Using Background Knowledge to Build Multistrategy Learners

Claude Sammut

School of Computer Science and Engineering
University of New South Wales
Sydney Australia 2052
claude@cse.unsw.edu.au

Abstract

This paper discusses the role that background knowledge can play in building flexible multistrategy learning systems. We contend that a variety of learning strategies can be embodied in the background knowledge provided to a general purpose learning algorithm. To be effective, the general purpose algorithm must have a mechanism for learning new concept descriptions which can refer to knowledge provided by the user or learned from some other task. The method of knowledge representation is a central problem in designing such a system since it should be possible to specify background knowledge in such a way that the learner can apply its knowledge to new information.

Introduction

There are many reasons why one may wish to combine a variety of learning strategies in one system. Many experiments have confirmed that we are yet to find a single learning algorithm that does best in all circumstances (for example, Michie, Spiegelhalter & Taylor, 1994). It is often more effective to use different algorithms at different times than to try to use one algorithm all the time. This approach raises several questions, including: under which circumstances is a particular method appropriate and how large does our library of algorithms have to be?

Often, multistrategy learning systems have a fixed set of algorithms to draw upon and their interaction is predetermined by the programmer. In other words, they are often designed to be specific to a particular type of learning problem. In this paper, we suggest that general purpose multistrategy learners can be constructed by allowing the learner to make use of complex background knowledge.

We contend that a great deal of flexibility can be gained by a machine learning system which uses a concept representation language that permits references to user defined background knowledge as well as knowledge derived from other learning tasks. First, we illustrate the power of complex background knowledge with an example of numerical reasoning in inductive logic programming. Next, we describe a machine learning system that encourages the use of learned as well as predefined background knowledge. Finally, we discuss the connection between background knowledge and multistrategy learning.

An Example: Numerical Reasoning in ILP

The effectiveness of background knowledge as a means of providing a variety of learning strategies can best be seen through an example. Sammut, Hurst, Kedzier and Michie (1992) describe a learning problem where a human pilot is required to fly an aircraft in a flight simulator. During the flight the actions of the pilot are logged, along with the situation in which the action is performed. Flight logs are used as input to an induction program which generates rules for an autopilot.

The task of piloting an aircraft through a complete flight plan is a complex task involving a number of stages each of which is defined by a particular goal. For example, the pilot may be told to take off; climb to a particular altitude; turn to some heading; maintain straight and level flight until a certain marker is reached, etc. For each stage of a flight, we build a separate set of control rules. Further complexity is added by the fact that an aircraft has a number of controls, eg. elevators, ailerons, rudder, flaps, throttle, etc. Within each stage we build rules for each control available to the pilot. The result is a two-level controller where the top level is an "executive" which invokes low level agents to perform a particular task. Thus, "Learning to Fly" applies machine learning in a number of ways to build a control system that has different strategies for different situations. While the result is a multistrategy controller, our original experiments only involved a single learning algorithm., C4.5 (Quinlan, 1993).

Although these experiments resulted in working autopilots capable of flying the aircraft from take-off to landing, the rules that were generated were often large, difficult to read and not always robust under different flight conditions. One reason for some of these difficulties is that only the raw data from the simulator were presented to the learning algorithm. The data obtained from the simulator include the position, orientation and velocities of the aircraft as well as the control settings. While these data are complete in the sense that they contain all the information necessary to describe the state of the system, they are not necessarily presented in the most convenient form. For

Table 1. Background Predicates

pos(P, T)	position, *P*, of aircraft at time, *T*.
before(T1, T2)	time, *T1*, is before time, *T2*.
regression(ListY, ListX, M, C)	least-square linear regression, which tries to find $Y = M \times X + C$ for the list of X and Y values.
linear(X, Y, M, C)	linear(X, Y, M, C) :- Y is M*X+C.
circle(P1, P2, P3, X, Y, R)	fits a circle to three points, specifying the centre (X, Y) and radius, R
=<, >=, abs	Prolog built-in predicates

example when a pilot is executing a constant rate turn, it makes sense to talk about trajectories as arcs of a circle. Induction algorithms, such as C4.5, can deal with numeric attributes to the extent that they can introduce in equalities, but they are not able to recognise trajectories as arcs or recognise any other kind of mathematical property of the data.

