An Algorithm for Learning Hierarchical Classifiers

Jyrki Kivinen, Heikki Mannila, Esko Ukkonen, Jaak Vilo

Department of Computer Science; P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland
email: {jkivinen,mannila,ukkonen,vilo}@cs.helsinki.fi

1 Introduction. In [4] an Occam algorithm ([1]) was introduced for PAC learning certain kind of decision lists from classified examples. Such decision lists, or *hierarchical rules* as we call them, are of the form shown in Fig.1. The purpose of the present paper is to discuss the practical implementation of the algorithm, to present a linguistic application (hyphenation of Finnish), and compare the learning result with an earlier experiment in which Angluin's k-reversible automata were used.

if $x \in C_1$ **or** $x \in C_2$ **or** ... **or** $x \in C_n$ **then** σ_1
else if $x \in C_{n+1}$ **or** $x \in C_{n+2}$ **or** ... **or** $x \in C_{n+m}$ **then** σ_2
...
else if $x \in C_{n+m+\ldots+1}$ **or** $x \in C_{n+m+\ldots+2}$ **or** ... **or** $x \in C_p$ **then** σ_k
else <**default**> σ_{k+1}

Fig. 1. Hierarchical rule of depth k.

The rule in Fig. 1 has k *levels*. Each level contains an *if*-statement that tests whether or not the instance x to be classified belongs to the union of some *basic concepts* C_i and if so, then gives a classification $\sigma_j \in \{+, -\}$ for x. There are k such levels and after them the default level with classification σ_{k+1} which is taken if x does not belong to any C_i appearing on the k ordinary levels. The number of basic concepts tested on each level is not restricted and the classifications σ_j alternates between consecutive levels.

2 Algorithm. To explain the intuition behind the algorithm, consider first the cases $k = 0$ and $k = 1$. For $k = 0$, the hierarchical rule consists of the default rule only. Such a rule can be consistent with the training examples only if all examples are positive or all are negative. Normally this is not the case. Rather, there are *exceptions* to any default rule. Our algorithm tries to classify the exceptions correctly using *exception rules* that are applied before the default rule. If $k = 1$, each consistent basic concept forms an exception rule. From such rules a minimal cost subset is selected such that it (exactly) covers all the exceptions. This gives a hierarchical rule of depth 1

if $x \in C_1$ **or** $x \in C_2$ **or** ... **or** $x \in C_n$ **then** σ_1 **else** σ_2,

where the exception rules are **if** $x \in C_1$ **then** σ_1, **if** $x \in C_2$ **then** $\sigma_1, \ldots,$ **if** $x \in C_n$ **then** σ_1. These exception rules are of depth 0. Concepts $C_1, C_2, \ldots, C_n$ are the *bases* of the rules.

A hierarchical rule of depth k is formed by covering the exceptions to the default rule by exception rules of depth $k - 1$. An exception rule of depth $k - 1$ is like a hierarchical rule of depth $k - 1$, with the default rule (which can be understood as rule **if** $x \in$ **true** **then** σ) replaced by rule **if** $x \in C$ **then** σ.

Concept C is the base of the exception rule. The rule has to be consistent with the examples it covers.

To get an Occam learning algorithm, we should find a *shortest* hierarchical rule that is consistent with the training sample. This is an NP-hard problem in general, but a good enough approximation can be found in polynomial time using the standard greedy heuristics for the set cover problem. This together with dynamic programming allows us to construct approximately shortest exception rules of depth $0, 1, \ldots$ for all basic concepts until enough rules are found to form a short consistent global rule.

Recall that a *weighted set cover problem* is defined by a domain D, an index set I and corresponding sets $D_i, i \in I$, with positive real costs $cost(D_i)$. A solution to the problem is the subset $J \subseteq I$ such that $\bigcup_{j \in J} D_j = D$ and the sum $\Sigma_{j \in J} cost(D_j)$ of costs for sets D_j is minimized. Chvatal ([2]) showed that if the minimal solution has cost M, then the greedy method obtains in polynomial time a solution with a cost at most $M \cdot H(|D|)$, where $H(n) = \sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$. Greedy algorithm builds the approximately minimal set cover incrementally adding one set (that covers some of the remaining elements of D by the lowest cost per element) at a time to the cover. We denote by *Weighted_Set_Cover*(D,I,$\{D_i\}$,$\{cost(D_i)\}$) a function that gives the greedy solution to the set cover problem.

