GSLT Machine Learning 2003, Project Report: An Implementation of FastTBL in Oz

Leif Grönqvist (lgr@msi.vxu.se),
Fredrik Kronlid (kronlid@ling.gu.se)

December 31. 2003

Contents

1 Introduction

This is the report of the course project for the GSLT course “Machine Learning”, fall 2003. The main part is an algorithm implementation in the programming language Oz (Van Roy & Haridi, April 11, 2003; Duchier et al, 2003) and the code is included in appendix.

Transformation Based learning is a machine learning method that can be applied to a great number of problems in linguistics and elsewhere. The original Brill implementation (Brill, 1995) has been shown to be relatively slow when compared to more optimized implementations (Lager, 1999).

Florian & Ngai have implemented what they call the FastTBL algorithm - a highly optimized version of the TBL (Transformation Based Learning) algorithm, describing their work in Ngai & Florian (2001).

We have implemented the algorithm in Oz (FoztTBL) in order to:

- obtain a more thorough understanding of TBL and FastTBL as well as
- improve our skill in Oz programming.

2 TBL

FastTBL is an optimized version of TBL and will obtain exactly the same result. This is supervised machine learning and to give a very short introduction to TBL we can describe it as:

- Given a tagged (any kind of annotation) corpus as training data and a set of rule templates of the type:
  - Change A to B at position X if X is tagged A and the context match properties like: the tag before is C and the word two positions after is D

  the TBL system will find the rule that match at least one of the templates, with the highest possible score. The rule candiadates are extracted from the templates and the corpus (for the example above, the variables A, B and C are replaced by tags, and D with a word). The number of rules to consider could be very high if the template contain many variables, but there is no point in generating rules that do not match any samples in the training corpus.

  The score for each rule is calculated as the number of correct tags improvement when the rule is applied to the corpus.
- This best rule is stored, and applied to the corpus. Then the process starts over, and we get a second rule, and so on.
- Continue until the new rule has a score below a threshold, or the wanted number of rules is reached.

One difference between for example Lager's and Brill's implementation is that Lager's templates are more flexible, which makes it possible to develop more kinds of taggers in his framework. Our implementation is not as flexible as Lager's μTBL, but could be easily extended.

For more details about TBL we recommend you to read (Brill, 1995) and (Lager, 1999).

3 The FastTBL Algorithm

3.1 Properties

The main problem with TBL is the long training time. A typical parts-of-speech tagger trained on one million tokens will typically take some tens of hours using a lot of memory with μTBL, and a few days with Brill's implementation. Ramshaw & Marcus (1994) demonstrate how the rules could be stored during the process, to avoid creating them over and over again. Unfortunately, it eats a lot of memory. FastTBL uses the same idea, but thanks to a clever way to store the sets of positions that improves or degrades the result when applying each rule, and the strategy to generate only the rules that corrects at least one sample, both the memory usage and the training time is cut compared to Ramshaw & Marcus (1994). The number of possible rules are kept at a minimum level thanks to this simple idea to generate them from the errors.

3.2 The Algorithm

Ngai & Florian (2001) presents an algorithm description, but it is really difficult to understand - especially the update part between each rule generation. It is simply not specified in an accurate way! So here we try to give a readable version instead. But feel free to read their version as well.

3.2.1 Notation

- A rule r(p, t) consists of a predicate p_r and a target t_r where p_r is the pattern to be matched, and t_r is the annotation to change to
- R_p: the set of predicates present in any rule in R
- good(r): number of samples changed from incorrect to correct when applying r

- bad(r): number of samples changed from correct to incorrect when applying r
- f(r) = good(r) - bad(r)
- V(s): The "vicinity" of the sample s, i.e. the set of samples on whose classification the sample s might depend
- b(s): The sample s corrected by the rule b
- T[s]: The correct tag for s
- C[s]: The tag value for s in the training corpus

3.2.2 The Top Level Description

Let us start with a high level description:

1. ∀ incorrectly tagged samples s:
   - generate all rules r from the templates that change s to the correct tag
   - set good(r):=1
   - This set of rules will be called R
2. ∀ correctly tagged samples s: $\forall r \epsilon R$ where s matches r_p: increase bad(r)
3. Find the rule $b = argmax_{r \epsilon R} f(r)$
4. If f(b) < Threshold or the wanted number of rules is learned, then quit
5. $\forall r \epsilon R$: Update good(r) and bad(r) to compensate for the application of b
6. Repeat from step 3

3.2.3 The update in step 5

This part has to cover a lot of cases so the description is not very fun to read if you are not going to implement it.

