Machine Learning 4, 99–106, 1989
© 1989 Kluwer Academic Publishers—Manufactured in The Netherlands

Efficient Specialization of Relational Concepts

KURT VANLEHN (VANLEHN@PSY.CMU.EDU)
Departments of Psychology and Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213 USA

Editor: Thomas Dietterich

Keywords: concept induction, specialization, version spaces.

Abstract. An algorithm is presented for a common induction problem, the specialization of overly general concepts. A concept is too general when it matches a negative example. The particular case addressed here assumes that concepts are represented as conjunctions of positive literals, that specialization is performed by conjoining literals to the overly general concept, and that the resulting specializations are to be as general as possible. Although the problem is NP-hard, there exists an algorithm, based on manipulation of bit vectors, that has provided good performance in practice.

In the process of including a concept from examples, it is sometimes necessary to make a concept less general because the concept matches a negative example. However, one usually wants to reduce the generality as little as possible. Thus, the so-called specialization problem is to find all specializations of a given concept that are maximally general and yet do not match the given negative example. This paper provides an algorithm for a restricted class of specialization problems.

This particular specialization problem was uncovered during the development of Sierra, a program that learns procedures from examples (VanLehn, 1987). Part of Sierra's job included inducing concepts, expressed in a restricted version of the predicate calculus, from positive and negative examples. In this respect, it was much like Winston's (1975) famous arch learner. Sierra used Mitchell's (1982) version space technique. Although Mitchell's presentation of the candidate elimination algorithm used propositional representations, it was straight forward to adapt them to the first-order representation used by Sierra. Unfortunately, the algorithms were very slow. The slowest one was the specialization algorithm that was used in the course of updating the G set. With the help of Johan de Kleer, a much faster algorithm was found. It allowed Sierra to reduce its learning time from 30 hours to a few minutes in some cases. This paper describes the particular specialization problem that the algorithm solves, then presents the algorithm itself, and finishes with a brief comment on the application of the algorithm to discimination learners, such as ID3 (Quinlan, 1986) and Prism (Langley, 1987).

Most of the speed in the algorithm is due to the particular representation of concepts it uses, so the problem will be defined in terms of it. A concept is represented by a conjunction of positive literals, where all variables are interpreted existentially. An example of a concept is:

Exists (X, Y) *such that color*(X, red) & *on*(X, Y) &

abuts(*bottom*(X), *top*(Y))

where variables are capaitalized and constants are not. This concept would be true of an example where there is a red block on top of a green block and a blue block on top of the table. Because the logical form is always the same—conjunctions of literals embedded in existential quantifiers—it is convenient to drop the logical symbols and represent concepts as sets of literals. The above concept would be represented as:

{*color*(X, red), *on*(X, Y), *abuts*(*bottom*(X), *top*(Y))}

Examples are represented in the same way as concepts except that variables are not allowed in examples (i.e., they are ground sentences of first-order logic).

A concept g is said to match another concept or example s if there exists a substitution for the variables of g such that every literal in g is equal to some literal in s. To put it more succinctly, g matches s if and only if there exists a substitution that makes g a subset of s. A concept is said to be consistent with a set of positive and negative examples if it matches each of the positive examples and none of the negative examples. Given two concepts, g and s, if g matches s, then g is said to be a generalization of concept s, and s is said to be a specialization of g.

In this version of the specialization problem, the only type of specialization allowed is the one just defined. Thus, the following concept would be a specialization of the one above:

{*color*(X, red), *on*$(X, b17)$, *abuts*(*bottom*(X), *top*$(b17)$), *color*$(Z, blue)$}

because one positive literal (the last one) has been added and one of the variables has been turned into a constant.

Lastly, and most importantly, variables are assumed to designate distinct objects. That is, there is an implicit inequality between each pair of variables so that $\{p(X, Y)\}$ really means $\{p(X, Y), X \neq Y\}$. This implies that $\{p(X, Y)\}$ is not a generalization of $\{p(V, V)\}$.

The assumption that distinct variables designate distinct objects implies that the set of all generalizations of a concept s corresponds to the set of all subsets of the literals of s. To see this, suppose that g is an arbitrary generalization of s. This means that g matches s, so there is some mapping of the variables of g into the variables of s such that the image of g under the mapping is a subset of s. This mapping must be a one-to-one mapping, because distinct variables must designate distinct objects. Thus, the mapping between g and its image in s is one-to-one and onto, which means that g is an alphabetic variant (i.e., the names of its variables have been systematically changed) of some subset of s. This shows that every generalization of s is isomorphic to some member of the power set of s.

