The μ-TBL System: Logic Programming Tools for Transformation-Based Learning

Torbjörn Lager
Department of Linguistics
Uppsala University
Torbjorn.Lager@ling.uu.se

Abstract

The μ-TBL system represents an attempt to use the search and database capabilities of the Prolog programming language to implement a generalized form of transformation-based learning. In the true spirit of logic-programming, the implementation is 'derived' from a declarative, logical interpretation of transformation rules. The μ-TBL system recognizes four kinds of rules, that can be used to implement various kinds of disambiguators, including Constraint Grammar disambiguators as well as more traditional 'Brill-taggers'. Results from a number of experiments and benchmarks are presented which show that the system is both flexible and efficient.

Introduction

Since Eric Brill first introduced the method of Transformation-Based Learning (TBL) it has been used to learn rules for many natural language processing tasks, such as part-of-speech tagging [Brill, 1995], PP-attachment disambiguation [Brill and Resnik, 1994], text chunking [Ramshaw and Marcus, 1995], spelling correction [Mangu and Brill, 1997], dialogue act tagging [Samuel *et al.*, 1998] and ellipsis resolution [Hardt, 1998]. Thus, TBL has proved very useful, in many different ways, and is likely to continue to do so in the future.

Moreover, since Brill generously made his own TBL implementation publicly available,[1] many researchers in need of an off-the-shelf retrainable part-of-speech tagger have found what they were looking for. However, although very useful, Brill's original implementation is somewhat opaque, templates are not compositional, and they are hard-wired into the program. Therefore, the program is difficult to modify and extend. What is more, it is fairly slow.

This paper is dedicated to the design and implementation of an alternative transformation-based learner system, called "the μ-TBL system" (pronounced "mutable"). The μ-TBL system is designed to be theoretically transparent, flexible and efficient. Transparency is achieved by performing a 'logical reconstruction' of TBL, and by deriving the system from there. Flexibility is achieved through the use of a compositional rule and template formalism, and 'pluggable' algorithms. As for the implementation, it turns out that transformation-based learning can be implemented very straightforwardly in a logic programming language such as Prolog. Efficient indexing of data, unification and backtracking search, as well as established Prolog programming techniques for building rule compilers and meta-interpreters, contribute to the making of a logically transparent, easily extendible, and fairly efficient system.[2]

The content of the paper is presented in a bottom-up fashion, starting from the semantics of transformation rules. First, I show that, contrary to what is often assumed, transformation rules can be given a declarative, logical interpretation. I then introduce the μ-TBL system, which in a manner of speaking is derived from this interpretation of rules. The template compiler, a part of the system which translates templates into efficient Prolog programs, is described, and by way of examples it is shown how a particular combination of training data and templates may be 'queried' from the Prolog prompt. Next, a number of variants of all-solutions predicates are specified, that deal with notions such as scores, rankings and thresholds. Since they appear to be independently useful – even useful outside TBL –

[1] Throughout this paper, when referring to Brill's TBL implementation, it is always his contextual-rule-learner – implemented in C – that I have in mind. It is available from `http://www.cs.jhu.edu/~brill/`. along with several other learners and utility programs.

[2] The μ-TBL system is available from `http://www.ling.gu.se/~lager/mutbl.html`.

they belong in a separate library. By combining predicates from these code libraries, a number of TBL-like algorithms are assembled, and benchmarks are run that show the μ-TBL system to be quite efficient. Finally, a small experiment using transformation-based learning to induce Constraint Grammars from corpora is performed.

The Semantics of Transformation Rules

The object of TBL is to learn an ordered sequence of *transformation rules*. The μ-TBL system supports four kinds of transformation rules.

Replacement rules dictate when – based on the context – one feature value for a word should be replaced with another feature value. An example would be "replace tag vb with nn if the word immediately to the left has a tag dt". Here is how this rule is represented in the μ-TBL system's compositional rule/template formalism:

```
tag:vb>nn <- tag:dt@[-1].
```

This is of course the exact counterpart of the transformation rule in Brill's original framework.

Addition rules specify when a feature value should be added to a word. An example would be "add tag nn to a word if the word immediately to the left has a tag dt":

```
tag:0>nn <- tag:dt@[-1].
```

Note that a feature value is actually added to a word only if it not already there.

Deletion rules dictate when a feature value should be removed from a word. An example would be "remove tag vb from a word if the word immediately to the left has a tag dt":

```
tag:vb>0 <- tag:dt@[-1].
```

Reduction rules reduce the set of feature values for a word with a certain value. An example would be "reduce a word's tag values with tag vb if the word immediately to the left has a tag dt":

```
tag:vb>1 <- tag:dt@[-1].
```

An important difference between *deletion* rules and *reduction* rules is that the latter will only remove a feature value from a word if it is not the last value for that feature. If vb is the last value the above rule is not applicable and the reduction will not take place. This should remind us of the kind of constraints that are central to the so called *reductionistic* approach to disambiguation, as represented by for example *Constraint Grammar* [Karlsson *et al.*, 1995]). Constraint grammars may indeed be possible to learn in the μ-TBL system, as I will show towards the end of this paper.

