Automated Software Engineering, 8, 131–137, 2001
© 2001 Kluwer Academic Publishers. Manufactured in The Netherlands.

Inductive Programming

PIERRE FLENER Pierre.Flener@dis.uu.se
Department of Information Science, Uppsala University, Box 513, S-751 20 Uppsala, Sweden

DEREK PARTRIDGE D.Partridge@exeter.ac.uk
Department of Computer Science, University of Exeter, Exeter, EX4 4PT, United Kingdom

What Is Inductive Programming?

The intent of this special issue was to bring together developments in *inductive programming* (Partridge, 1997) that have a direct bearing on software development, and to promote a broader usage of the term. Inductive programming, in our view, is not a return to the overly ambitious, and thus ultimately unworkable, schemes of automatically generating large software systems. Inductive programming is thus *not* just programming-by-example, nor just programming-by-demonstration, nor a panacea for software development-in-the-large. It is a mix of more subtle uses of induction to assist the software developer in a variety of ways. We describe some ingredients of the mixture below, but all need further development, and entirely new ones remain to be discovered. At this point in time, we can do no more than articulate our current understanding in the hope that this personal view will stimulate the requisite discussion and research needed to push the inductive-programming strategy forwards to become a set of practical options for the software engineer—not a replacement for mainstream methods, but a powerful adjunct in appropriate circumstances.

Scientific induction (which is not to be confused with mathematical induction) is the process of reasoning from the particular, which is known-to-be-incomplete information, to the general. Such inductive inference, just like abductive inference and analogical inference, is in general unsound. The term inductive programming may be contrasted with classical programming, which works from an assumed-to-be-complete specification to a particular implementation. Classical programming is thus a process that we might justifiably call deductive programming, which is a name that gains further credibility when we remember that the history of automatic programming is one of attempting to logically deduce correct implementations from (assumed-to-be-complete) so-called “formal specifications”.[1] Recall that deductive inference is always sound.

In inductive programming, we seek algorithms that survey known-to-be-incomplete information, say a set of input-output examples, and generate information *pertinent to* the construction of a generalised computational system for which these input-output examples are a representative sample. The information so extracted might thus be no more than problem features and decision logic for specification enhancement, or it might be a self-contained system module, but it does not have to be a complete software system in order to be useful.

Another definition of inductive programming, namely as programming-by-example, that is the extrapolation of "the correct" function from a subset of its input-output examples, is too restrictive in our opinion. It even seems ill-posed, as there never is a single correct function in such a setting, so that one has no guarantee that the obtained result is correct in any sense. This common prejudice is overly strong, and can be weakened by the following argument. There is not always a single correct function embedded in specifications suitable for deductive programming either. Indeed, such specifications *can* also be ambiguous (embed more than one function) or even internally inconsistent (embed no function at all).[2] An ambiguous or internally inconsistent specification is subjected to inspection and revision. If internal inconsistency persists, then it may just be because the problem has no solution. If ambiguity persists, then it may just be because several solutions are equally good. The hard rules of Formal Methods advocates, such as avoiding ambiguity and internal inconsistency, thus lead to absurdities: one cannot decide in advance what is good in *all* possible situations. (Known-to-be-)incomplete specifications can thus not be attacked for their (definite) ambiguity, due to their incomplete nature, because ambiguity and internal inconsistency *can* be desirable even for (assumed-to-be-)complete specifications.[3] The fundamental computer science notion of 'specification' is surprisingly complex (Partridge and Galton, 1995); this is a fact that goes largely unappreciated, to judge by the common and cavalier usage of this word.

Nothing thus says that no useful information nor valuable computational system can be forthcoming from applications of induction algorithms to data. Because the induction processes we contemplate are algorithmically specified, the information generated—either guidance for problem specification enhancement or an executable module—is generated automatically. This is the essence of what induction algorithms have to contribute to the field of automated software engineering (Flener and Popelínský, 1994; Partridge, 1997). It has been noted (Le Charlier and Flener, 1998) that the addition of input-output examples to even complete specifications may be beneficial (to human programmers), namely as a means of clarification. (The common fear that such examples may be inconsistent with the rest of the specification is unreasonable, as absence of internal inconsistency is not a guarantee of external consistency, whereas internal inconsistency is an undeniable indication that the considered problem either is ill-posed or has no solution.) What inductive programming aims at is the exploration of all synergies between complete and incomplete specifications.

