I

Identification

▶ Classification

Identity Uncertainty

▶ Entity Resolution
▶ Record Linkage

Idiot's Bayes

▶ Naïve Bayes

Immune Computing

▶ Artificial Immune Systems

Immune Network

A proposed theory that the immune system is capable of achieving immunological memory by the existence of a mutually reinforcing network of B-cells. This network of B-cells forms due to the ability of the paratopes, located on B-cells, to match against the idiotopes on other B-cells. The binding between the idiotopes and paratopes has the effect of stimulating the B-cells. This is because the paratopes on B-cells react to the idiotopes on similar B-cells, as it would an antigen. However, to counter the reaction there is a certain amount of suppression between the B-cells which acts as a regulatory mechanism. This interaction of the B-cells due to the network was said to contribute to a *stable* memory structure and account for the retainment of memory cells, even in the absence of antigen. This interaction of cells forms the basis of inspiration for a large number of AIS algorithms, for example aiNET.

Immune-Inspired Computing

▶ Artificial Immune Systems

Immunocomputing

▶ Artificial Immune Systems

Immunological Computation

▶ Artificial Immune Systems

Implication

▶ Entailment

Improvement Curve

▶ Learning Curves in Machine Learning

Incremental Learning

Paul E. Utgoff
University of Massachusetts, Amherst, MA, USA

Definition

Incremental learning refers to any ▶ online learning process that learns the same model as would be learned by a ▶ batch learning algorithm.

Motivation and Background

Incremental learning is useful when the input to a learning process occurs as a stream of distinct observations spread out over time, with the need or desire to be able to use the result of learning at any point in time, based on the input observations received so far. In principle, the stream of observations may be infinitely long, or the next observation long delayed, precluding any hope of waiting until all the observations have been received. Without the ability to forestall learning, one must commit to a sequence of hypotheses or other learned artifacts based on the inputs observed up to the present. One would rather not simply accumulate and store all the inputs and, upon receipt of each new one, apply a batch learning algorithm to the entire sequence of inputs received so far. It would be preferable computationally if the existing hypothesis or other artifact of learning could be updated in response to each newly received input observation.

Theory

Consider the problem of computing the balance in one's checkbook account. Most would say that this does not involve learning, but it illustrates an important point about incremental algorithms. One procedure, a batch algorithm based on the fundamental definition of balance, is to compute the balance as the sum of the deposits less the sum of the checks and fees. As deposit, check, and fee transactions accumulate, this definition remains valid. There is an expectation that there will be more transactions in the future, and there is also a need to compute the balance periodically to ensure that no contemplated check or fee will cause the account to become overdrawn. We cannot wait to receive all of the transactions and then compute the balance just once.

One would prefer an incremental algorithm for this application, to reduce the cost of computing the balance after each transaction. This can be accomplished by recording and maintaining one additional piece of information, the balance after the nth transaction. It is a simple matter to prove that the balance after n transactions added to the amount of transaction $n + 1$ provides the balance after $n + 1$ transactions. This is because the sums of the fundamental definition for $n + 1$ transactions can be rewritten as the sums of the fundamental definition for n transactions plus the amount of the nth transaction. This incremental algorithm reduces the computation necessary to know the balance after each transaction, but it increases the bookkeeping effort somewhat due to the need for an additional variable.

Now consider the problem of learning the mean of a real-valued variable from a stream of observed values of this variable. Though simple, most would say that this does involve learning, because one estimates the mean from observations, without ever establishing the mean definitively. The fundamental definition for the mean requires summing the observed values and then dividing by the number of observed values. As each new observation is received, one could compute the new mean. However, one can reduce the computational cost by employing an incremental algorithm. For n observations, we could just as well have observed exactly the n occurrences of the mean. The sum of these observations divided by n would produce the mean. If we were to be provided with an $n + 1$ observation, we could compute the new sum of the $n + 1$ observations as n cases of the mean value plus the new observation, divided by $n + 1$. This reduces the cost of computing the mean after each observation to one multiplication, two addition, and one division operations. There is a small increase in bookkeeping in maintaining the counter n of how many observations have been received and the mean m after n observations.

In both of the above examples, the need to record the fundamental data is eliminated. Only a succinct summary of the data needs to be retained. For the checkbook balance, only the balance after n transactions needs to be stored, making the specific amounts for the individual transactions superfluous. For the mean of a variable, only the mean m after n observations and the number n of observations need to be retained, making the specific values of the individual observations superfluous. Due to this characteristic, incremental algorithms are often characterized as memoryless, not because no memory at all is required but because no memory of the original data items is needed. An incremental algorithm is not required to be memoryless, but the incremental algorithm must operate by modifying its existing knowledge, not by hiding the application of the corresponding batch algorithm to the accumulated set of observations. The critical issue is the extent to which computation is reduced compared to starting with all the data observations and nothing more. An essential aspect for an incremental algorithm is that the obtained result be identical to that indicated by the fundamental definition of the computation to be performed.

A point of occasional confusion is whether to call an algorithm incremental when it makes adjustments to its data structures in response to a new data observation. The answer depends on whether the result is the same that one would obtain when starting with all the observations in hand. If the answer is no, then one may have an online learning algorithm that is not an incremental learning algorithm. For example, consider two alternative formulations of the problem mentioned above of learning the mean of a variable. Suppose that the count of observations, held in the variable n, is not permitted to exceed some constant, say 100. Then the mean after n observations coupled with the minimum of n and 100 no longer summarizes all n observations accurately. Consider a second reformulation. Suppose that the most recent 100 observations are held in a queue. When a new observation is received, it replaces the oldest of the 100 observations. Now the algorithm can maintain a moving average, but not the overall overage. These may be desirable, if one wishes to remain responsive to drift in the observations, but that is another matter. The algorithm would not be considered incremental because it does not produce the same result for all n observations that the corresponding batch algorithm would for these same n observations. The algorithm would be online, and it would be memoryless, but it would not be computing the same learned artifact as the batch algorithm.

These two latter reformulations raise the issue of whether the order in which the observations are received is relevant. It is often possible to determine this by looking at the fundamental definition of the computation to be performed. If the operator that aggregates the observations is commutative, then order is not important. For the checking account balance example above, the fundamental aggregation is accomplished in the summations, and addition is commutative, so the order of the transactions is not relevant to the resulting balance. If a fundamental algorithm operates on a set of observations, then aggregation of a new observation into a set of observations is accomplished by the set union operator, which is commutative. Can one have an incremental algorithm for which order of the observations is important? In principle, yes, provided that the result of the incremental algorithm after observation n is the same as that of the fundamental algorithm for the first n observations.

A final seeming concern for an incremental learning algorithm is whether the selection of future observations ($n+1$ and beyond) is influenced by the first n observations. This is a red herring, because for the n observations, the question of whether the learning based on these observations can be accomplished by a batch algorithm or a corresponding incremental algorithm remains. Of course, if one needs to use the result of learning on the first k instances to help select the $k + 1$ instance, then it would be good sense to choose an incremental learning algorithm. One would rather not apply a batch algorithm to each and every prefix of the input stream. This would require saving the input stream and it would require doing much more computation than is necessary.

We can consider a few learning scenarios which suit incremental learning. An ▶ active learner uses its current knowledge to select the next observation. For a learner that is inducing a classifier, the observation would be an unclassified instance. The active learner selects an unclassified instance, which is passed to an oracle that attaches a correct class label. Then the oracle returns the labeled instance as the next observation for the learner. The input sequence is no longer one of instances for which each was drawn independently according to a probability distribution over the possible instances. Instead, the distribution is conditionally dependent on what the learner currently believes. The learning problem is sequential in its nature. The observation can be delivered in sequence, and an incremental learning algorithm can modify its hypothesis accordingly. For the n observations received so far, one could apply a corresponding batch algorithm, but this would be unduly awkward.

▶ Reinforcement learning is a kind of online learning in which an agent makes repeated trials in a simulated or abstracted world in order to learn a good, or sometimes optimal, policy that maps states to actions. The learning artifact is typically a function V over states or a function Q over state-action pairs. As the agent moves from state to state, it can improve its function over time. The choice of action depends on the current V or Q and on the reward or punishment received at each step. Thus, the sequence of observations consists of state-reward pairs or state-action-reward triples. As with active learning, the sequence of observations can be seen as being conditionally dependent on what the learner currently believes at each step. The function V or Q can be modified after each observation, without retaining the observation. When the function is approximated in an unbiased manner, by using a lookup table for discrete points in the function domain, there is an analogy with the problem of computing a checkbook balance, as described above. For each cell of the lookup table, its value is its initial value plus the sum of the changes, analogously for transactions. One can compute the function value by computing this sum, or one can store the sum in the cell, as the net value of all the changes. An incremental algorithm is preferable both for reasons of time and space.

A k-nearest classifier (see ▶ Instance-Based Learning) is defined by a set of training instances, the observations, and a distance metric that returns the numeric distance between any two instances. The difference between the batch algorithm and the incremental algorithm is slight. The batch algorithm accepts all the observations at once, and the incremental algorithm simply adds each new observation to the set of observations. If, however, there were data structures kept in the background to speed computation, one could distinguish between building those data structures once (batch) and updating those data structures (incremental). One complaint might be that all of the observations are retained. However, these observations do not need to be revisited when a new one arrives. There is an impact on space, but not on time.

A ▶ decision tree classifier may be correct for the n observations observed so far. When the $n+1$ observation is received, an incremental algorithm will restructure the tree as necessary to produce the tree that the batch algorithm would have built for these $n + 1$ observations. To do this, it may be that no restructuring is required at all or that restructuring is needed only in a subtree. This is a case in which memory is required for saving observations in the event that some of them may be needed to be reconsidered from time to time. There is a great savings in time over running the corresponding batch algorithm repeatedly.

Applications

Incremental learning is pervasive, and one can find any number of applications described in the literature and on the web. This is likely due to the fact that incremental learning offers computational savings in both time and space. It is also likely due to the fact that human and animal learning takes place over time. There are sound reasons for incremental learning being essential to development.

Future Directions

Increasingly, machine learning is confronted with the problem of learning from input streams that contain many millions, or more, of observations. Indeed, the stream may produce millions of observations per day. Streams with this many instances need to be handled by methods whose memory requirements do not grow much or at all. Memoryless online algorithms are being developed that are capable of handling this much throughput. Consider transaction streams, say of a telephone company, or a credit card company, or a stock exchange, or a surveillance camera, or eye-tracking data, or mouse movement data. For such a rich input stream, one could sample it, thereby reducing it to a smaller stream. Or, one could maintain a window of observations, giving a finite sample that changes but does not grow over time. There is no shortage of applications that can produce rich input streams. New methods capable of handling such heavy streams have already appeared, and we can expect to see growth in this area.

Cross-References

▸ Active Learning
▸ Cumulative Learning
▸ Online Learning

Recommended Reading

Domingos P, Hulten G (2003) A general framework for mining massive data streams. J Comput Graph Stat 12:945–949

Giraud-Carrier C (2000) A note on the utility of incremental learning. AI Commun 13:215–223

Utgoff PE, Berkman NC, Clouse JA (1997) Decision tree induction based on efficient tree restructuring. Mach Learn 29:5–44

Indirect Reinforcement Learning

▸ Model-Based Reinforcement Learning

Induction

James Cussens
University of York, Heslington, UK

Definition

Induction is the process of inferring a general rule from a collection of observed instances. Sometimes it is used more generally to refer to any inference from premises to conclusion where the truth of the conclusion does not follow deductively from the premises, but where the premises provide evidence for the conclusion. In this more general sense, induction includes *abduction* where facts rather than rules are inferred. (The word "induction" also denotes a different, entirely deductive form of argument used in mathematics.)

Theory

Hume's Problem of Induction

The *problem of induction* was famously set out by the great Scottish empiricist philosopher David Hume (1711–1776), although he did not actually use the word "induction" in this context. With characteristic bluntness, he argued that:

> there can be no *demonstrative* arguments to prove *that those instances of which we have had no experience resemble those of which we have had experience* (Hume 1739, Part 3, Section 6).

Since scientists (and machine-learning algorithms) *do* infer future-predicting general laws from past observations, Hume is led to the following unsettling conclusion concerning human psychology (and statistical inference):

> It is not, therefore, reason, which is the guide of life, but custom. That alone determines the mind, in all instances, to suppose the future conformable to the past (Hume 1740).

That general laws cannot be demonstrated (i.e., deduced) from data is generally accepted. Hume, however, goes further: he argues that past observations do not even affect the *probability* of future events:

> Nay, I will go farther, and assert, that he could not so much as prove by any *probable* arguments, that the future must be conformable to the past. All probable arguments are built on the supposition, that there is this conformity betwixt the future and the past, and therefore can never prove it. This conformity is a *matter of fact*, and if it must be proved, will admit of no proof but from experience. But our experience in the past can be a proof of nothing for the future, but upon a supposition, that there is a resemblance betwixt them. This therefore is a point, which can admit of no proof at all, and which we take for granted without any proof (Hume 1740).

Induction and Probabilistic Inference

Hume's unwavering skepticism concerning prediction appears at variance with the predictive accuracy of machine learning algorithms: there is much experimental evidence that ML algorithms, once trained on "past observations," make predictions on unseen cases with an accuracy far in excess of what can be expected by chance. This apparent discrepancy between Hume's philosophy and practical experience of statistical inference can be explored using a familiar example from the literature on induction. Let e be the statement that all swans seen so far have been white and let h be the general rule that all swans are white. Since h implies e it follows that $P(e|h) = 1$ and so, using Bayes' theorem, we have that

$$P(h|e) = \frac{P(h)P(e|h)}{P(e)} = \frac{P(h)}{P(e)}. \qquad (1)$$

So $P(h|e) > P(h)$ as long as $P(e) < 1$ and $P(h) > 0$. This provides an explanation for the predictive accuracy of hypotheses supported by data: given supporting data they just have increased probability of being true. Of course, most machine learning outputs are not "noise-free" rules like h; almost always hypotheses claim a certain distribution for future data where no particular observation is ruled out entirely – some are just more likely than others. The same basic argument applies: if $P(h) > 0$ then as long as the observed data is more likely given the hypothesis than it is a priori, that is, as long as $P(e|h)/P(e) > 1$, then the probability of h will increase. Even in the (common) case where each hypothesis in the hypothesis space depends on real-valued parameters and so $P(h) = 0$ for all h, Bayes theorem still produces an increase in the probability *density* in the neighborhoods of hypotheses supported by the data.

In all these cases, it appears that e is giving "inductive support" to h. Consider, however, h' which states that all swans until now have been white and *all future swans will be black*. Even in this case, we have that $P(h'|e) > P(h')$ as long as $P(e) < 1$ and $P(h') > 0$, though h and h' make entirely contradictory future predictions. This is a case of Goodman's paradox. The paradox is the result of confusing probabilistic inference with inductive inference. Probabilistic inference, of which Bayes theorem is an instance, is entirely deductive in nature – the conclusions of all probabilistic inferences follow with absolute certainty from their premises (and the axioms of probability). $P(h|e) > P(h)$ for $P(e) < 1$ and $P(h) > 0$ essentially because e has (deductively) ruled out some data that might have refuted h, not because a "conformity betwixt the future and the past" has been established.

Good performance on unseen data can still be explained. Statistical models (equivalently machine learning algorithms) *make assumptions* about the world. These assumptions (so far!) often turn out to be correct. Hume noted that the principle "that like objects, placed in like circumstances, will always produce like effects" (Hume 1739, Part 3, Section 8) although not deducible from first principles, has been established by "sufficient custom." This is called the *uniformity of nature* principle in the philosophical literature. It is this principle which informs machine learning systems. Consider the standard problem of predicting class labels for attribute-value data using labeled data as training. If an unlabeled test case has attribute values which are "close" to those of many training examples all of which have the same class label then in most systems the test case will be labeled also with this class. Different systems differ in how they measure "likeness": they differ in their ▶ inductive bias. A system which posited h' above on the basis of e would have an inductive bias strongly at variance with the uniformity of nature principle.

These issues resurfaced within the machine learning community in the 1990s. This ML work focused on various "▶ *no-free-lunch theorems*." Such a theorem essentially states that a uniformity of nature assumption is required to justify any given inductive bias. This is how Wolpert puts in one of the earliest "no-free-lunch" papers:

> This paper proves that it is impossible to justify a correlation between reproduction of a training set and generalization error off of the training set using only a priori reasoning. As a result, the use in the real world of any generalizer which fits a hypothesis function to a training set (e.g., the use of back-propagation) is implicitly predicated on an assumption about the physical universe (Wolpert 1992).