In the μ-TBL system's rule formalism, conditions may refer to different symbol features, and complex conditions may be composed from simpler ones. For example, here is a rule saying "replace the tag for adverb with the tag for adjective, if the current word is "only", and if the previous tag, or the tag before that, is a determiner tag.":

```
tag:ab>jj <- wd:only@[0] & tag:dt@[-1,-2].
```

In this paper, I will break with the tradition to think about transformation rules in exclusively procedural terms, and instead try to think about them in declarative and logical terms. Transformation rules (partially) *describe* an ordered sequence of pairs of symbols, which I will refer to as a *relation*. Such a relation form *training data* for a TBL system. Here is a simple (and unrealistically small) example:

dt	vb	nn	dt	vb	kn	dt	vb	ab	dt	vb
dt	nn	vb	dt	nn	kn	dt	jj	kn	dt	nn

The sequence formed by the upper elements of the pairs will be referred to as S_1, and the sequence formed by the lower elements as S_n. Such sequences can be modelled by means of two sets of clauses, which relate positions in the sequences to symbol feature values:

$S_1(1,dt)$	$S_1(2,vb)$	$S_1(3,nn)$	...	$S_1(11,vb)$
$S_n(1,dt)$	$S_n(2,nn)$	$S_n(3,vb)$	...	$S_n(11,nn)$

A central point in this paper is the suggestion that the declarative semantics of transformation rules can be captured by *rule formulas* in the form of universally quantified implications, and that, for example, the meanings of the four very simple rules shown previously are captured by the following formulas:

Replacement

$$\forall p_0,p_1[S_1(p_0,vb) \wedge (p_1 = p_0 - 1) \wedge S_1(p_1,dt) \rightarrow S_n(p_0,nn)]$$

Addition

$$\forall p_0,p_1[\neg S_1(p_0,nn) \wedge (p_1 = p_0 - 1) \wedge S_1(p_1,dt) \rightarrow S_n(p_0,nn)]$$

Deletion

$$\forall p_0,p_1[S_1(p_0,vb) \wedge (p_1 = p_0 - 1) \wedge S_1(p_1,dt) \rightarrow \neg S_n(p_0,vb)]$$

Reduction

$$\forall p_0,p_1[S_1(p_0,vb) \wedge \exists x_0[S_1(p_0,x_0) \wedge (x_0 \neq vb)] \wedge (p_1 = p_0 - 1) \wedge S_1(p_1,dt) \rightarrow \neg S_n(p_0,vb)]$$

Rule formulas as such will not be put to any direct computational use, but the notion of a rule formula provides a starting point, from which computational tools can be derived.

A *rule instance* is a rule formula in which every variable has been replaced with a constant. Now, we may define the notions of *positive* and *negative instances* of rule formulas (and thus indirectly of transformation rules). A *positive rule instance* is a rule instance where the antecedent and the consequent are both true. Thus, the following formula is a positive instance of the formula corresponding to the simple replacement rule above:

$$S_1(2, vb) \wedge (1 = 2 - 1) \wedge S_1(1, dt) \rightarrow S_n(2, nn)$$

A *negative instance* of a rule is a rule instance where the antecedent is true but where the consequent is false, for example:

$$S_1(8, vb) \wedge (7 = 8 - 1) \wedge S_1(7, dt) \rightarrow S_n(8, nn)$$

Note that Brill's notion of a *neutral instance* of a rule, i.e. an instance of a rule that replaces an incorrect tag with another incorrect tag, is a negative instance in my terminology. (In practice, this does not seem to matter much, as I will show later.)

We now define two important rule evaluation measures. The *score* of a rule is the number of its positive instances minus the number of its negative instances:

$$score(R) = \mid pos(R) \mid - \mid neg(R) \mid$$

The *accuracy* of a rule is its number of positive instances divided by the total number of instances of the rule:

$$accuracy(R) = \frac{\mid pos(R) \mid}{\mid pos(R) \mid + \mid neg(R) \mid}$$

The notion of rule accuracy is well-known in rule induction and inductive logic programming, and towards the end of this paper we will see that it may have a role to play in the context of transformation-based learning too.

An Overview of the μ-TBL System

Through the use of unification and a particular search strategy (backtracking), a logic programming environment such as Prolog implements a constructive kind of inference which allows us to define predicates that are able to *recognize*, *generate* and *search* for positive and negative instances of transformation rules. Furthermore, a layer of *meta-logical predicates* provides a way to *collect* and *count* such instances, and thus a way to calculate the score and accuracy for any rule. Therefore, in a logic programming framework, transformation-based learning can be implemented in a very clear and simple way.

However, for such an implementation to become useful, we have to think about efficiency. Among other things, we need to think about how we *index* our training data. Assuming the part-of-speech tagging task, corpus data can be represented by means of three kinds of clauses:

`wd(P,W)` is true iff the word `W` is at position `P` in the corpus

`tag(P,A)` is true iff the word at position `P` in the corpus is tagged `A`

`tag(A,B,P)` is true iff the word at `P` is tagged `A` and the correct tag for the word at `P` is `B`

Although this representation may seem a bit redundant, it provides exactly the kind of indexing into the data that is needed.[3] A decent Prolog system can deal with millions of such clauses.

Rules that can be learned in TBL are instances of templates, such as "replace tag `A` with `B` if the symbol (e.g. the word) immediately to the left has tag `C`, where `A`, `B` and `C` are variables. Here is how we write this template in the μ-TBL system:

```
t3(A,B,C) # tag:A>B <- tag:C@[-1].
```

The term to the left of # is a unique *identifier* for the template. A *template instance* is a template in which every variable in the identifier has been replaced by a constant. If we strip the identifier we end up with a transformation rule again. The instantiated identifier uniquely identifies that rule.