Inductive Programming in Action

The first step in idealised software engineering is to abstract a precise (and assumed-to-be-complete) specification, which is then taken as the foundation for all subsequent development, such as coding and verification. However, complex specifications typically contain errors and approximations that lead to errors in the eventual software, errors that are not easily detected or eliminated before the software is subjected to operational testing.

However, many programming problems are manifest as sets of data values, namely inputs and corresponding outputs, divided into positive examples and negative examples, which can also be seen as a known-to-be-incomplete specification. Inductive programming techniques work from such data instances to the implementation without going through an assumed-to-be-complete specification. They thus offer the software engineer a means to avoid or rectify system errors that are due to specification faults, and perhaps even to circumvent the need for a complete specification, i.e., *some* system modules may be inductively generated from data where accurate specification proves difficult. Induction-based processing of problem data (i.e., data mining) may even be used to check and correct features of a potentially or assumed-to-be complete specification. Indeed, software engineering experience shows that the notion of 'complete specification' is nothing but a chimera.

Inductive software development will be particularly germane, if not essential, for (parts of) complex data-defined problems. These will arise in a data-rich domain and address very complex aspects of the world, such as the human body, complicated manufacturing processes, and complex dynamical situations. Human face recognition is one such problem. It is a priori plausible that a computerised face recogniser is possible. We are all good face recognisers, but we do not know how we do it. We may be able to specify what is desired but not at the level of detail required for classical algorithm design and subsequent implementation. The details could, however, be provided in the form of a set of input-output examples, and from such a set of instances an inductive programming technology could provide a generalised face-recognition system.

Inductive programming is not, however, trouble-free. There are difficult issues of 'understanding' automatically induced procedures so that implementation performance can be characterised. There are issues of data pre-processing to facilitate optimal application of a given inductive technique and to obtain an implementation with certain desired characteristics.

The 'understanding' issue is particularly important when the induction technology is that of distributed neural computing. It may be possible to train, say, a multilayer perceptron using the backpropagation algorithm to produce a good prediction module in a situation where classical programming has been unsuccessful. After training, no further weight updating is permitted, and then the trained neural network implements a deterministic computation. This module is then clear evidence of a systematic algorithmic solution to some hitherto intractable subfunction, but inspection of the trained network is unlikely to shed much light on *how*, in classical computational terms, this particular subfunction can be characterised (Partridge, 2000).

As a specific example, we are collaborating with National Air-Traffic Services of the UK to improve the performance of their Short-Term Conflict Alert (STCA) software system. The STCA system was designed to alert air-traffic controllers whenever two aircraft are likely to breach proximity restrictions. It must never miss a true alarm situation, and consequently it produces large numbers of false alarms. A common false-alarm situation occurs when one plane is ascending (or descending) towards a flight level where it can (and invariably does) safely level off, but the linear extrapolation of its flight path (before level-off) leads to a false alarm. The existence of a subfunction that could predict level-offs could cancel many false alarms, but the STCA system contains no such subfunction because no one knows how to specify it. However, given many examples of flight paths (and other objective data such as size of aircraft) for which planes level off and do not level off, it is possible to train a neural network to predict level-offs (not perfectly but quite well). But inspection of the trained networks yields no information to assist in the formulation of a classical specification for level-off prediction, and hence we are no closer to a classically programmed level-off prediction module to add to the STCA system. But we do then know which of the available features (such as speed, deceleration, and size) are important for predicting level-off, and use of automatic decision-tree induction algorithms is expected to reveal useful decision logic associated with those features.

