Prediction using Machine Learned Constraint Satisfaction Programs

John Charnley[⋆] jwc04@doc.ic.ac.uk
Simon Colton[⋆] sgc@doc.ic.ac.uk

[⋆] Department of Computing, Imperial College, 180 Queen's Gate, London, SW7 2AZ

1 Overview

Prediction is a well-researched area for Machine Learning applications. In these tasks, the aim is to predict the value for some unseen characteristic based upon the values of other, seen, characteristics for a given example. Machine learning has been extensively applied to these types of tasks by automating the derivation of a predictive function or a set of predictive rules. This predictive function can then be applied to new examples to estimate attribute values. Many techniques have been turned to this purpose. Inductive Logic Programming (ILP) (Muggleton (1999)), for instance, represents the attributes of given examples in first order logic and uses techniques such as inverse resolution to derive a set of first-order logic rules which can be used to logically deduce an attribute value from other attributes. Artificial Neural Networks (Lippmann (1987)) can be trained to predict attribute values. A net is pre-configured with interconnecting perceptron nodes and the weights associated with these nodes are adjusted to improve predictive accuracy over a training set. The learned artefact in this case is a neural net able to predict one attribute value. Decision Trees can predict attribute values by considering, in some stepwise order, the values of other attributes. One method of learning a decision tree is to apply the ID3 algorithm as implemented in the c4.5 decision tree learning program (Quinlan (1993)). The learned artefact is essentially a conjunction of implications which can be applied to a given example to predict the value for a single attribute. Each of these methods can be characterised by the specific ways they represent the task, the artefact they learn and how they learn it.

We consider here another type of predictive artefact together with another method of learning. The artefact we learn is a constraint satisfaction (CS) program, typically used for solving CS problems. A CS problem consists of a set of variables $\{x_1, x_2, \ldots, x_n\}$, a set of domains of values the variables can take and a set of constraints specifying which values the variables can take simultaneously. A solution to a CS problem is an assignment of values to each of the variables from their domains such that no constraints are broken. They find widespread use in science and industry. CS solvers allow users to specify CS problems in a particular syntax as CS programs and then search for solutions to the problem using a configurable search approach. We have adapted a CS program to be used for prediction. By encoding attributes as variables and machine learning appropriate constraints we obtain a predictive CS program. Given a new example for prediction, we add the values of all known attributes as variable value constraints to the predictive CS program. A CS solver can then determine allowable values, i.e. predictions, for unknown attributes. We machine learn constraints by considering combinations of attribute values, or *classifications*. By comparing how training examples fall into these classifications we can make conjectures about how different classifications relate to one-another and, through this, derive predictive relationships between the relative values of different attributes, which we then encode as constraints. In addition to a CS solver and machine learner, we use a SAT solver to filter conjectures, improving the efficiency of our CS programs.

We believe our approach has some benefits over the approaches we discussed above. Firstly, each of the above processes produces an artefact for predicting only one attribute of a given example. This means that should the user wish to predict for another attribute, they need not re-train a new predictor. By contrast, our learned CS programs can predict for any unknown attributes as it includes constraints learned with respect to all considered classifications. In addition, we believe our learned CS program may be more resistant to missing or corrupted data. This additional robustness is valuable as many real-world applications encounter instances of missing or corrupted data.

2 Approach and Implementation

Each investigation involves a number of training examples with various attributes for which values are known. The machine learning component of our system considers, using brute-force, various combinations of attributes and their possible values. It is computationally infeasible to consider all possible combinations of all possible attribute values. Consequently, we place a limit upon the maximum number of attributes and values to be considered. We refer to such combinations as *classification* as they effectively partition any given set of examples into two sets, those that exhibit those values and those that do not. We then consider the example sets to which each classification applies. Where one classification (A) is always a subset of another (B) then we conjecture that an example will always exhibit the latter if it exhibits the former, i.e. $A \rightarrow B$. This methodology replicates a subset of the functionality of the HR machine learning system (Colton (2002)) but is more efficient, as it is tailored to the specific task. The conjectures determined in this manner form the basis for the constraints we introduce into the CS program for prediction. We filter these conjectures in an attempt to find a set of maximally general conjectures. That is, we attempt to show whether any new conjecture can be deduced from existing conjectures. The most efficient method we have found to do this is to use a SAT solver, in our case Minisat (Een and Sorensson (2003)). We pass the negated conjecture together with all previous conjectures as background to the SAT solver. If the resulting SAT problem cannot be satisfied then we discard the new conjecture. As our search method is always from general to specific classifications such filtering should result in maximally general conjectures. In an effort to reduce over-fitting we have introduced a mechanism to relax the production of conjectures by considering, also, those conjectures where only a percentage of the classifications overlap. We refer to these as *near-conjectures*. We believe that including more relaxed constraints may allow for better predictive accuracy.

