# CHAPTER 3

# APPROACHES TO COMPUTATIONAL INDUCTION

*— in which it is demonstrated how problems from machine learning can be reformulated as problems of explanatory and confirmatory induction —*

IN THIS CHAPTER I will review and discuss some of the approaches to computational induction that can be found in the machine learning and knowledge representation literature. Broadly speaking, each of these approaches falls in one of two categories: the induction of logic programs (roughly corresponding to explanatory induction), and the induction of integrity constraints that are not primarily intended to explain the classification of the supplied examples. Many aspects of these approaches also occur in the context of the conceptually simpler problem of learning concepts from examples.

I will start with a discussion of the latter problem, and indicate how it can be reformulated in logical terms. In this way it becomes clear that concept learning can be viewed as a special case of induction of logic programs, which is discussed next. After that I turn to the problem of inducing non-classificatory logical theories. The chapter is ended with a discussion of the relations between these two forms of computational induction and the two forms of reasoning introduced in the previous chapter.

## §9. CONCEPT LEARNING FROM EXAMPLES

Traditionally, the problem of concept learning from examples has occupied a central position among computational approaches to induction. Informally, a *concept* is a description of a certain set of objects or *instances*, and an *example* is a description of an instance together with a classification (positive if the instance belongs to the concept, negative if it does not). Concept learning from examples is the problem of inferring an unknown concept from given positive and negative examples. This inference is inductive, since it proceeds from specific facts about a concept to a general definition of the concept.

### *The Version Space model*

Various formalisations of the problem of learning concepts from examples exist. The following formalisation was proposed by Mitchell (1982, p.204)[28]:

[28]For the purposes of the present discussion, Mitchell's terminology has been adapted (Mitchell used generalization for concept, training instance for example, and matching for covering).

## 3. Approaches to computational induction

*Problem*: Concept learning from examples.
*Given*: (1) A language in which to describe instances.
(2) A language in which to describe concepts.
(3) A covering predicate that matches concepts to instances.
(4) A set of positive and negative examples of a target concept to be learned.
*Determine*: Concepts within the provided language that are consistent with the presented examples (i.e., plausible descriptions of the target concept).

A concept is considered to be *consistent* with a set of examples if and only if it covers every positively classified instance and no negatively classified instance.

This problem can be reformulated in set-theoretical terms as follows. If the set of instances of a concept is called the *extension* of that concept, then a concept is consistent with a set of examples if the extension of the concept is a superset of the set of positively classified instances, while its intersection with the set of negatively classified instances is empty. Furthermore, concepts can be ordered by their extensions (Mitchell, 1982, p.206): a concept $C_1$ is said to be (extensionally) more *specific* than another concept $C_2$ if the extension of $C_1$ is a proper subset of the extension of $C_2$; we also say that $C_2$ is (extensionally) more *general* than $C_1$. Clearly this relation is transitive — we usually refer to the reflexive (i.e. non-proper) version of this relation as the (extensional) *generality ordering*, and say that $C_1$ is as specific as $C_2$ (or $C_2$ is as general as $C_1$). Note that the generality ordering is not, in general, antisymmetric, since different concepts may be extensionally equal (e.g., the concept 'square circle', and the concept 'unicorn'). However, it can be turned into a partial order by considering equivalence sets of extensionally equivalent concepts.

In fact, most of the regularities exhibited by concept extensions do not carry over to the set of concepts. For instance, the powerset of the set of instances forms a Boolean algebra, which means that there are well-defined operations to construct the smallest set of instances that contains two given sets of instances, and the largest set of instances that is contained in two given sets of instances (i.e., set-union and set-intersection). However, the corresponding operations on concepts (i.e. *least general generalisation* (LGG) and *most general specialisation* (MGS)) are usually not uniquely defined, which is a consequence of the circumstance that not every instance set is the extension of a concept, and that not every expression of the concept language has an extension. Thus, the complexity of concept learning is, to a large extent, determined by the languages involved.

Depending on the concept language $L_C$, it may or may not be possible to define a relation $\geq \subseteq L_C \times L_C$ of intensional generality or *subsumption*, which coincides with the extensional generality ordering. If $C_1 \geq C_2$ implies that $C_1$ is as general as $C_2$, the subsumption ordering may be called *sound*; if $C_1 \geq C_2$ whenever $C_1$ is as general as $C_2$, it may be called *complete*. Given this terminology, we can formulate the following result.

LEMMA 3.1. *If the subsumption ordering $\geq$ is sound, then the set of all concepts consistent with a given set of examples is convex wrt. $\geq$.*
*Proof.* We have to prove that if $C_1$ and $C_2$ are consistent concepts with $C_1 \geq C_2$, then so is any $C$ such that $C_1 \geq C \geq C_2$.

By the soundness of ≥, $C_1 \geq C$ implies that if $C_1$ does not cover any negatively classified instance, neither does $C$. Likewise, $C \geq C_2$ implies that if $C_2$ covers every positively classified instance, so does $C$. We conclude that $C$ is consistent.

