (using the commands from Section 5.1.1) and run the command:

```
fnTBL testfile.init -F <param_file>
```

If, in addition, you want to see which rules applied to which samples, add at the end the flag *-printRuleTrace*. This will add after each sample the character '|', followed by the indices associated with the rules that applied on that particular sample; the first index is 0. You could use the script *number-rules.prl*, provided in the distribution, to see the rules' indices.

Finally, the README file present in the test-case/baseNP contains all the command lines needed to run the chunker, and the chunker.wsj-small.rls and chunker.wsj-large.rls files contain the rules as learned by the program on the sections 15-19, respectively 02-21, of the WSJ corpus.

5.2 POS Tagging

The TBL solution to the POS tagging problem is broken down into 2 problems:

- First, the unknown words' POS are guesses, by making use of morphology – the prefixes, suffixes, infixes (of length 1) of the words, etc. – we will call this step lexical POS tagging, or simple lexical tagging;
- Second, starting from the output of the previous step, the words are examined in context to determine the best transformations – we will call this step the contextual POS tagging, or simple contextual tagging.

Though it would be possible for the two tasks to be combined into just one big task, there are good reasons for keeping them separate:

- The lexical tagging task examines the words in isolation, therefore the samples are independent, and the toolkit can solve this problem more efficiently because of the independence property;
- If there are no rule types that use features from both type of rules, it is not clear whether the joint estimation will produce different results from the one obtained by sequential estimation; combining the two types of rules (such as $word\_-1\ word :: ++2 \Rightarrow pos$), usually results in a very large number of rules, large enough so to make the application require too much memory to be practical;
- Since the lexical tagging refers only to unknown words, a much smaller training corpus can be selected (only the unknown words), which results in significantly shorter training time;

- The lexical tagging step uses data structures that are not necessary for the other step (e.g. lexical tries); storing them for the second step is wasting significant amounts of memory;
- Preliminary experiments showed no improvement from a joint estimation;
- The two steps might have different parameters (termination threshold, etc).

The next two sections will describe the type of rules that can be used for each process, and Section 5.2.3 will describe 2 scripts that automatize most of the POS tagging process, so, for people in a hurry, that is the section to read.

5.2.1 Lexical POS Tagging – Guessing the Unknown Words

The approach taken by fnTBL follows the guidelines set by Brill: the training data is split into 2 separate parts. Lexical priors (well, only the most likely sense) is learned using the first part, and then the second part is used to train a TBL system for the unknown words.

We propose that the initial POS assignment on the second part of the corpus be done according to:

$$POS\left(w\right) = \begin{cases} POS_t\left(w\right) & \text{if } C\left(w\right) \geq T \\ NNP & \text{if } C\left(w\right) < T \text{ and } w \text{ begins with a capital letter} \\ NN & \text{otherwise} \end{cases}$$

where w is a word, NNP is the POS tag corresponding to the proper noun, NN is the POS tag corresponding to the noun and $C\left(w\right)$ is the number of times that the word w appeared in the first part of the training set. Let us observe that if $T = 0$ then the assignment is done the same way Brill is doing it.

The rule types that can be used for lexical tagging are:

1. The prefix/suffix of the word is a specified sequence of characters:

   $$\langle feature_i \rangle :: \langle number \rangle \tilde{\ } \tilde{\ } \mid \langle feature_i \rangle :: \tilde{\ } \tilde{\ } \langle number \rangle$$

   For instance the rule

   $$word :: 3 \tilde{\ } \tilde{\ } = pre \tilde{\ } \tilde{\ } \Rightarrow pos = JJ$$

   fires on every word that starts with the letters *pre* and the rule

   $$word :: \tilde{\ } \tilde{\ } 4 = \tilde{\ } \tilde{\ } able \Rightarrow pos = JJ$$

   fires only on words that end with the letters *able* (unable, enjoyable).

2. Adding a specified prefix/suffix results in a word (i.e. is in a specified large vocabulary)

   $$\langle feature_i \rangle :: \langle number \rangle + + \mid \langle feature_i \rangle :: + + \langle number \rangle$$

   For instance

   $$word :: + + 2 = + + ly \Rightarrow pos = JJ$$

   will fires only on words like *like*, *converse*, etc (words to which by adding the suffix *ly* we obtain another word – *likely*, *conversely*);

3. Subtracting a specified prefix/suffix results in a word:

   $$\langle feature_i \rangle :: \langle number \rangle - - \mid \langle feature_i \rangle :: - - \langle number \rangle$$

   For instance the rule

   $$word :: 2 - - = un - - \Rightarrow pos = JJ$$

   will fire on words like *unspecified*, *undo*.

4. The word in question contains a specified character (such as -):

$$\langle feature \rangle :: \langle number \rangle \langle \rangle$$

For instance the rule

$$word :: 1 <>= . <> \Rightarrow pos = CD$$

will fire on each word that has the dot char '.' in it, transforming it into a cardinal.

