15

Automating the Design of Complex Systems

Summary. This chapter discusses the issue of whether it is possible to automate the design of rather complex workflows needed when addressing more complex data science tasks. The focus here is on symbolic approaches, which continue to be relevant. The chapter starts by discussing some more complex operators, including, for instance, conditional operators and operators used in iterative processing. Next, we discuss the issue of introduction of new concepts and the changes of granularity that can be achieved as a result. We review various approaches explored in the past, such as constructive induction, propositionalization, reformulation of rules, among others, but also draw attention to some new advances, such as feature construction in deep NNs. It is foreseeable that in the future both symbolic and subsymbolic approaches will coexist in systems exhibiting a kind of functional symbiosis. There are tasks that cannot be learned in one go, but rather require a sub-division into subtasks, a plan for learning the constituents, and joining the parts together. Some of these subtasks may be interdependent. Some tasks may require an iterative process in the process of learning. This chapter discusses various examples that can stimulate both further research and some practical solutions in this rather challenging area.

15.1 Introduction

The aim of this chapter is to discuss the issue of automating the design of more complex systems than the ones discussed in previous chapters. The aim is to present various techniques which have proved useful in the past research, as they can be regarded as fundamental architectural building blocks of both current and future intelligent systems.

Complex systems can be divided into two major groups. One includes systems that learn from relatively small sets of examples, while the others use large sets of examples (big data) in the learning process. Let us analyze each group in more detail.

As we have mentioned, the systems in the first group typically use modest numbers of examples. Some systems of this kind were discussed in the previous chapter on automating data science and, in particular, data wrangling. These systems often learn from human-provided input, in the form of input–output pairs or even individual steps showing how a particular problem is to be solved. They often exploit symbolic formulations (e.g., ILP) and may exploit background knowledge. This chapter contains material that is closely related to these systems.

P. Brazdil et al., *Metalearning*, Cognitive Technologies,
https://doi.org/10.1007/978-3-030-67024-5_15

The systems in the second group typically use large sets of examples (big data) in the learning process. In this category we find various systems that learn from big data and data streams (see Chapter 11 for details). Many current systems developed for complex tasks, such as vision or machine translation, do not use symbolic methods any more, but deep neural networks (DNNs), embeddings, etc. It is interesting to note that processing the input and constructing embeddings can be seen as a process of capturing the essential information. This knowledge is useful when dealing with new problems. So a question arises of whether symbolic approaches that are discussed in this chapter are at all relevant in the light of current developments.

In our view there are various reasons why they are relevant. One is to draw attention to various common principles that have proved useful in both symbolic and subsymbolic learning. The notion of, for instance, *constructive induction* discussed in Section 15.3 can be related to the introduction of new nodes higher up in the DNN.

The second reason is related to the issue of *explainability*. Many people find it important nowadays that the current systems are able to explain how they arrived at a particular recommendation or decision. As it is difficult to do this in some systems, including NNs, a symbolic model can be maintained to model the more complex system. This way, it is possible to provide the required explanation. If the symbolic model of the underlying complex process is *faithful* in the process of modeling, this solution is more satisfactory than some rather complex explanation that is difficult to comprehend. Explainability contributes towards what some people now call *trusted AI*.

The third reason is that humans use both subsymbolic and symbolic reasoning, and presumably this has a good motive. Possibly it is related to the human capacity called *creativity*. Currently, we can only postulate what this involves, but we believe that abstract concepts play an important role in that process.

It is foreseeable that future systems would benefit from a stronger interplay between the subsymbolic learning (e.g., embeddings) and symbolic counterpart (e.g., ontologies). So, subsymbolic and symbolic approaches could co-exist in a kind of functional symbiosis. Therefore, in our view, an overview of methods and techniques that have proved useful in the process of learning complex systems, albeit symbolic, continues to be relevant.

Overview of this chapter

More complex tasks may require plans with conditional operators or iterative processing. This issue is briefly reviewed in Section 15.2. Introduction of new concepts is addressed in Section 15.3. There are tasks that cannot be learned in one go, but rather require a sub-division into sub-tasks and then formulating a plan for learning the constituents and joining the parts together. This methodology is discussed in Section 15.4. Some tasks require an iterative process in the process of learning. More details on this can be found in Section 15.5. There are problems whose tasks are interdependent. One such problem is analyzed in Section 15.6.