∀ samples s, s' where s matches p_b and $s' \epsilon V(s)$

- If b changes s:
  - $\forall p \epsilon R_p$ where p(s') is true:
    - If s' is incorrectly classified:
      - If p(b(s')) = false then decrease good(r) where $p_r = p$ and $t_r = T[s']$
    - Else (If s' is correctly classified):
      - If p(b(s')) = false then $\forall r \epsilon R$ where $t_r \neq C[s']$: decrease bad(r)
  - $\forall p \epsilon R_p$ where p(b(s')) is true:
    - If b(s') is incorrectly classified:
      - If p(s') = false then increase good(r) where $p_r = p$ and $t_r = T[s']$
    - Else (If b(s') is correctly classified):
      - If p(s') = false then $\forall r \epsilon R$ where $t_r \neq C[b(s')]$: increase bad(r)
- Else (If b does not change s):
  - $\forall p \epsilon R_p$ where p(s') is true:
    - If s' is incorrectly classified:

        - If p(b(s')) = false or b(s') is correct then decrease good(r) where $p_r = p$ and $t_r = T[s']$
    - Else (If s' is correctly classified):
        - $\forall$ r where p_r = p and $t_r \neq C[s']$: decrease bad(r)
- $\forall p \epsilon R_p$ where p(b(s')) is true:
    - If b(s') is incorrectly classified:
        - If p(s') = false or s' is correct then increase good(r) where $p_r = p$ and $t_r = T[s']$
    - Else (If b(s') is correctly classified):
        - $\forall$ r where p_r = p and $t_r \neq C[b(s')]$: increase bad(r)

4 The Implementation

First we should mention that most of the code is written in functional programming style. For details about the programming language and the data structures we refer to: Van Roy & Haridi (April 11, 2003); Duchier et al (2003)

We have used a part-of-speech tagged corpus, prepared with correct tags and start tags, as test data. First it is read as a tuple and then converted to an array for easy access. The initial errors are easily found and stored in a list. The rules are generated from the list of errors and stored in a dictionary for fast access. To be able to compute the correct score for each rule, all possible application of rules on correctly tagged samples has to be found. After that the rule with the highest score is selected. Unfortunately, the update step is not yet implemented.

The implementation is realtively easy to read compared to C or Java implementations thanks to the high level data structures. The pattern matching is also very useful when working with data structures like our rule template, for example:

- template(wd(0 a b 0 0) tag(0 0 c 0 0))

together with a target forms a rule template (the zeros is interpreted as non dependent positions): change tag c if the word is b and the word before is a.

One strength with an implementation like this one in Oz is that the structures and rules may be stored in an understandable format, no strange codings are needed. This helps when debugging or exploring of the behavour of the algorithm.

5 Conclusion

Our implementation shows that the algorithm work, at least it finds the best rule. Our choice of programming language seem to be a good one - the templates and necessary data structures are easy to work with in Oz. We can not say very much about the performance, but that is covered enough in Ngai & Florian (2001). The main goal of this work was to explore the FastTBL algorithms and to gain more Oz programming experience.