We use this algorithm for finding a minimum cost set of exception rules of depth $k-1$ that covers the exceptions to the default rule. The cost of an exception rule is the sum of the costs of the basic concepts appearing in the rule, and the rule is assumed to cover the examples covered by its base. The same principle is also used for generating the exception rules of depths $0, 1, 2, \ldots, k-1$. The resulting algorithm is given in detail in Fig. 2. The exception rules are implicitly represented in table `Exception_Rules`. The final result is traced starting from entry `Exception_Rules[`$k+1$`][true ]`.

It can be shown that $T = O(k \cdot |R|^2 \cdot |S|^2)$ is an upper bound for the running time of the algorithm in Fig. 2. Here k is the number of levels, R is the set of the basic concepts and S the set of the training examples. Thus we can gain speed by decreasing the number of basic concepts (for example, often actually the same concepts can have different names in R; we can eliminate all except one) and the number of training examples (by standard windowing approach as in [6]). A memory-saving alternative is to re-compute the exception rules when the solution of finite cost is known to exist.

3 Empirical results. We have implemented and tested our algorithm for learning the hyphenation rules for a natural language. In this application the substrings of the training examples define basic concepts C as follows. Let w be any substring occurring in some training example. Then the concept with name w covers all strings that are representable as xwy, where x and y are arbitrary strings. In the hyphenation problem we are given a totally hyphenated word, as 'hy–phen–a–tion', for example. This word defines actually 10 classified examples:

```
h-yphenation -   hyph-enation -   hyphena-tion +   hyphenatio-n -
hy-phenation +   hyphe-nation -   hyphenat-ion -
hyp-henation -   hyphen-ation +   hyphenati-on -
```

INPUT: $S = S_+ \bigcup S_-$ // A sample: positive and negative examples
integer k // Depth of the hierarchical rule to be constructed
OUTPUT: Consistent, near minimal hierarchical rule of depth at most k

$R :=$ set of basic concepts $\cup$ {**true** } // Concepts that cover a nonempty subset of S
Rule_Cost: **array** [1..k+1][R] **of real** // Computed costs for rules
Exception_Rules: **array** [2..k+1][R] **of set of** concepts
D_+, D_-: **array**[R] **of set of** examples
for $r \in R$ compute:
$D_+[r] := S_+ \cap r$; $D_-[r] := S_- \cap r$; $Cost(r) :=$ the cost of r

for $\tau \in \{+, -\}$ // Choose the type of the highest level
$\sigma := \tau$ // In the following: $\overline{\sigma} \equiv$ if $(\sigma = +)$ then $-$ else $+$
for $r \in R$ // Initialize the first level. It can't have exception rules
if $(D_{\overline{\sigma}}[r] = \emptyset)$ **then** Rule_Cost[1][r] := $Cost(r)$
else Rule_Cost[1][r] := ∞ // Inconsistent rule
for level := 2 **to** k+1
$\sigma := \overline{\sigma}$ //Alternate the classification of the level
if (level=k+1) **then** H:={**true**} **else** H:=R
for $r \in H$
J := *Weighted_Set_Cover*($D_{\overline{\sigma}}[r]$, R, $D_{\overline{\sigma}}$, Rule_Cost[level-1])
Exception_Rules[level][r] := J
if no finite cover J exists **then** Rule_Cost[level][r] := ∞
else Rule_Cost[level][r] := $\sum_{j \in J}$ Rule_Cost[level-1][j] + $Cost(r)$

if (Rule_Cost[k+1][**true**] < ∞) // Output the rule
then
Concepts_at_Level[k+1] := { **true** }; Class_at_Level[k+1] := σ
for q := k **downto** 1
Concepts_at_Level[q] := $\bigcup_{r \in \text{Concepts\_at\_Level[q+1]}}$ Exception_Rules[q+1][r]
Class_at_Level[q] := $\overline{\sigma}$; $\sigma := \overline{\sigma}$
else Fail // No consistent rule of depth k and first level labeled by τ exists

Fig. 2. Algorithm that constructs a consistent, near-minimal hierarchical rule.