15.2 Exploiting a Richer Set of Operators

Workflows, sometimes also called pipelines, and their more general counterparts, networks, can be seen as plans to be executed. As sometimes it is necessary to verify certain

conditions before advancing with the execution, it is necessary to enrich the set of operators to be able to deal with such situations. The constructs in various programming languages suggest the main classes that need to be considered:

- Procedures
- Conditional operators
- Repeat operators

The first type, *procedures*, is related to abstract operators discussed in Chapter 7. Conditional operators have been used already in the early days of planning (Dean and Boddy, 1988). This is useful in many situations. Consider, for instance, a robot plan to go from one location to another. The plan could state: *If no obstacle is in the way, go straight, otherwise initiate an "avoidance plan".* Repeat operators are needed because, as the name suggests, some operations need to be repeated. Kietz et al. (2012), for instance, used this operator in their planning system to design a KDD workflow. It was used to repeat the operation of preprocessing each column of a given data table.

15.3 Changing the Granularity by Introducing New Concepts

In Chapter 7 we discussed methods that would come up with the best possible workflow for a given task. We note that the basic building blocks were given. This included both the descriptors of the data (features, attributes) and also the descriptions of given operators (e.g., a particular preprocessing operation or application of a particular classifier). A good question that we wish to address here is: where do concepts come from?

Before trying to answer this question, let us clarify what we mean by "concepts" here. Basically, one set of concepts describes *states* and another *events* or its special subcategories, namely *processes* and *actions*, that relate one state to another.

There are two possible ways these can be introduced in a system: One is by introducing them from the exterior. The other is by some autonomous process controlled by the agent. More details about the two alternatives follow.

Introducing new concepts from external sources

Researchers who study human learning in children note that many concepts are acquired from interactions with their progenitors. The new concepts undoubtedly shape the process of further learning.

Some AI systems use also external sources. One common source is the internet. Many information extraction tasks benefit by accessing Wikipedia pages. One famous system in this area is Watson, which is a question-answering system capable of answering questions posed in natural language (Ferrucci et al., 2013). This system was initially developed to answer questions on the quiz show Jeopardy!.

Although many systems exist that interact with the exterior, to the best of our knowledge not much work has been done up to now in the area that would combine the introduction of new concepts from the exterior with their introduction relying on autonomous methods.

Introducing new concepts autonomously

When talking about the introduction of concepts, we need to be aware that this is often a continued process, as De Raedt (2008) (Chapter 7) pointed out. Introduced concepts may undergo various phases of revision later. In the following subsections we provide pointers to some past work in this area.

15.3.1 Defining new concepts by clustering

Clustering (or cluster analysis) is a branch of machine learning that groups the data items that share certain attributes (features). It belongs to the so-called unsupervised learning methods, as the data points (examples) have not been classified, labeled, classified, or categorized. Instead of responding to the classes, clustering identifies commonalities in the data, represented by attributes (features). So, the clusters (groups) obtained by this process can be regarded as new concepts. The system can give internal names to these concepts (e.g., concept#12), but whenever it is necessary to communicate with other agents, including humans, commonly established names need to be adopted.

15.3.2 Constructive induction

Michalski is known for introducing the term *constructive induction*. Diettrich and Michalski (1983) have defined constructive induction as an induction process that changes the description space, that is, a process that produces new descriptors (e.g., features) that were not present in the input data.

15.3.3 Reformulation of theories consisting of rules

Reformulating theories by specialization

System ELM (Brazdil, 1981, 1984) could learn to solve arithmetic and algebraic problems. The given rules (clauses) were reformulated as a result of learning. Many operations involved could be described as specializations.

The set of problems given to the systems involved a certain order. Simpler problems preceded the more complex ones, as often the solutions of simpler problems could be reused later. These included some problems from the area of arithmetic, such as $4+(1+2)=X1$, and algebra, namely equations with one unknown such as, $(2+1)+X1=5$. In the learning phase the correct result was also given, together with some intermediate correct steps leading to the solution, referred to as a *trace*.