5. The word in question appears in to the left/right of a specified word in a long list of bigrams (e.g. appears after *the* somewhere in a list of bigrams makes it likely to be a noun):

$$\langle feature \rangle \hat{}\hat{} 1 \mid \langle feature \rangle \hat{}\hat{} - 1$$

For instance the rule

$$word \hat{}\hat{} - 1 = to \Rightarrow pos = VB$$

will fire on all words that appear after *the* in a list of bigrams.

In Table 5.2 a sample of how the rule template file should look like is presented – it is the actual rule file present in the distribution[3]:

There are several steps which need to be taken when performing the lexical training:

1. The training data (already in the column format) has to be split into 2 parts (equal or not);
2. From the first part, most likely information needs to be extracted;
3. Using the most likely POS information, assign the initial tag to the unknown words from the second part; a word from the second part is considered unknown if it didn't appear in the first part;
4. Generate the unigram list from the first part, or from some large corpus;
5. Generate a bigram list from the current corpus, a large corpus or both;
6. Create the rule template file and the parameter file;
7. Run the fnTBL-train program.

5.2.2 Contextual POS Tagging

After the initial stage, when POS is guessed for unknown words, a second learning process is to be applied, one that learns to correct POS in context. To do this, the learner can use a combination of one of the following basic predicate types[4]:

- The identity of one of the words in context:

$$word\_ < number >$$

where $< number >$ can be, in principle, a number ranging from -128 to 127 (we assume that the name of the "word" feature is $word$; otherwise, replace $word$ with the appropriate name); for instance, the rule

$$word\_ - 2 = as \Rightarrow pos = RB$$

will apply only on positions where 2 words back is the word as.

- The identity of one of the POS in context:

$$pos\_ < number >$$

where $< number >$ is the same as before (replace pos with the name of you POS feature, as defined in the file pointed by the parameter FILE_TEMPLATE, if different than pos);

[3] The file is presented here in double column, to save some space, but it's single column in the actual file.

[4] We are describing here the basic predicate types – the rules' predicates are made out of a conjunction of these predicate types.

```
word => pos
# This are the rules using 5
characters
# Suffix/prefix identity rules
pos word::~~5 => pos
pos word::5~~ => pos
# Suffix addition only at this
level
pos word::++5 => pos
# Suffix/prefix subtraction rules
pos word::--5 => pos
pos word::5-- => pos
# Rules at level 4, etc.
pos word::~~4 => pos
pos word::4~~ => pos
pos word::4++ => pos
pos word::++4 => pos
pos word::--4 => pos
pos word::4-- => pos
pos word::~~3 => pos
pos word::3~~ => pos
pos word::++3 => pos
pos word::3++ => pos
pos word::3-- => pos
pos word::--3 => pos
pos word::~~2 => pos
pos word::2~~ => pos
pos word::--2 => pos
pos word::2-- => pos
pos word::++2 => pos
pos word::2++ => pos
pos word::~~1 => pos
pos word::1~~ => pos
pos word::++1 => pos
pos word::1++ => pos
pos word::--1 => pos
pos word::1-- => pos
pos word::1<> => pos
pos wordˆˆ1 => pos
pos wordˆˆ-1 => pos
```

```
# The same rules as the preceding
ones,
# but without conditioning on the
POS
word::5~~ => pos
word::~~5 => pos
word::--5 => pos
word::5-- => pos
word::4~~ => pos
word::~~4 => pos
word::4++ => pos
word::++4 => pos
word::--4 => pos
word::4-- => pos
word::~~3 => pos
word::3~~ => pos
word::++3 => pos
word::3++ => pos
word::3-- => pos
word::--3 => pos
word::~~2 => pos
word::2~~ => pos
word::--2 => pos
word::2-- => pos
word::++2 => pos
word::2++ => pos
word::~~1 => pos
word::1~~ => pos
word::1<> => pos
word::++1 => pos
word::1++ => pos
word::--1 => pos
word::1-- => pos
# Bigram cooccurrence predicates:
# Test on the previous word
wordˆˆ-1 => pos
# Test on the next word
word1ˆˆ1 => pos
```