Note that in Bayesian approaches inductive bias is encapsulated in the prior distribution: once a prior has been determined all further work in Bayesian statistics is entirely deductive. Therefore it is no surprise that inductivists have sought to find "objective" or "logical" prior distributions to provide a firm basis for inductive inference. Foremost among these is Rudolf Carnap (1891–1970) who followed a logical approach – defining prior distributions over "possible worlds" (first-order models) which were in some sense uniform (Carnap 1950). A modern extension of this line of thinking can be found in Bacchus et al. (1996).

Popper

Karl Popper (1902–1994) accepted the Humean position on induction yet sought to defend science from charges of irrationality (Popper 1934). Popper *replaced* the problem of induction by the problem of criticism. For Popper, scientific progress proceeds by conjecturing universal laws and then subjecting these laws to severe tests with a view to refuting them. According to the *verifiability principle* of the logical positivist tradition, a theory is scientific if it can be experimentally confirmed, but for Popper confirmation is a hopeless task, instead a hypothesis is only scientific if it is *falsifiable*. All universal laws have prior probability of zero, and thus will eternally have probability zero of being true, no matter how many tests they pass. The value of a law can only be measured by how well-tested it is. The degree to which a law has been tested is called its degree of *corroboration* by Popper. The $P(e|h)/P(e)$ term in Bayes theorem will be high if a hypothesis h has passed many severe tests.

Popper's critique of inductivism continued throughout his life. In the *Popper–Miller* argument (Popper and Miller 1984), as it became known, it is observed that a hypothesis h is logically equivalent to:

$$(h \leftarrow e) \wedge (h \vee e)$$

for any evidence e. We have that $e \vdash h \vee e$ (where $\vdash$ means "logically implies") and also that (under weak conditions) $p(h \leftarrow e|e) < p(h \leftarrow e)$. From this Popper and Miller argue that

> . . . we find that what is left of h once we discard from it everything that is logically implied by e, is a proposition that in general is counterdependent on e (Popper and Miller 1987)

and so.

> Although evidence may raise the probability of a hypothesis above the value it achieves on background knowledge alone, every such increase in probability has to be attributed entirely to the *deductive connections* that exist between the hypothesis and the evidence (Popper and Miller 1987).

In other words if $P(h|e) > P(h)$ this is only because $e \vdash h \vee e$. The Popper–Miller argument found both critics and supporters. Two basic arguments of the critics were that (1) deductive relations only set limits to probabilistic support; infinitely many probability distributions can still be defined on any given fixed system of propositions and (2) Popper–Miller are mischaracterizing induction as the absence of deductive relations, when it actually means *ampliative inference*: concluding more than the premises entail (Cussens 1996).

Causality and Hempel's Paradox

The branch of philosophy concerned with how evidence can confirm scientific hypotheses is known as ▸ *confirmation theory*. Inductivists take the position (against Popper) that observing data which follows from a hypothesis not only fails to refute the hypothesis, but also *confirms* it to some degree: seeing a white swan confirms the hypothesis that all swans are white, because

$$\forall x : \text{swan}(x) \rightarrow \text{white}(x), \text{swan}(\text{white\_swan}) \vdash \text{swan}(\text{white\_swan}).$$

But, by the same argument it follows that observing any nonwhite, nonswan (say a black raven) also confirms that all swans are white, since:

$$\forall x : \text{swan}(x) \rightarrow \text{white}(x), \neg\text{white}(\text{black\_reven}) \vdash \neg(\text{black\_reven}).$$

This is Hempel's paradox to which there are a number of possible responses. One option is to accept that the black raven is a confirming instance, as one object in the universe has been ruled out as a potential refuter. The *degree* of confirmation is however of "a miniscule and negligible degree" (Howson and Urbach 1989, p. 90). Another option is to reject the formulation of the hypothesis as a material implication where $\forall x : \text{swan}(x) \rightarrow \text{white}(x)$ is just another way of writing $\forall x : \neg\text{swan}(x) \vee \text{white}(x)$. Instead, to be a scientific hypothesis of any interest the statement must be interpreted *causally*. This is the view of Imre Lakatos (1922–1974), and since any causal statement has a (perhaps implicit) *ceteris paribus* ("all other things being equal") clause this has implications for refutation also.

> ... "all swans are white," if true, would be a mere curiosity unless it asserted that swanness *causes* whiteness. But then a black swan would not refute this proposition, since it may only indicate *other causes* operating simultaneously. Thus "all swans are white" is either an oddity and easily disprovable or a scientific proposition with a ceteris paribus clause and therefore easily undisprovable (Lakatos 1970, p. 102).

Cross-References

▸ Abduction
▸ Classification

Recommended Reading

Bacchus F, Grove A, Halpern JY, Koller D (1996) From statistical knowledge bases to degrees of belief. Artif Intell 87(1–2):75–143

Carnap R (1950) Logical foundations of probability. University of Chicago Press, Chicago

Cussens J (1996) Deduction, induction and probabilistic support. Synthese 108(1):1–10

Howson C, Urbach P (1989) Scientific reasoning: the Bayesian approach. Open Court, La Salle

Hume D (1739) A treatise of human nature, book one (Anonymously published)

Hume D (1740) An abstract of a treatise of human nature. (Anonymously published as a pamphlet). Printed for C. Borbet, London

Lakatos I (1970) Falsification and the methodology of scientific research programmes. In: Lakatos I, Musgrave A (eds) Criticism and the growth of knowledge. Cambridge University Press, Cambridge, pp 91–196

Popper KR (1959) The logic of scientific discovery. Hutchinson, London (Translation of *Logik der Forschung*, 1934)

Popper KR, Miller D (1984) The impossibility of inductive probability. Nature 310:434

Popper KR, Miller D (1987) Why probabilistic support is not inductive. Philos Trans R Soc Lond 321: 569–591

Wolpert DH (1992) On the connection between in-sample testing and generalization error. Complex Syst 6: 47–94

Induction as Inverted Deduction

▸ Logic of Generality

Inductive Bias

Synonyms

Learning bias; Variance hint

Definition

Most ML algorithms make predictions concerning future data which cannot be deduced from already observed data. The inductive bias of an algorithm is what choses between different possible future predictions. A strong form of inductive bias is the learner's choice of hypothesis/model space which is sometimes called *declarative bias*. In the case of Bayesian analysis, the inductive bias is encapsulated in the prior distribution.

Cross-References

▸ Induction
▸ Learning as Search

Inductive Database Approach to Graphmining

Stefan Kramer
Technische Universität München, Garching b. München, Germany

Overview

The inductive database approach to graph mining can be characterized by (1) the concept of querying for (subgraph) patterns in databases of graphs, and (2) the use of specific data structures representing the space of solutions. For the former, a query language for the specification of the patterns of interest is necessary. The latter aims at a compact representation of the solution patterns.

Pattern Domain

In contrast to other graph mining approaches, the inductive database approach to graph mining (De Raedt and Kramer 2001; Kramer et al. 2001) focuses on simple patterns (paths and trees) and complex queries (see below), not on complex patterns (general subgraphs) and simple queries (minimum frequency only). While the first approaches were restricted to paths as patterns in graph databases, they were later extended toward unrooted trees (Rückert and Kramer 2003, 2004). Most of the applications are dealing with structures of small molecules and structure–activity relationships (SARs), that is, models predicting the biological activity of chemical compounds.

Query Language

The conditions on the patterns of interest are usually called *constraints* on the solution space. Simple constraints are specified by so-called *query primitives*. Query primitives express frequency-related or syntactic constraints. As an example, consider the frequency-related query primitive $f(p, D) \geq t$, meaning that a subgraph pattern p has to occur with a frequency of at least t in the database of graphs D. Analogously, other frequency-related primitives demand a maximum frequency of occurrence, or a minimum agreement with the target class (e.g., in terms of the information gain or the χ^2 statistic). Answering frequency-related queries generally requires database access. In contrast to frequency-related primitives, syntax-related primitives only restrict the syntax of solution (subgraph) patterns, and thus do not require database access. For instance, we may demand that a pattern p is more specific than "*c:c-Cl*" (formally $p \geq$ *c:c-Cl*) or more general than "*C-c:c:c:c:c-Cl*" (formally $p \leq$ *C-c:c:c:c:c-Cl*). The strings in the primitive contain vertex (e.g., "*C*," "*c*," "*Cl*"…) and edge labels (e.g., " : ," "-"…) of a path in a graph. Many constraints on patterns can be categorized as either monotonic or anti-monotonic. Minimum frequency constraints, for instance, are anti-monotonic, because all subpatterns (in our case: subgraphs) are frequent as well, if a pattern is frequent (according to some user-defined threshold) in a database. Vice versa, maximum frequency is monotonic, because if a pattern is not too frequent, then all superpatterns (in our case: supergraphs) are not too frequent either. Anti-monotonic or monotonic constraints can be solved by variants of level-wise search and APriori (De Raedt and Kramer 2001; Kramer et al. 2001; Mannila and Toivonen 1997). Other types of constraints involving convex functions, for example, related to the target class, can be solved by branch-and-bound algorithms (Morishita and Sese 2000). Typical query languages offer the possibility to combine query primitives conjunctively or disjunctively.

Data Structures

It is easy to show that solutions to conjunctions of monotonic and anti-monotonic constraints can be represented by *version spaces*, and in particular, borders of the most general and the most specific patterns satisfying the constraints (De Raedt and Kramer 2001; Mannila and Toivonen 1997). Version spaces of patterns can be represented in data structures such as *version space trees* (De Raedt et al. 2002; Rückert and Kramer 2003). For sequences, data structures based on *suffix arrays* are known to be more efficient than data structures based on version spaces (Fischer et al. 2006). Query languages allowing disjunctive normal forms of monotonic or anti-monotonic primitives yield multiple version spaces as solutions, represented by generalizations of version space trees (Lee and De Raedt 2003). The inductive database approach to graph mining can also be categorized as *constraint-based mining*, as the goal is to find solution patterns satisfying user-defined constraints.

Recommended Reading

De Raedt L, Kramer S (2001) The levelwise version space algorithm and its application to molecular fragment finding. In: Proceedings of the seventeenth international joint conference on artificial intelligence (IJCAI 2001). Morgan Kaufmann, San Francisco

De Raedt L, Jaeger M, Lee SD, Mannila H (2002) A theory of inductive query answering. In: Proceedings of the 2002 IEEE international conference on data mining (ICDM 2002). IEEE Computer Society, Washington, DC

Fischer J, Heun V, Kramer S (2006) Optimal string mining under frequency constraints. In: Proceedings of the tenth European conference on the principles and practice of knowledge discovery in databases (PKDD 2006). Springer, Berlin

Kramer S, De Raedt L, Helma C (2001) Molecular feature mining in HIV data. In: Proceedings of the seventh ACM SIGKDD international conference on knowledge discovery and data mining (KDD 2001). ACM, New York

Lee SD, De Raedt L (2003) An algebra for inductive query evaluation. In: Proceedings of the third IEEE international conference on data mining (ICDM 2003). IEEE Computer Society, Washington, DC

Mannila H, Toivonen H (1997) Levelwise search and borders of theories in knowledge discovery. Data Min Knowl Discov 1(3):241–258

Morishita S, Sese J (2000) Traversing itemset lattice with statistical metric pruning. In: Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems (PODS 2000). ACM, New York

Rückert U, Kramer S (2003) Generalized version space trees. In: Boulicaut J-F, Dzeroski S (eds) Proceedings of the second international workshop on knowledge discovery in inductive databases (KDID-2003). Berlin, Springer

Rückert U, Kramer S (2004) Frequent free tree discovery in graph data. In: Proceedings of the ACM symposium on applied computing (SAC 2004). ACM, New York

Inductive Inference

Sanjay Jain[1] and Frank Stephan[2]
[1]School of Computing, National University of Singapore, Singapore, Singapore
[2]Department of Mathematics, National University of Singapore, Singapore, Singapore

Definition

Inductive inference is a theoretical framework to model learning in the limit. The typical scenario is that the learner reads successively datum

$d_0, d_1, d_2, \ldots$ about a concept and outputs in parallel hypotheses $e_0, e_1, e_2, \ldots$ such that each hypothesis e_n is based on the preceding data $d_0, d_1, \ldots, d_{n-1}$. The hypotheses are expected to converge to a description for the data observed; here the constraints on how the convergence has to happen depend on the learning paradigm considered. In the most basic case, almost all e_n have to be the same correct index e, which correctly explains the target concept. The learner might have some preknowledge of what the concept might be, that is, there is some class $\mathcal{C}$ of possible target concepts – the learner has only to find out which of the concepts in $\mathcal{C}$ is the target concept; on the other hand, the learner has to be able to learn every concept which is in the class $\mathcal{C}$.

Detail

The above given scenario of learning is essentially the paradigm of inductive inference introduced by Gold (1967) and known as **Ex** (explanatory) learning. Usually one considers learning of recursive functions or recursively enumerable languages. Intuitively, using coding, one can code any natural phenomenon into subsets of $\mathbb{N}$, the set of natural numbers. Thus, recursive functions from $\mathbb{N}$ to $\mathbb{N}$ or recursively enumerable subsets of $\mathbb{N}$ (called languages here) are natural concepts to be considered.

Here we will mainly consider language learning. Paradigms related to function learning can be similarly defined and we refer the reader to Osherson et al. (1986) and Jain et al. (1999).

One normally considers data provided to the learner to be either full positive data (i.e., the learner is told about every element in the target language, one element at a time, but never told anything about elements not in the target language) or full positive data and full negative data (i.e., the learner is told about every element, whether it belongs or does not belong to the target language). Intuitively, the reason for considering only positive data is that in many natural situations, such as language learning by children and scientific exploration (such as in astronomy), one gets essentially only positive data.

A *text* is a sequence of elements over $\mathbb{N} \cup \{\#\}$. Content of a text T, denoted $\mathrm{ctnt}(T)$, is the set of natural numbers in the range of T. For a finite sequence σ over $\mathbb{N} \cup \{\#\}$, one can similarly define $\mathrm{ctnt}(\sigma)$ as the set of natural numbers in the range of σ. A text T is said to be for a language L if $\mathrm{ctnt}(T) = L$. Intuitively, a text T for L represents sequential presentation of all elements of L, with #'s representing pauses in the presentation. For example, the only text for $\emptyset$ is $\#^\infty$. $T[n]$ denotes the initial sequence of T of length n. That is, $T[n] = T(0)T(1)\ldots T(n-1)$. We let SEQ denote the set of all finite sequences over $\mathbb{N} \cup \{\#\}$. An *informant* I is a sequence of elements over $\mathbb{N} \times \{0,1\} \cup \{\#\}$, where for each $x \in \mathbb{N}$, exactly one of $(x, 0)$ or $(x, 1)$ is in the range of I. An informant I is for L if $\mathrm{range}(I) - \{\#\} = \{(x, \chi_L(x)) : x \in \mathbb{N}\}$, where χ_L denotes the characteristic function of L.

A learner M is a mapping from SEQ to $\mathbb{N} \cup \{?\}$. Intuitively, output of ? denotes that the learner does not wish to make a conjecture on the corresponding input. The output of e denotes that the learner conjectures hypothesis W_e, where $W_0, W_1, \ldots$ is some acceptable numbering of all the recursively enumerable languages. We say that a learner M converges on T to e if, for all but finitely many n, $\mathrm{M}(T[n]) = e$.

Explanatory Learning

A learner M **TxtEx** identifies a language L iff, for all texts T for L, M converges to an index e such that $W_e = L$. Learner M **TxtEx** identifies a class $\mathcal{L}$ of languages if M **TxtEx** identifies each language in the class $\mathcal{L}$. Finally, one says that a class $\mathcal{L}$ is **TxtEx** learnable if some learner **TxtEx** identifies $\mathcal{L}$. **TxtEx** denotes the collection of all **TxtEx**-learnable classes. One can similarly define **InfEx** identification, for learning from informants instead of texts. The following classes are important examples:

$$RE = \{L : L \text{ is recursively enumerable}\};$$

$$FIN = \{L : L \text{ is a finite subset of } \mathbb{N}\};$$

$$KFIN = \{L : L = K \cup H \text{ for some } H \in FIN\};$$

$$SD = \{L : W_{\min(L)} = L\};$$

$$COFIN = \{L : \mathbb{N} - L \text{ is finite}\};$$

$$SDSIZE = \{\{e + x : x = 0 \vee x < |W_e|\} : W_e \text{ is finite}\};$$

$$SDALL = \{\{e + x : x \in \mathbb{N}\} : e \in \mathbb{N}\}.$$

Here, in the definition of *KFIN*, K is the halting problem, a standard example of a set which is recursively enumerable but not recursive. The classes *FIN*, *SD*, *SDSIZE*, and *SDALL* are **TxtEx** learnable (Case and Smith 1983; Gold 1967): The learner for *FIN* always conjectures the set of all data observed so far. The learner for *SD* conjectures the least datum seen so far as, eventually, the least observed datum coincides with the least member of the language to be learned. The learner for *SDSIZE* as well as the learner for *SDALL* also find in the limit the least datum e to occur and translate it into an index for the e-th set to be learned. The class *KFIN* is not **TxtEx** learnable, mainly for computational reasons. It is impossible for the learner to determine if the current input datum belongs to K or not; this forces a supposed learner either to make infinitely many mind changes on some text for K or to make an error on $K \cup \{x\}$, for some $x \notin K$. The union *SDSIZE* $\cup$ *SDALL* is also not **TxtEx** learnable, although it is the union of two learnable classes; so it is one example of various nonunion theorems. Gold (1967) gave even a more basic example: *FIN* $\cup \{\mathbb{N}\}$ is not **TxtEx** learnable. Furthermore, the class *COFIN* is also not **TxtEx** learnable. However, except for *RE*, all the classes given above are **InfEx** learnable, so when being fed the characteristic function in place of only an infinite list of all elements, the learners become, in general, more powerful.