The given rules expressed valid operations in arithmetic and algebra, similar to Peano axioms. This includes a successor function and its inverse, i.e., a predecessor function. Besides, rules expressing associativity and commutativity were also given. Some rules also enabled the introduction of a new variable. For instance, the goal $X1+X2=X3$ could be transformed into $X1=X4\ \&\ X4+X2=X3$.

The system searched for a solution by expanding the search space. The given trace provided a useful help in the search, but if enough computing power were given, the system could reach the solution even without it. After the solution has been reached, the aim of the system was to reformulate the given rules, so as to avoid the wrong steps. This was done in two ways. The first one involved imposing some ordering constraints on the existing rules in the form of explicit commands, such as $r_i > r_j$, expressing that

rule r_i should be given preference to r_j. All rules followed in effect a partial order. So, the system employed certain aspects of *preference learning* (Fürnkranz and Hüllermeier, 2003; Fürnkranz and Hüllermeier, 2011).

Cycles were not permitted. To avoid them, the system used the second method, namely constraining the applicability of the rules. The constraints included, for instance, $int(X_i)$, expressing that X_i is an integer, $var(X_i)$, expressing that X_i is a variable, and $X_i :=: X_j$ expressing that X_i and X_j can be unified. One of the main ideas of this work was to analyze the solution traces and gather examples of situations where the given rule was used correctly, referred to as *selection contexts*, together with examples where the rule was applied incorrectly (*rejection contexts*).

The aim of the system was to find the most specific term that would be able to discriminate between the two. This system bears some similarities to system LEX (Mitchell, 1977, 1982), which was conceived around the same time. The concepts in ELM are not organized in a lattice according to their generality as in LEX, which is well known for the introduction of version spaces.

Folding and unfolding

Burstall and Darlington (1977) considered the issue of reformulating programs. They defined various operations (unfolding, folding) that enabled them to do this. Here we review briefly the operation of *folding*:

If $E \leftarrow E'$ and $F \leftarrow F'$ are equations and there is some occurrence in F' of an instance of E', replace it by the corresponding instance of E, obtaining F"; then add the equation $F \leftarrow F$".

The operation of *folding* is defined in a similar way, but instead of searching for an occurrence of an instance of E' in F', it searches for an instance of E.

Although the definition of $F \leftarrow F$" is new, we would be reluctant to regard it as a "new concept". For this, the concept would have to satisfy other criteria, such as having wide applicability across several different tasks.

Absorption

The operation of *absorption* defined by Sammut (1981) has a similar aim to *folding*. This operator was exploited later in a propositional learning system called DUCE (Muggleton, 1987) and its first-order upgrade CIGOL (Muggleton and Buntine, 1988) together with three other operators, namely identification, intra-construction, and inter-construction. Let us analyze one example that illustrates the use of absorption, which was adapted from De Raedt (2008) (Chapter 7). For instance, given the theory consisting of clauses

$$primate \leftarrow twoLegs, noWings$$
$$man \leftarrow twoLegs, noWings, notHairy, noTail$$

the application of the *absorption* operator transforms the second clause into

$$man \leftarrow primate, notHairy, noTail$$

We note that this operator changes the clause body in one of the clauses but does not introduce a new concept (a new clause with a new symbol in the clause head).

15.3.4 Introduction of new concepts expressed as rules

Muggleton and Buntine (1988) have observed that several rules containing a common conjunction of premises can be rewritten by replacing the common group by a new concept. This operation is captured by so-called *inter-construction* (Muggleton and Buntine, 1988; Muggleton, 1991). The operator was defined for both the propositional and first-order setting. Here we review briefly one propositional example adapted from (De Raedt, 2008) (Ch.7). Suppose the input theory is

$$man \leftarrow twoLegs, noWings, notHairy, noTail$$
$$gorilla \leftarrow twoLegs, noWings, Hairy, noTail, black$$

The combination of $twoLegs, noWings$ that appears in both definitions of man and $gorilla$ can be singled out and substituted by a new concept with an internal name (e.g., p). The system can ask the user to suggest an appropriate name, and let us assume that this name is $primate$. So the revised theory will be

$$primate \leftarrow twoLegs, noWings$$
$$man \leftarrow primate, notHairy, noTail$$
$$gorilla \leftarrow primate, Hairy, noTail, black$$

The inter-construction operator could be seen as a mechanism for constructing new features in the sense intended by Michalski when discussing constructive induction.