Mitchell calls the set of all concepts that are consistent with the examples the *Version Space*, 'because it contains all plausible versions of the emerging concept' (Mitchell, 1982, p.212). He notes that the convexity of the Version Space allows for a compact representation in terms of the following two sets:

$S = \{C \in L_C \mid$ $C$ is a concept that is consistent with the examples, and there is no concept which is both more specific than $C$ and consistent with the examples$\}$

$G = \{C \in L_C \mid$ $C$ is a concept that is consistent with the examples, and there is no concept which is both more general than $C$ and consistent with the examples$\}$

The sets $S$ and $G$ contain the most specific and most general concepts consistent with the examples. A concept $C$ is consistent with the examples if and only if it subsumes some member of $S$, and is subsumed by some member of $G$ (fig. *3.1*).

I will refer to the view of a hypothesis space as a partially ordered convex set with lower and upper bounds as the *Version Space model*. Every computational approach to induction exploits, in one way or the other, the subsumption ordering when searching the space of possible hypotheses. The Version Space model is therefore an important conceptual tool for describing such approaches.

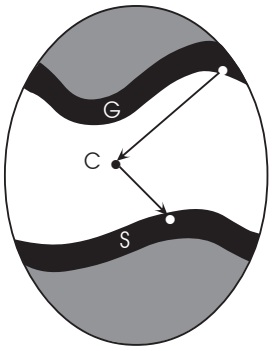


*Figure 3.1*. The Version Space model. $S$ and $G$ are the lower and upper boundaries of the set of concepts consistent with given examples: every consistent concept $C$ subsumes at least one member of $S$, and subsumed by at least one member of $G$ (the arrows point from the subsuming concept to the subsumed concept). The grey areas contain the inconsistent concepts.

## 3. Approaches to computational induction

*Attribute-value languages*

In his general definition of the concept learning problem, Mitchell distinguished between an instance language and a concept language. Indeed, the distinction between expressions describing instances and expressions describing sets of instances seems a crucial one, an observation which would suggest the use of a predicate-logical language. However, many approaches to concept learning employ in fact the same language for describing instances and concepts, a strategy that has been referred to as the *Single Representation Trick* (Dietterich *et al.*, 1982). This is, logically speaking, a non-obvious step that deserves some further attention.

The most frequently employed languages in computational approaches to concept learning belong to the class of *attribute-value languages*. These are propositional languages in which propositions are attribute-value pairs. Each attribute, such as colour, has a designated set of possible values, e.g. {red, yellow, blue}. Attribute-value pairs may be combined into expressions by means of the usual logical connectives, leading to expressions like

$$\texttt{colour=red} \wedge (\texttt{shape=square} \vee \texttt{shape=triangle}) \qquad (1)$$

Descriptions of instances may be restricted to conjunctions of attribute-value pairs, but this restriction is not essential.

What is important here is that the covering relation, which tests whether a concept covers an instance, is now equivalent to the subsumption relation, which tests whether a concept is more general than another concept. For instance, expression (1) subsumes the following expression

$$\texttt{colour=red} \wedge \texttt{shape=square} \wedge \texttt{size=small} \qquad (2)$$

If expression (2) describes an instance, we may conclude that this instance belongs to the concept described by (1). Alternatively, if (2) describes a concept, then this concept is more specific than the concept described by (1).

We may furthermore note that if attribute-value pairs are considered as propositions that may take on a truth value, every truth-assignment satisfying formula (2) also satisfies formula (1) — in other words, (2) logically entails (1). More generally, subsumption in attribute-value languages can be seen as a special case of logical entailment in propositional logic. It should be noted, however, that in some cases additional background knowledge is needed. For instance, consider the following expression:

$$\texttt{colour=red} \wedge \neg(\texttt{shape=round} \vee \texttt{size=big}) \qquad (3)$$

Intuitively, the concept described by (3) subsumes the concept described by (2). However, (2) only logically entails (3) if we know that $\texttt{shape=square} \rightarrow \neg(\texttt{shape=round})$ and $\texttt{size=small} \rightarrow \neg(\texttt{size=big})$ are logically valid (true in every interpretation). Axioms to this effect should be included for all attributes, and for all pairs of values.[29]

[29] By employing a higher-order logic, one could take care of all attributes at once — see (Flach, 1992a).

What has been demonstrated here is that, even if concept learning from examples proceeds from instances to sets of instances, it can be formalised by employing a simple propositional language that fails to recognise the difference. However, it should be made clear that something is lost along the way. For instance, consider a situation in which some of the examples are incomplete: the values of some attributes are left unspecified. It may happen that one of the missing attributes is crucial for explaining the classification of those instances — in fact, it is possible that two instances receive the same incomplete description, while one is classified positively and the other negatively! Clearly, in the above setting this would immediately result in logical inconsistency, because the two instances are described by syntactically equal expressions. Such inconsistencies can be avoided by employing a predicate-logical language, since such a language allows us to mention the instance explicitly.

*Concept learning in predicate logic*

In predicate logic instances are denoted by constants, and sets of instances are denoted by free variables. Following this observation, the concept denoted by attribute-value expression (1) could be represented in predicate logic as

$$\texttt{colour-red(X)} \wedge (\texttt{shape-square(X)} \vee \texttt{shape-triangle(X)}) \quad (4)$$

where `colour-red`, `shape-square` and `shape-triangle` are unary predicates. In general, a formula can have any number of free variables, describing concepts whose extensions contain tuples of instances — another advantage offered by predicate logic over propositional logic. The idea of representing concepts by *open* formulas can be traced back to Gottlob Frege (1893), and is extensively discussed in (Console & Saitta, 1994).