Note that the learner never knows when it has converged to its final hypothesis. If the learner is required to know when it has converged to the final hypothesis, then the criterion of learning is the same as finite learning. Here a finite learner is defined as follows: the learner keeps outputting the symbol ? while waiting for enough data to appear and, when the data observed are sufficient, the learner outputs exactly one conjecture different from ?, which then is required to be an index for the input concept in the hypothesis space. The class of singletons $\{\{n\} : n \in \mathbb{N}\}$ is finitely learnable; the learner just waits until the unique element n of $\{n\}$ has appeared and then knows the language. In contrast to this, the classes *FIN* and *SD* are not finitely learnable.

Blum and Blum (1975) obtained the following fundamental result: Whenever M learns L explanatorily from text, then L has a locking sequence for M. Here, a sequence σ is said to be a locking sequence for M on L if (a) $\text{ctnt}(\sigma) \subseteq L$, (b) for all τ such that $\text{ctnt}(\tau) \subseteq L$, $\text{M}(\sigma) = \text{M}(\sigma\tau)$, and (c) $W_{\text{M}(\sigma)} = L$. If only the first two conditions are satisfied, then the sequence is called a *stabilizing sequence* for M on L (Fulk 1990). It was shown by Blum and Blum (1975) that if a learner M **TxtEx** identifies L, then there exists a locking sequence σ for M on L. One can use this result to show that certain classes, such as *FIN* $\cup \{\mathbb{N}\}$, are not **TxtEx** learnable.

Beyond Explanatory Learning

While **TxtEx** learning requires that the learner syntactically converges to a final hypothesis, which correctly explains the concept, this is no longer required for the more general criterion of behaviorally correct learning (called **TxtBc** learning). Here, the learner may not syntactically converge, but it is still required that all its hypothesis after sometime are correct; see Bārzdiņš (1974b), Case and Lynes (1982), Case and Smith (1983), Osherson et al. (1986), and Osherson and Weinstein (1982). So there is semantic convergence to a final hypothesis. Thus, a learner M **TxtBc** identifies a language L if for all texts T for L, for all but finitely many n, $W_{\text{M}(T[n])} = L$. One can similarly define **TxtBc** learnability of classes of languages and the collection **TxtBc**. Every **TxtEx**-learnable class is **Bc** learnable, but the classes *KFIN* and *SDSIZE* $\cup$ *SDALL* are **TxtBc** learnable but not **TxtEx** learnable. Furthermore, **InfEx** $\not\subseteq$ **TxtBc**, for example, *FIN* $\cup \{\mathbb{N}\}$ is **InfEx** learnable but not **TxtBc** learnable. On the other hand, every class that is finitely learnable from informant is also **TxtEx** learnable (Sharma 1998).

An intermediate learning criterion is **TxtFex** learning (Case 1999) or vacillatory learning, which is similar to **TxtBc** learning except that we require that the number of distinct hypotheses output by the learner on any text is finite. Here one says that the learner $\mathbf{TxtFex}_n$ learns the language L if the number of distinct hypotheses that appears infinitely often on any text T for L is bounded by n. Note that $\mathbf{TxtFex}_* = \mathbf{TxtFex}$. Case (1999) showed that

$$\mathbf{TxtEx} = \mathbf{TxtFex}_1 \subset \mathbf{TxtFex}_2 \subset \mathbf{TxtFex}_3 \subset \ldots \subset \mathbf{TxtFex}_* \subset \mathbf{TxtBc}.$$

For example, the class $SD \cup SDALL$ is actually $\mathbf{TxtFex}_2$ learnable and not **TxtEx** learnable. The corresponding notion has also been considered for function learning, but there the paradigms of explanatory and vacillatory learning coincide (Case and Smith 1983).

Blum and Blum (1975), Case and Lynes (1982), and Case and Smith (1983) also considered allowing the final (or final sequence of) hypothesis to be anomalous; Blum and Blum (1975) considered $*$ anomalies, and (Case and Lynes 1982; Case and Smith 1983) considered the general case. Here the final grammar for the input language may not be perfect, but may have up to a anomalies. A grammar n is a anomalous for L (written $W_n =^a L$) iff card $((L - W_n) \cup (W_n - L)) \leq a$. Here one also considers finite anomalies, denoted by $*$-anomalies, where card$(S) \leq *$ just means that S is finite. Thus, a learner M $\mathbf{TxtEx}^a$ identifies a language L iff, for all texts T for all L, M on T converges to a hypothesis e such that $W_e =^a L$. One can similarly define $\mathbf{TxtBc}^a$-learning criteria. It can be shown that

$$\mathbf{TxtEx} = \mathbf{TxtEx}^0 \subset \mathbf{TxtEx}^1 \subset \mathbf{TxtEx}^2 \subset \ldots \subset \mathbf{TxtEx}^*$$

and

$$\mathbf{TxtBc} = \mathbf{TxtBc}^0 \subset \mathbf{TxtBc}^1 \subset \mathbf{TxtBc}^2 \subset \ldots \subset \mathbf{TxtBc}^*.$$

Let $SD_n = \{L : W_{\min(L)} =^n L\}$. Then one can show (Case and Lynes 1982; Case and Smith 1983) that $SD_{n+1} \in \mathbf{TxtEx}^{n+1} - \mathbf{TxtEx}^n$. However, there is a trade-off between behaviorally correct learning and explanatory learning for learning with anomalies. On one hand, $\mathbf{TxtBc} \not\subseteq \mathbf{TxtEx}^*$, but on the other hand $\mathbf{TxtEx}^{2n+1} \not\subseteq \mathbf{TxtBc}^n$ and $\mathbf{TxtEx}^{2n} \subseteq \mathbf{TxtBc}^n$. However, for learning from informants, we have $\mathbf{InfEx}^* \subseteq \mathbf{InfBc}$ (see Case and Lynes (1982) for the above results).

Consistent and Conservative Learning

Besides the above basic criteria of learning, researchers have also considered several properties that are useful for the learner to satisfy.

A learner M is said to be *consistent* on L iff, for all texts T for L, ctnt$(T[n]) \subseteq W_{\mathrm{M}(T[n])}$. That is, the learner's hypothesis is consistent with the data seen so far. There are three notions of consistency considered in the literature: (a) **TCons**, in which the learner is expected to be consistent on all inputs, irrespective of whether they represent some concept from the target class or not (Wiehagen and Liepe 1976); (b) **Cons**, in which the learner is just expected to be consistent on the languages in the target class being learned, though the learner may be inconsistent or even undefined on the input outside the target class (Bārzdiņš 1974a); and (c) **RCons**, in which the learner is expected to be defined on all inputs, but required to be consistent only on the languages in the target class (Jantke and Beick 1981). It can be shown that $\mathbf{TCons} \subset \mathbf{RCons} \subset \mathbf{Cons} \subset \mathbf{TxtEx}$ (Jantke and Beick 1981; Wiehagen and Liepe 1976; Bārzdiņš 1974a; Wiehagen and Zeugmann 1995).

A learner M is said to be *conservative* (Angluin 1980) if it does not change its mind unless the data contradicts its hypothesis. That is, M conservatively learns L iff, for all texts T for L, if $\mathrm{M}(T[n]) \neq \mathrm{M}(T[n+1])$, then ctnt$(T[n+1]) \not\subseteq W_{\mathrm{M}(T[n])}$. It can be shown that conservativeness is restrictive, that is, there are classes of languages, which can be **TxtEx** identified but not conservatively identified. An example of a class that can be identified explanatorily but not conservatively is the class containing all sets from *SDALL*, that is, the sets of the form $\{e, e+1, e+2, \ldots\}$, and all sets with minimum k_s and up to s elements where $k_0, k_1, k_2, \ldots$ is a recursive one-one enumeration of K. The general idea why this class is not conservatively learnable is that when the learner reads the data $e, e+1, e+2, \ldots$, it will, after some finite time based on data $e, e+1, e+2, \ldots, e+s$, output a conjecture which contains these data plus $e+s+1$; but conservative learning would then imply that $e \in K$ iff $e = k_r$ for some $r \leq s$, contradicting the non-recursiveness of K.

Monotonicity

Related notions to conservativeness are the various notions on monotonic learning that impose certain conditions on whether the previous hypothesis is a subset of the next hypothesis or not. The following notions are the three main ones.

- A learner M is said to be strongly monotonic (Jantke 1991) on L iff, for all texts T for L, $W_{\mathrm{M}(T[n])} \subseteq W_{\mathrm{M}(T[n+1]}$. Intuitively, strong monotonicity requires that the hypothesis of the learner grows with time.
- A learner M is said to be monotonic (Wiehagen 1990) on L iff, for all texts T for L, $W_{\mathrm{M}(T[n])} \cap L \subseteq W_{\mathrm{M}(T[n+1])} \cap L$. In monotonicity, the growth of the hypothesis is required only with respect to the language being learned.
- A learner M is said to be weakly monotonic (Jantke 1991) on L iff, for all texts T for L, if $\mathrm{ctnt}(T[n+1]) \subseteq W_{\mathrm{M}(T[n])}$, then $W_{\mathrm{M}(T[n])} \subseteq W_{\mathrm{M}(T[n+1])}$. That is, the learner behaves strongly monotonically, as long as the input data is consistent with the hypothesis.

An example for a strong monotonically learnable class is the class *SDALL*. When the learner currently conjectures $\{e, e+1, e+2, \ldots\}$ and it sees a datum $d < e$, then it makes a mind change to $\{d, d+1, d+2, \ldots\}$ which is a superset of the previous conjecture; it is easy to see that all mind changes are of this type. It can be shown that strong monotonic learning implies monotonic learning and weak monotonic learning, though monotonic learning and weak monotonic learning are incomparable (and thus both are proper restrictions of **TxtEx** learning). For example, consider the class $\mathcal{C}$ consisting of the set $\{0, 2, 4, \ldots\}$ of all even numbers and, for each n, the set $\{0, 2, 4, \ldots, 2n\} \cup \{2n+1\}$ consisting of the even numbers below $2n$ and the odd number $2n+1$. Then, $\mathcal{C}$ is monotonically but not strong monotonically learnable.

Lange et al. (1992) also considered the dual version of the above criteria, where dual strong monotonicity learning of L requires that, for all texts T for L, $W_{\mathrm{M}(T[n])} \supseteq W_{\mathrm{M}(T[n+1])}$; dual monotonicity requires that, for all texts T for L, $W_{\mathrm{M}(T[n])} \cap (\mathbb{N} - L) \supseteq W_{\mathrm{M}(T[n+1])} \cap (\mathbb{N} - L)$, and dual weak monotonicity requires that, if $\mathrm{ctnt}(T[n+s]) \subseteq W_{\mathrm{M}(T[n])}$, then $W_{\mathrm{M}(T[n])} \supseteq W_{\mathrm{M}(T[n+s])}$.

In a similar fashion, various other properties of learners have been considered. For example, reliability (Blum and Blum 1975; Minicozzi 1976) postulates that the learner does not converge on the input text unless it learns it; prudence (Fulk 1990; Osherson et al. 1986) postulates that the learner outputs only indices of languages, which it also learns; and confidence (Osherson et al. 1986) postulates that the learner converges on every text to some index, even if the text is for some language outside the class of languages to be learned.

Indexed Families

Angluin (1980) initiated a study of learning indexed families of recursive languages. A class of languages (along with its indexing) $L_0, L_1, \ldots$ is an indexed family if membership questions for the languages are uniformly decidable, that is, $x \in L_i$ can be recursively decided in x and i. Angluin gave an important characterization of indexed families that are **TxtEx** learnable.

Suppose a class $\mathcal{L} = \{L_0, L_1, \ldots\}$ (along with the indexing) is given. Then, S is said to be a *tell-tale set* (Angluin 1980) of L_i iff S is finite, and for all j, if $S \subseteq L_j$ and $L_j \subseteq L_i$, then $L_i = L_j$. It can be shown that for any class of languages that are learnable (in **TxtEx** or **TxtBc** sense), there exists a tell-tale set for each language in the class. Moreover, Angluin showed that for indexed families, $\mathcal{L} = L_0, L_1, \ldots$, one can **TxtEx** learn $\mathcal{L}$ iff one can recursively enumerate a tell-tale set for each L_i, effectively from i. Within the framework of learning indexed families, a special emphasis is given to the hypothesis space used; so the following criteria are considered for defining the learnability of a class $\mathcal{L}$ in dependence of the hypothesis space $\mathcal{H} = H_0, H_1, \ldots$. The class $\mathcal{L}$ is

- *Exactly learnable* iff there is a learner using the same hypothesis space as the given class, that is, $H_n = L_n$ for all n;
- *Class-preservingly learnable* iff there is a learner using a hypothesis space $\mathcal{H}$ with $\{L_0, L_1, \ldots\} = \{H_0, H_1, \ldots\}$ – here the order and the number of occurrences in the hypothesis space can differ, but the hypothesis space must consist of the same languages as the class to be learned, and no other languages are allowed in the hypothesis space;
- *Class-comprisingly learnable* iff there is a learner using a hypothesis space $\mathcal{H}$ with $\{L_0, L_1, \ldots\} \subseteq \{H_0, H_1, \ldots\}$ – here the hypothesis space can also contain some further languages not in the class to be learned and the learner does not need to identify these additional languages;
- *Prescribed learnable* iff for every hypothesis space $\mathcal{H}$ containing all the languages from $\mathcal{L}$, there is a learner for $\mathcal{L}$ using this hypothesis space;
- *Uniformly learnable* iff for every hypothesis space $\mathcal{H}$ with index e containing all the languages from $\mathcal{L}$ one can synthesize a learner M_e which succeeds to learn $\mathcal{L}$ using the hypothesis space $\mathcal{H}$.

Note that in all five cases $\mathcal{H}$ only ranges over indexed families. This differs from the standard case where $\mathcal{H}$ is an acceptable numbering of all recursively enumerable sets. We refer the reader to the survey of Lange et al. (2008) for an overview on work done on learning indexed families (**TxtEx** learning, learning under various properties of learners, as well as characterizations of such learning criteria) and to (Jain et al. 2008; Lange and Zeugmann 1993). While for explanatory learning and every class $\mathcal{L}$, all these five notions coincide, these notions turn out to be different for other learning notions like those of conservative learning, monotonic learning, and strong monotonic learning. For example, the class of all finite sets is not prescribed conservatively learnable: one can make an adversary hypothesis space where some indices contain large spurious elements, so that a learner is forced to do nonconservative mind change to obtain correct indices for the finite sets. The same example as above works for showing the limitations of prescribed learning for monotonic and strong monotonic learning.

The interested reader is referred to the textbook *Systems that Learn* (Jain et al. 1999; Osherson et al. 1986) and the papers below as well as the references found in these papers for further reading. Complexity issues in inductive inference like the number of mind changes necessary to learn a class or oracles needed to learn some class can be found under the entries *Computational Complexity of Learning* and *Query-Based Learning*. The entry *Connections between Inductive Inference and Machine Learning* provides further information on this topic.

Cross-References

▸ Connections Between Inductive Inference and Machine Learning

Acknowledgements Sanjay Jain was supported in part by NUS grant numbers C252-000-087-001, R146-000-181-112, R252-000-534-112. Frank Stephen was supported in part by NUS grant numbers R146-000-181-112, R252-000-534-112.