One possible rule that is consistent with these examples,

if –ph **or** n–a **or** a–t **then** + **else** <**default**> –,

says, for example, that there is a correct hyphenation point between n and a.

We have experimented with the hyphenation of Finnish. The training data, correctly hyphenated words from a lecture containing computer science oriented technical language (1706 words), has been used earlier also to learn the hyphenation rule by Angluin's synthesis algorithm for k-reversible finite-state automata (see [3]). The hyphenation automata had very high accuracy for proposed hyphenation points (about 98%), though they might miss some of the possible hyphens. Unfortunately the automata were large: in average 100 states and 250 transitions for 3-reversible language. That is too much for easy understanding by humans.

In our experiments to learn the hyphenation by hierarchical rules, each example contained one hyphen and was classified as either positive or negative

depending from whether the hyphen was allowable in that position or not. As the basic concepts we used the substrings found from the examples as well as the same substrings transformed so that all original characters were mapped to consonants (K) and vowels (E). Then, for example, concept '–KE' covers all the possible strings where consonant-vowel pair is preceded by a hyphen. We used the windowing approach and started from 0.1 fraction of all the 14880 examples. The algorithm constructed the rule that was consistent with that window. Then the window was enlarged by the examples from remaining part of examples that were misclassified by constructed rule. After 5 such iterations the method resulted 1, 2 and 3-level hyphenation rules that were consistent with all 14880 examples. The final sample consisted of 1378 positive examples and 2397 negative ones. Our algorithm produced an easy to read and understand rule that contained 100 patterns with average length of 4 characters:

```
if
  -ene, -eni, -est, -nas, -nomai, -notta, -salg, -sets, -sos, -täk, -x,
  -yd, ama-, e-no, eru-, g-, i-sar, it-r, ite-o, ity-, la-u, mu-s, p-r,
  tu-s, vyy-, yvi-
then -
else if
  äe-, ö-ym, -alg, -arvo, -d, -ets, -g, -j, -lex, -ma, -omai, -ong, -osa,
  -po, -pr, -rar, -spi, -ti, -to, -v, a-aske, a-e, a-ilm, a-o, a-uks, a-us,
  ais-a, bs-, e-ä, e-a, e-o, e-uks, e-us, e-utt, e-y, ea-as, en-o, i-ä, i-a,
  i-en, i-es, i-o, i-tr, intö-, k-k, k-t, kom-, mus-, n-as, n-kr, n-otta,
  n-st, o-a, o-e, rus-, s-s, t-äk, t-t, ta-aj, ttö-ä, tu-it, tus-, u-a,
  u-e, umo-, y-ä, y-e, E-EEKE, E-KE, E-KEE, EK-KE, KE-KEKK, KEEK-KE, KEK-KE
then +
else default -
```

The hyphenation algorithm (for English) of TEX [5] is a 5-level hierarchical classifier that in some respects is similar to our rules. The synthesis algorithm of [5] for finding the classifier counts probabilities for patterns that allow or prohibit the hyphens, and the resulting rule doesn't have to be totally consistent with the data.

References

1. A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Occam's razor. *Information Processing Letters*, 24:377–380, April 1987.
2. V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of the Operation Research*, 4(3):233–235, August 1979.
3. R. Kankkunen, H. Mannila, M. Rantamäki, and E. Ukkonen. Experience in inductive inference of a hyphenation algorithm for Finnish. In *Proceedings of the Finnish Artificial Intelligence Symposium (STEP'90)*, pages 183–193. Oulu, Finland, 1990.
4. J. Kivinen, H. Mannila, and E. Ukkonen. Learning hierarchical rule sets. In *Proc. of the 5th Annual ACM Workshop on Computational Learning Theory.*, pages 37–44. Pittsburgh, Pennsylvania, July 27–29 1992.
5. M. F. Liang. *Word Hy-phen-a-tion by Com-put-er.* PhD thesis, Stanford University, 1983.
6. J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.