A major source of incipient technologies to develop for inductive programming is the field of Artificial Intelligence (AI). This is because the problems of AI have long been acknowledged as unspecifiable with the precision and completeness typically demanded by software engineers. Michie (1991), for example, makes explicit connection between “machine learning” technologies and software maintenance. A recent collection entitled “Computational Intelligence in Software Engineering” (Pedrycz and Peters, 1998) contains a number of inductive technologies applied to various aspects of software development. The efficient solving of constraint satisfaction problems (Tsang, 1993) is an important sub-field of AI, because of the ubiquity of these often NP-complete problems; to cope with the instance sensitivity of heuristics, recent industry-strength solver generators (Ellman et al., 1998; Minton, 1996) also use training instances, and thus feature a productive mix of inductive and deductive inference.

In some inductive programming settings, mere input-output examples may be too weak specification information, either because the search space of induction then becomes too large, or because the specifier knows a few more things. To overcome the many negative results on inductive inferability from examples alone, many researchers have proposed additional specification information, such as oracles, properties, and background knowledge. Indeed, the induction algorithm may construct its own additional examples and submit them to an oracle (usually the specifier) for classification as positive or negative. Or the specifier may wish to impart that the sought function is believed to satisfy a certain property, such as transitivity. This may be useful for communicating known *intrinsic* information: for instance, the $\leq$ relation is intrinsic to number-list sorting, as it appears in *all* sorting programs, but a partitioning function is extrinsic to it, as it *only* appears in quicksort programs.[4] Finally, background knowledge may increase the power of induction by making reusable programs available.

Other information is often added to reduce the search space. For instance, declarative bias is used to control the search and language during induction: a deterministic program may be preferred, or a program that fits a certain schema[5] (Flener, 1995; Flener, 1997). A note of caution is necessary about the addition of hints at what relations from the background knowledge may or should be used during the induction. Indeed, hinting at the *exactly* necessary background knowledge in a problem-*specific* way amounts to “specifying the solution” (which is an oxymoron), and thus misses the usual objective of specifying the problem. Especially in Inductive Logic Programming (ILP), some systems require such use in a *teacher setting* (as opposed to a *specifier setting*) (Flener and Yılmaz, 1999) and are thus essentially deductive synthesisers masquerading as inductive ones. (Their search spaces are intractable otherwise.) Of course, there are scenarios where the specifier feels that some specific background programs may or do have to be reused, but does not know exactly how to combine them to achieve the desired computation, so prefers to hand over to an inductive programming tool to figure it out. We then get a hybrid approach between inductive programming and programming-by-demonstration.

The Future of Inductive Programming?

Prediction is always difficult (as someone said), especially of the future, but it may be worth a try. As stated above, the field of AI is a rich source of potential technologies for inductive programming. But AI technologies are notoriously fragile and often come with no formal underpinning, which can make a success no more than a pleasant surprise. In inductive programming, we require robustness and we require that the scope and limitations of a candidate technology can be circumscribed, so that applicability is not based on chance and that success comes with some assurance of reliability. Part of the future can thus be seen as development of inductive technologies that AI has demonstrated. A further aspect of this development must be scalability: techniques that work on small demonstration examples must also succeed on realistically large examples.

Work in Neural Computing (Bishop, 1995), which associates an “error bar” with each neural-net computation, offers promise of the necessary assurances for a practical software system. By modelling the training data and the inductive algorithm used to develop a computational module, accurate confidence measures can be associated with every new computation. Much work remains to be done on accurate and efficient data modelling, but the potential reward, namely an accurate ‘confidence’ value associated with each computed result, will be valuable information for the software engineer. A similar strand of research is concentrating on the further development of Bayesian networks, which hold the promise of illuminating logical decision structures as well as accurate confidence measures (Jensen, 1996).

Overview of the Accepted Papers

The three papers that have been selected bring an interesting variety of applications of inductive technologies to bear on the central problem of software development.

The first paper—by Hernández-Orallo and Ramírez-Quintana—tackles the issue of software specification from the viewpoint that there will always be scope for improvement. In their view, software development methodologies have an unhealthy tendency to treat the specification as some given foundation upon which everything else is built and with respect to which crucial notions like implementation correctness are defined. They propose a new model for software development that is inspired by the ideas of incremental learning emanating from the Machine Learning subfield of Artificial Intelligence. Inductive technology is used to move the specification into the evolutionary loop of incremental modification alongside design and implementation, which are the more traditional elements of an evolutionary software development paradigm.