Positive instances of rules that are instances of the above template can be efficiently recognized, generated and searched for, by means of the following clause:

```
positive(t3(A,B,C)) :-
    tag(A,B,P0), P1 is P0-1, tag(P1,C).
```

Negative instances are handled as follows:

```
negative(t3(A,B,C)) :-
    tag(A,X,P0), dif(X,B), P1 is P0-1,tag(P1,C).
```

It should be clear how these clauses use the representation described above, and that they respect the semantics exemplified in the previous section. Clauses corresponding to other templates and other types of rules can be defined accordingly.

Tied to each template is also an *update procedure* that will apply rules that are instances of this template, and thus update sequences, by replacing feature values with other feature values, adding to the feature values, or removing from them. For example:

```
apply(t3(A,B,C)) :-
    (tag(A,X,P), P1 is P-1, tag(P1,C),
    retract(tag(A,X,P)), retract(tag(P,A)),
    assert(tag(B,X,P)), assert(tag(P,B)),
    fail ; true).
```

[3] Assuming a Prolog with first argument indexing.

To write clauses such as these by hand for large sets of templates would be tedious and prone to errors and omissions. Fortunately, since the formalism is compositional, it is easy to write a template compiler that generates them automatically. The μ-TBL system uses well-known Prolog compiler writing techniques to expand templates written in the compositional high-level notation into clauses that can be run as programs. Thus, the convenience and flexibility of a high-level notation for templates and rules does not compromise performance. A *template grammar* defines the exact relation between a template and a set of clauses. As an illustration, the following grammar rules are used to expand a template into a Prolog clause defining `positive/1`, `negative/1` and `apply/1`, for that template:

```
term_expansion((ID # A<-Cs),
        [(positive(ID) :- G1),
         (negative(ID) :- G2),
         (apply(ID) :- (G3,fail;true))]) :-
    pos((A<-Cs),L1,[]), list2goal(L1,G1),
    neg((A<-Cs),L2,[]), list2goal(L2,G2),
    app((A<-Cs),L3,[]), list2goal(L3,G3).

pos((F:A>B<-Cs)) -->
    {G =.. [F,A,B,P]},[G], cond(Cs,P).

neg((F:A>B<-Cs)) -->
    {G =.. [F,A,X,P]}, [dif(X,B),G], cond(Cs,P).

app((F:A>B<-Cs)) -->
    {G1 =.. [F,A,X,P], G2 =.. [F,P,A],
     G3 =.. [F,B,X,P], G4 =.. [F,P,B]},
    [G1], cond(Cs,P), [retract(G1),
    retract(G2), assert(G3), assert(G4)].

cond((C&Cs),P) --> cond(C,P), cond(Cs,P).
cond(FA@Pos,P0) --> pos(Pos,P0,P), feat(FA,P).

pos(Pos,P0,P) -->
    [member(Offset,Pos), P is P0+Offset].

feat(F:A,P) --> {G =.. [F,P,A]}, [G].
```

A modern Prolog system will compile the resulting clauses all the way down to machine code. Thus, a TBL-system implemented in Prolog can be quite efficient.

The μ-TBL Template Compiler

When a file containing transformation rules is consulted or compiled, each transformation rule is expanded into several Prolog clauses.[4] As a result of this, a large number of predicates becomes available, some of which are documented in Figure 1.

Using the predicates generated by the template compiler, the training data in combination with the templates may be queried. By backtracking through the solutions to a call to `positive/1` we may for example verify that there are ten ways to instantiate our example template in our example data (for space reasons, I show only the first three solutions):

```
| ?- positive(RuleID), RuleID # Rule.
     Rule = tag:vb>nn <- tag:dt@[-1] ? ;
     Rule = tag:nn>vb <- tag:vb@[-1] ? ;
     Rule = tag:dt>dt <- tag:nn@[-1] ? ;
     . . .
```

Alternatively, we might be interested only in instances where the aligned feature values (`A` and `B`) are different, and there are six of those:[5]

`pair(?A,?B)`
`pair(?A,?B,?P)`

`A` is aligned with `B` at a position `P` in the current data.

`positive(?RuleID)`
`positive(?RuleID,?A,?B)`
`positive(?RuleID,?A,?B,?P)`

`RuleID` names a rule which has a positive instance in the current data at a position `P`, where `A` is aligned with `B`. The rule is an instance of a template, which is identified by the functor of `RuleID`. A call to this predicate usually has many solutions, and the order in which solutions are returned on backtracking is determined by the order in which templates are presented to the system, and the order of symbols in the training data.

`sample(?RuleID)`
`sample(?RuleID,?A,?B)`
`sample(?RuleID,?A,?B,?P)`

As `positive/{1,2,3}`, except that rules are randomly sampled.

`sample_R(+R,?A,?B,-Rules)`

Binds `Rules` to a list with `R` randomly sampled rules.

`negative(?RuleID)`
`negative(?RuleID,?A,?B)`
`negative(?RuleID,?A,?B,?P)`

`RuleID` names a rule which has a negative instance in the data at a position `P`, where `A` is aligned with `B`.

`apply(+RuleID)`

The rule `RuleID` is applied to the current data.