Recommended Reading

Angluin D (1980) Inductive inference of formal languages from positive data. Inf Control 45:117–135

Bārzdiņš J (1974a) Inductive inference of automata, functions and programs. In: Proceedings of the international congress of mathematics, Vancouver, pp 771–776
Bārzdiņš J (1974b) Two theorems on the limiting synthesis of functions. In: Theory of algorithms and programs, vol 1. Latvian State University, Riga, pp 82–88 (In Russian)
Blum L, Blum M (1975) Toward a mathematical theory of inductive inference. Inf Control 28:125–155
Case J (1999) The power of vacillation in language learning. SIAM J Comput 28:1941–1969
Case J, Lynes C (1982) Machine inductive inference and language identification. In: Nielsen M, Schmidt EM (eds) Proceedings of the 9th international colloquium on automata, languages and programming. Lecture notes in computer science, vol 140. Springer, Heidelberg, pp 107–115
Case J, Smith C (1983) Comparison of identification criteria for machine inductive inference. Theor Comput Sci 25:193–220
Fulk M (1990) Prudence and other conditions on formal language learning. Inf Comput 85:1–11
Gold EM (1967) Language identification in the limit. Inf Control 10:447–474
Jain S, Osherson D, Royer J, Sharma A (1999) Systems that learn: an introduction to learning theory, 2nd edn. MIT Press, Cambridge
Jain S, Stephan F, Ye N (2008) Prescribed learning of indexed families. Fundam Inf 83:159–175
Jantke KP (1991) Monotonic and non-monotonic inductive inference. New Gener Comput 8:349–360
Jantke KP, Beick H-R (1981) Combining postulates of naturalness in inductive inference. J Inf Process Cybern (EIK) 17:465–484
Lange S, Zeugmann T (1993) Language learning in dependence on the space of hypotheses. In: Proceedings of the sixth annual conference on computational learning theory, Santa Cruz, pp 127–136
Lange S, Zeugmann T, Kapur S (1992) Class preserving monotonic language learning. Technical report 14/92, GOSLER-Report, FB Mathematik und Informatik, TH Leipzig
Lange S, Zeugmann T, Zilles S (2008). Learning indexed families of recursive languages from positive data: a survey. Theor Comput Sci 397: 194–232
Minicozzi E (1976) Some natural properties of strong identification in inductive inference. Theor Comput Sci 2:345–360
Osherson D, Weinstein S (1982) Criteria of language learning. Inf Control 52:123–138
Osherson D, Stob M, Weinstein S (1986) Systems that learn, an introduction to learning theory for cognitive and computer scientists. Bradford–The MIT Press, Cambridge
Sharma A (1998) A note on batch and incremental learnability. J Comput Syst Sci 56:272–276
Wiehagen R (1990) A thesis in inductive inference. In: Dix J, Jantke K, Schmitt P (eds) Nonmonotonic and inductive logic, 1st international workshop. Lecture notes in artificial intelligence, vol 543. Springer, Berlin, pp 184–207
Wiehagen R, Liepe W (1976) Charakteristische Eigenschaften von erkennbaren Klassen rekursiver Funktionen. J Inf Process Cybern (EIK) 12:421–438
Wiehagen R, Zeugmann T (1995) Learning and consistency. In: Jantke KP, Lange S (eds) Algorithmic learning for knowledge-based systems (GOSLER), final report. Lecture notes in artificial intelligence, vol 961. Springer, Heidelberg, pp 1–24

Inductive Inference Rules

► Logic of Generality

Inductive Learning

Synonyms

Statistical learning

Definition

Inductive learning is a subclass of machine learning that studies algorithms for learning knowledge based on statistical regularities. The learned knowledge typically has no deductive guarantees of correctness, though there may be statistical forms of guarantees.

Inductive Logic Programming

Luc De Raedt
Department of Computer Science, Katholieke Universiteit Leuven, Heverlee, Leuven, Belgium

Abstract

Inductive logic programming is the subfield of machine learning that uses ► First-Order Logic to represent hypotheses and data.

Because first-order logic is expressive and declarative, inductive logic programming specifically targets problems involving structured data and background knowledge. Inductive logic programming tackles a wide variety of problems in machine learning, including classification, regression, clustering, and reinforcement learning, often using "upgrades" of existing propositional machine learning systems. It relies on logic for knowledge representation and reasoning purposes. Notions of coverage, generality, and operators for traversing the space of hypotheses are grounded in logic; see also ▶Logic of Generality. Inductive logic programming systems have been applied to important applications in bio- and chemo-informatics, natural language processing, and web mining.

Synonyms

Learning in logic; Multi-relational data mining; Relational data mining; Relational learning

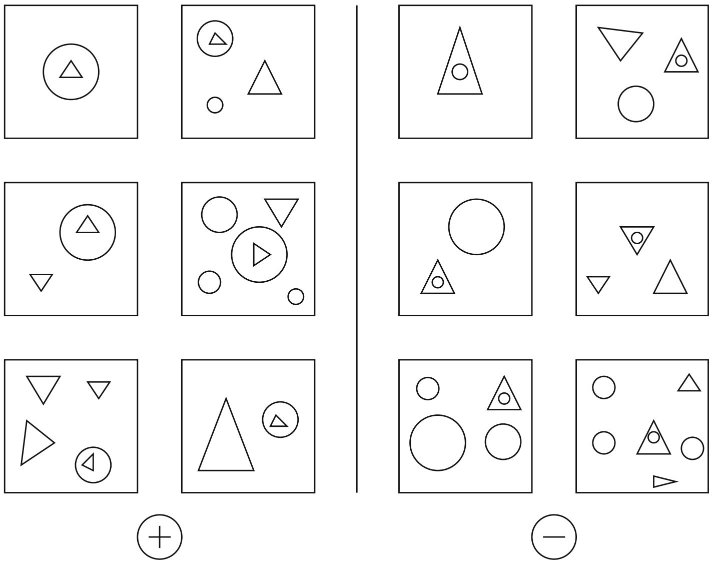
Motivation

The first motivation and most important motivation for using inductive logic programming is that it overcomes the representational limitations of attribute-value learning systems. Such systems employ a table-based representations where the instances correspond to rows in the table, the attributes to columns, and for each instance, a single value is assigned to each of the attributes. This is sometimes called the *single-table single-tuple* assumption. Many problems, such as the Bongard problem shown in Fig. 1, cannot elegantly be described in this format. Bongard (1970) introduced about a hundred concept learning or pattern recognition problems, each containing six positive and six negative examples. Even though Bongard problems are toy problems, they are similar to real-life problems such as structure–activity relationship prediction, where the goal is to learn to predict whether a given molecule (as represented by its 2D graph structure) is active or not. It is hard – if not, impossible – to squeeze this type of problem into the single-table single-tuple format for various reasons. Attribute-value learning systems employ a fixed number of attributes and also assume that these attributes are present in all of the examples. This assumption does not hold for the Bongard problems as the examples possess a variable number of objects (shapes). The singe-table single-tuple representation imposes an implicit order on the attributes, whereas there is no natural order on the objects in the Bongard problem. Finally, the relationships between the objects in the Bongard problem are essential and must be encoded as well. It is unclear how to do this within the single-table single-tuple assumption. First-order logic and relational representations allow one to encode problems involving multiple objects (or entities) as well as the relationships that hold them in a natural way.

The second motivation for using inductive logic programming is that it employs logic, a declarative representation. This implies that hypotheses are understandable and interpretable. By using logic, inductive logic programming systems are also able to employ background knowledge in the induction process. Background knowledge can be provided in the form of definitions of auxiliary relations or predicates that may be used by the learner. Finally, logic provides a well-understood theoretical framework for knowledge representation and reasoning. This framework is also useful for machine learning, in particular, for defining and developing notions such as the covers relation, generality, and refinement operators; see also ▶Logic of Generality.

Theory

Inductive logic programming is usually defined as concept learning using logical representations. It aims at finding a hypothesis (a set of rules) that covers all positive examples and none of the negatives, while taking into account a background theory. This is typically realized by searching a



Inductive Logic Programming, Fig. 1 A complex classification problem: Bongard problem 47, developed by the Russian scientist Bongard (1970). It consists of 12 scenes (or examples), 6 of class ⊕ and 6 of class ⊖. The goal is to discriminate between the two classes

space of possible hypotheses. More formally, the traditional inductive logic programming definition reads as follows:

Given

- A language describing hypotheses $\mathcal{L}_h$
- A language describing instances $\mathcal{L}_i$
- Possibly a background theory B, usually in the form of a set of (definite) clauses
- The *covers* relation that specifies the relation between $\mathcal{L}_h$ and $\mathcal{L}_i$, that is, when an example e is covered (considered positive) by a hypothesis h, possibly taking into account the background theory B
- A set of positive and negative examples $E = P \cup N$

Find a hypothesis $h \in \mathcal{L}_h$ such that for all $p \in P : covers(B, h, p) = true$ and for all $n \in N : covers(B, h, n) = false$.

This definition can, as for ▸ Concept-Learning in general, be extended to cope with noisy data by relaxing the requirement that all examples be classified correctly.

There exist different ways to represent learning problems in logic, resulting in different learning settings. They typically use definite clause logic as the hypothesis language $\mathcal{L}_i$ but differ in the notion of an example. One can learn from entailment, from interpretations, or from proofs, cf. ▸ Logic of Generality. The most popular setting is *learning from entailment*, where each example is a clause and $covers(B, h, e) = true$ if and only if $B \cup h \models e$.

The top leftmost scene in the Bongard problem of Fig. 1 can be represented by the clause:

```
positive :- object(o1),
            object(o2),
            circle(o1),
            triangle(o2),
            in(o1, o2),
            large(o2).
```

The other scenes can be encoded in the same way. The following hypothesis then forms a solution to the learning problem:

```
positive :- object(X),
            object(Y),
            circle(X),
            triangle(Y),
            in(X,Y).
```

It states that those scenes having a circle inside a triangle are positive. For some more complex Bongard problems, it could be useful to employ background knowledge. It could, for instance, state that triangles are polygons.

```
 polygon(X) :- triangle(X).
```

Using this clause as background theory, an alternative hypothesis covering all positives and none of the negatives is

```
positive :- object(X),
            object(Y),
            circle(X),
            polygon(Y),
            in(X,Y).
```

An alternative for using long clauses as examples is to provide an identifier for each example and to add the corresponding facts from the condition part of the clause to the background theory. For the above example, the facts such as

```
object(e1,o1).
object(e1,o2).
circle(e1,o1).
triangle(e1,o2).
in(e1,o1,o2).
large(e1,o2).
```

would be added to the background theory, and the positive example itself would then be represented through the fact `positive(e1)`, where `e1` is the identifier. The inductive logic programming literature typically employs this format for examples and hypotheses.

Whereas inductive logic programming originally focused on concept learning – as did the whole field of machine learning – it is now being applied to virtually all types of machine learning problems, including regression, clustering, distance-based learning, frequent pattern mining, reinforcement learning, and even kernel methods and graphical models.

A Methodology

Many of the more recently developed inductive logic programming systems have started from an existing attribute-value learner and have upgraded it toward the use of first-order logic (Van Laer and De Raedt 2001). By examining state-of-the-art inductive logic programming systems, one can identify a methodology for realizing this (Van Laer and De Raedt 2001). It starts from an attribute-value learning problem and system of interest and takes the following two steps. First, the problem setting is upgraded by changing the representation of the examples, the hypotheses as well as the covers relation toward first-order logic. This step is essentially concerned with defining the learning setting, and possible settings to be considered include the already mentioned learning from *entailment*, *interpretations*, and *proofs* settings. Once the problem is clearly defined, one can attempt to formulate a solution. Thus, the second step adapts the original algorithm to deal with the upgraded representations. While doing so, it is advisable to keep the changes as minimal as possible. This step often involves the modification of the operators used to traverse the search space. Different operators for realizing this are introduced in the entry on the ▸ Logic of Generality.

There are many reasons why following the methodology is advantageous. First, by upgrading a learner that is already effective for attribute-value representations, one can benefit from the experiences and results obtained in the propositional setting. In many cases, for instance, decision trees, this implies that one can rely on well-established methods and findings, which are the outcomes of several decades of machine learning research. It will be hard to do better starting from scratch. Second, upgrading an existing learner is also easier than starting from scratch as many of the components (such as heuristics and search strategy) can be recycled. It is therefore also economic in terms of man power. Third, the upgraded system will be able to emulate the original one, which provides guarantees that the output hypotheses will perform well on attribute-value learning problems. Even more important is that it will often also be able to emulate extensions of the original systems. For instance, many systems that extend frequent item-set mining toward using richer representations, such as sequences, intervals, the use of taxonomies, graphs, and so on, have been developed over the past decade. Many of them can be emulated using the inductive logic programming upgrade of Apriori (Agrawal et al. 1996) called Warmr (Dehaspe and Toivonen 2001). The upgraded inductive logic programming systems will typically be more flexible than the systems it can emulate but typically also less efficient because there is a price to be paid for expressiveness. Finally, it may be possible to incorporate new features in the attribute-value learner by following the methodology. One feature that is often absent from propositional learners and may be easy to incorporate is the use of a background theory.

It should be mentioned that the methodology is not universal, that is, there exist also approaches, such as Muggleton's Progol (1995), which have directly been developed in first-order logic and for which no propositional counterpart exists. In such cases, however, it can be interesting to follow the inverse methodology, which would specialize the inductive logic programming system.

FOIL: An Illustration

One of the simplest and best-known inductive logic programming systems is FOIL (Quinlan 1990). It can be regarded as an upgrade of a rule learner such as CN2 (Clark and Niblett 1989). FOIL's problem setting is an instance of the learning from entailment setting introduced above (though it restricts the background theory to ground facts only and does not allow functors).

Like most rule-learning systems, FOIL employs a separate-and-conquer approach. It starts from the empty hypothesis, and then repeatedly searches for one rule that covers as many positive examples as possible and no negative example, adds it to the hypothesis, removes the positives covered by the rule, and then iterates. This process is continued until all positives are covered. To find one rule, it performs a hill-climbing search through the space of clauses ordered according to generality. The search starts at the most general rule, the one stating that all examples are positive, and then repeatedly specializes it. Among the specializations, it then selects the best one according to a heuristic evaluation based on information gain. A heuristic, based on the minimum description length principle, is then used to decide when to stop specializing clauses.

The key differences between FOIL and its propositional predecessors are the representation and the operators used to compute the specializations of a clause. It employs a refinement operator under θ-subsumption (Plotkin 1970) (see also ▸Logic of Generality). Such an operator essentially refines clauses by adding atoms to the condition part of the clause or applying substitutions to a clause. For instance, the clause

```
positive :- triangle(X),
            in(X,Y),
            color(X,C).
```

can be specialized to

```
positive :- triangle(X),
            in(X,Y),
            color(X,red).



positive :- triangle(X),
            in(X,Y),
            color(X,C),
            large(X).
positive :- triangle(X),
            in(X,Y),
            color(X,C),
            rectangle(Y).
  ...
```

The first specialization is obtained by substituting the variable `C` by the constant `red`, the other two by adding an atom (`large(X)`, `rectangle(Y)`, respectively) to the condition part of the rule. Inductive logic programming systems typically also employ syntactic restrictions – the so-called – that specify which clauses may be used in hypotheses. For instance, in the above example, the second argument of the `color` predicate belongs to the type *Color*, whereas the arguments of `in` are of type *Object* and consist of object identifiers.

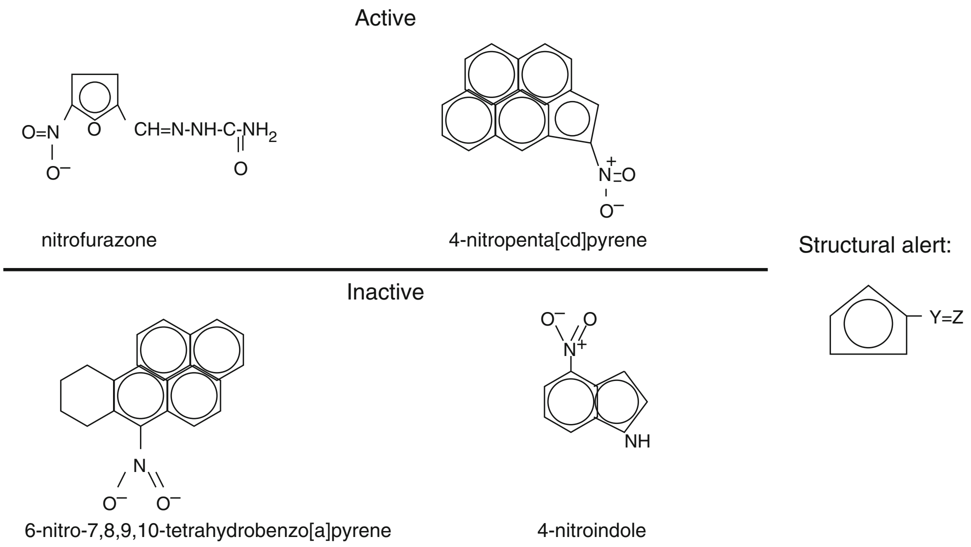
Application

Inductive logic programming has been successfully applied to many application domains, including bio- and chemo-informatics, ecology, network mining, software engineering, information retrieval, music analysis, web mining, natural language processing, toxicology, robotics, program synthesis, design, architecture, and many others. The best-known applications are in scientific domains. For instance, in structure–activity relationship prediction, one is given a set of molecules together with their activities, and background knowledge encoding functional groups, that is particular components of the molecule, and the task is to learn rules stating when a molecule is active or inactive. This is illustrated in Fig. 2 (after Srinivasan et al. 1996), where two molecules are active and two are inactive. One then has to find a pattern that discriminates the actives from the inactives. Structure–activity relationship (SAR) prediction is an essential step in, for instance, drug discovery. Using the general purpose inductive logic programming system Progol (Muggleton 1995) *structural alerts*, such as that shown in Fig. 2, have been discovered. These alerts allow one to distinguish the actives from the inactives – the one shown in the figure matches both of the actives but none of the inactives – and at the same time they are readily interpretable and provide useful insight into the factors determining the activity. To solve structure–activity relationship prediction problems using inductive logic programming, one must represent the molecules and hypotheses using the logical formalisms introduced above. The resulting representation is very similar to that employed in the Bongard problems: the objects are the atoms and relationships the bonds. Particular functional groups are encoded as background predicates.