An instance, like the one expressed by attribute-value expression (2), could be represented by a variable-free formula, thereby naming the instance:

$$\texttt{colour-red(a)} \wedge \texttt{shape-square(a)} \wedge \texttt{size-small(a)} \quad (5)$$

Instance `a` can be shown to belong to the concept defined by (4) by substituting `a` for `X`, and showing that the resulting formula is logically implied by (5). Similarly, one concept can be shown to subsume another by substituting an arbitrary constant for the variable in both formulas, and to show that the latter logically entails the former.

The expressiveness of predicate logic can be further exploited by including a designated predicate expressing the classification of instances. For instance, the following formula expresses that `b` is a positive example of the concept ‘pretty things’:

$$\texttt{colour-red(b)} \wedge \texttt{size-small(b)} \rightarrow \texttt{pretty(b)} \quad (6)$$

The following formula gives a definition of that concept:

$$\forall \texttt{X:}\ (\texttt{colour-red(X)} \vee \texttt{colour-blue(X)}) \leftrightarrow \texttt{pretty(X)} \quad (7)$$

This definition is consistent with positive example (6), since the latter is logically

entailed by the former, or equivalently, the consequent of implication (6) is logically entailed by its antecedent combined with (7).

Another definition that is consistent with example (6) is the following:

$$\forall \texttt{X: colour-red(X)} \leftrightarrow \texttt{pretty(X)} \qquad (8)$$

Clearly, the concept defined by (8) is more specific than the one defined by (7) — however, neither of them logically entails the other. The usual solution to this problem is to replace the equivalences $\leftrightarrow$ with implications $\rightarrow$, expressing *sufficient* conditions for concept membership:

$$\forall \texttt{X: (colour-red(X)} \vee \texttt{colour-blue(X))} \rightarrow \texttt{pretty(X)} \qquad (7')$$

$$\forall \texttt{X: colour-red(X)} \rightarrow \texttt{pretty(X)} \qquad (8')$$

It is now obvious that (7′) logically entails (8′), hence the former defines a more general concept than the latter.[30] Representing a concept definition by sufficient conditions only means that a negative classification follows from a failure to prove a positive classification, similar to negation as failure in logic programming. Alternatively, one could add explicit necessary conditions — if these are different from the sufficient conditions, one would effectively obtained a three-valued concept.

I have indicated several ways in which the problem of concept learning from examples can be reformulated in predicate logic. One possibility is to represent concepts by open formulas with free variables. An alternative to this 'open formula' approach is to represent sufficient and necessary conditions by closed formulas, by introducing a designated predicate naming the concept to be learned. This 'closed formula' approach has the distinct advantage that a more general concept corresponds to a logically stronger formula, which is intuitively more appealing.

As for the examples, in the open formula approach these are represented by variable-free descriptions, while in the closed formula approach there are again two alternatives. Consider formula (6) above, which describes an instance `b` as belonging to the concept `pretty`. One alternative is to consider the complete implication as the example, which should be entailed by the concept definition (the 'examples as implications' approach). Another option is to split the implication in two parts, the description of the object (here: $\texttt{colour-red(b)} \wedge \texttt{size-small(b)}$), and its classification (here: `pretty(b)`), and to consider only the latter as constituting the example, while the former is part of the background knowledge. This 'examples as classifications' approach has the additional advantage that instances need not be described by variable-free formulas only: their properties could be deduced from a general background theory.

Combining the 'closed formula' and 'examples as classifications' approaches, we obtain the following predicate logical paraphrase of the concept learning problem:

[30] Under this representation, reasoning from specific to general corresponds to reasoning from logically weaker to logically stronger, which seems very natural. However, notice that this is at variance with the attribute-value and open formula representations of concepts, where the more specific concept entails the more general one. This has been a source of much confusion and debate — see e.g. (Niblett, 1988; Flach, 1992b; Console & Saitta, 1994).

### *§9. Concept learning from examples*

*Problem*: Concept learning from examples in predicate logic.
*Given*: (1) A predicate-logical language.
(2) A predicate representing the target concept.
(3) A set of ground positive and negative literals of this predicate, representing the positive and negative examples.
(4) A background theory from which descriptions of instances can be deduced.
*Determine*: Concepts within the provided language that are consistent with the presented examples.

A concept is consistent with the examples if, together with the background theory, it entails every positive example, without misclassifying any of the negative examples. This presupposes that concepts are represented as sufficient conditions only, which is usually the case.

Note that such a sufficient condition corresponds to a definite clause in logic programming, and a concept definition corresponds to a definite program. Thus, the above problem statement is general enough so as to cover the problem of inferring a logic program from examples. The latter problem constitutes a considerably more difficult induction task, since in general both background theory and target program may be recursive. Furthermore, if function symbols are involved, the universe of instances is infinite.

## §10. INDUCTION OF LOGIC PROGRAMS

I should like to stress that representing an inductive hypothesis as a logic program, i.e. a set of definite or normal clauses, does not necessarily imply that the inductive hypothesis is meant to be **used** as a computer program. For instance, a medical doctor might be interested in a definition of the concept 'symptoms indicating a rheumatoid disease' without intending to execute it as a Prolog program. The choice for logic programs as description formalism is a matter of representation rather than pragmatics. On the other hand, the synthesis of Prolog programs is one possible application of what has come to be called *inductive logic programming* (LavraË & Dæeroski, 1984).