Table 5.2: Sample of lexical rule template file

pos_0 word_0 word_1 word_2 => pos pos_0 word_-1 word_0 word_1 => pos pos_0 word_0 word_-1 => pos pos_0 word_0 word_1 => pos pos_0 word_0 word_2 => pos pos_0 word_0 word_-2 => pos pos_0 word:[1,2] => pos pos_0 word:[-2,-1] => pos pos_0 word:[1,3] => pos pos_0 word:[-3,-1] => pos pos_0 word_0 pos_2 => pos pos_0 word_0 pos_-2 => pos pos_0 word_0 pos_1 => pos pos_0 word_0 pos_-1 => pos pos_0 word_0 => pos pos_0 word_-2 => pos pos_0 word_2 => pos pos_0 word_1 => pos pos_0 word_-1 => pos pos_0 pos_-1 pos_1 => pos	pos_0 pos_1 pos_2 => pos pos_0 pos_-1 pos_-2 => pos pos_0 pos_1 => pos pos_0 pos_-1 => pos pos_0 pos_-2 => pos pos_0 pos_2 => pos pos_0 pos:[1,3] => pos pos_0 pos:[1,2] => pos pos_0 pos:[-3,-1] => pos pos_0 pos:[-2,-1] => pos pos_0 pos_1 word_0 word_1 => pos pos_0 pos_1 word_0 word_-1 => pos pos_-1 pos_0 word_-1 word_0 => pos pos_-1 pos_0 word_0 word_1 => pos pos_-2 pos_-1 pos_0 => pos pos_-2 pos_-1 word_0 => pos pos_1 word_0 word_1 => pos pos_1 word_0 word_-1 => pos pos_0 pos_1 pos_2 => pos pos_0 pos_1 pos_2 word_1 => pos

Example of contextual rule templates

- The identity of one of the previous/next words:

$$word : [< number >, < number >]$$

where $< number >$ can be a integer number in $[-128, 127]$; for example, the rule

$$pos\_0 = VBP\ word : [-2, -1] = a \Rightarrow pos = NN$$

will change the POS of a word from VBP to NN if one of the previous 2 words is a;

- The identity of one of the previous/next POS tags:

$$pos : [< number >, < number >]$$

where $< number >$ is the same as before; e.g.

$$pos\_0 = NN\ pos : [-2, -1] = MD \Rightarrow pos = VB$$

An example of template file can be found in Table 5.2.2 .

The steps that need to be done to run the contextual tagging are:

1. Compute, from the training data, the most likely POS tag associated with the words. This is probably well enough computed at the previous step, but if you want to compute it again, using the entire data, now is the time[5].

2. Use the lexical rules obtained at the previous step to guess the POS of all the unknown words in the training data – this is done by applying ***fnTBL*** to the training data, with appropriate flags:

$$\text{fnTBL}\ < training\_data >\ < lexical\_rules >\ -F\ \langle param\_file\rangle\ -o\ \langle train\_output\_file\rangle$$

3. Run the learning command

$$\text{fnTBL} - \text{train}\ \langle train\_output\_file\rangle\ \langle context\_rules\rangle\ -F\ \langle context\_param\_file\rangle\ \langle options\rangle$$

where the options are presented in Section 4.2.

[5]One might want to do this, as the distribution on the first part of the training data might be different than the distribution on the second part. However, one should exclude the words that appear only on the second part, as it will create conditions similar to the testing time situation, i.e. do not use the true tag for the unknown words, but rather the guessed tag.

Some observations:

- As discussed earlier, TBL suffers from the lack of output probability estimation, and the fix is to enforce a class output limited to the choices seen in training data. While this is appropriate for high frequency words, it might not be for low frequency ones. Therefore, it might be wise to create the most likely file using a threshold, such that words that are below the frequency threshold are not limited to be assigned a POS associated with them in the training data.
- Depending on the number of predicate templates and/or how long the context in the predicate templates is, the program might need relatively large amounts of memory[6]. This behavior is due in part to the STL (Standard Template Library) and in part to the fact that all the possible rules that correct at least one error are generated at the start of the program. An improvement of the program would be to restrict the initial generation of the rules to the ones that apply at least a certain number of times, since only those ones can be chosen at the starting phase. This modification will probably be present in the version 2.0 of the toolkit.

5.2.3 TBL POS Tagging Without Headaches

Fortunately, there are 2 scripts provided, ***pos-train.prl*** and ***pos-apply.prl***, which do most of these tasks, with little intervention.

Learning the Rule List

For the purpose of training a TBL POS tagger, one can use the script *pos-train.prl*. It should be called as:

pos-train.prl <*parameters*> <*train_file*>

where the options are:

- Mandatory parameters:
  - ▷ -F <*lexical_param_file*>,<*context_param_file*> – defines the lexical parameter file and the contextual parameter file. As a starting point for these files, one could use the provided *tbl.lexical.params* and *tbl.context.params*.
  - ▷ <*train_file*> is the training file, 2 columns per line (first the word, second the true POS); sentences should be separated by an empty line.
- Optional parameters:

-B <*bigram_file*>	defines the file containing the large quantity of bigrams; if undefined, the bigrams are extracted from the training file;
-*D* <*directory*>	defines a directory where all the parameters are saved - highly recommended, as it will keep everything tidy and make sure that additional runs do not affect the learned parameters
-f <*cutoff*>	defines the cutoff to be used for contextual features with the contextual training;
-o <*outfile1*>,<*outfile2*>	defines the names the two split files will have; by default they are <*train_file*>.part1, <*train_file*>.part2;
-r <*ratio*>	defines the ratio of the split between the first file and second file (0.3 would split it 0.3/0.7, with 0.3 being the one from which the counts are extracted);
-t <*NC*>,<*NP*>	defines the POS symbols for the common noun and proper noun; by default the values are NN and NNP, but they need to be specified for other languages;
-*R* <*lexrulefile*>,<*contextrulefile*>	specifies the names of the 2 rule files to be output: the lexical rule file and the contextual rule file - by default lexical.rls and context.rls;
-T <*thresh1*>,<*tresh2*>	defines the learning thresholds for the lexical/contextual tagging;
-u <*unigram_file*>	defines the file containing the large vocabulary; if not present, the unigrams are extracted from the training file;
-v	turns on the verbose output - useful for debugging and/or see what sequence of commands the program executes;

Some observations about the training procedure:

[6] Everything is relative nowadays, though. It is debatable whether is better to invest time in making the program run in smaller amounts of memory or to wait until computers with a large amount of memory become standard.

- The cutoff we regularly used is 10, but your mileage may vary. Experiment with different values (5 is also a good value), as it may improve the performance of the classifier;
- The split ratio as proposed by Brill is 0.5 (50%). We found it better to use something like 0.35-0.4, as it more words will be "unseen", and therefore the program will "generalize" more;
- The default threshold is 3: the learner will stop if it obtains a rule with a score less than 3. The lower the threshold, the longer the training time; while this algorithm speeds up considerably as the best rule has lower score, there are a (exponentially) larger number of rules with lower score (as suggested by Zipf's law). A threshold of 0 will force the algorithm to learn to completion (until no more rules are possible); it is debatable if rules that apply once during training have good generalization properties[7]. Experimentation is needed to determine if those rules help or hurt – they usually help in cases where the training data is scarce, and don't help where there is a lot of training data;
- If the POS is done in a language where more types of common noun are possible, select the one that is most common to be assigned. The proper noun will be assigned to the data if the word starts with a capital letter – assign the same type of common noun if some other nouns start with capital letters (e.g. like in German);
- It is useful to filter the bigram file such that at least one of the words is a frequent word, because infrequent words will probably not help in figuring the POS of unknown words (because they are infrequent).

Applying a Rule List

The application of a rule list to a corpus consists of 2 phases also: first, we apply the lexical rules, to guess the POS for unknown words, and then we apply the contextual rules. One can call the fnTBL program directly, or call the ***tbl-apply.prl*** script. This script is called as:

tbl-apply.prl *<options> <test-file>*

where the options are:

- Mandatory parameters:

-F *<lexparam>,<contextparam>*	defines the 2 parameter files; as a start, one could use the tbl.lexical.params and tbl.context.pos.params provided with the distribution;
-t *<NC>,<NP>*	defines the POS symbols for the common noun and proper noun; by default the values are NN and NNP, but they need to be specified for other languages;
-*R <lexrulefile>,<contextrulefile>*	specifies the 2 rule files: the lexical rule file and the contextual rule file;
<test_file>	the text to be tagged; can be in 1 or 2 column format.

- Optional parameters:

-m *<most_likely_tag_file>*	defines which file should be used to initialize the test data - it should the "largest" data available.
-D *<training_directory>*	The directory defined during training
-o *<output_file>*	defines the name of the output file; if not present the output will be in a file called *<test_file>.res*;
-v	turns on verbose output - useful for debugging and to see what the script does.

Some observations about the process of applying a POS rule list:

- The test file can be in either 1 column format (1 word per line) or 2 column format (word and truth, if testing is desired); sentences should be broken by an empty line;
- The same observations about the most frequent common noun and the most common proper noun made for the script *pos-train.prl* apply here as well;
- The most likely file that is applied to the test data should be extracted from the entire training data; the less unknown words, the more accurate the process;
- If you want to see what the script is doing, turn on the verbose flag.

[7] But it should be mentioned that the fact they applied only one on the data also says that they did not apply on the rest of the samples, which means that they should not make a big negative impact on the test data, as they should not apply too often. However, if the training and test data are not drawn from the same distribution, they can hurt the performance.

5.3 Word Sense Disambiguation

Word sense disambiguation is the task of assigning predefined senses to particular words in context. It can be cast as either a lexical choice task (where the goal is to determine the sense for one particular word in a given context) or a all-words task (assign senses to all nouns, verbs, adjectives and adverbs in a sentence) – the task we are presenting here is the lexical choice task. In this task, the samples are independent, as the classification of one instance of a word can be considered independent from the other classifications[8].