This completes the definition of the representation language for concepts and examples. It is a rather ordinary representation language, similar in most important respects to the semantic net representations used by Winston and others. Given this representation language, the specialization problem can now be stated.

In Mitchell's version space approach to concept formation, the set, V, of all concepts consistent with the examples given so far (i.e., the version space) is represented by two subsets of V, called G and S. The G set is the set of all maximally *general* elements of V (i.e., for each $g \in$ G, there does not exist any element of V more general than g). The S set is the set of all maximally specific elements of V (i.e., for each $s \in$ s, there is no element of V more specific than s). Moreover, Mitchell (1978) proves that G and S form the boundaries of an interval, so that for each g in G, there is some concept s in S such that g is a generalization of s, and for each concept s in S there is a concept g in G such that s is a specialization of g.

The specialization problem to be discussed occurs when the G set must be modified in order to make its members consistent with a newly received negative example. Suppose a concept $g \in$ G matches the negative example. This means it is too general and must be specialized. Given the assumptions above, specializing it means finding literals from some $s \in$ S that can be added to g. This leads to the following specialization problem:

Given:
  n, a negative example,
  g, a member of G that matches n, and
  s, a member of S that does not match n and is a specialization of g.
Find a set of concepts C such that:
  no concept in C matches n,
  every concept in C is a specialization of g,
  every concept in C is a generalization of s, and
  no concept in C is a generalization of any other concept in C.

The G and S set often have multiple members, so the above problem may have to be solved for more than one g/s pair and the resulting C sets must be merged to form the new G set.[1]

A direct, but inefficient way to solve the problem is to use a brute-force search that adds a single literal from s to the evolving g, then checks to see if the resulting concept matches n. If it still does, then the search continues. If the concept does not match n, then the search has found one candidate for C, so it backtracks to find more. The branching factor of the search is $|s| - |g|$ where $|x|$ is the number of literals in x. At one time, Sierra used this technique for updating its version spaces. Since $|s| - |g|$ ranged between 30 and 150, Sierra often took 30 hours or more just to handle a few negative examples. With the new algorithm, Sierra completes the same calculations in a minute or two.

The algorithm uses the same search as before, and thus has the same branching factor, but the search step is made significantly faster. The slowest step in the search is checking to see whether a newly built concept matches the negative example. However, this check always occurs between *n* and a concept that is some subset of *s*. The algorithm takes advantage of this by precomputing a bit-vector representation that allows the fast parallel bit-vector computation provided by most machines to be used in place of the slow matching step. Thus, although the search still has the same exponential combinatorics as before, it is sped up by such a large constant factor that the overall speed becomes quite acceptable. Since the problem is NP-hard, it doubtful that a polynomial solution will be found.[2]

There are two steps to the initialization that precede the search. The first step is to enumerate all substitutions of objects in *n* for variables in *s*. For example, if there are four variables in *s* and seven objects in *n*, then there are $7 \times 6 \times 5 \times 4$ substitutions.[3] Having enumerated the substitutions, each literal in *s* is assigned a bit vector. The bit vector has one bit position for each substitution. If the literal is in *n* under a given substitution, then the bit is zero. If the literal is not in *n*, then the bit is one. If any literal has a bit vector that is all ones, then it is not in *n* under any substitution, so it does not match *n*. The result of this step of the initialization is a set, call it *s-pairs*, that consists of the literals of *s* paired with their bit vectors.

The second step in the initialization is to convert *g* into a subset of *s* by finding a substitution of its variables for variables in *s*. If there is more than one such substitution, then the search must be run once for each substitution. Let g' be a version of *g* with variables from *s* substituted for its original variable. Let *g'-pairs* be a set consisting of the literals of g' paired with their bit vectors.

Attaching bit vectors to literals converts the matching problem into a well-known problem, the set covering problem: Given a target set and a collection of subsets of it, find a cover for the target set, where a *cover* is a set of subsets such that the union of those subsets equals the target set. In this case, the target set is represented by a bit vector that is all ones, and the collection of subsets is represented by the bit vectors of *s-pairs*. The goal is to find a set, call it *c-pairs,* which is a superset of *g'-pairs* and a subset of *s-pairs*, such that the logical Or of the bit vectors of *c-pairs* is all ones. When the logical Or (union) of the bit vectors of *c-pairs* is all ones, then the conjunction of literals of *c-pairs* does not match *n* because those literals are not a subset of the literals of *n* under any substitution. Hence, this conjunction is almost the concept needed: it is a specialization of *g* because it is a superset of the literals of g'; it is a generalization of *s* because it is a subset of the literals of *s*; and it does not match *n*. The only criterion left to meet is that the concepts be maximally general with respect to each other. The criterion can be partially met by using an appropriate set covering algorithm, which is the topic we turn to next.