The second paper—by Hamfelt, Nilsson, and Oldager—presents a new approach to inductive synthesis of logic programs. They contrast their scheme with that of the traditional ones, which attempt to generalise programs from a set of examples. The proposed method works through examples but uses problem-decomposition and problem-reduction principles to yield a practically viable alternative, provided that the programmer can supply appropriate auxiliary predicates to maintain the overall search space within reasonable bounds.

The third contribution—by McCluskey and West—again addresses the initial phases of software development, namely the specification and management of requirements. Their application domain, namely air-traffic management over the North Atlantic, with its demanding safety requirements puts heavy emphasis upon the accuracy of a requirements domain theory. The central concern is the refinement and improvement of this domain theory so that it better fits the intentions of air-traffic control officers as reflected in operational examples. They present a novel theory refinement algorithm that uses the logs of expert decisions and so permits validation of the requirements.

Acknowledgments

We express our gratitude to all the authors who submitted a paper to this special issue, as well as to all the anonymous referees who accepted to participate in the review process. Finally, we would like to thank the Editorial Board members who commented on a draft of this introduction, as well as Bashar Nuseibeh, the Editor-in-Chief of this journal, for his continuous support.

Notes

1. See (Le Charlier and Flener, 1998) for an argument why “formal specifications” cannot really be considered to be actual specifications, in the classical engineering sense.
2. Note that internal inconsistency is *impossible* with input-output examples.
3. Similarly, both kinds of specification cannot be attacked for their potential external inconsistency with respect to the intentions or real world.
4. Our passive SYNAPSE (Flener, 1995) and interactive DIALOGS (Flener, 1997) inductive program synthesisers demonstrate the benefits of having such properties in addition to examples.
5. See (Flener and Yılmaz, 1999) for an overview of schema-guided inductive synthesisers of recursive logic programs.

References

Bishop, C.M. 1995. *Neural Networks for Pattern Recognition*. Clarendon Press.

Ellman, T., Keane, J., Banerjee, A., and Armhold, G. 1998. A transformation system for interactive reformulation of design optimization strategies. *Research in Engineering Design*, 10(1):30–61.

Flener, P. 1995. *Logic Program Synthesis from Incomplete Information*. Kluwer Academic Publishers.

Flener, P. 1997. Inductive logic program synthesis with DIALOGS. In S. Muggleton (Ed.), *Proc. of ILP’96*, pp. 175–198. LNAI 1314, Springer-Verlag.

Flener, P. and Popelínský, L. 1994. On the use of inductive reasoning in program synthesis: Prejudice and prospects. In L. Fribourg and F. Turini (Eds.), *Proc. of LOPSTR/META’94*, pp. 69–87. LNCS 883, Springer-Verlag.

Flener, P. and Yılmaz, S. 1999. Inductive synthesis of recursive logic programs: Achievements and prospects. *Journal of Logic Programming*, 41(2–3):141–195.

Jensen, F. 1996. *An Introduction to Bayesian Networks*. UCL Press.

Le Charlier, B. and Flener, P. 1998. Specifications are necessarily informal, or: Some more myths of formal methods. *Journal of Systems and Software*, 40(3):275–296.

Michie, D. 1991. Methodologies from machine learning in data analysis and software. *The Computer Journal*, 34(6):559–565.

Minton, S. 1996. Automatically configuring constraint satisfaction programs: A case study. *Constraints*, 1(1–2):7–43.

Partridge, D. 2000. Non-programmed computation. *Comm. of the ACM* 43(11es):293–302.

Partridge, D. 1997. The case for inductive programming. *IEEE Computer*, 30(1):36–41.

Partridge, D. and Galton, A. 1995. The specification of 'specification.' *Minds and Machines*, 5(2):243–255.

Pedrycz, W. and Peters, J.F. (Eds.), 1998. *Computational Intelligence in Software Engineering*. World Scientific.

Tsang, E.P.K. 1993. *Foundations of Constraint Satisfaction*. Academic Press.