5.3.1 Data Preparation

The samples in this task consist of a number of sentences around or before the word of interest; a number of such samples is presented for each word and are (supposed to be) balanced on the senses. As preprocessing we applied the following steps:

- The text was tokenized and end-of-sentence delimiters were inserted;
- Samples were POS tagged, lemmatized and tagged with sentence chunks (using the fnTBL, of course ☺).
- Syntactic information associated with the selected words was extracted (e.g. the verb to which the noun in question is object to, etc).

For more information on the processing, see [YCF+01].

5.3.2 Rule Types

The features that the TBL system has access to the following feature type:

- words in a 3 positions window (word-3left . . . word-3right);
- part-of-speech information in a 3-word window (pos-3left . . . pos-3right);
- lemmas in a 3-word window (lemma-3left . . . lemma-3right);
- a set of 7 words around the word in question (word-in-a-3-word-window);
- a set of 7 lemmas around the word in question (lemma-in-a-3-word-window);
- a set of 100 words (not stopwords) around the word to be disambiguated (word-in-a-100-word-window);
- a set of 100 lemmas (not stopwords also) around the word to be disambiguated (lemma-in-a-100-word-window);
- the syntactic features (that also depend on the POS of the word) described earlier.

The difference between the labeled words (e.g. word-3left) and the unlabeled ones (word-in-a-3-word-window) is that a rule depending on the predicate *word-3left=manufacturing* will fire if the *manufacturing* appears exactly 3 words back from the word position, while the predicate *word-in-a-3-word-window=manufacturing* will be true if the word *manufacturing* appears anywhere in a 3 word window around the desired word.

One way to create this data is the following (we assume that we want 2 window-sizes for the set features 7 and 100):

- define in the file template some features named $f\_1 \ldots f\_201$;
- for each sample:
  - ▷ Identify the 7 words around the word of interest; eliminate the stop words and duplicates; put the resulting words in the fields $f\_1 \ldots f\_7$, adding the character '-' if necessary[9].

[8] This is true if the instances are not part of the same document, where the "one sense per discourse" [Yar95]; as the task was presented in Senseval2 [CEKP01], the samples are independent.

[9] This character was selected because it still keeps the sample line readable, but you can replace it with any character you like.

▷ Identify the 201 words around the word of interest, eliminating stop words and duplicates, and also the words that were determined at the first step; write them in the fields corresponding to the variables $f\_8 \ldots f\_201$, adding the character '-' if necessary.

and repeat the process for lemmas as well. Then add the following line to the parameter file:

$$\text{NULL\_FEATURES} = -;$$

this will prevent the toolkit from considering predicates of the form *word-in-a-3-word-window=-*, which are not really useful and will just increase the computation time[10].

Since the input to the task can vary considerably, we did not provide any tools to prepare the data; it should not be very hard for the reader to create the data files, as they require just normal perl processing. In the test-cases/wsd directory we provided a parameter file, together with the file.templ and rule.templ files corresponding to the 3 and 100 windows described in this Section. Also provided are training and test data for the verb *strike*, as extracted from the Senseval2 data.

Another interesting observation is that, in the Senseval data, some samples had multiple classifications. fnTBL toolkit can handle this case if one separates the values with a special character, which should be defined in the parameter file. For instance,

$$\text{TRUTH\_SEPARATOR} = |;$$

will defined the character separator to be '|' — values such as *strike%2:35:03::|strike_a_chord%2:31:00::*[11] will be interpreted correctly as one of the senses and rules will be generated such that they correct at least one of the senses; the concatenation is not treated as a new truth. If the parameter file does not contain such a definition, then the values such as the one described above will be considered to be classifications.

5.3.3 Training the System

The training is done in the usual way, once the files are created. The command to be run is

```
fnTBL-train myfile.init wsd.rls -F <param_file> [-threshold <n1>]
[-allPositiveRules <n2> [-minPositiveScore <n3>]]
```

Some observations:

- For this task, using a threshold of 0 seems beneficial, as the data is sparse.
- This is a classic example where the original TBL algorithm performs poorly, partly because it does not allow for redundancy. If one uses the flag *-allPositiveRules*, then one will allow the algorithm to select those rules that, at the end, have a "good" application count[12] over the specified threshold and do not have any bad application – see Section 4.6 for further details.
- In case the argument of the *-allPositiveRules* flag either contains a '%' sign or is negative, one can use the *-minPositiveScore* flag to enforce the minimum count over which rules are output.

When TBL is allowed to consider the redundant rules its performance is very similar to the one of a decision list-based algorithm, similar to the one described in [Yar95], that has access to the same features as the TBL algorithm.

5.3.4 Testing the System

To test the system, one runs the command:

```
fnTBL mytest.init wsd.rls -F <param.file> [-printRuleTrace] [-o <outfile>]
```

The directory test-cases/wsd contains a README file which has all the commands needed to run the fnTBL system on the verb *strike* data provided.

[10] Actually, we observed a significant decrease in running time when using this feature; its use is highly recommended in cases such as this one.