There are several different versions of the set covering problem, depending on the kind of cover desired. An *irredundant* cover is a cover that is not properly included in any other cover (i.e., none of its subsets is redundant in that it can be removed from the cover without affecting the cover's equality to the target set).

Table 1. Well's algorithm for finding irredundant covers.

```
(Defun FindCover (Cover Covered Duplicates Usable)
(And Usable
 (Let
  ((Candidate (Car Usable))
   (NewCover (Cons Candidate Cover))
   (NewCovered (LogicalOr Covered (BitVector Candidate)))
   (NewDuplicates
     (LogicalOr Duplicates (LogicalAnd Covered (BitVector Candidate))))
   (NewUsable (Cdr Usable))
  (Cond
   ((For X in NewCover thereis
         (AllOnes
            (LogicalOr
               (LogicalNot (BitVector X))
               NewDuplicates))))
    (FindCover Cover Covered Duplicates NewUsable))
   ((AllOnes NewCovered)
    (Cons NewCover
      (FindCover Cover Covered Duplicates NewUsable))
   (T
    (Append
      (FindCover NewCover NewCovered NewDuplicates NewUsable)
      (FindCover Cover Covered Duplicates NewUsable)))))))
```

Let C′ stand for all possible irredundant covers of the all-one bit vector using subsets of *s-pairs* and supersets of *g′-pairs*. Then the concepts represented by C′ are maximally general. If it were otherwise, then there would be two elements of C′, call them *a* and *b*, such that the literals of *a* are a proper subset of the literals of *b*. But then *b* would not be an irredundant cover, because some of its elements could be removed and yet the logical Or of the remaining bit vectors would still be all ones. So C′ represents concepts that are maximally general with respect to themselves, are specializations of *g′* and generalizations of *s*, and do not match *n*.[4]

As noted earlier, the G set and the S set might have multiple members, and each *g* might have multiple *g′*. In order to get a new G set, all the C′ generated from specific *s*/*g′* pairs must be merged.

In order to generate irredundant covers, Sierra uses an algorithm from Wells (1971, section 6.4.3), which is based on depth-first search (see table 1). The algorithm keeps the cover generated so far in the variable Cover and the possible cover elements that have not been used yet in the variable Usable. It also maintains in the variable Cover a bit vector representing the substitutions that have been covered so far. The trick to Well's algorithm is to prune the search whenever adding a new cover element to the cover would cause the cover to become redundant. In order to detect this, it maintains in the variable Duplicates a bit vector of the substitutions that have been covered by two or more cover elements. This piece of search pruning is achieved in the first cond clause of table 1. In order to calculate C′, FindCover is called with the following argument values:

Cover = *g′-pairs*
Covered = the bit vector that is the logical Or of the bit vectors of *g′-pairs*

Table 2. Concepts and bit vectors used during FindCover

Concept	X=b1 Y=b2	X=b1 Y=t	X=b2 Y=b1	X=b2 Y=t	X=t Y=b1	X=t Y=b2
on(X,Y)	0	1	1	0	1	1
table(Y)	1	0	1	0	1	1
red(X)	0	0	1	1	1	1
red(X),on(X,Y)	0	1	1	1	1	1
red(X),on(X,Y),table(Y)	1	1	1	1	1	1
red(X),table(Y)	1	0	1	1	1	1

Duplicates = the bit vector that is the logical Or of the bit vectors of *g'-pairs*
Usable = *s-pairs* with *g'-pairs* removed

As an illustration of the algorithm's execution, consider the following problem:

n = {block(b1), red(b1), block(b2), green(b2), table(t), on(b1,b2), on (bt,t)}
g' = {red(X)}
s = {red(X), on(X,Y), table(Y)}

The concept g' matches the negative example, and it is a specialization of s with the same variables as s. Table 2 shows several concepts with their associated bit vectors. The columns correspond to bits in the bit vectors. Each column is labeled with the substitution that its bit position represents. Since there are two variables in s, and three constants in n, there are 3 × 2 substitutions. The first three rows of the table correspond to the elements of *s-pairs*. Notice that a bit is one if the literal does *not* match n under the substitution corresponding to that bit position. The algorithm is initialized with Cover equal to *g'-pairs*, whose only element corresponds to the third row in the table. This means that Covered is the bit vector shown in the third row. Because Covered is not all ones, the search takes the first element of Usable (which is initialized to the first two rows of the table), and conjoins it with Cover. This produces the concept shown in the fourth row of the table. This concept's vector is not all ones, so the algorithm again adds an element of Usable, resulting in the concept shown in the fifth row. This concept's bit vector is all ones, so it is saved for later output. The search continues, eventually producing the sixth row in the table. However, all these later search paths fail because Usable empties before the search finds any new covers. Thus, FindCover returns a singleton list consisting of the cover corresponding to the fifth row of the table. Although this example illustrates the bit-vector representation and the algorithm's flow of control, it does not exhibit the search pruning caused by Duplicates.