Figure 1: Extract from the manual

[4] Also, if the user does not provide them, template identifiers are constructed automatically.

[5] `dif/2` is a built-in predicate in SICStus Prolog. A call to `dif(X,Y)` constrains `X` and `Y` to represent different terms. Calls to `dif/2` either succeed, fail, or are blocked depending on whether `X` and `Y` are sufficiently instantiated.

```
| ?- dif(A,B), positive(ID,A,B), ID # Rule.
     . . .
```

Or, we might be interested only in template instances where the aligned symbols have feature values nn and vb, respectively. There is only one such rule:

```
| ?- positive(RuleID,nn,vb), RuleID # Rule.
     Rule = tag:nn>vb <- tag:vb@[-1] ? ;
```

Sometimes, a *random* sample of a positive rule might be more useful:

```
| ?- sample(RuleID,nn,vb), RuleID # Rule.
     Rule = tag:nn>vb <- tag:vb@[-1]
```

As for negative instances, we may want to know if the rule `tag:vb>nn <- tag:dt@[-1]` has any negative instances in the training data, and indeed there is one at position 8, where vb is aligned with jj rather than nn:

```
| ?- RuleID # (tag:vb>nn <- tag:dt@[-1]),
     negative(RuleID,A,B,P).
     A = vb, B = jj, P = 8 ?
```

Library *ranking*

Library *ranking* is a package for scoring and ranking rules. It was written for the specific purpose of scoring transformation rules in the context of TBL, but is likely to be more generally useful, hence deserving its status as a library. The basic notions are defined as follows:

A *score* is an integer > 0

A *ranking entry* is a pair S-R such that S is a score and R is a rule

A *ranking* is an ordered sequence of ranking entries where each rule occurs only once.

The score of a rule is determined by counting the solutions returned by goals containing the rule (or rather its ID). Thus, many predicates in library *ranking* are meta-predicates that work much the same way as the so called all-solutions predicates that are built into Prolog. Figure 2 lists some of the predicates available in library *ranking*.

The library encapsulates some of Brill's own ways of optimizing transformation-based learning – optimizations which are possible to perform for the ranking of rules in general.

The predicates in library *ranking* interact in a straightforward way with the predicates generated by the template compiler, as the following examples will show. Here, for instance, is how we compute (and print) a ranking on the basis of the goal involving a call to positive/3:

`count(?R,+Goal,-N)`

Binds N to the number of solutions for Goal, unless it fails for lack of solutions. If there are uninstantiated variables in Goal, then a call to count/3 may backtrack, generating alternative values for N corresponding to different instantiations of the free variables of Goal. Defined as:

```
count(R,Goal,N) :-
    bagof(R,Goal,Solutions),
    length(Solutions,N).
```

`rank(?R,+Goal,+ST,-Ranking)`

Computes the score for each instance of R and ranks the instances. However, instances with scores less than the score threshold (ST) are not ranked. Defined as:

```
rank(R,Goal,ST,Rnkng) :-
    setof(N-R,(count(_,Goal,N),N>ST),RnkngO),
    reverse(RnkngO,Rnkng).
```

`penalize(?R,+Goal,+Rnkng,+ST,+AT,-NewRnkng)`

Re-ranks the rules in Rnkng by subtracting from their scores, giving a new ranking NewRnkng. However, any rule with a score $<$ ST or an accuracy $<$ AT is just dropped.

`at_position(+N,+Rnkng,-Rule,-Score)`

Retrieves the Nth rule in the ranking, and its score.

`highscore(?R,+PGoal,+NGoal,+ST,+AT,?WR,?WRS)`

Among the different instances of R, WR is the rule with the highest score (i.e. the 'winning rule'), and WRS is its score, defined as the number of solutions to the goal PGoal minus the number of solutions to NGoal. However, if WRS $<$ ST, or if no rule clears the accuracy threshold (AT), highscore/7 fails. Works as if defined by:

```
highscore(R,PGoal,NGoal,ST,AT,WR,WRS) :-
    rank(R,PGoal,ST,Rnkng),
    penalize(R,NGoal,Rnkng,ST,AT,NewRnkng),
    at_position(1,NewRnkng,WR,WRS).
```

In fact, highscore/7 is implemented in a more efficient way. It keeps track of a *leading* rule and its score, and thus only has to generate and count solutions to NGoal for rules for which the number of positive instances is greater than the score for the leading rule. Moreover, the score threshold (ST) for the counting of solutions of NGoal can be set to the number of solutions to PGoal minus the score for the leading rule.

Figure 2: Extract from the manual

```
| ?- rank(R,A^B^(dif(A,B),positive(R,A,B)),1,L),
     print_ranking(L).
     3     tag:vb>nn <- tag:dt@[-1]
     1     tag:vb>jj <- tag:dt@[-1]
     . . .
```

And here is how we find the highest scoring rule:

```
| ?- highscore(R,A^B^(dif(A,B),positive(R,A,B)),
            negative(R),1,WR,WRS).
     WR = tag:vb>nn <- tag:dt@[-1], WRS = 3  ? ;
```

This concludes the demonstration of how the template compiler and the ranking library allows a particular combination of templates and training data to be interactively explored from the Prolog prompt.