Srinivasan and Camacho (forthcoming) have shown how such trajectories can be recognised with the aid of a sophisticated mechanism for making use of background knowledge. This mechanism is part of the inductive logic programming system Progol (Muggleton, 1995). The feature of Progol that most interests us here is that the learning algorithm is embedded in a custom built Prolog interpreter. Concepts learned by Progol are expressed in the form of Prolog Horn clauses and background knowledge is also expressed in the same form. Moreover, the Prolog interpreter's built-in predicates may appear in either learned or background clauses.

Like Marvin (Sammut, 1981; Sammut & Banerji, 1986), Progol begins by taking the description of a single positive example and *saturates* the description by adding to it literals derived from background knowledge. These new literals result from predicates in the background knowledge being satisfied by data in the example. Suppose the example contained the numbers 2 and 3 somewhere in the example description. Also suppose that the built-in predicate '<' were supplied as background knowledge. Saturation might add a new literal, $X < Y$, to the description of the example. X has been substituted for 2 and Y for 3. Progol permits *mode* declarations to restrict the application of predicates to avoid an explosion in the number of literals that the saturation procedure may try. Note that *any* of Prolog's built-in predicates can be declared as background knowledge and any user defined Prolog program may also be declared as background knowledge. Furthermore, since Progol concept representation language is Horn clauses, any learned concepts may also be entered as background knowledge.

When saturation is completed, Progol has a *most specific clause* to serve as a bound on a general-to-specific search. Beginning with the most general clause, ie. one with an empty body, Progol tries to find a subset of the most specific clause that satisfies a minimum description length criterion for the best clause. Muggleton (1995) describes the search procedure.

Progol's ability to use background knowledge can be used in a multistrategy approach to learning to pilot an aircraft. Srinivasan and Camacho applied Progol to the problem of learning to predict the roll angle of an aircraft during a constant rate turn at a fixed altitude. To do this effectively, the target concept must be able to recognise the trajectory as an arc of a circle. The predicates shown in Table 1 are included in the background knowledge.

The *pos* predicate is the 'input' to the learner since it explicitly describes the trajectory of the aircraft as a sequence of points in space. These points are derived from flight logs. The *before* predicate imposes an ordering on the points in the trajectory. The *regression* predicate provides the key to finding the relationship between the roll angle and the radius of the turn. The mode declaration for *regression* specifies that the first two arguments are lists which described the sequence of pairs of coordinates for the aircraft during the turn. The mode declaration causes Progol to generate these lists and invokes *regression* which performs a least-square regression to find the coefficients of the linear equation which relates roll angle and radius. *Regression* must be accompanied by another background predicate, *linear*, which implements the calculation of the formula.

The regression predicate is an intermediate relationship that does not appear in the final description of the learned concept. During saturation, *regression* recognises the relationship between the angle and radius, given the sequence of aircraft positions. Once the coefficients of the linear equation are available, Progol can generate a reference to *linear*. Thus the theory produced is[1]:

```
roll_angle(T1, Angle) :-
   pos(P1, T1), pos(P2, T2), pos(P3, T3),
   before(T1, T2), before (T2, T3),
   circle(P1, P2, P3, _, _, Radius),
   linear(Angle, Radius, 0.043, -19.442).
```

The *circle* predicate recognises that P1, P2 and P3 fit a circle of radius, *Radius* and regression finds a linear approximation for the relationship between *Radius* and *Angle* which is:

$$Angle = 0.043 \times Radius - 19.442$$

[1] The theory as been siplified slightly to avoid lengthy explanations of details not relevant to this discussion.

The '_' arguments for *circle* are "don't cares" which indicate that, for this problem. we are not interested in the centre of the circle.

Machine learning algorithms generally tend to be poor at numerical reasoning. This is usually left to scientific discovery systems. However, the flight trajectory example illustrates why, for many domains, it is important to incorporate numerical reasoning into induction. One way to do this is by adding background knowledge for a variety of equation solvers.