State-of-the-Art

The upgrading methodology has been applied to a wide variety of machine learning systems and problems. There exist now inductive logic programming systems that:

- Induce logic programs from examples under various learning settings. This is by far the most popular class of inductive logic programming systems. Well-known systems include Aleph (Srinivasan 2007) and Progol (Muggleton 1995) as well as various variants of FOIL (Quinlan 1990). Some of these systems, especially Progol and Aleph, contain many features that are not present in propositional learning systems. Most of these systems focus



Inductive Logic Programming, Fig. 2 Predicting mutagenicity (Srinivasan et al. 1996) on a classification setting and learn the definition of a *single* predicate.

- Induce logical decision trees from examples. These are binary decision trees containing conjunctions of atoms (i.e., queries) as tests. If a query succeeds, then one branch is taken, else the other one. Decision tree methods for both classification and regression exist (see Blockeel and De Raedt 1998; Kramer and Widmer 2001).
- Mine for frequent queries, where queries are conjunctions of atoms. Such queries can be evaluated on an example. For instance, in the Bongard problem, the query `?- triangle (X), in (X, Y)` succeeds on the leftmost scenes and fails on the rightmost ones. Therefore, its frequency would be 6. The goal is then to find all queries that are frequent, that is, whose frequencies exceed a certain threshold. Frequent query mining upgrades the popular local pattern mining setting due to Agrawal et al. (1996) to inductive logic programming (see Dehaspe and Toivonen 2001).
- Learn or revise the definitions of theories, which consist of the definitions of multiple predicates, at the same time (cf. Wrobel 1996), and the entry in this encyclopedia. Several of these systems have their origin in the model inference system by Shapiro (1983) or the work by Angluin (1987).

Current Trends and Challenges

There are two major trends and challenges in inductive logic programming. The first challenge is to extend the inductive logic programming paradigm beyond the purely symbolic one. Important trends in this regard include:

- The combination of inductive logic programming principles with graphical and probabilistic models for reasoning about uncertainty. This is a field known as *statistical relational learning*, *probabilistic logic learning*, or *probabilistic inductive logic programming*. At the time of writing, this is a very popular research stream, attracting a lot of attention in the wider artificial intelligence community, cf. the entry ▸ Statistical Relational Learning in this encyclopedia. It has resulted in many relational or logical upgrades of well-known graphical models including Bayesian networks, Markov networks, hidden Markov models, and stochastic grammars.
- The use of relational distance measures for classification and clustering (Ramon and Bruynooghe 1998; Kirsten et al. 2001). These distances measure the similarity between two examples or clauses, while taking into account the underlying structure of the instances. These distances are then combined with standard classification and clustering methods such as k-nearest neighbor and k-means.
- The integration of relational or logical representations in reinforcement learning, known as ▸ Relational Reinforcement Learning (Dzeroski et al. 2001).

The power of inductive logic programming is also its weakness. The ability to represent complex objects and relations and the ability to make use of background knowledge add to the computational complexity. Therefore, a key challenge of inductive logic programming is tackling this added computational complexity. Even the simplest method for testing whether one hypothesis is more general than another – that is, θ-subsumption (Plotkin 1970) – is NP-complete. Similar tests are used for deciding whether a clause covers a particular example in systems such as FOIL. Therefore, inductive logic programming and relational learning systems are computationally much more expensive than their propositional counterparts. This is an instance of the expressiveness versus efficiency trade-off in computer science. Because of these computational difficulties, inductive logic programming has devoted a lot of attention to efficiency issues. On the theoretical side, there exist various results about the polynomial learnability of certain subclasses of logic programs (cf. Cohen and Page 1995, for an overview). From a practical perspective, there is quite some work on developing efficient methods for searching the hypothesis space and especially for evaluating the quality of hypotheses. Many of these methods employ optimized inference engines based on Prolog or database technology or constraint satisfaction methods (cf. Blockeel and Sebag 2003 for an overview).

Cross-References

▶ Multi-Relational Data Mining

Recommended Reading

A comprehensive introduction to inductive logic programming can be found in the book by De Raedt (2008) on logical and relational learning. Early surveys of inductive logic programming are contained in Muggleton and De Raedt (1994) and Lavrač and Džeroski (1994) and an account of its early history is provided in Sammut (1993). More recent collections on current trends can be found in the proceedings of the annual *Inductive Logic Programming Conference* (published in Springer's *Lectures Notes in Computer Science Series*) and special issues of the *Machine Learning Journal*. A summary of some key future challenges is given in Muggleton et al. (2012). An interesting collection of inductive logic programming and multi-relational data mining works are provided in Džeroski and Lavrač (2001). The upgrading methodology is described in detail in Van Laer and De Raedt (2001). More information on logical issues in inductive logic programming are given in the entry ▶ Logic of Generality in this encyclopedia, whereas the entries ▶ Statistical Relational Learning and ▶ Graph Mining are recommended for those interested in frameworks tackling similar problems using other types of representations.

Agrawal R, Mannila H, Srikant R, Toivonen H, Verkamo AI (1996) Fast discovery of association rules. In: Fayyad U, Piatetsky-Shapiro G, Smyth P, Uthurusamy R (eds) Advances in knowledge discovery and data mining. MIT Press, Cambridge, pp 307–328

Angluin D (1987) Queries and concept-learning. Mach Learn 2:319–342

Blockeel H, De Raedt L (1998) Top-down induction of first order logical decision trees. Artif Intell 101(1–2):285–297

Blockeel H, Sebag M (2003) Scalability and efficiency in multi-relational data mining. SIGKDD Explor 5(1):17–30

Bongard M (1970) Pattern recognition. Spartan Books, New York

Clark P, Niblett T (1989) The CN2 algorithm. Mach Learn 3(4):261–284

Cohen WW, Page D (1995) Polynomial learnability and inductive logic programming: methods and results. New Gener Comput 13:369–409

De Raedt L (2008) Logical and relational learning. Springer, Berlin

Dehaspe L, Toivonen H (2001) Discovery of relational association rules. In: Džeroski S, Lavrač N (eds) Relational data mining. Springer, Berlin/Heidelberg, pp 189–212

Džeroski S, De Raedt L, Driessens K (2001) Relational reinforcement learning. Mach Learn 43(1/2): 5–52

Džeroski S, Lavrač N (eds) (2001) Relational data mining. Springer, Berlin/New York

Kirsten M, Wrobel S, Horvath T (2001) Distance based approaches to relational learning and clustering. In: Džeroski S, Lavrač N (eds) Relational data mining. Springer, Berlin/Heidelberg, pp 213–232

Kramer S, Widmer G (2001) Inducing classification and regression trees in first order logic. In: Džeroski S, Lavrač N (eds) Relational data mining. Springer, Berlin/Heidelberg, pp 140–159

Lavrač N, Džeroski S (1994) Inductive logic programming: techniques and applications. Ellis Horwood, Chichester

Muggleton S (1995) Inverse entailment and Progol. New Gener Comput 13:245–286

Muggleton S, De Raedt L (1994) Inductive logic programming: theory and methods. J Log Program 19(20):629–679

Muggleton S, De Raedt L, Poole D, Bratko I, Flach P, Inoue K, Srinivasan A (2012) ILP Turns 20. Mach Learn 86:2–23

Plotkin GD (1970) A note on inductive generalization. In: Machine intelligence, vol 5. Edinburgh University Press, Edinburgh, pp 153–163

Quinlan JR (1990) Learning logical definitions from relations. Mach Learn 5:239–266

Ramon J, Bruynooghe M (1998) A framework for defining distances between first-order logic objects. In: Page D (ed) Proceedings of the eighth international conference on inductive logic programming. Lecture notes in artificial intelligence, vol 1446. Springer, Berlin/Heidelberg, pp 271–280

Sammut C (1993) The origins of inductive logic programming: a prehistoric tale. In: Muggleton S (ed) Proceedings of the third international workshop on inductive logic programming. J. Stefan Institute, Ljubljana, pp 127–148

Shapiro EY (1983) Algorithmic program debugging. MIT Press, Cambridge

Srinivasan A (2007) The Aleph Manual. http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph_toc.html

Srinivasan A, Muggleton S, Sternberg MJE, King RD (1996) Theories for mutagenicity: a study in first-order and feature-based induction. Artif Intell 85(1/2):277–299

Van Laer W, De Raedt L (2001) How to upgrade propositional learners to first order logic: a case study. In: Džeroski S, Lavrač N (eds) Relational data mining. Springer, Berlin/Heidelberg, pp 235–261

Wrobel S (1996) First-order theory refinement. In: De Raedt L (ed) Advances in inductive logic programming. Frontiers in artificial intelligence and applications, vol 32. IOS Press, Amsterdam, pp 14–33

Inductive Process Modeling

Ljupčo Todorovski
University of Ljubljana, Ljubljana, Slovenia

Synonyms

Process-based modeling

Definition

Inductive process modeling is a machine learning task that deals with the problem of learning quantitative *process models* from ▸ time series data about the behavior of an observed dynamic system. Process models are models based on ordinary differential equations that add an explanatory layer to the equations. Namely, scientists and engineers use models to both predict and explain the behavior of an observed system. In many domains, models commonly refer to processes that govern system dynamics and entities altered by those processes. Ordinary differential equations, often used to cast models of dynamic systems, offer one way to represent these mechanisms and can be used to simulate and predict the system behavior, but fail to make the processes and entities explicit. In response, process models tie the explanatory information about processes and entities to the mathematical formulation, based on equations, that enables simulation.

Inductive Process Modeling, Table 1 A process model of Predatory–Prey interaction between foxes and rabbits. The notation $d[X, t]$ indicates the time derivative of variable X

model predation;
entities fox{population}, rabbit{population};
process rabbit_growth;
entites rabbit;
*equations d[rabbit.conc,t] = 1.81 * rabbit.conc **
*(1 − 0.0003 * rabbit.conc);*
process fox_death;
entites fox;
*equations d[fox.conc,t] = −1.04 * fox.conc;*
process fox_rabbit_predation;
entities fox, rabbit;
equations
*d[fox.conc,t] = 0.03 * rabbit.conc * fox.conc;*
*d[rabbit.conc,t] = −1 * 0.3 * rabbit.conc * fox.conc;*

Table 1 shows a process model for a predator–prey interaction between foxes and rabbits. The three processes explain the dynamic change of the concentrations of both species (represented in the model as two *population* entities) through time. The *rabbit_growth* process states that the reproduction of rabbit is limited by the fixed environmental capacity. Similarly, the *fox_death* process specifies an unlimited exponential mortality function for the fox population. Finally, the *fox_rabbit_predation* process refers to the predator–prey interaction between foxes and rabbits that states that the prey concentration decreases and the predator one increases proportionally with the sizes of the two populations. The process model makes the structure of the model explicit and transparent to scientists; while at the same time it can be easily transformed in to a system of two differential equations by additively combining the equations for the time derivatives of the system variables *fox.conc* and *rabbit.conc*. Given initial values for these variables, one can simulate the equations to produce trajectories that correspond to the population dynamics through time.

The processes from Table 1 instantiate more general generic processes, that can be used for modeling any ecological system. For example: is a general form of the *fox_rabbit_predation* process from the example model in Table 1. Note that in the generic process, the parameters are replaced with numeric ranges and the entities with identifiers of generic entities (i.e., *Predator* and *Prey* are identifiers that refer to instances of the generic entity *population*).

```
generic process predation;
  entities Predator{population}, Prey{population};
  parameters ar[0.01, 10], ef[0.001, 0.8];
  equations
        d[Predator.conc,t] = ef * ar * Prey.conc *
Predator.conc;
        d[Prey.conc,t] = −1 * ar * Prey.conc * Preda-
tor.conc;
```

Having defined entities and processes on an example, one can define the task of inductive process modeling as: Given

- Time series observations for a set of numeric system variables as they change through time
- A set of entities that the model might include
- Generic processes that specify casual relations among entities
- Constraints that determine plausible relations among processes and entities in the model

Find a specific process model that explains the observed data and the simulation of which closely matches observed time series.

There are two approaches for solving the task of inductive process modeling. The first is the transformational approach that transforms the given knowledge about entities, processes, and constraints to ▸ language bias for equation discovery and uses the Lagramge method for ▸ equation discovery in turn (Todorovski and Džeroski 1997, 2007). The second approach performs search through the space of candidate process models to find the one that matches the given time series data best.

Inductive process modeling methods IPM (Bridewell et al. 2008) and HIPM (Todorovski et al. 2005) follow the second approach. IPM is a naïve method that exhaustively searches the space of candidate process models following the ▸ learning as search paradigm. The search space of candidate process models is defined by the sets of generic processes and of entities in the observed system specified by the user. IPM first matches the type of each entity against the types of entities involved in each generic process and produces a list of all possible instances of that generic process. For example, the generic process *predation*, from the example above, given two *population* entities *fox* and *rabbit*, can be instantiated in four different ways (*fox_fox_predation*, *fox_rabbit_predation*, *rabbit_fox_predation*, and *rabbit_rabbit_predation*). The IPM search procedure collects the set of all possible instances of all the generic processes and uses them as a set of candidate model components. In the search phase, all combinations of these model components are being matched against observed ▸ time series. The matching involves the employment of gradient-descent methods for nonlinear optimization to estimate the optimal values of the process model parameters. As output, IPM reports the process models with the best match.

Trying out all components' combinations is prohibitive in many situations since it obviously leads to combinatorial explosion. HIPM employs constraints that limit the space of combinations by ruling-out implausible or forbidden combinations. Examples of such constraints in the predator–prey example above include rules that a proper process model of population dynamics should include a single growth and a single mortality process per species, the predator–prey process should relate two different species, and different predator–prey interaction should refer to different population pairs. HIPM specifies the rules in a hierarchy of generic processes where each node in the hierarchy specifies a rule for proper combination/selection of process instances.

Cross-References

▸ Equation Discovery

Recommended Reading

Bridewell W, Langley P, Todorovski L, Džeroski S (2008) Inductive process modeling. Mach Learn 71(1):1–32
Todorovski L, Džeroski S (1997) Declarative bias in equation discovery. In: Fisher DH (ed) Proceedings of the fourteenth international conference on machine learning,Nashville
Todorovski L, Džeroski S (2007) Integrating domain knowledge in equation discovery. In: Džeroski S, Todorovski L (eds) Computational discovery of scientific knowledge. LNCS, vol 4660. Springer, Berlin
Todorovski L, Bridewell W, Shiran O, Langley P (2005) Inducing hierarchical process models in dynamic domains. In: Veloso MM, Kambhampati S (eds) Proceedings of the twentieth national conference on artificial intelligence, Pittsburgh

Inductive Program Synthesis

▸ Inductive Programming

Inductive Programming

Pierre Flener[1] and Ute Schmid[2]
[1]Department of Information Technology, Uppsala University, Uppsala, Sweden
[2]Faculty of Information Systems and Applied Computer Science, University of Bamberg, Bamberg, Germany

Abstract

Inductive programming is introduced as a branch of program synthesis which is based on inductive inferece where recursive, declarative programs are constructed from incomplete specifications, especially from input/output examples. Inductive logic programming as well as inductive functional programming are addressed. Central concepts such as predicate invention and background knowledge are defined. Two worked-out examples are presented to illustrate inductive logic as well as inductive functional programming.

Most of the work by this author was done while on leave of absence in 2006/07 as a Visiting Faculty Member and Erasmus Exchange Teacher at Sabancı University, Turkey.