Unfortunately the algorithm description in Ngai & Florian (2001) is not enough to be able to implement the complete algorithm, since there are plenty of missing details in it. For example:

- what is supposed to be stored and how? An algorithm description in a logical framework, is it really an algorithm description until we know the evaluation strategy?
- it is unclear if the annotations in the training data is supposed to be changed during the training, which is usual in normal TBL
- notations like b(s) and p(b(s)) where b is a rule, p a predicate, and s a sample is not obvious at all. If we treat b as a function, then what would it return? And if it does not match?
- the so called vicinity (the set of samples on whose classification the sample s might depend) is not defined in detail

One day when we have more time to experiment with implementation details we will finish the job since an implementation of FastTBL in Oz would be of general interest.

Bibliography

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

Duchier, D., Gardent, C. & Niehren, J. (2003), *Concurrent Constraint Programming in Oz for Natural Language Processing*, DFKI and SICS.

Lager, T. (1999), The Mju-Tbl System - Logic Programming Tools for Transformation-Based Learning, *in* 'Presented at the Third International Workshop on Computational Natural Language Learning'.

Ngai, G. & Florian, R. (2001), Transformation-Based Learning in the Fast Lane, *in* 'Proceedings of the 39th ACL Conference'.

Ramshaw, L. & Marcus, M. (1994), 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'.

Van Roy, P. & Haridi, S. (April 11, 2003), *Concepts, Techniques, and Models of Computer Programming*, Draft.