Because Progol permits the use of arbitrary Prolog code in its representation of concepts, it can invoke a variety of methods for fitting data which go beyond Progol's own ILP style of learning. Linear regression is just one example of a statistical method for fitting data. A library of such predicates gives a system like Progol the ability to use different strategies depending on the type of data available. Since the components of the library are just Prolog programs, the learning algorithm does not have to be modified to permit different data fitting methods to be added. Furthermore, the library may even include other learning algorithms. In the following section, we describe an ILP system that has such a library of learning algorithms.

Using Induction to Build Background Knowledge

An experimental ILP system is currently being developed which is actually a Prolog interpreter with a variety of machine learning and statistical algorithms included as built-in predicates. The algorithms included so far are: Aq (Michalski, 1973; 1986), ID3 with pruning (Quinlan, 1979; 1993), Induct-RDR (Gaines, 1989), naive Bayes, regression trees (Breiman et al, 1984), linear discriminant, DUCE (Muggleton, 1987). All of these algorithms can be invoked as predicates which are available as background knowledge to an ILP system similar to Progol.

First, we give a brief example of the individual use of some these algorithms and then we describe how they may be combined in a multistrategy learner. Like most attribute/value based systems, a description of the attributes and their legal values is required. Following the example of Cendrowksa (1987):

```
mode lens(
   age(young, pre_presbyopic, presbyopic),
   prescription(myope, hypermetrope),
   astigmatism(not_astigmatic, astigmatic),
   tear_production(reduced, normal),
   lens(hard, soft, none)
).
```

The task here is to learn to predict whether a person should wear a hard or soft contact lens or wear no contact lens. The class value is always the last attribute in the list. Examples are entered as ground unit clauses in Prolog's database. A small sample for the problem described above is:

```
lens(young, myope, not_astigmatic, reduced, none).
lens(young, hypermetrope, not_astigmatic, reduced, none).
lens(young, hypermetrope, not_astigmatic, normal, soft).
lens(pre_presbyopic, myope, astigmatic, reduced, none).
lens(pre_presbyopic, myope, astigmatic, normal, hard).
lens(presbyopic, myope, not_astigmatic, reduced, none).
lens(presbyopic, myope, not_astigmatic, normal, none).
lens(presbyopic, hypermetrope, astigmatic, normal, none).
```

To run an induction algorithm, we simply invoke it as a procedure call in Prolog. Thus, to build a decision tree, the following call is executed:

```
id(lens)?
```

The output of all the induction algorithms is a Prolog clause which captures, the decision tree or set of rules. Since the output is in standard Prolog form, it can be asserted into the database and used as an ordinary program.

```
lens(Age, Prescription, Astigmatism, TearProduction, Lens) :-
   (TearProduction = reduced -> Lens = none
   |TearProduction = normal ->
      (Astigmatism = not_astigmatic -> Lens = soft
      |Astigmatism = astigmatic ->
         (Prescription = myope -> Lens = hard
         |Prescription = hypermetrope ->
            |Age = pre_presbyopic -> Lens = none
            |Age = presbyopic -> Lens = none))).
```

ILP learning systems are typically used in domains where relational information is important. These are domains in which the common attribute/value representation of traditional induction algorithms makes representation of examples and concepts difficult. However, when a relational representation is not essential, propositional learning algorithms are often still preferable because of their speed and ability to handle noise and numeric data.

We have seen how Srinivasan and Camacho used linear regression to fit numeric data. In the same way, it is possible to use regression tree methods as background knowledge to construct a function. The following data are samples from a flight log. Each clause gives the position of the aircraft and the pitch angle in 10ths of degrees.

```
pitch(500, -2461, -977, -7).
pitch(498, -2422, -955, -17).
pitch(489, -2385, -932, -20).
pitch(413, -2219, -801, -20).
pitch(397, -2190, -771, -18).
pitch(369, -2131, -709, -19).
pitch(265, -1925, -494, -15).
pitch(252, -1895, -463, -13).
pitch(231, -1835, -400, -12).
pitch(211, -1771, -339, -10).
```

```
pitch(202, -1737, -311, -8).
pitch(195, -1702, -283, -7).
```

Suppose we wish to predict the pitch of the aircraft, based on its current position. This is a gross simplification of what we would really want to know since pitch is usually not a function of position alone.