Simple TBL

Full transformation-based learning is just a small snippet of code away. Given corpus data, templates and values for the thresholds (ST and AT), the predicate `tbl/3` implements learning of a sequence of rules:

Program 1

```
tbl(ST,AT,WRs) :-
    (   highscore(Rule,
            A^B^(dif(A,B),positive(Rule,A,B)),
            negative(Rule),
            ST,
            AT,
            WR,
            WRS)
    ->  apply(WR),
        tbl(ST,AT,WRs1),
        WRs = [WR|WRs1]
    ;   WRs = []
    ).
```

This predicate, defined entirely in terms of predicates generated by the template compiler and predicates from library *ranking*, combines all the important principles of TBL into a complete learning program, that repeatedly instantiates rule templates in training data, scores rules on the basis of counts of positive and negative instances of them, selects the highest scoring rule on the basis of this ranking, and applies it to the training data.

Consider our small example once again. Here are the three rules learned (with the score threshold set to 1)

```
tag:vb>nn <- tag:dt@[-1].
tag:ab>kn <- tag:nn@[-1].
tag:nn>vb <- tag:nn@[-1].
```

and here are the transformations that the upper sequence of the training data goes through, when the rules are applied in the given order:

```
dt vb nn dt vb kn dt vb ab dt vb
dt nn nn dt nn kn dt nn ab dt nn
dt nn nn dt nn kn dt nn kn dt nn
dt nn vb dt nn kn dt nn kn dt nn
```

It is interesting to regard what is happening here as a *decomposition* of a relation S_1-S_n, into a number of relations S_1-$S_2 \circ \ldots \circ S_{n-1}$-$S_n$, corresponding to a number of rules $R_1 \circ \ldots \circ R_{n-1}$.

In general, for such a decomposition S_i-S_n = S_i-$S_{i+1} \circ S_{i+1}$-S_n, it holds that if a rule R_i has P positive and N negative instances in S_i-S_n, then (i) R_i will have $P + N$ positive and no negative instances in S_i-S_{i+1}, and (ii) R_i will have no positive nor negative instances in S_{i+1}-S_n. Clearly, (i) follows from the fact that the update procedure associated with R_i changed the negative instances of R_i in S_i-S_n into positive ones, and (ii) from the fact that the antecedent of R_i must be false in S_{i+1}-S_n. As a corollary to (ii) it follows that R_i will not be selected next.

For each step, as long as $P > ST + N$, then S_i will become more similar to S_n. Note, in our example, that there is one rule, `tag:nn>jj <- tag:dt@[-1]`, that would remove the only remaining difference between S_4 and S_n. However, this rule also has three negative instances, and thus the rule gets a score below the threshold.

Scaling Up

Program 1 is indeed small, simple and transparent. But what about efficiency? How well does it scale up to handle real world tasks, such as part-of-speech tagging? In one small test the learner was operating on annotated Swedish corpora[6] of three different sizes, with 23 different tags, and the 26 templates that Brill uses in his distribution:

```
tag:A>B <- tag:C@[-1].
tag:A>B <- tag:C@[1].
tag:A>B <- tag:C@[-2].
tag:A>B <- tag:C@[2].
tag:A>B <- tag:C@[-1,-2].
tag:A>B <- tag:C@[1,2].
tag:A>B <- tag:C@[-1,-2,-3].
tag:A>B <- tag:C@[1,2,3].
tag:A>B <- tag:C@[-1] & tag:D@[1].
tag:A>B <- tag:C@[-1] & tag:D@[-2].
tag:A>B <- tag:C@[1] & tag:D@[2].
tag:A>B <- wd:C@[0] & tag:D@[-2].
tag:A>B <- wd:C@[0] & wd:D@[-2].
tag:A>B <- wd:C@[-1].
tag:A>B <- wd:C@[1].
tag:A>B <- wd:C@[-2].
tag:A>B <- wd:C@[2].
tag:A>B <- wd:C@[-1,-2].
tag:A>B <- wd:C@[1,2].
tag:A>B <- wd:C@[0] & wd:D@[-1].
```

[6] I have used selected parts of the Stockholm-Umea Corpus (SUC). Here is a key to the part-of-speech tags appearing in the present paper: `nn` = noun, `vb` = verb, `pp` = preposition, `pm` = proper name. `dt` = determiner, `pn` = pronoun, `ie` = infinitive marker, `sn` = subjunction. `jj` = adjective, `ab` = adverb, `hp` = relative pronoun, `kn` = conjunction. See [Ejerhed *et al.*, 1992] for further details of this corpus.

```
tag:A>B <- wd:C@[0] & wd:D@[1].
tag:A>B <- wd:C@[0] & tag:D@[-1].
tag:A>B <- wd:C@[0] & tag:D@[1].
tag:A>B <- wd:C@[0].
tag:A>B <- wd:C@[0] & tag:D@[2].
tag:A>B <- wd:C@[0] & wd:D@[2].
```

Below, I show the first thirteen rules, as they are reported by the μ-TBL system (during training on the 30kw corpus). Each rule is preceded by its score (first column), and by its accuracy (second column).

```
130   1.00   tag:dt>pn <- tag:vb@[1].
114   0.82   tag:ie>sn <- tag:vb@[2].
58    0.85   tag:pn>dt <- wd:det@[0] & tag:jj@[1].
37    1.00   tag:ie>sn <- tag:dt@[1].
37    0.97   tag:ie>sn <- tag:nn@[1].
34    0.95   tag:dt>pn <- tag:pp@[1].
29    1.00   tag:ie>sn <- tag:pn@[1].
21    1.00   tag:ie>sn <- tag:pm@[1].
19    1.00   tag:jj>pn <- wd:bland@[-1].
18    1.00   tag:dt>pn <- tag:hp@[1].
17    0.84   tag:pp>sn <- wd:om@[0] & tag:pn@[1].
15    0.94   tag:sn>ie <- tag:vb@[1].
14    0.63   tag:hp>kn <- wd:som@[0] & tag:nn@[1].
```