A The Oz-code

We have tried to add comments at crucial points in the code.

```
declare

% Import data - access tuple by Corpus.corpus

[CorpusAsTuple] = {Module.link ['./data/10kW_suc.ozf']}

proc {Main}
   local
      MaxRules = 10
      Threshold = 2
      Corpus = {CorpusToArray CorpusAsTuple.corpus 2000}
      BestRules
   in
      BestRules = {FoztblTrain Corpus MaxRules Threshold}
   end
end

fun {FoztblTrain Corpus MaxRules Threshold}
   local
      Errors Rules BestRules
   in
      % Find all samples s where C[s]!=T[S]
      Errors = {GetErrorPositions Corpus}
      {Inspect Errors}

      % A Dictionary of rules: for all samples s (in Errors),
      % generate all rules r that correct the classification of s;
      % increase good(r)
      Rules = {Temps2Rules Corpus Errors Templates}

      % For all correctly classified samples s, generate all predicates
      % p s.t. p(s)=true; for each rule r s.t. p_r=p and t_r!=C[s];
      % increase bad[r]
      {CheckBadApplications Rules Corpus}

      % Find the best rules, see description in BestRules
      BestRules = {List.reverse {GetBestRules Rules Threshold MaxRules}}
   end
end

fun {CorpusToArray C Mx}
   Lo=1 Hi Init=0 NewCorpus in
   if Mx>0 then
      Hi = Mx
   else
      Hi = {Width C}
   end
   NewCorpus = {NewArray Lo Hi Init}
   for I in 1..Hi do
      NewCorpus.I := C.I
   end
   NewCorpus
end

proc {ApplyRuleNoCheck Corpus Rule Pos}
   Target Wd T in
   rule(_ target(Target) _ _) = Rule
   Corpus.Pos = Wd#_#T
   Corpus.Pos := Wd#Target#T
   {Inspect Corpus.Pos}
end

%Run thru Corpus and check for bad applications of rules.
```

```
proc {CheckBadApplications Rules Corpus}
   for Position in {Array.low Corpus}..{Array.high Corpus} do
      for Template in Templates do
         {CheckBadApplicationAt Rules Corpus Position Template}
      end
   end
end

proc {CheckBadApplicationAt Rules Corpus Pos Template}
   UpdateRules in
   proc {UpdateRules Rules C P}
      Keys in
      Keys = {Dictionary.keys Rules}
      for I in 1..{Length Keys} do
         Target in
         Target = {Nth Keys 1}
         if {CorrectClass Target C P} then
            skip
         else
            Rules.Target := {IncreaseBad Rules.Target}
         end
      end
   end
   if {CorrectSample Corpus Pos} then
      Pred ApplicableRules in
      Pred = {TemplateToPredicate Template Corpus Pos}
      ApplicableRules
      = {Dictionary.condGet Rules {PredicateToAtom Pred} {NewDictionary}}
      {UpdateRules ApplicableRules Corpus Pos}
   end
end

fun {GetBestRules RuleDict Threshold MaxNum}
   local
      Rule
   in
      % Find the best rule b=argmax_R f(r)
      Rule = {GetBestRule RuleDict}
      {Inspect Rule}

      % If f(b)<Threshold
      if {FScoreRaw Rule}>=Threshold then
         Rule | {GetBestRules RuleDict Threshold MaxNum-1}

      % Update good and bad for all rules
      else
         nil
      end
   end
end

fun {GetBestRule RuleDict}
   Keys GetBestRuleA GetBestInSubDict GetBestInSubDictA BestRule in
   % RuleDict = dictionary of dictionaries of rules
   % Keys the keys of RuleDict
   % Pos the Pos in Keys
   % BestSoFar the best rule so far, init = 'dummy'
   fun {GetBestRuleA RuleDict Keys Pos BestSoFar}
      if Pos > {Length Keys}
      then BestSoFar
      else {GetBestRuleA RuleDict Keys Pos + 1
            {MaxRule BestSoFar {GetBestInSubDict RuleDict.{Nth Keys Pos}}}}
      end
   end
   fun {GetBestInSubDictA Dict Keys Pos BestSoFar}
      if Pos > {Length Keys}
      then BestSoFar
```

```
      else
         {GetBestInSubDictA Dict Keys Pos + 1
          {MaxRule BestSoFar Dict.{Nth Keys Pos}}}
      end
   end
   fun {GetBestInSubDict Dict}
      Keys = {Dictionary.keys Dict}
   in
      {GetBestInSubDictA Dict Keys 1 'dummy'}
   end

   Keys = {Dictionary.keys RuleDict}
   BestRule = {GetBestRuleA RuleDict Keys 1 'dummy'}
   BestRule = rule(_ _ _ used)
end

fun {MaxRule R S}
   if {FScore R} >= {FScore S} then
      R
   else
      S
   end
end

fun {FScoreRaw Rule}
   Good Bad in
   Rule = rule(_ _ f(Good Bad) _)
   Good-Bad
end

fun {FScore Rule}
   if Rule == dummy orelse {RuleIsUsed Rule} then
      ~999
   else
      Good Bad in
      Rule = rule(_ _ f(Good Bad) _)
      Good-Bad
   end
end

fun {RuleIsUsed Rule}
   Used in
   Rule = rule(_ _ _ Used)
   {IsDet Used}
end

fun {Temps2Rules Corpus Errors Templates}
   Temps2RulesA AddNewRules in
   proc {AddNewRules RulesDict RuleList}
      AddOrUpdateRule in