Running the CART regression algorithm, we obtain the following regression tree[1]:

```
pitch(Alt, Dist, X, Pitch) :-
    (Alt <= 216 ->
        (Alt <= 21.5 ->
            Pitch = -3.86;
            (Alt <= 83.5 ->
                Pitch = -7.70;
                Pitch = -6.11));
        Pitch = -17.18).
```

As with linear regression, regression trees can be built during saturation. Two background predicates are required to achieve this. Like linear regression, one predicate is needed to invoke the regression tree algorithm during saturation and the second predicate is needed for execution of the regression tree, eg. pitch. In this case, however, the second predicate is built by the first predicate during saturation. Note that when predicates of this form are built, the system is performing quite sophisticated constructive induction.

Structured Induction

The present solution to the "Learning to Fly" task is an example of *structured induction*. Shapiro (1987) introduced the idea of structured induction as a means of simplifying a learning task and for making the result of learning more human readable. His experiments were conducted in the familiar domain of chess end-games. Quinlan (1979) had previously demonstrated that it was possible to induce decision trees for this domain. However, they were usually large and end-game experts could find little in them that corresponded to their own intuition. To overcome this problem, Shapiro obtained the help of a chess master who was able to describe high-level features that players looked for.

Armed with this knowledge, Shapiro induced decision trees for each of the high-level features and organised the whole knowledge base as a tree of trees. The top-level tree was hand-crafted from the knowledge obtained by the chess expert. The subtrees were built by induction. The result was an accurate solution which also made sense to chess experts.

Shapiro used as uniform representation and the same type of induction algorithm throughout his analysis. This is also true of the original flight control system. All the control agents were synthesised using the same learning algorithm. However, we have seen that different algorithms may be used when one is more appropriate than another, thus extending the notion of structured induction.

It should be noted that, multistrategy learning in the context described here is not fully automatic, but rather, is a collaboration between human and machine. In principle, it is possible to give systems like Progol very little guidance about what kinds of predicates to try to use in the saturated clause. In practice, however, the run time for an unconstrained search is prohibitive for large problems. Thus, the user normally provides mode declarations to specify the type of predicate to look for.

Background Knowledge and Generalisation

Having seen some specific examples of the use of background knowledge in multistrategy learning, what lesson can be learned?

Often when one sees many specialised strategies doing subtly different things, one is tempted to ask if there is some underlying principle that is common to all cases.

To illustrate this, let us consider Michalski's (1983) categorisation a number of generalisation operations. For example, the "climbing a generalisation tree" is given as a distinct generalisation operator. In the following, we define a simple type hierarchy in terms of Horn clauses.

```
living_thing(X) :- plant(X).
living_thing(X) :- animal(X).

animal(X) :- mammal(X).
animal(X) :- fish(X).

mammal(X) :- elephant(X).
mammal(X) :- whale(X).
```

Suppose 'fred' is an example of the concept 'p' which we wish to learn. Fred has the property that it is an elephant.

```
p(fred) :- elephant(fred).
```

Saturation proceeds as follows, *mammal(fred)* is true since elephant(fred) is given. Having concluded *mammal(fred)*, *animal(fred)* follows and consequently so does *living_thing(fred)*. A system like Progol would construct a *most specific clause* of the form:

```
p(fred) :-
        elephant(fred),
        mammal(fred),
        animal(fred),
        living_thing(fred).
```

It then becomes a matter for the search procedure to determine which subset of literals in the *most specific clause* is the target concept. Thus appropriate background

[1] Prolog's standard notation for the logial 'or' is ';'. We use this symbol and '|' interchangably.

knowledge gives us generalisation by climbing a type hierarchy. In what way does this differ from the following implementation of the "closing the interval" rule?

The background knowledge consists of the clauses:

X in L..H :- X =< L, L =< H.

and the example,

p(2, 1, 3).

generalises to:

p(N, X, Y) :- N in X..Y.

since *2 in 1..3* is satisfied. Thus the "closing the interval" rule can also be obtained from background knowledge.

In fact, linear regression and regression trees can also be seen as generalisation rules since they are implemented in exactly the same way as these "basic" generalisation rules are. So the main lesson to be learned is that while different forms of generalisation may be appropriate for different types of data, these forms can be described using a uniform representation. This means that they can all share a common mechanism for applying them. This, in turn, leads to a very flexible framework for learning.