If a concept is represented by a logic program consisting of clauses with the target predicate in the head, its extension is represented by the minimal Herbrand model of the program plus the background theory, restricted to the ground atoms of the target predicate. In other words, the extension of such a predicate definition consists of the set of ground instances of the target predicate logically entailed by that definition. It follows that the extensional generality ordering is a special case of logical entailment between predicate definitions, relative to the background theory. However, a subsumption relation between sets of clauses is computationally expensive, and many approaches to induction of logic programs employ a subsumption relation between single clauses instead.

In the case of non-recursive clauses, there is a very elegant intensional generality relation exactly matching extensional generality, which is called *θ-subsumption*. If $C_1$ and $C_2$ are two clauses, $C_1$ θ-subsumes $C_2$ if and only if there exists a substitution that can be applied to $C_1$, such that every literal in the resulting clause occurs in $C_2$. For instance,

consider the following two clauses:

```
element(X,[Y|Z]).
element(V,[V|W]):-list(W).
```

The first clause states that X is an element of any non-empty list; the second clause states that the head of the list is an element, provided the tail is a list. The first clause can be made equal to the head of the second by applying the substitution {X→V,Y→V,Z→W}; thus, the first clause θ-subsumes the second. One could say that in the first clause the head and the tail of the list are more constrained: the head by unifying it with the first argument, the tail by the literal in the body.

For recursive clauses, however, θ-subsumption is sound but incomplete. For instance, consider the following two clauses:

```
list([X,Y|Z]):-list(Z).
list([V|W]):-list(W).
```

If *P* is a predicate definition of `list` containing the first clause, and *P′* is obtained from *P* by replacing the first clause with the second, it is clear that *P′* is extensionally more general than *P*. However, the first clause is not θ-subsumed by the second.

On the other hand, the intensional generality ordering of θ-subsumption has the advantage that it behaves more regularly than its extensional counterpart. This is illustrated by the following two clauses:

```
list([A,B|C]):-list(C).
list([P,Q,R|S]):-list(S).
```

Under θ-subsumption, there is a unique least general clause subsuming both of these clauses:

```
list([X,Y|Z]):-list(V).
```

However, the following clause is also extensionally more general than the first two, without θ-subsuming them:

```
list([X|Y]):-list(Y).
```

Thus, under θ-subsumption the operation of forming a least general generalisation or LGG is uniquely defined, which is a clear computational advantage. An extensive analysis of the relation between θ-subsumption and logical implication can be found in (Idestam-Almquist, 1994).

There are basically two approaches to induction of logic programs from examples, that can be understood in terms of the Version Space model. The so-called *top-down* methods search the generality ordering from the top downwards, constructing a most general program that is consistent with the examples, gradually specialising the program when more examples become available. Such methods rely heavily on the availability of negative examples, in order to prevent overgeneralisation. *Bottom-up* methods, on the other hand, construct a most specific program that implies the positive examples. Both methods will be briefly reviewed below.

*Top-down induction*

Shapiro's Model Inference System (MIS) was the first system to infer logic programs consisting of definite clauses from examples (Shapiro, 1981; 1983). It was mainly intended for incrementally synthesising Prolog programs. MIS performs a breadth-first search of the space of possible clauses, ordered by θ-subsumption. I will describe the operation of MIS by means of an example. Suppose we are learning the `element` predicate, then MIS starts with the most general clause `element(X,Y)`. Upon receipt of the first negative example, this clause is retracted and a list of possible *specialisations* (called refinements by Shapiro) is added to the search agenda. Under θ-subsumption, a clause can be specialised in two ways:

(*i*) by adding a literal to the clause;
(*ii*) by applying a substitution to the clause.

One possible specialisation of `element(X,Y)` is `element(X,[V|W])`, obtained by applying the substitution {`Y`→`[V|W]`}. Somewhere in the search process this clause will be considered; if it is still too general, it can subsequently be specialised to `element(X,[X])` by applying the substitution {`V`→`X`, `W`→`[]`}. This clause is true in the intended interpretation, so it will never be refuted by a negative example. However, this single-clause program will be incomplete with respect to most positive examples concerning longer lists, so the search for an additional clause continues. Another specialisation of `element(X,[V|W])`, obtained by adding a literal to the body, is `element(X,[V|W]):-element(X,W)`. This clause is also true in the intended interpretation; together, the two constructed clauses form a correct predicate definition of `element`.

The main drawback of the MIS system was its inefficiency, mostly due to the employment of a breadth-first search strategy. FOIL is a much more efficient top-down induction algorithm, that operates in function-free definite clause logic, employing a hill-climbing search using an information-gain heuristic (Quinlan, 1990). Below we will consider De Raedt's extension of MIS to the problem of inducing integrity constraints.

*Bottom-up induction*

One approach to bottom-up induction applies operators that invert the deductive inference rule of resolution, and is called *inverse resolution* (Muggleton & Buntine, 1988). One inverse resolution operator constructs a clause which, together with a given clause, produces a given resolvent. The main problem that has to be solved is to prevent trivial solutions that can be constructed by inverting propositional resolution with empty substitutions. To this end, the CIGOL system hypothesises inverse substitutions, which turn non-variable terms at specific positions into variables.