We produce a CS program using the method first considered for solving general first-order logic problems by expressing them as CS problems (Charnley and Colton (2006)). For each attribute we introduce one or more CS problem variables. For example, binary attributes would be expressed using one binary variable and multi-valued attributes are either scaled as binary variables or represented as finite domain variables. The conjectures are then translated into constraints using the syntax of the CS solver, in our case Minion (Gent et al. (2006)). For instance, a conjecture over three binary variables, $A \rightarrow B \wedge C$, which states that both B and C should be true whenever A is, could be translated into the constraint $reifyimply(sum([x1, x2], 2), x0)$, where $x0$, $x1$ and $x2$ are binary variables representing the values of attributes A, B and C respectively. This constraint ensures that $x1$ and $x2$ are both 1 in any solution of the CS program where $x0$ is 1. We construct a CS program by combining the definition of the problem variables and the constraints generated from the conjectures we have machine learned. This CS program can be used to make predictions by introducing, for each new case, constraints representing values of the attributes of that case. A CS solver then gives possible values for the attributes we would like predicted. In the example above, if we have a new example for which A is true then the CS solver would only ever find solutions for which B and C were both true, therefore predicting their values. Similarly, if the example shows either B or C false then the solver would predict A to also be false.

3 Experiments and Future Direction

Our preliminary studies have used the moral reasoning dataset [1]. This dataset primarily concerns the prediction of the guilt or otherwise of suspects based upon various facts of the case. The machine learned CS program we generate is able to make predictions for any attribute. This would not be possible using any of the methods we described earlier where, traditionally, we would learn a single predictor solely for guilt. If we wanted, for instance, to have decision trees able to predict for each attribute then we would need to machine learn one for each and select the appropriate tree as required. As noted earlier, we believe that the machine learned CS program exhibits robustness to missing data. If we consider a learned decision tree, it's ability to predict depends on the availability of information to choose at each branch. The decision tree is rendered useless if any of this information is missing. This would not be the case for a learned CS program, where supplementary constraints may well be able to form a prediction in the absence of some data.

We have, currently, implemented the system as described above. We have fully automated the production of constraint programs according to our method, including near conjectures, but we have yet to do detailed testing on the predictive accuracy of the approach. We envisage a key aspect of our further work will be in determining how we best select the conjectures to apply from those we have machine learned. This could encompass using metrics obtained during the machine learning phase, such as the relative sizes of the classification sets or *nearness* of conjectures. We also need to investigate how we interepret the results of the CS solver in forming a prediction. We could increase the sophistication by, for instance, considering how many constraints an example triggers to give some notion of strength to the prediction. The representation we have chosen provides at least the level of granularity required to encode a similar decision tree predictor and, we believe, more. We should, hopefully, be able to generate at least as good predictive accuracy whilst gaining the additional benefits we have outlined.

References

J Charnley and S Colton. Expressing general problems as CSPs. In *CSP Modelling Workshop at ECAI*, 2006.

S Colton. *Automated Theory Formation in Pure Mathematics.* Springer-Verlag, 2002.

N. Een and N. Sorensson. An extensible sat-solver. In *Satisfiability Workshop*, 2003.

I. P. Gent, C. Jefferson, and I. Miguel. MINION a fast, scalable, constraint solver. In *Proceedings of the 17th ECAI*, 2006.

R. P. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, pages 4–22, 1987.

S. Muggleton. Inductive Logic Programming. In *MIT Encyclopedia of the Cognitive Sciences*. MIT Press, 1999.

J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kauffman, 1993.

[1] UCI Repository of machine learning databases, www.ics.uci.edu/∼mlearn/MLRepository.html