Note that the actual accuracy of a learned rule can sometimes be well below 1.00. (The accuracy threshold was set to 0.5 in this experiment.) The sequence of rules works well anyway, since the damage done by an incorrect rule can be repaired by rules later in the sequence. (In fact, a small experiment confirmed that the setting of the accuracy threshold to 1.00 generates a tagger which performs less well.)

For each corpus, the accuracy of the learned sequence of rules was measured on a test corpus consisting of 40,000 words, with an initial-state accuracy of 93.3%. The system was running on a Sun Ultra Enterprise 3000 with a 250Mhz processor. Table 1 summarizes the results of the tests:

Size	ST	Runtime	Mem.req.	#(Rs)	Acc.
30k	2	15 min	23M	99	95.5%
60k	4	24 min	38M	81	95.7%
120k	6	54 min	71M	88	95.8%

Table 1: μ-TBL performance – the simple algorithm

The performance of the μ-TBL system was compared with Brill's learner running on the same machine, with the same templates, score thresholds and training data. Table 2 gives the figures.

These tests verify that the program works as expected, and also that it is quite efficient, despite its small size and simple design. In fact, the tests show that μ-TBL learner is an order of magnitude faster than Brill's original learner for this particular task.

Size	ST	Runtime	Mem.req.	#(Rs)	Acc.
30k	2	90 min	24M	104	95.5%
60k	4	185 min	43M	89	95.7%
120k	6	560 min	82M	98	95.8%

Table 2: Performance of Brill's learner

TBL à la Eric Brill

This algorithm is perhaps the one that resembles Brill's own algorithm the most. It differs from the simple algorithm in that to learn one rule, it ranks the error types that occur in the training data (using `rank/4` from library *ranking* to do so), and then it searches top-to-bottom in this ranking, entry by entry, for a rule which fixes the type of error recorded by the entry, always keeping track of a leading rule and its score. When the score for a ranking entry drops below the leading rule's score, the search is abandoned, and the leader is declared winner. This effectively prunes the search space without losing completeness, and it also saves a lot of memory, since only rules for one kind of error at a time have to be held in memory.

Program 2

```
tbl(ST,AT,WRs) :-
    (   rank((A,B),(dif(A,B),pair(A,B)),ST,Rnkng),
        learn_one(Rnkng,dummy,0,AT,WR,WRS),
        WRS >= ST
    ->  apply(WR),
        tbl(ST,AT,WRs1),
        WRs = [WR|WRs1]
    ;   WRs = []
    ).

learn_one(Rnkng0,LR,LRS,AT,WR,WRS) :-
    (   Rnkng0 = [N-(A,B)|Rnkng],
        N > LRS
    ->  (   highscore(R,
                positive(R,A,B),
                negative(R,A,A),
                LRS,
                AT,
                LR1,
                LRS1)
        ->  learn_one(Rnkng,LR1,LRS1,AT,WR,WRS)
        ;   learn_one(Rnkng,LR,LRS,AT,WR,WRS)
        )
    ;   WR = LR, WRS = LRS
    ).
```

The benchmark results, using the same setup as with the simple algorithm, are shown in Table 3.

As can be seen from Table 3, the optimized algorithm is significantly faster than the simple one, and it uses less memory. However, as pointed out in [Ramshaw and Marcus, 1995], the effect of this particular optimization method depends on the size of the

Size	ST	Runtime	Mem.req.	#(Rs)	Acc.
30k	2	10 min	17M	99	95.5%
60k	4	20 min	22M	85	95.7%
120k	6	50 min	39M	92	95.8%

Table 3: μ-TBL performance – the optimized algorithm

tag set. The larger the tag set, the more benefit we can expect. Thus, we can expect to see even greater improvements for many learning tasks.

Note also that in contrast with the simple algorithm, this algorithm uses Brill's notion of negative rule instance. The call `negative(R,A,A)` ensures that neutral instances are not counted as negative. However, it appears that the way negative instances are counted does not matter much, at least not for this application. The rules look pretty much the same as the rules generated by Brill's learner, and in fact, the first ten rules are identical.

Monte Carlo TBL

The original TBL algorithm suffers from the fact that the number of candidate rules to consider grows very fast with the number of rule templates, and in practice only a small number of templates can be handled. [Samuel *et al.*, 1998] presents a novel twist to the algorithm, in order to solve this problem. The idea is to *randomly sample* from the space of possible rules, rather than generating them all. The better the rule is, the greater the chance that it is included in the sample. Thus, the system is likely to find the best rules first. An implementation of this algorithm can be assembled by replacing the definition of `learn_one/6` in Program 2 with the following definition:

Program 3

```
learn_one(Rnkng0,LR,LRS,AT,WR,WRS) :-
    (   Rnkng0 = [N-(A,B)|Rnkng],
        N > LRS
    ->  sample_R(16,A,B,Rs),
        (   highscore(R,
                (member(R,Rs),positive(R,A,B)),
                negative(R,A,A),
                LRS,
                AT,
                LR1,
                LRS1)
        ->  learn_one(Rnkng,LR1,LRS1,AT,WR,WRS)
        ;   learn_one(Rnkng,LR,LRS,AT,WR,WRS)
        )
    ;   WR = LR, WRS = LRS
    ).
```

That is, `highscore/7` picks the rules that it evaluates from the (here) 16 rules that are sampled. The amount of work that `highscore/7` has to perform, and the memory requirements, no longer depends on how many templates there are.