Due to the soundness of resolution as a deductive inference rule, the hypothesised clause logically entails the resolvent at the root of the tree, given the other clauses. Since resolution is refutation-complete, it seems that a full inversion of resolution is capable of constructing every clause that entails a given resolvent, given a number of other clauses[31]. However, notice that some of these clauses may have to be used several times

[31]See (Nienhuys-Cheng & Flach, 1991) for a discussion of the completeness of inverse

in order to derive the given resolvent. The CIGOL system, on the other hand, constructs inverse proof trees in which the hypothesised clause occurs only once; it was proved by Gottlob (1987) that the resulting generality relation is θ-subsumption (relative to the given clauses) rather than logical entailment. (Muggleton, 1992b) discusses techniques to invert logical entailment.

The GOLEM system (Muggleton & Feng, 1990) embodies a different approach to bottom-up induction. Rather than constructing clauses completing an inverse proof tree, they construct minimal generalisations of pairs of positive examples by means of relative least general generalisation (RLGG). Briefly, such an RLGG is the LGG under θ-subsumption of two ground clauses with the positive examples in the head, and literals from a ground background theory in the body. Syntactical restrictions, such as determinacy of variables in the body of the hypothesised clause, are applied in order to restrict the size of candidate clauses. Upon construction of a clause, the positive examples it covers are removed, and search proceeds in order to cover the remaining examples.

## §11. INDUCTION OF INTEGRITY CONSTRAINTS

Both concept learning and induction of logic programs are classification-oriented forms of induction. Their main goal is to induce a theory that is capable of partitioning a universe of instances (or tuples of instances) into positive and negative instances. By introducing a classification predicate, such classification theories naturally correspond to a definite logic program. However, not every logical theory is intended to perform classification — for instance, integrity constraints, i.e. indefinite clauses (having more than one positive literal) and denials (having no positive literal), do not unequivocally define a classification predicate. In order to induce such non-classificatory logical theories one needs a perspective that differs from the traditional concept learning setting.

### *Induction as nonmonotonic inference*

Such an alternative perspective was provided by Helft (1989). He noted that 'upon observing a number of birds and their ability to fly, people might generate the rule that *all birds fly* simply as a conclusion of the observations, grounded on their similarities, rather than as an explanation of the fact that, for example, Tweety flies knowing that it is a bird' (p.149). Furthermore, Helft observed that

> 'induction assumes that the similarities between the observed data are representative of the rules governing them (…). This assumption is like the one underlying default reasoning in that a priority is given to the information present in the database. In both cases, some form of "closing-off" the world is needed. However, there is a difference between these: loosely speaking, while in default reasoning the assumption is "what you are not told is false", in similarity-based induction it is "what you are not told looks like what you are told".' (Helft, 1989, p.149)

---

resolution operators.

On the basis of these observations, Helft arrives at the following problem definition:

*Problem*: Induction of generalisations (Helft)
*Given*: (1) A predicate-logical language.
(2) A set $\Delta$ of formulas.
*Determine*: A set $\Gamma$ of generalisations within the provided language such that $\Delta \vdash_{ND} \Gamma$, where $\vdash_{ND}$ is a certain rule of inference that embodies the assumptions underlying induction.

Essentially, the solution provided by Helft runs as follows. Given evidence $\Delta$ the set of generalisations $\Gamma$ consists of those clauses that are true in every truth-minimal model[32] of $\Delta$ (the 'strong' generalisations), and those clauses that are true in some truth-minimal model of $\Delta$ (the 'weak' generalisations). Additional requirements guarantee that none of the clauses in $\Gamma$ can be deduced from $\Delta$, and that $\Gamma$ contains no clauses that can be deduced from other generalisations in $\Gamma$.

The 'rule of inference' $\vdash_{ND}$ has a certain resemblance to what I call a conjectural consequence relation in this thesis. However, when defining this logical relation between evidence and generalisations Helft is concerned first and foremost with practical issues. For instance, $\Delta$ is restricted to a set of (possibly indefinite) clauses having certain properties, ensuring that every such $\Delta$ has a finite set of finite truth-minimal models. Furthermore, $\Gamma$ is limited to (possibly indefinite) clauses having a certain 'injectivity' property, ensuring that there are no infinite descending chains of such clauses under $\theta$-subsumption. These restrictions ensure, for example, that $\Gamma$ is always finite.

*Discovery of clausal theories*

De Raedt has implemented the system CLAUDIEN, which is capable of constructing a general theory consisting of clauses (including integrity constraints) from a given database consisting of range-restricted normal clauses[33] (De Raedt & Bruynooghe, 1993). Such a theory is complete in the sense that it entails every clause in the language that is satisfied by the given database.

*Problem*: Induction of a clausal theory (De Raedt)
*Given*: (1) A predicate-logical language.
(2) A database $D$ of range-restricted normal clauses.
*Determine*: A theory $T$ within the provided language such that:
($i$) $Comp(D) \vdash T$;
($ii$) for all formulas $f$ in the language, if $Comp(D) \vdash f$ then $T \vdash f$.

In addition, $T$ is required to be minimal (i.e., no clause can be removed from $T$ without giving up the completeness property).