Although the examples presented here include concepts like “man”, “gorilla”, etc., there is no reason why the operations discussed here, including e.g., inter-construction, could not be applied also to sequences of operators.

15.3.5 Propositionalisation

The term *propositionalization* was introduced in first-order learning/inductive logic programming (ILP). The aim was to make ILP learning more effective. This technique was exploited in LINUS (Lavrač et al., 1991; Lavrač and Džeroski, 1994). This system employs propositional learners in a more expressive logic programming framework. The algorithm solves ILP problems in the following three steps:

- Transform the learning problem from relational to attribute-value (propositional) form
- Solve the transformed problem by an attribute-value (propositional) learner
- Transform the learned concept back into relational form

Propositionalization is probably quite close in spirit to what Michalski intended, since this does result in new features, which are then used to represent the given data instances.

15.3.6 Automatic feature construction in deep NNs

Deep neural networks have become popular in recent years, mainly due to various successful applications in image and speech recognition. Each layer of nodes is involved in training a distinct set of features while exploiting the output of previous layers. The higher layers typically contain more complex features, as they aggregate and recombine features from the previous layer. The features can be exported from the NN and reused in other applications.

We note that this mechanism can be compared to the method of reusing solutions of subtasks in further learning, discussed in Section 15.4. It indicates that the mechanism is quite general and can be of use in various, rather different settings.

15.3.7 Reusing new concepts to redefine ontologies

Various approaches that enable the definition of new concepts that were discussed in this chapter can be applied also to operator sequences. The operator *inter-construction* of Muggleton and Buntine (1988) can be used to identify common subsequences of operators and replace them by a new *abstract operator*. If this process is continued, it is possible to envisage that in the end we can obtain an ontology of abstract and concrete operators that represent an important part of many current HTN planning systems.

15.4 Reusing New Concepts in Further Learning

The idea of reusing the solutions of subtasks in further learning permeates the whole area of ML. Let us consider again Figure 7.2. It suggests that a "DM operation" can be accomplished by carrying out a *preprocessing operation*, followed by *model-building operation* and *post-processing operation*. The figure gives more details at lower levels. For instance, it determines what kind of preprocessing operations can be considered.

As we have pointed out in Chapter 7, the ontology captures certain declarative and procedural bias, which is useful for constructing solutions to new tasks.

Example: using acquired skills in learning more complex behavior

The notion that solutions of sub-problems can be reused when solving other, more complex problems was discussed by various other authors. Stone (2000), for instance, discusses a strategy called *layered learning* which starts with learning basic skills. The acquired skills can be considered as primitive actions. After these have been learned, the system proceeds to learn more complex actions.

A similar position was also put forward by Lake et al. (2017), who expressed the view that one should reuse approaches that worked well before. The consequence of this is that, with every new skill learned, learning new skills becomes easier.

Let us analyze an example of simulated soccer discussed by Stone (2000) concerned with passing a ball to another agent, referred to as the *receiving agent*, who must intercept the ball. Intercepting the ball represents a skill that has been acquired before. The learned ability can be reused.

As there are several players in the field, the passer must decide to whom the ball should be passed. Here, the identifier of the player can be regarded as the parameter that needs to be learned.

The passer announces his intention to pass, and the teammates reply when they are ready to receive. The passer chooses a receiver randomly during training and announces to whom it is passing. The receiver and four nearest opponents attempt to get the ball using the learned interception skill. The training example is classified as a success if the receiver manages to advance the ball towards the opponent's goal, and a failure otherwise. The authors point out that the performance can be increased further if the passer is given other options, besides just passing the ball. That is, if there are no conditions to pass, the agent may decide to continue to dribble the ball.

15.5 Iterative Learning

The notion of using the output of learning in further learning seems quite general. In this subsection we discuss the problem of learning recursive definitions that follows this idea. This is an important issue, as many concepts are best represented this way.