Scientific Discovery

Induction and scientific discovery are often thought of as related but separate sub-fields of machine learning. This need not be the case. It is clear from the use of linear regression that finding models for numerical data can be accommodated in an induction setting. Indeed, a model finding algorithm becomes another generalisation operation. The search for combinations of variables to create formulae can be guided by background knowledge to indicate what classes of formulae to look for.

Srinivasan (personal communication) has used a scheme similar to the one described for flight trajectories to learn the equations of motion for a pole and cart system. A simulation of a pole and cart was run with random actions applied to push the cart left or right. A trace of the systems behaviour was input to Progol with the result that equations of motion predicting the next state from the current state were synthesised.

Discussion

This discussion as taken place in the setting of inductive logic programming. However, the issues raised here go beyond a debate about the suitability or otherwise of logic as a representation language. The key issue are that:

- the background knowledge and the concept description language should be the same and
- the expressions in the language should be executable as programs.

Uniformity of representation ensures that knowledge can be reused. That is, learned concepts can be added to background knowledge for further learning. The user can also provide pre-defined background knowledge and the system will not have to know what is user defined and what has been learned because both are treated the same way.

If the representation language is executable and therefore can be used to write programs, the background knowledge can describe very powerful concepts, including other learning and data modelling algorithms.

It happens that Horn clause logic has these features. For all the limitations of this form of representation, it does provide an excellent framework for building flexible learning systems.

Acknowledgments

Thanks to Ashwin Srinivasan and Mike Bain for their assistance in understanding the intricacies of Progol.

References

Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. 1984. *Classification and Regression Trees*. Belmont, CA: Wadsworth International Group.

Cendrowkska, J. 1987. An algorithm for Inducing Modular Rules. *International Journal of Man-Machine Studies*, 27(4): 349-370.

Gaines, B. R. 1989. An ounce of knowledge is worth a ton of data. In A. M. Segre (Ed.), *Proceedings of the Sixth International Workshop on Machine Learning*, 156-159. Ithaca, New York: Morgan Kaufmann.

Michalski, R. S. 1973. Discovering Classification Rules Using Variable Valued Logic System VL1. In *Third International Joint Conference on Artificial Intelligence*, 162-172.

Michalski, R. S. 1983. A Theory and Methodology of Inductive Learning. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach*. Palo Alto: Tioga.

Michalski, R. S., Mozetic, I., Hong, J., & Lavrac, N. 1986. The multi-purpose incremental learning system AQ15 and its testing application to three medical domains. In *Proceedings of AAAI-86*, Philadelphia: Morgan Kaufmann.

Michie, D., Spiegelhalter, D. J., & Taylor, C. C. (Ed.). 1994. *Machine Learning, Neural and Statistical Classification*. Elis Horwood.

Muggleton, S. 1987. Duce, An oracle based approach to constructive induction. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 287-292. Milan: Morgan Kaufmann.

Muggleton, S. 1995. Inverse entailment and Progol. *New Generation Computing*. 13:245-286.

Quinlan, J. R. 1979. Discovering rules by induction from large collections of examples. In D. Michie (Eds.), *Expert Systems in the Micro-Electronic Age*. Edinburgh: Edinburgh University Press.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Sammut, C. A. 1981. Concept Learning by Experiment. In *Seventh International Joint Conference on Artificial Intelligence*. Vancouver.

Sammut, C. A., & Banerji, R. B. 1986. Learning Concepts by Asking Questions. In R. S. Michalski Carbonell, J.G. and Mitchell, T.M. (Eds.), *Machine Learning: An Artificial Intelligence Approach, Vol 2*. 167-192. Los Altos, California: Morgan Kaufmann.

Sammut, C., Hurst, S., Kedzier, D., & Michie, D. 1992. Learning to Fly. In D. Sleeman & P. Edwards (Ed.), *Proceedings of the Ninth International Conference on Machine Learning*, Aberdeen: Morgan Kaufmann.

Shapiro, A. 1987. *Structured Induction in Expert Systems*. Addison-Wesley.

Srinivasan, A & Camacho, R. Numerical Reasoning in ILP. In S. Muggleton, K. Furukawa & D. Michie (Ed.), *Machine Intelligence 15*. Oxford University Press. Forthcoming