Synonyms

Example-based programming; Inductive program synthesis; Inductive synthesis; Programming by examples; Program synthesis from examples

Definition

Inductive programming is the inference of an algorithm or program featuring recursive calls or repetition control structures, starting from information that is known to be incomplete, called the *evidence*, such as positive and negative input-output examples or clausal constraints. The inferred program must be correct with respect to the provided evidence, in a **generalization** sense: it should be neither equivalent to it nor inconsistent. Inductive programming is guided explicitly or implicitly by a **language bias** and a **search bias**. The inference may draw on background knowledge or query an oracle. In addition to **induction**, **abduction** may be used. The restriction to algorithms and programs featuring recursive calls or repetition control structures distinguishes inductive programming from **concept learning** or **classification**.

We here restrict ourselves to the inference of declarative programs, whether functional or logic, and dispense with repetition control structures in the inferred program in favor of recursive calls.

Motivation and Background

Inductive program synthesis is a branch of the field of *program synthesis*, which addresses a cognitive question as old as computers, namely, the understanding of the human act of computer programming, to the point where a computer can be made to help in this task (and ultimately to enhance itself). See Flener (2002) and Gulwani et al. (2014) for surveys; the other main branches of program synthesis are based on deductive inference, namely, *constructive program synthesis* and *transformational program synthesis*. In such *deductive program synthesis*, the provided information, called the *specification*, is assumed to be complete (in contrast to inductive program synthesis where the provided information is known to be incomplete), and the presence of repetitive or recursive control structures in the synthesized program is not imposed.

Research on the inductive synthesis of recursive *functional* programs started in the early 1970s and was brought onto firm theoretical foundations with the seminal THESYS system of Summers (1977) and work of Biermann (1978), where all the evidence is handled non-**incrementally**. Essentially, the idea is first to infer computation *traces* from input-output examples (**instances**) and then to use a **trace-based programming** method to fold these traces into a recursive program. The main results until the mid-1980s were surveyed in Smith (1984). Due to limited progress with respect to the range of programs that could be synthesized, research activities decreased significantly in the next decades. However, a new approach that formalizes functional program synthesis in the term rewriting framework and that allows the synthesis of a broader class of programs than the classical approaches is pursued in Kitzelmann and Schmid (2006).

The advent of *logic* programming brought a new lan but also a new direction in the early 1980s, especially due to the MIS system of Shapiro (1983), eventually spawning the new field of **inductive logic programming** (ILP). Most of this ILP work addresses a wider class of problems, as the focus is *not* only on recursive logic programs: more adequate designations are inductive **theory revision** and *declarative program debugging*, as an additional input is a possibly empty initial theory or program that is **incrementally** revised or debugged according to each newly presented piece of evidence, possibly in the presence of background knowledge or an oracle. The main results on the inductive synthesis of recursive logic programs were surveyed in Flener and Yılmaz (1999).

Structure of Learning System

The core of an inductive programming system is a mechanism for constructing a recursive **generalization** for a set of input/output examples (**instances**). Although we use the vocabulary of logic programming, this method also covers the synthesis of functional programs.

The input, often a set of input/output examples, is called the *evidence*. Further evidence may be queried from an *oracle*. Additional information, in the form of predicate symbols that can be used during the synthesis, can be provided as *background knowledge*. Since the **hypothesis space** – the set of legal recursive programs – is infinite, a **language bias** is introduced. One particularly useful and common approach in inductive programming is to provide a statement bias by means of a *program schema*.

The evidential synthesis of a recursive program starts from the provided evidence for some predicate symbol and works essentially as follows. A program schema is chosen to provide a template for the program structure, where all yet undefined predicate symbols must be instantiated during the synthesis. Predefined predicate symbols of the background knowledge are then chosen for some of these undefined predicate symbols in the template. If it is deemed that the remaining undefined predicate symbols cannot all be instantiated via purely structural generalization by non-recursive definitions, then the method is recursively called to infer recursive definitions for some of them (this is called **predicate invention** and amounts to shifting the *vocabulary bias*); otherwise the synthesis ends successfully right away. This generic method can backtrack to any choice point for synthesizing alternative programs.

In the rest of this section, we discuss this basic terminology of inductive programming more precisely. In the next section, instantiations of this generic method by some well-known methods are presented.

The Evidence and the Oracle

The evidence is often limited to ground positive examples of the predicate symbols that are to be defined. Ground negative examples are convenient to prevent overgeneralization, but should be used constructively and not just to reject candidate programs. A useful generalization of ground examples is evidence in the form of a set of (nonrecursive) clauses, as variables and additional predicate symbols can then be used.

Example 1 The $delOdds(L, R)$ relation, which holds if and only if R is the integer list L without its odd elements, can be incompletely described by the following clausal evidence:

$$\begin{aligned} delOdds([\,],[\,]) &\leftarrow true \\ delOdds([X],[\,]) &\leftarrow odd(X) \\ delOdds([X],[X]) &\leftarrow \neg odd(X) \\ delOdds([X,Y],[Y]) &\leftarrow odd(X),\ \neg odd(Y) \\ delOdds([X,Y],[X,Y]) &\leftarrow \neg odd(X),\ \neg odd(Y) \\ false &\leftarrow delOdds([X],[X]),\ odd(X) \end{aligned} \tag{1}$$

The first clause is a ground positive example, whereas the second and third clauses generalize the infinity of ground positive examples, such as $delOdds([5],[\,])$ and $delOdds([22],[22])$, for handling singleton lists, while the fourth and fifth clauses summarize the infinity of ground positive examples for handling lists of two elements, the second one being even: these clauses make explicit the underlying filtering relation (*odd*) that is *intrinsic* to the problem at hand but cannot be provided via ground examples and would otherwise have to be guessed. The sixth clause summarizes an infinity of ground negative examples for handling singleton lists, namely, where the only element of the list is odd but not filtered.

In some methods, especially for the induction of functional programs, the first n positive input-output examples with respect to the underlying data type are presented (e.g., for linear lists, what to do with the empty list, with a one-element list, up to a list with three elements); because of this ordering of examples, no explicit presentation of negative examples is then necessary.

Inductive program synthesis should be monotonic in the evidence (more evidence should never yield a less complete program, and less evidence should not yield a more complete program) and should not be sensitive to the order of presentation of the evidence.

Program Schemas

Informally, a *program schema* contains a template program and a set of axioms. The *template* abstracts a class of actual programs, called *instances*, in the sense that it represents their dataflow and control flow by means of placeholders, but does not make explicit all their actual computations nor all their actual data structures. The *axioms* restrict the possible instances of the placeholders and define their interrelationships. Note that a schema is problem independent. Let us here take a **first-order logic** approach and consider templates as open logic programs (i.e. programs where some placeholder predicate symbols are left undefined or *open*; a program with no open predicate symbols is said to be *closed*) and axioms as first-order specifications of these open predicate symbols.

Example 2 Most methods of inductive synthesis are biased by program schemas whose templates have clauses of the forms in the following generic template:

$$\begin{aligned} r(X,Y,Z) \leftarrow\ & c(X,Y,Z), \\ & p(X,Y,Z) \\ r(X,Y,Z) \leftarrow\ & d(X,H,X_1,\ldots,X_t,Z), \\ & r(X_1,Y_1,Z),\ \ldots,\ r(X_t,Y_t,Z), \\ & q(H,Y_1,\ldots,Y_t,Z,Y) \end{aligned} \tag{2}$$

where c, d, p, q are open predicate symbols, X is a nonempty sequence of terms, and Y, Z are possibly empty sequences of terms. The intended semantics of this generic template can be informally described as follows. For an arbitrary relation r over parameters X, Y, Z, an instance of this generic template is to determine the values of *result parameter* Y corresponding to a given value of *induction parameter* X, considering the value of *auxiliary parameter* Z. Two cases arise: either the c test succeeds and X has a value for which Y can be easily directly computed through p, or X has a value for which Y cannot be so easily directly computed and the *divide-and-conquer principle* is applied:

1. *divide* X through d into a term H and t terms $X_1, \ldots, X_t$ of the same type as X but smaller than X according to some well-founded relation;
2. *conquer* through t recursive calls to r to determine the values of $Y_1, \ldots, Y_t$ corresponding to $X_1, \ldots, X_t$, respectively, considering the value of Z;
3. *combine* through q the terms $H, Y_1, \ldots, Y_t, Z$ to build Y.

Enforcing this intended semantics must be done manually, as any instance template by itself has no semantics, in the sense that any program is an instance of it (it suffices to define c by a program that always succeeds and p by the given program). One way to do this is to attach to a template some axioms (see Smith (1985) for the divide-and-conquer axioms), namely, the set of specifications of its open predicate symbols: these specifications refer to each other, including the one of r, and are generic (because even the specification of r is unknown), but can be manually abduced once and for all according to the informal semantics of the schema.

Predicate Invention

Another important **language bias** is the available vocabulary, which is here the set of predicate symbols mentioned in the evidence set or actually defined in the background knowledge (and possibly mentioned by the oracle). If an inductive synthesis fails, other than backtracking to a different program schema (i.e., shifting the statement bias), one can try and shift the vocabulary bias by inventing new predicate symbols and inducing programs for them in the extended vocabulary; this is also known as performing **constructive induction**. Only the invention of recursively defined predicate symbols is *necessary*, as a non-recursive definition of a predicate symbol can be eliminated by substitution (under **resolution**) for its calls in the induced program (even though that might make the program longer).

In general, it is undecidable whether **predicate invention** is necessary to induce a finite program in the vocabulary of its evidence and background knowledge (as a consequence of Rice's theorem, 1953), but introducing new predicate symbols always allows the induction of a finite program (as a consequence of a result by Kleene), as shown in Stahl (1995). The necessity of shifting the vocabulary bias is only decidable for some restricted languages (but the bias shift attempt might then be unsuccessful), so in practice one often has to resort to heuristics. Note that an inductive synthesizer of recursive algorithms may be recursive itself: it may recursively invoke itself for a necessary new predicate symbol.

Other than the decision problem, the difficulties of predicate invention are as follows. First, adequate formal parameters for a new predicate symbol have to be identified among all the variables in the clause using it. This can be done instantaneously by using precomputations done manually once and for all at the template level. Second, evidence for a new predicate symbol has to be **abduced** from the current program using the evidence for the old predicate symbol. This usually requires an oracle for the old predicate symbol, whose program is still unfinished at that moment and cannot be used. Third, the abduced evidence may be less numerous than for the old predicate symbol (note that if the new predicate symbol is in a recursive clause, then no new evidence might be abduced from the old evidence that is covered by the base clauses) and can be quite sparse, so that the new synthesis is more difficult. This *sparseness problem* can be illustrated by an example.

Example 3 Given the positive ground examples $factorial(0,1)$, $factorial(1,1)$, $factorial(2,2)$, $factorial(3,6)$, and $factorial(4,24)$ and given the still open program:

$$
\begin{aligned}
&\mathit{factorial}(N,F) \leftarrow N=0,\ F=1\\
&\mathit{factorial}(N,F) \leftarrow \mathit{add}(M,1,N),\\
&\qquad\qquad\qquad\qquad \mathit{factorial}(M,G),\\
&\qquad\qquad\qquad\qquad \mathit{product}(N,G,F)
\end{aligned}
$$

where *add* is known but *product* was just invented (and named so only for the reader's convenience), the abduceable examples are *product*$(1,1,1)$, *product*$(2,1,2)$, *product*$(3,2,6)$, and *product* $(4,6,24)$, which is hardly enough for inducing a recursive program for *product*; note that there is one less example than for *factorial*. Indeed, examples such as *product*$(3,6,18)$, *product*$(2,6,12)$, *product*$(1,6,6)$, etc. are missing, which puts the given examples more than one resolution step apart, if not on different **resolution** paths. This is aggravated by the absence of an oracle for the invented predicate symbol, which is not necessarily intrinsic to the task at hand (although *product* actually is intrinsic to] *factorial*).

Background Knowledge

In an inductive programming context, background knowledge is particularly important, as the inference of recursive programs is more difficult than the inference of **classifiers**. For the efficiency of synthesis, it is crucial that this collection of definitions of the predefined predicate symbols be *annotated* with information about the *types* of their arguments and about whether some *well-founded relation* is being enforced between some of their arguments, so that semantically suitable instances for the open predicate symbols of any chosen program schema can be readily spotted. (This requires in turn that the types of the arguments of the predicate symbols in the provided evidence are declared as well.) The background knowledge should be problem independent, and an inductive programming method should be able to perform *knowledge mobilization*, namely organizing it dynamically according to relevance to the current task.

In data-driven, analytical approaches, background knowledge is used in combination with **explanation-based learning** (EBL) methods, such as **abduction** (see Example 4) or systematic rewriting of input/output examples into computational traces (see Example 5).

Background knowledge can also be given in the form of constraints or an explicit inductive bias as in meta-interpretative learning (Muggleton and Lin 2013) or in using higher-order patterns (Katayama 2006).

Programs and Data

Example 4 The DIALOGS (Dialogue-based Inductive-Abductive LOGic program Synthesizer) method (Flener 1997) is interactive. The main design objective was to take all extra burden from the specifier by having the method ask for exactly and only the information it needs, default answers being provided wherever possible. As a result, no evidence needs to be prepared in advance, as the method invents its own candidate evidence and queries the oracle about it, with an opportunity to declare (at the oracle/specifier's risk) that enough information has been provided. All answers by the oracle are stored as *judgments*, to prevent asking the same query twice. This is suitable for all levels of expertise of human users, as the queries are formulated in the specifier's initially unknown conceptual language, in a way such that the specifier must know the answers if she really feels the need for the wanted program. The method is schema-biased, and the current implementation has two schemas. The template of the *divide-and-conquer* schema has the generality of the generic template (2). The template of the *accumulate* schema extends this by requiring an accumulator in the sequence Z of auxiliary parameters. The evidence language (**observation language**) is (non-recursive) logic programs with negation. Type declarations are provided as a **language bias**. The program language (**hypothesis language**) is recursive logic programs with negation, with possibly multiple base cases and recursive cases.

For instance, starting from the empty program for the relation *delOdds* in Example 1, the algorithm design choices of using the *divide-*

and-conquer schema with R as result parameter and L as induction parameter decomposed (with $t = 1$) through head-tail decomposition by d lead to the following intermediate open program:

$$\begin{aligned} delOdds(L,R) &\leftarrow c(L,R),\ p(L,R) \\ delOdds(L,R) &\leftarrow d(L,H,L_1), \\ &\quad\ delOdds(L_1,R_1), \\ &\quad\ q(H,R_1,R) \\ c(\_,\_) &\leftarrow true \\ d(L,H,T) &\leftarrow L = [H|T] \end{aligned} \tag{3}$$

The first five evidential clauses for *delOdds* in (1) are then implicitly interactively acquired from the oracle/specifier by the following question and answer dialogue, leading the specifier to reveal the intrinsic predicate symbol *odd*:

DIALOGS: When does $delOdds([\,], R)$ hold?
Specifier: If $R = [\,]$.
DIALOGS: When does $delOdds([X], R)$ hold?
Specifier: If $(odd(X)$ *and* $R = [\,])$ *or* $(\neg odd(X)$ *and* $R = [X])$.
DIALOGS: When does $delOdds([X,Y], R)$ hold, assuming $odd(Y)$?
Specifier: If $(odd(X)$ *and* $R = [\,])$ *or* $(\neg odd(X)$ *and* $R = [X])$.
DIALOGS: When does $delOdds([X,Y], R)$ hold, assuming $\neg odd(Y)$?
Specifier: If $(odd(X)$ *and* $R = [Y])$ *or* $(\neg odd(X)$ *and* $R = [X,Y])$.

Next, **abduction** infers the following evidence set for the still open predicate symbols p and q:

$$\begin{aligned} p([\,],[\,]) &\leftarrow true \\ p([X],[\,]) &\leftarrow odd(X) \\ p([X],[X]) &\leftarrow \neg odd(X) \\ p([X,Y],[Y]) &\leftarrow odd(X),\ \neg odd(Y) \\ p([X,Y],[X,Y]) &\leftarrow \neg odd(X),\ \neg odd(Y) \\ q(X,[\,],[\,]) &\leftarrow odd(X) \\ q(X,[\,],[X]) &\leftarrow \neg odd(X) \\ q(X,[Y],[Y]) &\leftarrow odd(X) \\ q(X,[Y],[X,Y]) &\leftarrow \neg odd(X) \end{aligned}$$

From this, induction infers the following closed programs for p and q:

$$\begin{aligned} p([\,],[\,]) &\leftarrow true \\ q(H,L,[H|L]) &\leftarrow \neg odd(H) \\ q(H,L,L) &\leftarrow odd(H) \end{aligned} \tag{4}$$

The final closed program is the union of the programs (3) and (4), as no predicate invention is deemed necessary. Sample syntheses with predicate invention are presented in Flener (1997) and Flener and Yılmaz (1999).