To test the algorithm, the system was run with 260 templates with the 60,000 word corpus, and a comparison was made with the optimized algorithm. The outcome of this experiment is reported in Table 4.

Algorithm	Runtime	Mem.req.	#(Rs)	Acc.
Lazy	48 min	21M	202	95.7%
Brill	427 min	84M	126	95.7%

Table 4: μ-TBL performance – the lazy algorithm vs. the à la Brill algorithm.

As can be seen from the table, although the other algorithm did not perform too bad with 260 templates, the 'lazy' algorithm was an order of magnitude faster. Accuracy was not compromised, although the number of rules grew.

As a sidenote, let me describe a convenient μ-TBL system feature which makes it possible to train with very many templates without actually writing them all down. Instead of loading a set of templates into the system, the user may load a couple of *template declarations*, which, in terms of 'window' sizes and ranges of relative positions over which windows 'slide', *constrain* the relation between templates and clauses, defined by the template grammar. Constrained in this way, the grammar can be used to generate templates. Without going into any further details, let me just show the declarations which causes the system to generate the 260 templates used above:

```
:- head(tag:A>B).
:- window_size(tag,3).
:- window_size(wd,2).
:- range(tag,[-3,-2,-1,1,2,3]).
:- range(wd,[-2,-1,1,2]).
:- anchors([-1,0,1]).
```

Learning Constraint Grammars

In another experiment, the μ-TBL system was run with a number of templates for *reduction* rules, in order to see if something resembling a Constraint Grammar could be induced from training data. Each word token in a training corpus of 30,000 words was assigned the set of part-of-speech tags that it can have according to a lexicon. The training data also indicated which member of this set was the correct one.

The system was run with the following four templates:

```
tag:A>1 <- unique(tag:C@[-1]).
tag:A>1 <- unique(tag:C@[1]).
tag:A>1 <- wd:C@[0] & unique(tag:D@[-1]).
tag:A>1 <- wd:C@[0] & unique(tag:D@[1]).
```

The use of the `unique/1` wrapper in the conditions of the rules has the effect that a rule will trigger only if the assignments of tags to words in the relevant surroundings are non-ambiguous. (As Karlsson et al. (1995) put it, the rules are run in "careful application mode".)

As mentioned earlier, *replacement* rules do not have to be very accurate: if a rule early in a sequence of replacement rules makes some errors, the errors can often be 'fixed' by rules later in the sequence. By contrast, in a sequence of *reduction* rules there are no rules that can add tags once they have been removed. Therefore, in order to maximize the accuracy of the whole sequence of rules, it must be induced under a validation bias which sees to it that each rule is as accurate as possible. In the μ-TBL system, this is taken care of by setting the accuracy threshold to a very high value. However, a sequence of rules induced in this way will typically leave many words with more than one tag. If we want instead to minimize the tags per words ratio, the accuracy threshold can be set to a lower value, but then a lower tagging accuracy will naturally result. In the experiment, the accuracy threshold was set to 0.99 (which still allows for a bit of noise in the data) and to 0.85. Program 2 was used.

Below, I show the first ten rules that were learned by the system (with the accuracy threshold set to 0.99):

```
451  1.00  tag:rg>1 <- wd:i@[0] & unique(nn@[1]).
451  1.00  tag:pl>1 <- wd:i@[0] & unique(nn@[1]).
274  1.00  tag:pn>1 <- wd:en@[0] & unique(nn@[1]).
274  0.99  tag:ab>1 <- wd:en@[0] & unique(nn@[1]).
230  1.00  tag:pl>1 <- unique(dl@[-1]).
222  1.00  tag:ab>1 <- wd:av@[0] & unique(nn@[-1]).
221  1.00  tag:rg>1 <- wd:i@[0] & unique(nn@[-1]).
219  1.00  tag:pl>1 <- wd:i@[0] & unique(nn@[-1]).
200  1.00  tag:pl>1 <- wd:p@[0] & unique(nn@[-1]).
166  0.99  tag:rg>1 <- wd:en@[0] & unique(jj@[1]).
```

The induced sequences of rules were tested on a corpus of 11,000 words. Both the accuracy and the tags per word ratio in the test corpus were measured.[7] The initial tags per word ratio in the test corpus was 1.35. The results of the tests are given in Table 5.

Size	ST	AT	#(Rs)	Runtime	Acc.	T/W
30k	6	0.99	410	75 min.	99.6%	1.17
30k	6	0.85	215	20 min.	98.1%	1.04

Table 5: Result of Constraint Grammar induction

[7] A word is deemed to be accurately tagged if the correct tag is an element in the set of tags that the word has been assigned.