[11] The values presented here represent the wordnet 1.7 tags [Fel00]associated with the senses.

[12] In this case defined as would convert the value to correct, if applied, regardless of the actual value of the classification – usually, rules are not considered to apply if the classification is already correct.

Acknowledgements

The authors would like to thank to all the people that were involved in the development of this software: the NLP and CLSP groups at Hopkins as a whole, for valuable interaction and suggestions over the years, and also to particular individuals:

- David Yarowsky for his continued support and for allowing us to pursue writing this code;
- Eric Brill for long discussions on the behavior of Transformation Based Learning;
- John Henderson, for lengthy discussions and suggestions, and also for being responsible in the largest part for the probabilistic behavior of the TBL;
- Ellen Riloff, who was one of the first users of this toolkit, for valuable suggestions regarding the behavior of the fnTBL;
- Charles Schafer, for useful suggestions and comments on the fnTBL behavior;
- Mary D. Taffet, for identifying several problems in the packaging, documentation and scripts, and also for suggesting improvements;
- Cristina Nita-Rotaru for help on setting up the web page, and for many other things.

Appendix A

Appendix

A.1 Parameter File Variables

- CONSTRAINTS_FILE – defines a set of constraints on the data – rules will not be applied to a particular sample if some constraint is broken. For instance, if a constraint of the form

  the DT

  is imposed on a POS task, then no rule will change the tag of the word *the* to anything else but a determiner. See Section 3.2.5 for more information on constraints and why they are useful for TBL.

- COOCCURRENCE_CONFIGURATION_FILE – defines the bigram – file used by lexical tagging – see Section 5.2.1. A sample of such a file is presented in Table A.1.

- ELIMINATION_THRESHOLD – if defined, it sets a threshold on the number of rules that are generated initially. Every time the number of generated rules is a multiple of a specified number, all rules with count less than ELIMINATION_THRESHOLD are eliminated – this feature has the advantage of keeping the initial number of rules down.

- ERASE_USELESS_RULES – if defined, the rules that have 0 good counts and a number of bad counts greater than the value of this parameter are eliminated – this helps keep the space represented by the rules smaller, but it will make the program run a little slower. The big advantage is that probably the space needed to represent the rules is not going to grow considerably beyond what is needed to store the initial rule set (as many rules that have low good scores are eliminated).

- FILE-TEMPLATE – defines the file that holds the names assigned to the features. The file should contain one line describing the sample, and it should have the following format:

  $$\langle feature_1 \rangle \dots \langle feature_n \rangle \ \langle class_1 \rangle \dots \langle class_m \rangle => \langle truth_1 \rangle \dots \langle truth_m \rangle$$

  where $\langle feature_i \rangle$ is the name associated with feature number i, $\langle class_k \rangle$ is the name associated with the k^{th} class and $\langle truth_k \rangle$ is the name associated with truth number k. Let us notice that there are exactly as many classifications as true values, and that there can be more than one classification, since the *fnTBL* toolkit can handle multiple simultaneous classifications. The names provided here will be used to generate the rules (the features will be named using these names).

word_0 word_1 ./unseen_bigrams.dat
word_-1 word_0 ./unseen_bigrams.dat

Table A.1: Bigram cooccurrence file

- LARGE_WORD_VOCABULARY – defines the large vocabulary file used by the lexical tagging program – see Section 5.2.1. The file should contain a list of words, one per line, that can appear in the language – such a file is easily extracted from a large unanotated corpus, using a command such as

$$\text{tr}\,'\,'\,'nn'\ <\ file \mid \text{sort}\ -\text{u} \mid \text{perl}\ -\text{pe}\ '\$\_ = \ ''\ ''\ \text{unless}\ /nS/' > outputfile$$

- MINIMUM_ENTROPY_GAIN – this parameter is used to control the further expansion of the probabilistic tree (see [FHN00]) – the expansion is stopped when the gain of the current split is less than the value of this parameter;
- NULL_FEATURES – When considering a set-like predicate, some of the features might not be "real" features, but rather the absence of a feature. These features are ignored when rules are generated – it speeds up the program considerably, while eliminating rules that make no sense at all. The features to be ignored are to be separated by a single comma (,).
- ORDER_BASED_ON_SIZE — determines how the ties[1] are broken. Possible values:
  - ▷ 0 – the rule whose predicate has more atomic predicates is chosen first;
  - ▷ 1 – the rule whose predicate has less atomic predicates is chosen first;
  - ▷ 2 – the rule whose template is declared first in the rule template file is chosen first.
- REASONABLE_SPLIT – this parameter is required by the probabilistic tree generation (see [FHN00]) – and controls the threshold under which rules do not split the data: if the rule applied less than the value of this parameter, then the rule is not considered in the split;
- REASONABLE_DT_SPLIT – this parameter is required by the further expansion of the probabilistic tree (see [FHN00]); the expansion is stopped when the any of the generated nodes has less than the value of this parameter samples;
- RULE_TEMPLATES – defines the file that holds the rule templates. The file should list a series of rules, one on a line, respecting the following pattern:

$$\langle pred_1\rangle \ldots \langle pred_l\rangle => \langle class_1\rangle \ldots \langle class_p\rangle$$

  where $\langle pred_i\rangle$ is a basic predicate template and $\langle class_j\rangle$ is a valid name for a classification (as defined by the file contained in the FILE_TEMPLATE). The basic (also called atomic) predicates are checking one particular feature (i.e. they have one argument), and the rule is formed as a conjunction of these atomic predicates. The rule template defines the search space of the algorithm – it will find particular instantiations of these rules that correct the most errors at each step. A valid research question that can be raised in this context is how to obtain these patterns automatically, and not have the user specify them.
- TRUTH_SEPARATOR – in some cases, it might be useful to have multiple values for the truth. For instance, in the task of word sense disambiguation, words might have multiple senses (e.g. "Roger *Rabbit*" will have both the proper name sense – a cartoon character – and the real rabbit sense – animal). In this case, the multiple true values are to be separated by the character defined in this parameter. For instance, if TRUTH_SEPARATOR=|, then the value P|bar%1:14:00:: is a valid value as the truth for a sample. The fnTBL training program will interpret this as a union of senses, rather than a single value, and will generate the appropriate rules.

A.2 Additional Tools Provided

There are a number of additional tools provided with the toolkit, mainly perl scripts that are built to solve one problem or the other. Here is a (partial) list of scripts, together with their description:

- ***mcreate_lexicon.prl*** – generates the feature lexicon (all the possible classes for a given feature); it is used to create the restriction files. It should be run as:

$$\text{mcreate\_lexicon.prl}\ [-\text{d}\ \langle n_1\rangle \Rightarrow \langle n_2\rangle]\ [-n\ \langle number\rangle]\ \ [-c]\,\langle file\rangle$$

  where :

[1] Tie = a case where 2 rules have the same score

- ▷ $\langle file \rangle$ is a feature text file, with the features separated by whitespace; $\langle file \rangle$ can also be '-', in which case the input is read from stdin (useful when used in pipes);
- ▷ -d $\langle n_1 \rangle \Rightarrow \langle n_2 \rangle$ will generate a list for feature number $\langle n_1 \rangle$ and classification $\langle n_2 \rangle$ (it assumes that column $\langle n_1 \rangle$ contains features while on column $\langle n_2 \rangle$ classifications are listed)
- ▷ -n $\langle number \rangle$ sets a listing threshold – any feature with a frequency less than $\langle number \rangle$ is not displayed;
- ▷ -c – if this flag is present, classification frequencies are also output, together with the classification.

The output is done at stdout, so you might want to redirect it to a file. For each feature with a frequency over the threshold, it outputs the feature and a list of classifications separated by spaces. The list is output in the reverse order of occurrence (i.e. the most likely one first).

- ***most_likely_tag.prl*** – applies a lexicon file to a feature file. For instance, it can be used to generate the most likely POS tag associated with a sequence of words, given a lexicon file created with the previous program (***mcreate_lexicon.prl***). Its command line is

$$\text{most\_likely\_tag.prl } [-l \ \langle lexicon\_file \rangle] \ [-p \ \langle pattern \rangle] \ [-t \ \langle NC \rangle, \langle NP \rangle] \ \langle file \rangle$$

where:

- ▷ $\langle file \rangle$ is a feature file, with features separated by whitespace; $\langle file \rangle$ can also be "-", in which case the input is read from stdin (useful for pipelining);
- ▷ $-l \ \langle lexicon\_file \rangle$ – uses the specified $\langle lexicon\_file \rangle$; if this parameter is unspecified, it will use the input file to first extract the most likely tag, given the provided pattern;
- ▷ $-p \ \langle pattern \rangle$ – uses this pattern to assign the most likely classification; a pattern has the form $\langle n_1 \rangle \Rightarrow \langle n_2 \rangle$ (similar to the -d option of the ***mcreate_lexicon.prl*** script). If this pattern is not specified, $\langle n_1 \rangle$ is considered 0 and $\langle n_2 \rangle$ is considered 1.
- ▷ $-t \ \langle NC \rangle, \langle NP \rangle$ is used to specify the most likely tag in case the feature classification is undefined. There are 2 values specified, one for common nouns, and one for proper nouns. If this doesn't make sense (it does only for POS), then specify just one value, the value of the most likely classification overall.

