Chapter IV

ILP Applications to Software Engineering

Daniele Gunetti, Università degli Studi di Torino, Italy

Abstract

Though inductive logic programming (ILP for short) should mean the “induction of logic programs”, most research and applications of this area are only loosely related to logic programming. In fact, the automatic synthesis of “true” logic programs is a difficult task, since it cannot be done without a lot of information on the sought programs, and without the ability to describe in a simple way well-restricted searching spaces. In this chapter, we argue that, if such knowledge is available, inductive logic programming can be used as a valid tool for software engineering, and we propose an integrated framework for the development, maintenance, reuse, testing, and debugging of logic programs.

Introduction to Inductive Logic Programming

Inductive logic programming is normally defined as the research area at the intersection of machine learning and logic programming. An ILP system receives as input some examples of the program’s input/output behavior (plus other kinds of information about the desired
program) and must produce a logic program that behaves as expected on the given examples, or at least on a high percentage of them. Typically, the obtained programs will then be used on new examples, not given to the ILP system during the learning phase. From the point of view of a computer scientist, this is just a form of program synthesis from examples, but the view presented in this chapter emphasizes the fact that examples are absolutely not the only input to practical ILP methods. Another important source of information comes from a priori knowledge about the target program, including partially-developed software components, and properties of the needed sub-procedures, such as the number and the type of the arguments and of the returned values. To stress this observation, and also the fact that practical tools will have to be embedded in more complex environments, this chapter looks at ILP as logic program development with the help of examples, and not just automatic programming from examples.

In other words, ILP may be seen as a modern approach to automatic programming, and as a software engineering tool for logic program development. The choice of Prolog as a programming language for the learned programs is a good one. This was noticed early on by Ehud Shapiro when he developed the first automatic logic programming method, called MIS, during the early 1980s (Shapiro, 1983). Prolog’s simple syntax identifies procedures with clauses, which can be expanded simply by adding literals to the antecedents. Examples are just predicates applied to constant arguments, and no complicated example specification language is needed. On the other hand, one is not limited to providing input/output examples for the top-level procedure only. Examples of predicates corresponding to sub-procedures will follow the same simple syntax. For instance, for learning multiplication, one may provide:

\[ \text{times}(2,3,6), \text{times}(1,3,3), \text{add}(3,3,6). \]

Thus, we may give the ILP system information for the main computation (times), and also for the sub-calls that are needed (add). Finally, nondeterminism makes the code more concise and substantially shorter than in imperative languages; as a consequence, learning a program may require fewer steps.

Examples are not the only source of information: previous knowledge about parts or properties of the target program may be used by the ILP system. In other words, we consider a scenario where a competent programmer, with a diverse set of programming tools, will also consider examples and inductive inference methods as an option. Assistance during software development is probably the most natural, but certainly not the only, software engineering application of ILP. First of all, it must be noted that the kind of software development that is achieved through ILP techniques includes a simple form of maintenance. After an initial version of a program has been coded or generated automatically, it may be used on specific cases that arise in the end application. If an error is detected, the corresponding and corrected input/output example may be added to the example base, and the program may be corrected automatically, or a new program may be learned from scratch. ILP techniques have been also used for test-case generation (Bergadano & Gunetti, 1996b) “reversing” the induction process: we go from programs to relevant, input values instead of learning a program from some of its input/output examples. If examples are seen as a specification
Improving Ontology Alignment through Genetic Algorithms
www.igi-global.com/chapter/improving-ontology-alignment-through-genetic/43155?camid=4v1a