Example 5 The THESYS method (Summers 1977) was one of the first methods for the inductive synthesis of functional (Lisp) programs. Although it has a rather restricted scope, it can be seen as the methodological foundation of many later methods for inducing functional programs. The noninteractive method is schema biased, and the implementation has two schemas. Upon adaptation to functional programming, the template of the *linear recursion* schema is the instance of the generic template (2) obtained by having X as a sequence of exactly one induction parameter and Z as the empty sequence of auxiliary parameters, and by dividing X into $t = 1$ smaller value X_t, so that there is only $t = 1$ recursive call. The template of the *accumulate* schema extends this by having Z as a sequence of exactly one auxiliary parameter, playing the role of an accumulator. The evidence language (**observation language**) is sets of ground positive examples. The program language (**hypothesis language**) is recursive functional programs, with possibly multiple base cases, but only one recursive case. The only primitive functions are *nil*, *cons*, *head*, *tail*, and *empty*, because the implementation is limited to the list data type, inductively defined by $list \equiv nil \mid cons(x, list)$, under the axioms $empty(nil) = true$, $head(cons(x,y)) = x$, and $tail(cons(x,y)) = y$. There is no function invention.

For instance, from the following examples of a list unpacking function:

$$\begin{aligned}
&\mathit{unpack}(\mathit{nil}) &&= \mathit{nil}\\
&\mathit{unpack}((A)) &&= ((A))\\
&\mathit{unpack}((A\ B)) &&= ((A)\ (B))\\
&\mathit{unpack}((A\ B\ C)) &&= ((A)\ (B)\ (C))
\end{aligned}$$

the **abduced** traces are:

$$\begin{aligned}
&\mathit{empty}(X) &&\rightarrow \mathit{nil}\\
&\mathit{empty}(\mathit{tail}(X)) &&\rightarrow \mathit{cons}(X, \mathit{nil})\\
&\mathit{empty}(\mathit{tail}(\mathit{tail}(X))) &&\rightarrow \mathit{cons}(\mathit{cons}(\mathit{head}(X), \mathit{nil}), \mathit{cons}(\mathit{tail}(X), \mathit{nil}))\\
&\mathit{empty}(\mathit{tail}(\mathit{tail}(\mathit{tail}(X)))) &&\rightarrow \mathit{cons}(\mathit{cons}(\mathit{head}(X), \mathit{nil}), \mathit{cons}(\mathit{cons}(\mathit{head}(\mathit{tail}(X)), \mathit{nil}),\\
& &&\quad\ \mathit{cons}(\mathit{tail}(\mathit{tail}(X)), \mathit{nil})))
\end{aligned}$$

and the **induced** program is:

$$\begin{aligned}
\mathit{unpack}(X) = {}&\mathit{empty}(X) &&\rightarrow \mathit{nil},\\
&\mathit{empty}(\mathit{tail}(X)) &&\rightarrow \mathit{cons}(X, \mathit{nil}),\\
&\mathit{true} &&\rightarrow \mathit{cons}(\mathit{cons}(\mathit{head}(X), \mathit{nil}), \mathit{unpack}(\mathit{tail}(X)))
\end{aligned}$$

A modern extension of THESYS is the IGOR method (Kitzelmann and Schmid 2006). The underlying program template describes the set of all functional programs with the following restrictions: built-in functions can only be first-order, and no nested or mutual recursion is allowed. IGOR adopts the two-step approach of THESYS. Synthesis is still restricted to structural problems, where only the structure of the arguments matters, but not their contents, such as in list reversing. Nevertheless, the scope of synthesizable programs is considerably larger. For instance, tree-recursive functions and functions with hidden parameters can be induced. Most notably, programs consisting of a calling function and an arbitrary set of further recursive functions can be induced. The first step of synthesis (trace construction) is therefore expanded such that traces can contain nestings of conditions. The second step is expanded such that the synthesis of a function can rely on the invention and synthesis of other functions (i.e., IGOR uses a technique of function invention in correspondence to the concept of **predicate invention** introduced above). An extension, IGOR2, relies on constructor term rewriting techniques. The two synthesis steps are merged into one and make use of background knowledge. Therefore, the synthesis of programs for semantic problems, such as list sorting, becomes feasible.

Applications

In the framework of *software engineering*, inductive programming is defined as the inference of information that is pertinent to the construction of a generalized computational system for which the provided evidence is a representative sample (Flener and Partridge 2001). In other words, inductive programming does *not* have to be a panacea for software development in the large and infer a complete software system in order to be useful: it suffices to induce, for instance, a self-contained system module while programming in the small, problem features and decision logic for specification acquisition and enhancement or support for debugging and testing. Inductive programming is then not always limited to programs with repetitive or recursive control structures. There are opportunities for synergy with manual programming and deductive program synthesis, as there are sometimes system modules that no one knows how to specify in a complete way, or that are harder to specify or program in a complete way, and yet where incomplete information such as input-output examples is readily available. More examples and pointers to the literature are given in Flener (2002, Section 5) and Flener and Partridge (2001).

In the context of *end-user programming*, inductive programming methods can be used to enable nonexpert users to take advantage of the more sophisticated functionalities offered by their software. This kind of application is in the focus of **programming by demonstration** (PBD).

Finally, it is worth having an evidential synthesizer of recursive algorithms invoked by a more general-purpose machine learning method when necessary predicate invention is detected or conjectured, as such general methods require a lot of evidence to infer reliably a recursively defined hypothesis.

Future Directions

Inductive programming is still mainly a topic of basic research, exploring how the intellectual ability of humans to infer generalized recursive procedures from incomplete evidence can be captured in the form of synthesis methods. Already a variety of promising methods are available. A necessary step should be to compare and analyze the current methods. A first extensive comparison of different ILP methods for inductive programming was presented some years ago (Flener and Yılmaz 1999). An up-to-date analysis should take into account not only ILP methods but also methods for the synthesis of functional programs, using classical (Kitzelmann and Schmid 2006) as well as evolutionary (Olsson 1995) methods. The methods should be compared with respect to the required quantity of evidence, the kind and amount of background knowledge, the scope of programs that can be synthesized, and the efficiency of synthesis. Such an empirical comparison should result in the definition of characteristics that describe concisely the scope, usefulness, and efficiency of the existing methods in different problem domains. A first step toward such a systematic comparison was presented in Hofmann et al. (2009).

Since only a few inductive programming methods can deal with semantic problems, it should be useful to investigate how inductive programming methods can be combined with other machine learning methods, such as kernel-based **classification**.

Finally, the existing methods should be adapted to a broad variety of application areas in the context of programming assistance, as well as in other domains where recursive data structures or recursive procedures are relevant.

Cross-References

▶ Explanation-Based Learning
▶ Inductive Logic Programming
▶ Programming by Demonstration
▶ Programming by Example (PBE)
▶ Trace-Based Programming

Recommended Reading

- Online Platform of the Inductive Programming Community: http://www.inductive-programming.org/.
- Journal of *Automated Software Engineering, Special Issue on Inductive Programming*, April 2001: Flener and Partridge (2001), http://user.it.uu.se/~pierref/ase/.
- *Biannual Workshops on Approaches and Applications of Inductive Programming*: http://www.cogsys.wiai.uni-bamberg.de/aaip/.
- *Journal of Machine Learning Research, Special Topic on Approaches and Applications of Inductive Programming*, February/March 2006: http://jmlr.csail.mit.edu/papers/topic/inductive_programming.html.
- *Dagstuhl Report 3/12 on Approaches and Applications of Inductive Programming* http://drops.dagstuhl.de/opus/volltexte/2014/4507/.

Biermann AW (1978) The inference of regular LISP programs from examples. IEEE Trans Syst Man Cybern 8(8):585–600

Flener P (1997) Inductive logic program synthesis with DIALOGS. In: Muggleton SH (ed) Revised selected papers of the 6th international workshop on inductive logic programming (ILP 1996), Stockholm. Volume 1314 of lecture notes in artificial intelligence. Springer, pp 175–198
Flener P (2002) Achievements and prospects of program synthesis. In: Kakas A, Sadri F (eds) Computational logic: logic programming and beyond; essays in honour of Robert A. Kowalski. Volume 2407 of lecture notes in artificial intelligence. Springer, Berlin/New York, pp 310–346
Flener P, Partridge D (2001) Inductive programming. Autom Softw Eng 8(2):131–137
Flener P, Yılmaz S (1999) Inductive synthesis of recursive logic programs: achievements and prospects. J Log Program 41(2–3):141–195
Gulwani S, Kitzelmann E, Schmid U (2014) Approaches and Applications of Inductive Programming (Dagstuhl Seminar 13502). Dagstuhl Reports 3/12, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl
Hofmann M, Kitzelmann E, Schmid U (2009) A unifying framework for analysis and evaluation of inductive programming systems. In: Goerzel B, Hitzler P, Hutter M (eds) Proceedings of the second conference on artificial general intelligence (AGI-09, Arlington, Virginia, 6–9 March 2009), Amsterdam. Atlantis Press, pp 55–60
Katayama S (2005) Systematic search for lambda expressions. In: Trends in functional programming. Intellect, Bristol, pp 111–126
Kitzelmann E, Schmid U (2006) Inductive synthesis of functional programs – an explanation based generalization approach. J Mach Learn Res 7(Feb): 429–454
Muggleton SH, Lin D (2013) Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. In: Rossi F (ed) IJCAI 2013, proceedings of the 23rd international joint conference on artificial intelligence, Beijing, 3–9 Aug 2013. IJCAI/AAAI, pp 1551–1557
Olsson JR (1995) Inductive functional programming using incremental program transformation. Artif Intell 74(1):55–83
Shapiro EY (1983) Algorithmic program debugging. The MIT Press, Cambridge
Smith DR (1984) The synthesis of LISP programs from examples: a survey. In: Biermann AW, Guiho G, Kodratoff Y (eds) Automatic program construction techniques. Macmillan, New York, pp 307–324
Smith DR (1985) Top-down synthesis of divide-and-conquer algorithms. Artificial Intelligence, 27(1):43–96
Stahl I (1995) The appropriateness of predicate invention as bias shift operation in ILP. Mach Learn 20(1–2):95–117
Summers PD (1977) A methodology for LISP program construction from examples. J ACM 24(1): 161–175

Inductive Synthesis

▶ Inductive Programming

Inductive Transfer

Ricardo Vilalta[1], Christophe Giraud-Carrier[2], Pavel Brazdil[3], and Carlos Soares[3,4]
[1]Department of Computer Science, University of Houston, Houston, TX, USA
[2]Department of Computer Science, Brigham Young University, Provo, UT, USA
[3]LIAAD-INESC Tec/Faculdade de Economia, University of Porto, Porto, Portugal
[4]LIAAD-INESC Porto L.A./Faculdade de Economia, University of Porto, Porto, Portugal

Abstract

We describe different scenarios where a learning mechanism is capable of acquiring experience on a source task, and subsequently exploit such experience on a target task. The core ideas behind this ability to transfer knowledge from one task to another have been studied in the machine learning literature under different titles and perspectives. Here we describe some of them under the names of inductive transfer, transfer learning, multitask learning, meta-searching, meta-generalization, and domain adaptation.

Synonyms

Domain adaptation; Multitask learning; Transfer learning; Transfer of knowledge across domains

Definition

Inductive transfer refers to the ability of a learning mechanism to improve performance on the current or *target* task after having learned a different but related concept or skill on a previous *source* task. Transfer may additionally occur between two or more learning tasks that are being undertaken concurrently. The object being transferred may refer to instances, features, a particular form of search bias, an action policy, background knowledge, etc.

Motivation and Background

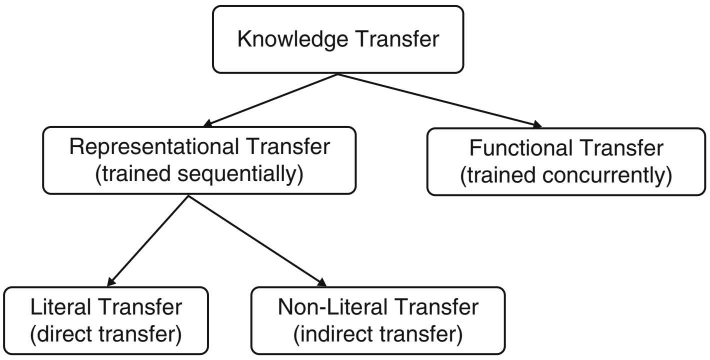
Learning is not the result of an isolated task that starts from scratch with every new problem. Instead, a learning algorithm should exhibit the ability to adapt through a mechanism dedicated to transfer knowledge gathered from previous experience. The problem of transfer of knowledge is central to the field of machine learning and is also known as *inductive transfer*. In this case, knowledge can be understood as a collection of patterns observed across tasks. One view of the nature of patterns across tasks is that of invariant transformations. For example, image recognition of a target object is simplified if the object is invariant under rotation, translation, scaling, etc. A learning system should be able to recognize a target object on an image even if previous images show the object in different sizes or from different angles. Hence, inductive transfer studies know how to improve learning by detecting, extracting, and exploiting (meta)knowledge in the form of invariant transformations across tasks.

Similarly, in competitive games involving teams of robots (e.g., RoboCup Soccer), transferring knowledge learned from one task to another task is crucial to acquire skills necessary to beat the opponent team. Specifically, imagine a situation where a team of robots has been taught to keep a soccer ball away from the opponent team. To achieve that goal, robots must learn to keep the ball, pass the ball to a close teammate, etc., always trying to remain at a safe distance from the opponents. Now let us assume that we wish to teach the same team of robots to be efficient at scoring against a team of defending robots. Knowledge gained during the first activity can be transferred to the second one. Specifically, a robot can prefer to perform an action learned in the past over actions proposed during the current task, because the past action has a significant higher merit value. For example, a robot under the second task may learn to recognize that it is preferable to shoot than to pass the ball because the goal is very close. This action can be learned from the first task by recognizing that the precision of a pass is contingent upon the proximity of the teammate.

Structure of the Learning System

The main idea behind a learning architecture using knowledge transfer is to produce a source model from which knowledge can be extracted and transferred to a target model. This allows for multiple scenarios (Brazdil et al. 2009; Pratt and Thrun 1997). For example, the target and source models can be trained at different times in such a way that the transfer takes place after the source model has been trained. In this case there is an explicit form of knowledge transfer, also called *representational transfer*. In contrast, we use the term *functional transfer* to denote the case where two or more models are trained simultaneously; in this case the models share (part of) their internal structure during learning (see Neural Networks below). Under representational transfer, we denote as *literal transfer* the case when the source model is left intact and as *nonliteral transfer* the case when the source model is modified before knowledge is transferred to the target model. In nonliteral transfer some processing takes place on the source model before it is used to initialize the target model (see Fig. 1).

Neural Networks. A learning paradigm amenable to test the feasibility of knowledge transfer is that of neural networks (Caruana 1993). A popular form of (functional) knowledge transfer is effected through multitask learning, where the output nodes in the multilayer network represent more than one task. In such a scenario, internal nodes are shared by different tasks dynamically during learning. As an illustration, consider the problem of learning to classify astronomical objects from images mapping

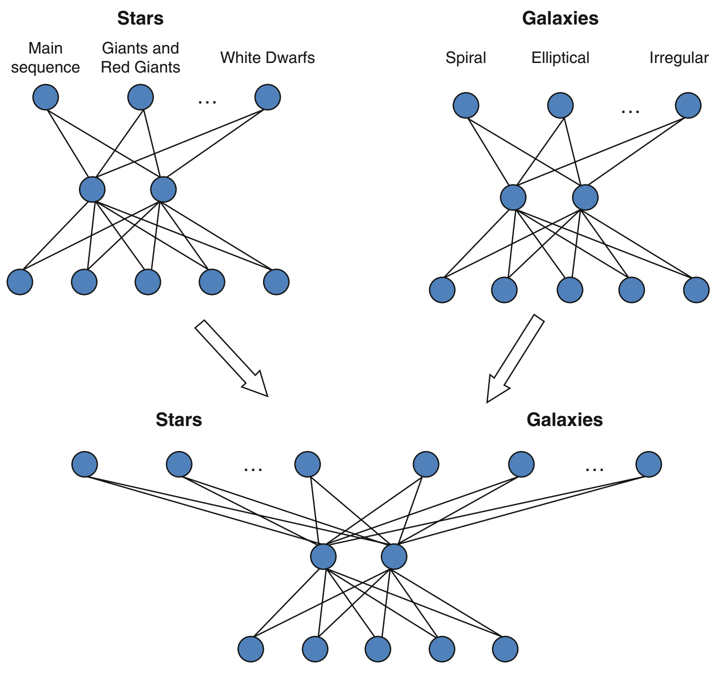

Knowledge Transfer
Representational Transfer (trained sequentially)
Functional Transfer (trained concurrently)
Literal Transfer (direct transfer)
Non-Literal Transfer (indirect transfer)

Inductive Transfer, Fig. 1 A taxonomy of inductive transfer

the sky into multiple classes. One task may be in charge of classifying a star into several classes (e.g., main sequence, dwarf, red giant, neutron, pulsar, etc.). Another task can focus on galaxy classification (e.g., spiral, barred spiral, elliptical, irregular, etc.). Rather than separating the problem into different tasks where each task is in charge of identifying one type of luminous object, one can combine the tasks together into a single parallel multitask problem where the hidden layer of a neural network shares patterns that are common to all classification tasks (see Fig. 2). The reasons explaining why learning often improves in accuracy and speed in this context is that training with many tasks in parallel on a single neural network induces information that accumulates in the training signals; if there exists properties common to several tasks, internal nodes can serve to represent common sub-concepts simultaneously.