The output is done, again, at the standard out. The output corresponding to a feature sequence is done as follows: the feature on position $\langle n_2 \rangle$ is replaced with the most likely classification associated with the feature present on position $\langle n_1 \rangle$; if no feature list is associated with the feature value on position $\langle n_1 \rangle$ (i.e. the feature is "unknown"), then the most likely tag is output on position $\langle n_2 \rangle$ (with a difference for POS tagging, where if the word begins with a capital letter, the second value provided is output, otherwise the first one is output).

- ***brill-to-tbl.prl*** converts the rules generated by the Eric Brill's POS tagger into rules that can be used by the ***fnTBL*** toolkit. Provided just for fun – and comparison. It receives the list of rules in Brill format and outputs at stdout the list of rules in ***fnTBL*** format.

- ***rm-to-tbl.prl*** converts the rules generated by the Ramshaw & Marcus baseNP chunking TBL program (see [RM94]) into the corresponding rules in ***fnTBL*** format. It accepts a file as argument, the rules in Ramshaw&Marcus format and outputs the corresponding rules in the toolkit format.

- ***mcompute_error.prl*** evaluates the performance of the ***fnTBL*** output for a given task. Obviously, one needs to have the correct classification for this feature to work properly. The necessary output is the one obtained from the ***fnTBL*** command. The running command is:

$$\text{mcompute\_error.prl } [-\text{i } \langle i_1 \rangle, \langle i_2 \rangle, \langle n \rangle] \ \langle file \rangle$$

where

- ▷ $\langle file \rangle$ is the output of the ***fnTBL*** program (assuming the last n columns correspond to the true classifications); the value can also be "-", in which case the input is read from stdin;

- ▷ $-$i $\langle i_1 \rangle, \langle i_2 \rangle, \langle n \rangle$ specifies which fields are the system's output and which ones are the true classification: $\langle i_1 \rangle$ is the index where the system's output begins, $\langle i_2 \rangle$ is the index where the true classification begins and $\langle n \rangle$ are the number of elements to be classified; the default values are $\langle n \rangle = 1$, $\langle i_1 \rangle = \langle \text{second} - \text{to} - \text{last} - \text{position} \rangle$, $\langle i_2 \rangle = \langle \text{last} - \text{position} \rangle$.

- ***number-rules.prl*** a really silly program that numbers the rule file such that the rule numbers correspond to the numbers output with the option $-printRuleTrace$ of the ***fnTBL*** command.

- ***pos-train.prl*** and ***pos-apply.prl*** are described in detail in Section 5.2.3, and we will not bore the reader with another description.

- A number of other simple, less-than-useful scripts. Please feel free to poke around them to see what they do $\ddot{\smile}$... .

Bibliography

[Bri95] E. Brill. Transformation-based error-driven learning and natural language processing: A case study in part of speech tagging. *Computational Linguistics*, 21(4):543–565, 1995.

[CEKP01] S. Cotton, P. Edmonds, A. Kilgarriff, and M. Palmer. Second international workshop on evaluating word sense disambiguation systems. web page, http://www.sle.sharp.co.uk/senseval2/, July 2001.

[Fel00] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, 2000.

[FHN00] R. Florian, J.C. Henderson, and G. Ngai. Coaxing confidence from an old friend: Probabilistic classifications from transformation rule lists. In *Proceedings of EMNLP-SIGDAT 2000*, Hong Kong, October 2000. Association of Computational Linguistics.

[FN01] R. Florian and G. Ngai. Multidimensional transformational-based learning. *Proceedings of the fifth Conference on Computational Natural Language Learning, CoNLL 2001*, pages 1–8, July 2001.

[Lag99] T. Lager. The μ-tbl system: Logic programming tools for transformation-based learning. In *Proceedings of the 3rd International Workshop on Computational Natural Language Learning*, Bergen, 1999.

[NF01] G. Ngai and R. Florian. Transformation-based learning in the fast lane. In *Proceedings of North Americal ACL 2001*, pages 40–47, June 2001.

[RM94] L. Ramshaw and M. Marcus. Exploring the statistical derivation of transformational rule sequences for part-of-speech tagging. In *The Balancing Act: Proceedings of the ACL Workshop on Combining Symbolic and Statistical Approaches to Language*, New Mexico State University, July 1994.

[Sam98] K. Samuel. Lazy transformation-based learning. In *Proceedings of the 11th Interational Florida AI Research Symposium Conference*, pages 235–239, Florida, USA, 1998.

[SV99] E. F. Tjong Kim Sang and J. Veenstra. Representing text chunks. *Proceedings of the Eastern Chapter of the Association for Computional Linguistics*, 1999.

[Yar95] D. Yarowsky. Unsupervised word sense disamibuation rivalring supervised methods. *Proceedings of the 33rd Meeting of the Association for Computational Linguistics*, pages 189–196, 1995.

[YCF+01] D. Yarowsky, S. Cucerzan, R. Florian, C. Schafer, and R. Wicentowski. The jhu system in senseval 2. forthcoming, September 2001.