These results are promising. But before it would be fair to compare with other methods for inducing Constraint Grammars from annotated corpora, e.g. the methods described in [Samuelsson *et al.*, 1996] or in [Lindberg and Eineborg, 1998], it remains to determine the optimal set of templates and the optimal settings of the accuracy threshold. Very likely, the learning process (applied to the learning of *reduction* rules) can also be optimized for speed. In short, a lot more has to be done, but at least this section has shown how easily an experiment like this can be set up in the μ-TBL environment.

Summary and Conclusions

The μ-TBL system is not just a re-implementation of original TBL in another programming language. Rather it should be seen as an attempt to use the reasoning and database capabilities of Prolog to do TBL in a more high-level way. The μ-TBL system is:

General – The system supports four types of rules by means of which not only traditional 'Brill-taggers', but also Constraint Grammar disambiguators, are possible to train.

Easily extendible – Through its support of a compositional rule/template formalism and 'pluggable' algorithms, the system can easily be tailored to different learning tasks.

Transparent – Rules have a declarative, logical semantics which, among other things, has proved to be of great value during the implementation work.

Efficient – A number of benchmarks have been run which show that the system is fairly efficient – an order of magnitude faster than Brill's contextual-rule learner.

Interactive – Prolog is an interactive language and this is something that the μ-TBL system inherits.

Small – Thanks to the choice of implementation language, the system's code base can be kept quite small. Indeed, a 'light' version of the μ-TBL system, consisting of just one page of Prolog code, has been implemented [Lager, 1999].

In short, the μ-TBL system is a powerful environment in which to experiment with transformation-based learning.

Acknowledgements

Thanks to Lars Borin, Mats Dahllöf and Natalia Zinovjeva at Uppsala University for comments and suggestions. Joakim Nivre in Göteborg also provided me with valuable insights and advice.

References

[Brill and Resnik, 1994] Brill, E. and Resnik, P., 1994. A transformation-based approach to prepositional phrase attachment disambiguation. In *Proceedings of COLING'94*, Kyoto, Japan.

[Brill, 1995] Brill, E., 1995, Transformation-Based Error-Driven Learning and Natural Language Processing: A Case Study in Part of Speech Tagging. *Computational Linguistics*, December 1995.

[Ejerhed *et al.*, 1992] Ejerhed, E., Källgren, G., Wennstedt, O. and Åström, M., 1992, The Linguistic Annotation System of the Stockholm-Umea Project. Department of Linguistics, University of Ume.

[Hardt, 1998] Hardt, D. 1998, Improving Ellipsis Resolution with Transformation-Based Learning. To appear, AAAI Fall Symposium.

[Karlsson *et al.*, 1995] Karlsson, F., Voutilainen, A., Heikkilä, J., Anttila, A. (eds.). 1995, *Constraint Grammar. A Language-Independent System for Parsing Unrestricted Text.* Mouton de Gruyter.

[Lager, 1999] Lager, T. 1999, μ-TBL Lite: A Small, Extendible Transformation-Based Learner, In *Proceedings of the Ninth Conference of the European Chapter of the Association for Computational Linguistics* (EACL'99), Bergen, June 8 - 12, 1999.

[Lindberg and Eineborg, 1998] Lindberg, N. and Eineborg M., Learning Constraint Grammar-style disambiguation rules using Inductive Logic Programming. In *Proceedings of COLING/ACL'98.*

[Mangu and Brill, 1997] Mangu, L. and Brill, E., 1997, Automatic Rule Acquisition for Spelling Correction, In *Proceedings of The Fourteenth International Conference on Machine Learning*, ICML 97, Morgan Kaufmann.

[Ramshaw and Marcus, 1995] Ramshaw, L. A. and Marcus, M., P., 1995, Text Chunking using Transformation-Based Learning, In *Proceedings of the ACL Third Workshop on Very Large Corpora*, June 1995, pp. 82-94.

[Samuel *et al.*, 1998] Samuel, K., Carberry, S. and Vijay-Shanker, K., 1998. Dialogue Act Tagging with Transformation-Based Learning. In *Proceedings of COLING/ACL'98*, pp. 1150-1156.

[Samuelsson *et al.*, 1996] Samuelsson, C., Tapanainen. P. and Voutilainen. A., 1996, Inducing Constraint Grammars. In: Laurent, M. and de la Higuera. C. (eds.) *Grammatical Inference: Learning Syntax from Sentences*, Springer Verlag.

Appendix: The μ-TBL User Interface

Although it certainly helps to be familiar with the Prolog programming language, the μ-TBL system is actually designed to be usable also by those who lack Prolog experience.

The system has a simple command line interface, depicted in Figure 3, from which commands can be given and queries be made. Furthermore, there are several flags which control the way in which the system carries out its tasks.

```
**************************************************
        The MUTBL System, version 0.7
          (c) Torbjoern Lager, 1999
 Dept. of Linguistics, Uppsala University, Sweden
The MUTBL System comes with absolutely no warranty.
**************************************************

FLAGS:

training_data='data/testcorpus'
test_data='data/10kw_test'
algorithm='algorithms/brill'
templates='templates/brill_dist_templates'
score_threshold=3
accuracy_threshold=0.5
verbosity=2

COMMANDS:

load       - loads data, compiles templates, etc.
train      - starts training process
test       - tests result of training
set F=V    - sets flag F to the value V
help       - shows this menu
flags      - shows settings of flags
commands   - lists available commands
predicates - lists available predicates

**************************************************

| ?-
```

Figure 3: The μ-TBL User Interface