Various approaches have been proposed in the literature. Burstall and Darlington (1977) have proposed a system for reformulating recursive definitions for a given task. Model inference system (MIS) of Shapiro (1996) conducted a general-to-specific search for clauses covering examples and was able to synthesize recursive definitions. Quite a detailed account of this system is presented by De Raedt (2008). Various other systems followed, including, for instance, RTL (Baroglio et al., 1992), CRUSTACEAN (Aha et al., 1994), and SKILit (Jorge and Brazdil, 1996; Jorge, 1998), among others.

Here we focus on one method mentioned above (SKILit) that is able to synthesize the correct definitions with a relatively high probability on the basis of a few randomly chosen examples. However, in principle, other systems (e.g., ALEPH) could have been used too. The method does not assume any a priori knowledge of the solution except the domain-specific knowledge in the form of a clause-structure grammar similar to grammars discussed in Section 8.3.

Suppose we are interested in synthesizing an algorithm for processing structured objects, such as lists. Before doing this, we would want to capture (and exploit) the following idea: *If you want to process a structured object using some procedure (P), decompose it into parts, then invoke the same procedure recursively, and then join the partial solutions.*

We can conceive a grammar to capture this. In this domain, the general structure of a clause (possibly recursive) can be specified as follows:

$$body(P) \rightarrow decomp, test, recursion(P), comp$$

The grammar specifies that, in this domain, four different groups of literals are needed. The first group decomposes a list into parts. The second one carries out a test whose outcome is either true or false. The third group enables the introduction of the recursive call. The fourth group consists of composition literals that enable us to construct the output from parts.[1] Symbol P carries the name of the predicate of the head literal whose definition we wish to synthesize. If, for instance, the aim is to synthesize *insertion sort*, this symbol would represent *isort*.

Let us examine just the definition of the recursion group, but skip all other details, as these can be found in the original paper.

$$recursion(P) \rightarrow recursive\_lit(P).$$
$$recursion(P) \rightarrow recursive\_lit(P), recursion(P).$$
$$recursive\_lit(P) \rightarrow [P].$$

We note that the recursive group contains a link to itself, as the name suggests. So, a question arises about how to adapt the method that follows a bottom-up approach, discussed in Section 15.4, namely learning sub-concepts before learning higher-level concepts. The objective of this subsection is to clarify just this point.

Wherever a link exists to itself in the grammar or the corresponding graph, the solution consists of repeating the learning cycle more than once. In each step a tentative theory, T_i, is produced and subsequently reused as background knowledge in the next cycle of the induction process (see Figure 15.1). If the stopping criterion is satisfied, the process terminates.

[1] The reader can compare this with the grammar used in the design of KDD workflows (Section 7.2).

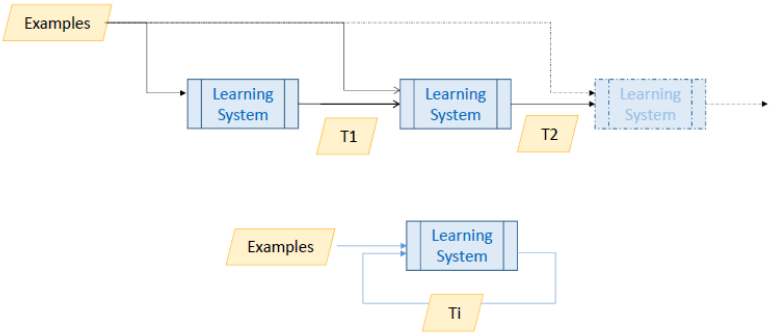

Fig. 15.1: Iterative learning

Example: learning the definition of *insertion sort*

Let us examine how the method of iterative induction helped synthesizing the definition of insertion sort (*isort/2*) (Jorge and Brazdil, 1996). First, let us see which predicates were made available to the system as domain-specific metaknowledge.

Initially, these include some basic list handling predicates, including *split/4* and *concat/3*, among others. Some properties generated in the first step of iterative induction, corresponding to theory $T1$, are shown below:

$$isort([A,B],[A,B]) \leftarrow A < B.$$
$$isort([A,B],[B,A]) \leftarrow B < A$$

The properties induced by the system represent a correct (but specific) program for sorting two-element lists. The first clause establishes that, if the list is already sorted (i.e., the first element is smaller than the second one), the order should be maintained. The second clause takes care of swapping the two elements when necessary. It is easy to see that both properties generalize many concrete examples. Thanks to these discovered properties, the system was often able to generate the correct definition in the next step.