As mentioned earlier, this algorithm was used with success as part of a version space maintenance module. It may also be useful in inductive concept learners that do discrimination learning, such as ID3 (Quinlan, 1986) and PRISM (Langley, 1987). These programs were initially developed with propositional concept representations, just as the version space algorithm was. The set covering technique could be adapted to let them use the first-order representations defined above. The following paragraphs sketch this application.

Suppose for the sake of illustration that the original version of the discrimination learning problem is to build a concept that discriminates positive from negative examples (i.e., binary discrimination, rather than multi-ary) and that concepts are represented as feature sets. Then the main step in the search is to take a concept (e.g., tree node) that is not yet capable of discriminating the positive from the negative examples, find a single feature that discriminates as many positive from negative examples as possible, and extend the concept with that feature.

Now suppose that concepts are represented in the first-order language defined above. The discrimination learner's search would now work with literals rather than features. However, adding a literal to a set of literals does not have the same easily calculated effect on a concept's extension as adding a feature to a set of features. Thus, the search would have to recalculate the extension of a concept after selecting a literal and adding it to the concept. It could do this by matching the new concept against every negative and positive example in the data. Clearly, this would be quite expensive.

However, the same bit-vector representation used above should help here as well. In this case, there are multiple negative examples to match, so each bit position stands for the result of matching to one example with one substitution. This will make the bit vectors longer in this case than in the candidate elimination algorithm, which processes one example at a time. However, the extension of a concept can still be calculated by taking the logical Or of the old concept's bit vector and the literal's bit vector. Presumably, this would still be faster than matching even if the bit vectors were long. Obviously,.specialized parallel hardware could increase the speed even more.

Notes

1. Merging the C set consists of more than a simple union, however. Although the definition of C specifies that the concepts in C are maximally general with respect to each other, when multiple C sets are unioned in order to get the new version of the G set, the result must be filtered one last time in order to remove concepts that are specialized by others in the set. (It would be interesting to see if leaving these "almost" maximally general concepts in the G set speeds up the candidate elimination algorithm.)
2. As shown later, this problem can be converted to a set covering problem. In theorem 10.2.9 of Aho, Hopcroft and Ullman (1974), it is shown that set covering is NP-complete.
3. Sierra's concept representation enforces additional constraints that reduces the number of substitutions still further. This reduction is crucial, because Sierra's concepts usually had between 10 and 50 variables. In retrospect, the same reduction could be achieved more elegantly by assigning types to variables and objects, then enumerating only substitutions that paired objects and variables of the same type.

4. For version space maintenance, the definition of the G set implies the covering algorithm desired will be one that finds irredundant covers. For other applications of this technique, one might consider an algorithm for calculating minimal covers, which are covers with the fewest number of elements. This would cause the specialization algorithm to prefer maximally "simple" concepts instead of maximally general ones.

References

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1974). *The design and analysis of computer algorithms.* Reading, MA: Addison-Wesley.

Langley, P. (1987). A general theory of discrimination learning. In D. Klahr, P. Langley, & R. Neches (Eds.), *Production system models of learning and development.* Cambridge, MA: MIT Press.

Mitchell, T. M. (1978). *Version spaces: An approach to concept learning* (Tech. Rep. STAN-CS-78-711). Palo Alto, CA: Stanford University, Computer Science Department.

Mitchell, T. M. (1982). Generalization as search. *Artificial Intelligence, 18,* 203–226.

Quinlan, J. R. (1986). The effect of noise on concept learning. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* (Vol. 2). Los Altos, CA: Morgan Kaufmann.

VanLehn, K. (1987). Learning one subprocedure per lesson. *Artificial Intelligence, 31,* 1–40.

Wells, M. B. (1971). *Elements of combinatorial computing.* New York: Pergamon Press.

Winston, P. H. (1975). Learning structural descriptions from examples. In P. H. Winston (Ed.), *The psychology of computer vision.* New York: McGraw-Hill.