In this problem setting $Comp(D)$ denotes the *predicate completion* of $D$ (Clark, 1978), which is essentially a theory expressed in predicate logic, obtained from $D$ by adding

[32] These are not necessarily Herbrand models.
[33] A *normal* clause is a clause with a single literal in the head, possibly including negated literals in the body. A *range-restricted* clause is a clause in which every variable in the head also occurs in the body.

sufficient conditions for the predicates in *D*. Predicate completion establishes a semantics for normal logic programs, for which SLDNF resolution (SLD resolution with *negation as failure*) provides a sound proof procedure. Typically the completion of a normal program results in a single model (restricted to the vocabulary of the program); as De Raedt notes, the completion semantics might be replaced by another semantics for logic programs selecting a particular model. All such semantics coincide when the given program is definite, and result in the unique truth-minimal model of the program, which assigns **`false`** to all ground atoms of which the truthvalue is unknown.

The main algorithm of the CLAUDIEN system is very similar to MIS, and operates by searching the space of possible clauses in a top-down manner, applying refinement operators to specialise clauses that are violated by the given database.

*Induction of attribute dependencies*

My own approach to inducing attribute dependencies in databases also falls in the non-classificatory category. Since it will be fully elaborated in chapter 8, I will only make a few brief remarks here. Two different kinds of attribute dependencies are distinguished: functional dependencies, and multivalued dependencies. A functional dependency states that the value a tuple takes on for a certain attribute is completely determined by the value it takes on for other attributes. Consequently, if two tuples in a relation satisfying the functional dependency have identical values for the latter attributes, they also have an identical value for the determined attribute. The induction problem is to determine the set of functional dependencies that hold for a given relation.

*Problem*: Induction of functional dependencies.
*Given*: (1) A relation scheme *R*.
(2) A set *E* of tuples over *R*.
*Determine*: A set *H* of functional dependencies over *R* such that:
(*i*) for all $h \in H$: *h* is satisfied by *E*;
(*ii*) for every functional dependency *g* satisfied by *E*: *H* logically entails *g*.

Note that the phrase '*h* is satisfied by *E*' takes on its standard logical meaning: functional dependencies correspond to definite clauses, and relations correspond to sets of ground facts, such that a relation satisfies a functional dependency if and only if the corresponding set of ground facts is a Herbrand model of the corresponding definite clause. Similarly, logical entailment in condition (*ii*) denotes entailment between definite clauses.

Multivalued dependencies generalise functional dependencies by stipulating that the function governing the determined attributes is set-valued. The details need not concern us here; suffice it to say that the problem statement for induction of multivalued dependencies is analogous to the one for functional dependencies above. However, the situation changes if tuples are processed one at a time: in such an *incremental* setting, dependencies should possibly be refuted by known examples only, and not by means of making possibly unwarranted assumptions about unseen instances. This will be further elaborated in the next section.

## §12. DISCUSSION

I will now analyse the computational approaches to induction presented in this chapter from the point of view of this thesis. In particular, I will categorise these approaches according to the two forms of induction distinguished in chapter 2: explanatory reasoning and confirmatory reasoning. After that, I will discuss an important property that can be used to distinguish some of these approaches: incrementality.

*Induction of logic programs as explanatory reasoning*

Both concept learning and induction of logic programs aim at formulating a hypothesis that classifies the examples correctly. We might say that such a hypothesis provides explanations for the classification of every single example, in the sense that the classification is obtained by a deductive proof. As a result, the classified instances themselves become obsolete once a hypothesis has been adopted.

In the case of induction of logic programs, where only sufficient conditions are induced, a positive classification is obtained by setting up a deductive proof, while a negative classification results whenever such a proof cannot be constructed (negation as failure). This means that negative classifications are not really explained: this would require necessary conditions as well. Rather, the induced hypothesis should be logically compatible with the negative examples, represented as negated ground literals — for practical purposes, they can be simply added to the background theory. This pragmatic view of negative examples into our logical framework corresponds to the way they are used in practice: as a means to prevent overgeneralisation, rather than as full-fledged premisses in a conjectural argument[34].

We thus arrive at the following abstract problem statement:

*Problem*: Induction of logic programs from examples and background knowledge.
*Given*: (1) A predicate-logical language.
(2) A set $E$ of ground positive literals of the target predicate.
(3) A background theory $T$, defining auxiliary predicates as well as negative examples (ground negative literals of the target predicate).
*Determine*: Hypotheses $H$ within the provided language such that:
($i$) for all $e \in E$: $T \cup H \models e$;
($ii$) $H$ is compatible with $T$.

Notice that this problem statement could be easily adapted to learning necessary conditions by replacing ‘positive’ with ‘negative’ and *vice versa*; likewise, one would obtain a hypothesis stating both sufficient and necessary conditions by including all examples, positive as well as negative, in $E$.

[34]As argued in (De Raedt & Bruynooghe, 1992), negative examples can be generalised into constraints with which the hypothesis should be compatible.

## 3. Approaches to computational induction

I will now show that inference from *E* to *H* relative to *T* is a form of explanatory reasoning, since it obeys the adequacy conditions formulated in chapter 2. These conditions are reproduced below, with the term 'observation report' replaced with 'set of examples'.

(E1) *Converse entailment condition*: a set of examples is explained by every consistent hypothesis entailing it.