In another experiment (Jorge, 1998), the predicate $insertb(A,B,C)$ was added to the background knowledge. This predicate inserts element A into list B and generates C as a result. In this run, the property

$$isort([A,B],C) \leftarrow insertb(A,[B],C)$$

was generated as theory $T2$. In the next cycle the correct theory $T3$ was generated:

$$isort([],[]).$$
$$isort([A|B],C) \leftarrow isort(B,D), insertb(A,D,C).$$

The method described was able to synthesize the correct definitions (with relatively high probability) of various predicates on the basis of a few examples. Overall, the accuracies were of the order of 90% or more when only five positive examples were given.

15.6 Learning to Solve Interdependent Tasks

There are situations where we need to control two or more processes in a coordinated manner. Let us consider some situations when this occurs. Consider, for instance, how we put a car into motion (assuming that the car does not use automatic gear change). Having started the motor, we need to keep releasing the clutch and pressing the accelerator. Both actions need to be carried out in a coordinated manner, as otherwise the car would stall.

A similar situation occurs when controlling a simulated plane, as some maneuvres involve more than one control. For instance, if the aim is to turn left, it is necessary to determine whether to adjust not only the *elevators*, but also the *ailerons* and the *thrust*.

Let us analyze a situation discussed by Camacho and Brazdil (2001) which involves two controls, control i and control j, which affect one another. The general situation is illustrated in Figure 15.2. The interdependence of the two controls is illustrated by a link interconnecting them.

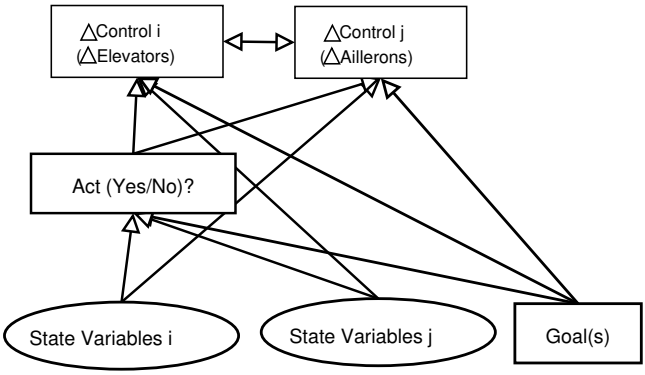

Fig. 15.2: Concept graph with two interdependent controls

If we were to ignore the effect of control j ($aillerons$), learning the change of control i ($elevators$) at time t would involve the state variables at time $t-1$. In addition, we would need to consider the goals and whether there is a need to act (values at time $t-1$). As our aim is to capture the effect of the other controls, we need to add the relevant information to the model. Here we need to add information about the change of control j ($aillerons$) used at time $t-1$ to the model for control i ($elevators$).

A similar approach is adopted when learning the change of control j ($aillerons$). The information used involves the state variables at time $t-1$, the current goal, and the current value of the change of control i ($elevators$) at time $t-1$.

The method outlined was validated using extensive experiments (Camacho and Brazdil, 2001). It was shown that, if the strategy outlined is followed, it is possible to acquire the ability to deal with several controls at the same time in an apparently coordinated manner.

The approach follows the basic methodology of bottom-up learning (or layered learning) discussed earlier in Section 15.4. Due to the interdependence of concepts, the approach can be regarded as a variant of iterative learning discussed in Section 15.5. Some concepts learned are used as input in the next phase of learning.

References

Aha, D. W., Lapointe, S., Ling, C. X., and Matwin, S. (1994). Inverting implication with small training set. In Bergadano, F. and De Raedt, L., editors, *Machine Learning: ECML-94, European Conference on Machine Learning, Catania, Italy*, volume 784 of *Lecture Notes in Artificial Intelligence*, pages 31–48. Springer.

Baroglio, C., Giordana, A., and Saitta, L. (1992). Learning mutually dependent relations. *Journal of Intelligent Information Systems*, 1:159–176.

Brazdil, P. (1981). *Model of Error Detection and Correction*. PhD thesis, University of Edinburgh.