      proc {AddOrUpdateRule R Dict}
         Target in
         R = rule(_ target(Target) _ _)
         if {Dictionary.member Dict Target} then
            Pred Good Bad UsedFlag in
            rule(Pred _ f(Good Bad) UsedFlag) = Dict.Target
            Dict.Target := rule(Pred Target f(Good + 1 Bad) UsedFlag)
         else
            Dict.Target := R
         end
      end

      case RuleList
      of nil then skip
      [] F|R then
         HashKey TargetDict in
         HashKey = {RulePredToAtom F}
```

```
      % If dict has no entries for that key, return empty list.
         TargetDict = {Dictionary.condGet RulesDict HashKey {NewDictionary}}
         {AddOrUpdateRule F TargetDict}
         RulesDict.HashKey := TargetDict
         {AddNewRules RulesDict R}
      end
   end

   fun {Temps2RulesA Corpus Errors Templates Rules}
      case Errors
      of nil then Rules
      [] First|Rest then
         RulesForError in
         RulesForError = {InstantiateTemplates First Templates Corpus}
         {AddNewRules Rules RulesForError}
         {Temps2RulesA Corpus Rest Templates Rules}
      end
   end
   {Temps2RulesA Corpus Errors Templates {NewDictionary}}
end

% Pos > 0
fun {CorrectClass Class Corpus Pos}
   Corpus.Pos.3 == Class
end

% Pos > 0
fun {CorrectSample Corpus Pos}
   Corpus.Pos.2 == Corpus.Pos.3
end

% Pos > 0
fun {InCorrectSample Corpus Pos}
   Corpus.Pos.2 \= Corpus.Pos.3
end

fun {InstantiateTemplates Error Templates Corpus}
   InstantiateTemplatesA in
   fun {InstantiateTemplatesA Error Templates Corpus Rules}
      case Templates
      of nil then Rules
      [] Template|Rest then
         NewRule in
         NewRule = {TemplateToRule Template Corpus Error}
         if {Member NewRule Rules} then
            {InstantiateTemplatesA Error Rest Corpus Rules}
         else
            {InstantiateTemplatesA Error Rest Corpus NewRule|Rules}
         end
      end
   end
   {InstantiateTemplatesA Error Templates Corpus nil}
end

fun {TemplateToRule Template Corpus Position}
   rule({TemplateToPredicate Template Corpus Position}
        target(Corpus.Position.3)
        f(1 0)
        _)
end


fun {InstTemplatePart Position Part Corpus Type}
   local Out in
      Out = {MakeTuple {Record.label Part} {Width Part}}
      for I in 1..{Width Part} do
         if Part.I \= 0
```

```
         then Index in
            Index = (Position - ({Width Part} div 2) - 1 + I)
            if {And Index > 0 Index < {Array.high Corpus} -1}
            then
               Out.I = Corpus.Index.Type
            else
               Out.I = 'EMPTY'
            end
         else
            Out.I = 'EMPTY'
         end
      end
      Out
   end
end

fun {TemplateToPredicate Template Corpus Position}
   local Wd Tag in
      Template = template(Wd Tag)
      predicate({InstTemplatePart Position Wd Corpus 1}
                {InstTemplatePart Position Tag Corpus 2})
   end
end

fun {IncreaseBad Rule}
   P T G B S in
   Rule = rule(P T f(G B) S)
   rule(P T f(G B + 1) S)
end

fun {DecreaseBad Rule}
   P T G B S in
   Rule = rule(P T f(G B) S)
   rule(P T f(G B - 1) S)
end

fun {IncreaseGood Rule}
   P T G B S in
   Rule = rule(P T f(G B) S)
   rule(P T f(G + 1 B) S)
end

fun {DecreaseGood Rule}
   P T G B S in
   Rule = rule(P T f(G B) S)
   rule(P T f(G - 1 B) S)
end


fun {RulePredToAtom Rule}
   Predicate in
   rule(Predicate _ _ _) = Rule
   {PredicateToAtom Predicate}
end

fun {PredicateToAtom Pred}
   {StringToAtom {TupleToString Pred}}
end

fun {TupleContentsToString Tup Pos Str}
   if Pos > {Width Tup} then
      Str
   else
      if {IsTuple Tup} then
         {TupleContentsToString Tup Pos + 1
          {Append Str {TupleToString Tup.Pos}}}
      else
```

```
         {TupleContentsToString Tup Pos + 1
          {Append Str {AtomToString Tup.Pos}}}
      end
   end
end

fun {TupleToString Tuple}
   {Append {AtomToString {Label Tuple}} {TupleContentsToString Tuple 1 nil}}
end

fun {GetErrorPositions Corpus}
   {GetErrorPositionsA Corpus nil {Array.low Corpus} {Array.high Corpus}}
end

fun {GetErrorPositionsA Corpus Positions PosLow Pos}
   if Pos < PosLow
   then Positions
   else if {InCorrectSample Corpus Pos} then
           {GetErrorPositionsA Corpus Pos|Positions PosLow Pos-1}
        else {GetErrorPositionsA Corpus Positions PosLow Pos-1}
        end
   end
end

Templates = [
             template(wd(0 0 a 0 0) tag(0 0 b 0 0))
             template(wd(0 a b 0 0) tag(0 0 c 0 0))
            ]

{Main}
```