(E2) *Converse consequence condition*: if a set of examples is explained by a hypothesis *H*, then it is also explained by every consistent formula entailing *H*.

(E3) *Special consistency condition*: a set of examples is compatible with every hypothesis by which it is explained.

(E4) *Equivalence condition for observations*: if a set of examples *E* is explained by a hypothesis *H*, then any set of examples logically equivalent with *E* is also explained by *H*.

The converse entailment condition (E1) states that consistent entailment is a special case of explanation. For the cases discussed in this chapter this condition can actually be strengthened to equivalence; I will demonstrate in chapter 7 that the same effect can be obtained by adding additional conditions. On the other hand, in some cases explanation is indeed a more comprehensive concept than consistent entailment; this will also be discussed in chapter 7.

The converse consequence condition (E2) expresses that from a given explanation *H*, one might go up along the generality ordering in the Version Space, as long as one does not go beyond the line established by the G set (recall that consistency is relative to the background theory including negative examples). It should be noted however that some forms of explanatory reasoning violate the converse consequence condition, for instance if explanations are only plausible, rather than deductive, consequences. In such cases, the Version Space contains 'holes': hypotheses that are in between the S and G boundaries, yet do not explain some positive example.

The special consistency condition (E3) basically states that explanations must be compatible with the negative examples. Finally, condition (E4) expresses that the logical form of the examples is immaterial; this may seem trivial in the context of clausal logic, in which formulas are expressed in conjunctive normal form, but note that logical equivalence should be interpreted relative to the background theory.

We may conclude that induction of logic programs conforms to the adequacy conditions for explanatory reasoning. Further conditions will be introduced in §24, which will enable a more detailed categorisation of different approaches to explanatory induction.

### *Induction of integrity constraints as confirmatory reasoning*

Each of Helft's, De Raedt's, and my own problem setting discussed in §11 is an instance of the following abstract setting:

## *§12. Discussion*

*Problem*: Confirmatory induction.
*Given*: (1) A predicate-logical language.
(2) Evidence *E*.
*Determine*: A hypothesis *H* within the provided language such that:
(*i*) *H* is confirmed by *E*;
(*ii*) for all *g* within the language, confirmed by *E*: *H* logically entails *g*.

The relation '*H* is confirmed by *E*' is a parameter in this scheme, that can be instantiated to either '*H* is satisfied by the truth-minimal model of *E*' (Helft, De Raedt) or '*H* is a set of attribute dependencies satisfied by the relation *E*'. The use of the term *confirmation* is justified by the fact that each of these instantiations satisfies Hempel's adequacy conditions, reproduced below[35]:

(C1) *Entailment condition*: any sentence which is entailed by consistent evidence is confirmed by it.

(C2) *Consequence condition*: if the evidence confirms every one of a set *K* of sentences, then it also confirms any sentence which is a logical consequence of *K*.

(C3) *Consistency condition*: consistent evidence is compatible with the set of all the hypotheses which it confirms.

(C4) *Equivalence condition for observations*: if evidence *E* confirms a hypothesis *H*, then any evidence logically equivalent with *E* also confirms *H*.

The entailment condition (C1) states that logical entailment is a special case of confirmation. This is obviously true for the minimal model setting: if every model of *E* is a model of *H*, certainly the minimal one is, which exists if *E* is consistent. For the attribute dependency setting this condition is essentially void, since such a sentence will be outside the hypothesis language. However, if we generalise this setting to 'the evidence, being a set of ground facts, forms a model for the hypothesis', then clearly (C1) holds for this setting as well.

The consequence condition (C2) states that confirmation is closed under logical consequence, which is true for each of the three settings. Interestingly, this means that also here the Version Space model applies, with a trivial S set (the equivalence class of tautologies). Condition (*ii*) in the problem statement for confirmatory induction above then essentially states that we are interested in constructing the G set of this Version Space. Consistency condition (C3) is obviously valid, since all hypotheses share the same model with the evidence; the same can be said about (C4).

We may conclude that each of the three problem settings for induction of non-classificatory theories can be seen as establishing a form of confirmatory reasoning. However, there are also difference between these approaches, that hinge upon the question

[35]I have replaced the term 'observation report' with 'evidence', because in Helft's and De Raedt's settings the input need not be restricted to ground facts.

whether it is allowed to process examples one by one. This question will be addressed below.

*Incremental induction*

In each of the problem settings discussed previously, hypotheses were constructed on the basis of a set of examples $E$. Now suppose a new example $e$ is presented. If the induction algorithm is able to re-use its previous calculations for $E$ in order to construct one or more hypotheses for $E \cup \{e\}$, the algorithm is said to be interactive or incremental. If, on the other hand, the induction algorithm merely repeats its calculations for the extended set of examples $E \cup \{e\}$, the algorithm is said to be empirical or non-incremental.

Mitchell's candidate elimination algorithm, which calculates the S and G sets delimiting the Version Space, is a nice example of an incremental algorithm. After receipt of a new example, the new S and G sets are calculated from the old sets and the example. The rationale behind this algorithm is that hypotheses below the S set or above the G set, that were refuted by some previous example, never become candidate hypotheses again. In other words: the Version Space is monotonically non-increasing when the set of examples increases. Notice that this is a necessary condition for the algorithm to converge upon a unique solution.