Brazdil, P. (1984). Use of derivation trees in discrimination. In O'Shea, T., editor, *ECAI 1984 - Proceedings of 6th European Conference on Artificial Intelligence*, pages 239–244. North-Holland.

Burstall, R. M. and Darlington, J. (1977). A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67.

Camacho, R. and Brazdil, P. (2001). Improving the robustness and encoding complexity of behavioural clones. In De Raedt, L. and Flach, P., editors, *Proceedings of the 12th European Conference on Machine Learning (ECML '01)*, LNAI 2167, pages 37–48, Freiburg, Germany. Springer.

De Raedt, L. (2008). *Logical and Relational Learning*. Springer.

Dean, T. and Boddy, M. (1988). Reasoning about partially ordered events. *Artificial Intelligence*, 36:375–399.

Diettrich, T. and Michalski, R. (1983). A comparative review of selected methods for learning from examples. In Michalski, R., Carbonell, J., and Mitchell, T., editors, *Machine Learning: An Artificial Intelligence Approach*, pages 41–82. Tioga Publishing Company.

Ferrucci, D., Levas, A., Bagchi, S., Gondek, D., and Mueller, E. (2013). Watson: Beyond Jeopardy! *Artificial Intelligence*, 199:93–105.

Fürnkranz, J. and Hüllermeier, E. (2003). Pairwise preference learning and ranking. In Lavrač, N., Gamberger, D., Blockeel, H., and Todorovski, L., editors, *Proceedings of the 14th European Conference on Machine Learning (ECML2003)*, volume 2837 of *LNAI*, pages 145–156. Springer-Verlag.

Fürnkranz, J. and Hüllermeier, E. (2011). *Preference Learning*. Springer-Verlag.

Jorge, A. M. (1998). *Iterative Induction of Logic Programs*. PhD thesis, Faculty of Sciences, University of Porto.

Jorge, A. M. and Brazdil, P. (1996). Architecture for iterative learning of recursive definitions. In De Raedt, L., editor, *Advances in Inductive Logic Programming*, volume 32 of *Frontiers in Artificial Intelligence and applications*. IOS Press.

Kietz, J.-U., Serban, F., Bernstein, A., and Fischer, S. (2012). Designing KDD-Workflows via HTN-Planning for Intelligent Discovery Assistance. In Vanschoren, J., Brazdil, P., and Kietz, J.-U., editors, *PlanLearn-2012, 5th Planning to Learn Workshop WS28 at ECAI-2012, Montpellier, France*.

Lake, B. M., Ullman, T. D., Tenenbaum, J. B., and Gershman, S. J. (2017). Building machines that learn and think like people. *Beh. and Brain Sciences*, 40.

Lavrač, N. and Džeroski, S. (1994). *Inductive Logic Programming: Techniques and Applications*, chapter 5. LINUS: Using attribute-value learners in an ILP framework, pages 81–122. Ellis Horwood.

Lavrač, N., Džeroski, S., and Grobelnik, M. (1991). Learning non-recursive definitions of relations with LINUS. In *Proceedings of the 5th Working Session on Learning*, pages 265–281. Springer.

Mitchell, T. (1977). *Version spaces: A candidate elimination approach to rule learning*. PhD thesis, Electrical Engineering Department, Stanford University.

Mitchell, T. (1982). Generalization as Search. *Artificial Intelligence*, 18(2):203–226.

Muggleton, S. (1987). Duce: an oracle-based approach to constructive induction. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 287–292. Morgan Kaufmann.

Muggleton, S. and Buntine, R. (1988). Machine invention of first-order predicated by inverting resolution. In *Proceedings of the 5th International Workshop on Machine Learning*, pages 339–351. Morgan Kaufmann.

Muggleton, S. H. (1991). Inverting resolution principle. In Hayes, J. E., Michie, D., and Tyugu, Ė., editors, *Machine Intelligence 12: Towards an Automated Logic and Thought*.

Sammut, C. (1981). Concept learning by experiment. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence, Vancouver*.

Shapiro, E., editor (1996). *Algorithmic Program Debugging*. MIT Press.

Stone, P. (2000). *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. MIT Press.