Now consider a situation in which only positive examples are available. In order to prevent overgeneralisation, one might hypothesise that certain instances, that have not been presented as positive examples, are actually negative examples. This would rule out those hypotheses that cover the assumed negative examples. However, if examples are supplied one at a time, it might be the case that an instance that was once considered negative turns out to be positive. Consequently, some of the hypotheses previously refuted become candidates again. If this happens, it is clear that the complete Version Space has to be rebuilt from scratch. The important conclusion must be that an incremental approach is incompatible with a strategy in which hypotheses are refuted on the basis of assumed classifications of instances.

However, this is exactly the strategy that is applied in the approaches of Helft and De Raedt, when they assume that every ground fact that is not known to be true must be false. Also, in my attribute dependency setting every tuple that is not known to be in the relation is assumed not to be in it. Does this imply that confirmatory reasoning is inherently non-incremental? The answer is: no, it does not — confirmatory reasoning can very well be incremental. For one thing, despite appearances, induction of functional dependencies can be done incrementally. The reason for this is that the only way to refute a functional dependency is by demonstrating that two tuples, that have equal values for the determining attributes but unequal values for the determined attribute, are in the relation. In other words, functional dependencies are refuted by positive examples only. Since we only make assumptions about unseen tuples being negative, functional dependencies are never refuted on the basis of assumptions, but always by hard facts. The set of functional dependencies satisfied by a relation is monotonically non-increasing when the relation increases.

For multivalued dependencies the situation is different. As will be explained in chapter 8, a multivalued dependency predicts that, if the relation contains two tuples which have

the same values for the determining attributes, a certain third tuple that can be constructed from those two must also be in the relation. Thus, a multivalued dependency is refuted by refuting one of its predictions, which requires a negative example, or else an assumption that any tuple that is not known the be in the relation is out of it. The latter assumption necessitates a non-incremental strategy. However, incrementality can be restored by not making any assumption about a tuple being in the relation or not, unless it is explicitly given by a positive or negative example. This can be formalised by introducing the notion of a *partial relation*, which is a pair $\langle E,N\rangle$ of positive tuples $E$ and negative tuples $N$. This leads to the following problem definition.

*Problem*: Incremental induction of multivalued dependencies.
*Given*: (1) A relation scheme $R$.
(2) A set $E$ of positive tuples over $R$.
(3) A set $N$ of negative tuples over $R$.
*Determine*: A set $H$ of multivalued dependencies over $R$ such that:
($i$) for all $h \in H$: $h$ is satisfied by the partial relation $\langle E,N\rangle$
($ii$) for every multivalued dependency $g$ satisfied by $\langle E,N\rangle$: $H$ logically entails $g$.

What it means for a multivalued dependency to be satisfied by a partial relation will be explained in chapter 8.

The same idea can be used to obtain an incremental version of Helft's and De Raedt's approach[36], as will be worked out in chapter 7. Rather than demanding that the hypothesis be satisfied by the truth-minimal model of the evidence, we require in the incremental setting that the hypothesis not be falsified by the information-minimal *partial* model of the evidence. In this model, the truthvalue of the unseen instances is undefined, preventing us from making unwarranted assumptions. Hypotheses are only refuted if they explicitly disagree with the seen examples, and thus the Version Space shrinks with growing evidence.

## §13. SUMMARY AND CONCLUSIONS

In this chapter I have discussed induction from a computational viewpoint. Every computational approach to induction presupposes a well-defined problem, such as concept learning from examples, induction of logic programs, induction of non-classificatory theories, and induction of attribute dependencies. It has been demonstrated that concept learning can be seen as a special case of induction of logic programs, which in turn is a prototypical form of explanatory induction, conforming to the four adequacy conditions (E1–4) listed in §8. Furthermore, I have demonstrated that each of the remaining problems can be seen as an instance of confirmatory induction, aimed at constructing a confirmed theory entailing all confirmed formulas. This has been similarly achieved by demonstrating that each of those instances satisfies Hempel's adequacy conditions (H1–4).

[36] In a different sense these approaches are incremental: if the evidence is seen as constituting one or more **models** of the intended hypotheses, rather than logical statements (De Raedt, personal communication).

### *3. Approaches to computational induction*

In the course of the discussion I have frequently referred to Mitchell's Version Space model, which has been developed in the context of concept learning, but has a much broader conceptual significance. Indeed, in explanatory reasoning the set of possible hypotheses will be convex relative to logical implication whenever explanation is identified with deductive entailment. In confirmatory reasoning this set is also convex, with a fixed upper bound.

Finally, the important notion of incrementality has been discussed. Incrementality is a property both of an induction algorithm and of a general problem setting. An algorithm is incremental whenever it processes examples one by one, constructing at each step a hypothesis that is based on the previous one. Such an algorithm will never reconsider previously refuted hypotheses, therefore it is important that hypotheses are never refuted on the basis of assumptions (which may turn out to be wrong) — this may be called incrementality of the problem setting. Explanatory reasoning is incremental in this sense: hypotheses are only refuted by known positive or negative examples. The confirmatory settings of Helft and De Raedt are non-incremental, since every instance not known to be positive is assumed to be negative. My own, incremental approach to induction of attribute dependencies in databases suggests a formalisation of incremental confirmatory induction in terms of partial models.

* * *