Other Paradigms. Knowledge transfer can be performed using other learning and data analysis paradigms –mainly in the form of representational transfer– such as kernel methods, probabilistic methods, clustering, etc. (Raina et al. 2006; Evgeniou et al. 2005). For example, inductive transfer can take place in learning methods that assume a probabilistic distribution of the data by guaranteeing a form of relatedness among the distributions adopted across tasks (Raina et al. 2006). As an illustration, if learning to classify stars and galaxies both assume a mixture of normal densities to model the input-output or example-class distribution, one can force both distributions to have sets of parameters that are as similar as possible while preserving good generalization performance. In that case, shared knowledge can be interpreted as a set of assumptions about the data distribution for all tasks under analysis. The concept of knowledge transfer is also related to the problem of introducing new intermediate concepts during rule induction. In the inductive logic programming (ILP) setting, this is referred to as *predicate invention* (Stahl 1995).

Meta-Searching for Problem Solvers. A different research direction in inductive transfer explores complex scenarios where the software architecture itself evolves with experience (Schmidhuber 1997). The main idea is to divide a program into different components that can be reused during different stages of the learning process. As an illustration, one can work within the space of (self-delimiting binary) programs to propose an optimal ordered problem solver. The goal is to solve a sequence of problems, deriving one solution after the other, as optimally as possible. Ideally the system should be capable of exploiting previous solutions and of incorporating them into the solution to the current problem. This can be done by allocating computing time to the search for previous solutions that, if useful, become transformed into building blocks. We assume the current problem can be solved by copying or invoking previous pieces of code (i.e., building blocks or knowledge). In that case the mechanism will accept those solutions with substantial savings in computational time.



Inductive Transfer, Fig. 2 Example of multitask learning (functional transfer) applied to astronomical images

Domain Adaptation. A recent research direction in representational transfer seeks to adjust the model obtained in a source domain to account for differences exhibited in a new target domain. Unlike traditional studies in classification where both training and testing sets are assumed as realizations of the same joint input-output distribution, this *domain adaptation* approach either weakens or completely disregards such assumption (Ben-David et al. 2007, Daumé, et al. 2006, Storkey 2009). In addition, domain adaptation commonly assumes an abundance of labeled examples in the source domain, but little or no class labels in the target domain.

An example of these concepts lies in light curve classification from star samples obtained from different galaxies. A classification task set to differentiate different types of stars in a nearby source galaxy –where class labels are available– will experience a change in distribution as it moves to a target galaxy lying farther away –where class labels are unavailable. A major reason for such change is that at greater distances, less luminous stars fall below the detection threshold and more luminous stars are preferentially detected. The corresponding dataset shift (Quinonero-Candela et al. 2009) precludes the direct utilization of one single model across galaxies; it calls for a form of model adaptation to compensate for the change in the data distribution.

Domain adaptation has gained much attention recently, mainly due to the pervasive character of problems where distributions change over time. It assumes that the learning task remains constant, but the marginal and class posterior distributions between source and target domain may differ (as opposed to traditional transfer learning where tasks can in addition exhibit different input representations, i.e., different input spaces). Domain adaptation has been attacked from different angles: by searching for a single representation that unifies both source and target domains (Glorot et al. 2011); by proving error bounds as a function of empirical error and the distance between source and target distributions (Ben-David et al. 2010), within a co-training framework where target vectors are incorporated into the source training set based on confidence (Chen et al. 2011), by re-weighting source instances (Mansour et al. 2009), by using regularization terms to learn models that perform well on both source and target domains (Daumé et al. 2010), and several others.

Theoretical Work

Several studies have provided a theoretical analysis of the case where a learner uses experience from previous tasks to learn a new task. This process is often referred to as meta-learning or meta-generalization. The aim is to understand the conditions under which a learning algorithm can provide good generalizations when embedded in an environment made of related tasks. Although the idea of knowledge transfer is normally made implicit in the analysis, it is clear that the meta-learner extracts and exploits knowledge on every task to perform well on future tasks. Theoretical studies fall within a Bayesian model and within a probably approximately correct (PAC) model. The idea is to find not only the right hypothesis in a hypothesis space (base learning), but in addition to find the right hypothesis space in a family of hypothesis spaces (meta-learning).

We briefly review the main ideas behind these studies (Baxter 2000). We begin by assuming that the learner is embedded in a set of related tasks that share certain commonalities. Going back to the problem where a learner is designed for recognition of astronomical objects, the idea is to classify objects (e.g., stars, galaxies, nebulae, and planets) extracted from images mapping certain region of the sky. One way to transfer learning experience from one astronomical center to another is by sharing a meta-learner that carries a bias toward recognition of astronomical objects. In traditional learning, we assume a probability distribution $\mathbf{p}$ that indicates which examples are more likely to be seen in such a task. Now we assume that there is a more general distribution $\mathbf{P}$ over the space of all possible distributions. In essence, the meta-distribution $\mathbf{P}$ indicates which tasks are more likely to be found within the sequence of tasks faced by the meta-learner (distribution $\mathbf{p}$ indicates which examples are more likely to be seen in one task). In our example, the meta-distribution $\mathbf{P}$ peaks over tasks corresponding to classification of astronomical objects. Given a family of hypothesis spaces $\{\mathbf{H}\}$, the goal of the meta-learner is to find a hypothesis space $\mathbf{H^*}$ that minimizes a functional risk corresponding to the expected loss of the best possible hypothesis in each hypothesis space. In practice, since we ignore the form of $\mathbf{P}$, we need to draw samples $\mathbf{T_1}, \mathbf{T_2}, \ldots, \mathbf{T_n}$ to infer how tasks are distributed in our environment. To summarize, in the transfer learning scenario, our input is made of samples $\mathbf{T} = \{\mathbf{T_i}\}$, where each sample $\mathbf{T_i}$ is composed of examples. The goal of the meta-learner is to output a hypothesis space with a learning bias that generates accurate models for a new task.

Future Directions

The research community faces several challenges on how to efficiently transfer knowledge across tasks. One challenge involves devising learning architectures with an explicit representation of knowledge about models and algorithms, i.e., meta-knowledge. Most systems that integrate knowledge transfer mechanisms make an implicit assumption about the type of knowledge being transferred. This is indeed possible when strong assumptions are made on the relationship between the source and target tasks. For example, most approaches to domain adaptation work under strong assumptions about the similarity between the source and target tasks, imposing similar class posterior distributions, marginal distributions, or both. Ideally we would like to track the evolution of the source task to the target task to be able to justify any assumptions about their differences.

From a global perspective, it seems clear that proper treatment of the inductive transfer problem requires more than just statistical or mathematical techniques. Inductive transfer can be embedded in a complex artificial intelligence system that incorporates important components such as knowledge representation, search, planning, reasoning, etc. Without the incorporation of artificial intelligence components, we are forced to work with a large hypothesis space and a set of stringent assumptions about the nature of the discrepancy between the source and target tasks.

Cross-References

► Metalearning

Recommended Reading

Baxter J (2000) A model of inductive learning bias. J Artif Intell Res 12:149–198
Ben-David S, Blitzer J, Crammer K, Pereira F (2007) Analysis of representations for domain adaptation. Adv Neural Inf Process Syst 19:137–144
Ben-David S, Blitzer J, Crammer K, Kulesza A, Pereira F, Wortman J (2010) A theory of learning from different domains. Mach Learn Spec Issue Learn Mult Sources 79:151–175
Brazdil P, Giraud-Carrier C, Soares C, Vilalta R (2009) Metalearning: applications to data mining. Springer, Berlin
Caruana R (1993) Multitask learning: a knowledge-based source of inductive bias. In: Proceedings of the 10th international conference on machine learning (ICML), Amherst, pp 41-48
Chen M, Weinberger KQ, Blitzer J (2011) Co-training for domain adaptation. In: Advances in neural information processing systems (NIPS), Granada
Dai W, Yang Q, Xue G, Yu Y (2007) Boosting for transfer learning. In: Proceedings of the 24th international conference on machine learning (ICML), Corvallis, pp 193–200
Daumé H, Marcu D (2006) Domain adaptation for statistical classifiers. J Mach Learn Res 26:102–126
Daumé H, Kumar A, Saha A (2010) Co-regularization based semi-supervised domain adaptation. In: Advances in neural information processing systems (NIPS), Whistler
Evgeniou T, Micchelli CA, Pontil M (2005) Learning multiple tasks with kernel methods. J Mach Learn Res 6:615–637
Glorot X, Bordes A, Bengio Y (2011) Domain adaptation for large-scale sentiment classification: a deep learning approach. In: Proceedings of the 28th international conference on machine learning (ICML), Bellevue, pp 513–520
Mansour Y, Mohri M, Rostamizadeh A (2009) Domain adaptation with multiple sources. In: Advances in neural information processing systems (NIPS), Whistler, pp 1041–1048
Mihalkova L, Huynh T, Mooney RJ (2007) Mapping and revising markov logic networks for transfer learning. In: Proceedings of the 22nd AAAI conference on artificial intelligence, Vancouver, pp 608–614
Oblinger D, Reid M, Brodie M, de Salvo Braz R (2002) Cross-training and its application to skill-mining. IBM Syst J 41(3):449–460
Pratt L, Thrun S (1997) Second special issue on inductive transfer. Mach Learn 28:4175
Quinonero-Candela J, Sugiyama M, Schwaighofer A, Lawrence ND (2009) Dataset shift in machine learning. MIT Press, Cambridge
Raina R, Ng AY, Koller D (2006) Constructing informative priors using transfer learning. In: Proceedings of the 23rd international conference on machine learning (ICML), Pittsburgh, pp 713–720
Reid M (2004) Improving rule evaluation using multitask learning. In: Proceedings of the 14th international conference on ILP, Porto, pp 252–269
Schmidhuber J (1997) Shifting inductive bias with success-story algorithm, adaptive levin search, and incremental self-improvement. Mach Learn 28: 105–130
Stahl I (1995) Predicate invention in inductive logic programming. In: De Raedt L (ed) Advances in inductive logic programming. IOS Press, Amsterdam/Washington, DC, pp 34–47
Storkey A (2009) When training and test sets are different. In: Quinonero-Candela J, Sugiyama M, Schwaighofer A, Lawrence ND (eds) Dataset shift in machine learning. MIT Press, Cambridge, pp 3–28

Inequalities

► Generalization Bounds

Information Retrieval

Information retrieval (IR) is a set of techniques that extract from a collection of documents those that are relevant to a given query. Initially addressing the needs of librarians and specialists, the field has evolved dramatically with the advent of the World Wide Web. It is more general than *data retrieval*, whose purpose is to determine which documents contain occurrences of the keywords that make up a query. Whereas the syntax and semantics of data retrieval frameworks is strictly defined, with queries expressed in a totally formalized language, words from a natural language given no or limited structure are the medium of communication for information retrieval frameworks. A crucial task for an IR system is to index the collection of documents to make their contents efficiently accessible. The documents retrieved by the system are usually ranked by expected relevance, and the user who examines some of them might be able to provide feedback so that the query can be reformulated and the results improved.

In-Sample Evaluation

Synonyms

Within-sample evaluation

Definition

In-sample evaluation is an approach to ▶ algorithm evaluation whereby the learned model is evaluated on the data from which it was learned. This provides a biased estimate of learning performance, in contrast to ▶ holdout evaluation.

Cross-References

▶ Algorithm Evaluation

Instance

Synonyms

Case; Example; Item; Object

Definition

An *instance* is an individual object from the universe of discourse. Most learners create a model by analyzing a ▶ training set of instances. Most machine learning models take the form of a function from an ▶ instance space to an output space. In ▶ attribute-value learning, each instance is often represented as a vector of ▶ attribute values, each position in the vector corresponding to a unique attribute.

Instance Language

▶ Observation Language

Instance Space

Synonyms

Example space; Item space; Object space

Definition

An *instance space* is the space of all possible ▶ instances for some learning task. In ▶ attribute-value learning, the instance space is often depicted as a geometric space, one dimension corresponding to each attribute.

Instance-Based Learning

Eamonn Keogh
University of California-Riverside, Riverside, CA, USA

Synonyms

Analogical reasoning; Case-based learning; Memory-based; Nearest neighbor methods; Non-parametric methods

Definition

Instance-based learning refers to a family of techniques for ▶ classification and ▶ regression, which produce a class label/predication based on the similarity of the query to its nearest neighbor(s) in the training set. In explicit contrast to other methods such as ▶ decision trees and ▶ neural networks, instance-based learning algorithms do not create an abstraction from specific instances. Rather, they simply store all the data, and at query time derive an answer from an examination of the query's ▶ nearest neighbor (s).

Somewhat more generally, instance-based learning can refer to a class of procedures for solving new problems based on the solutions of similar past problems.

Motivation and Background

Most instance-based learning algorithms can be specified by determining the following four items:

1. Distance measure: Since the notion of similarity is being used to produce class label/prediction, we must explicitly state what similarity/distance measure to use. For real-valued data, Euclidean distance is a popular choice and may be optimal under some assumptions.
2. Number of neighbors to consider: It is possible to consider any number from one to all neighbors. This number is typically denoted as k.
3. Weighting function: It is possible to give each neighbor equal weight, or to weight them based on their distance to the query.
4. Mapping from local points: Finally, some method must be specified to use the (possibly weighted) neighbors to produce an answer. For example, for regression the output can be the weighted mean of the k nearest neighbors, or for classification the output can be the majority vote of the k nearest neighbors (with some specified tie-breaking procedure).

Since instance-based learning algorithms defer all the work until a query is submitted, they are sometimes called lazy algorithms (in contrast to eager learning algorithms, such as decision trees). Beyond the setting of parameters/distance measures/mapping noted above, one of the main research issues with instance-based learning algorithms is mitigating their expensive classification time, since a naïve algorithm would require comparing the distance for the query to every point in the database. Two obvious solutions are indexing the data to achieve a sublinear search, and numerosity reduction (data editing) (Wilson and Martinez 2000).

Further Reading

The best distance measure to use with an instance-based learning algorithms is the subject of active research. For the special case of time series data alone, there are at least one hundred methods (Ding et al. 2008). Conferences such as ICML, SIGKDD, etc. typically have several papers each year which introduce new distance measures and/or efficient search techniques.

Recommended Reading

Aha DW, Kibler D, Albert MK (1991) Instance-based learning algorithms. Mach Learn 6:37–66
Ding H, Trajcevski G, Scheuermann P, Wang X, Keogh EJ (2008) Querying and mining of time series data: experimental comparison of representations and distance measures. PVLDB 1(2):1542–1552
Wilson DR, Martinez TR (2000) Reduction techniques for exemplar-based learning algorithms. Mach Learn 38(3):257–286

Instance-Based Reinforcement Learning

William D. Smart
Washington University in St. Louis, St. Louis, MO, USA

Synonyms

Kernel-based reinforcement learning

Definition

Traditional reinforcement-learning (RL) algorithms operate on domains with discrete state spaces. They typically represent the value function in a table, indexed by states, or by state–action pairs. However, when applying RL to domains with continuous state, a tabular representation is no longer possible. In these cases, a common approach is to represent the value function by storing the values of a small set of states (or state–action pairs), and interpolating these values to other, unstored, states (or state–action pairs). This approach is known as instance-based reinforcement learning (IBRL). The instances are the explicitly stored values, and the interpolation is typically done using well-known instance-based supervised learning algorithms.

Motivation and Background

Instance-Based Reinforcement Learning (IBRL) is one of a set of value-function approximation techniques that allow standard RL algorithms to deal with problems that have continuous state spaces. Essentially, the tabular representation of the value function is replaced by an instance-based supervised learning algorithm and the rest of the RL algorithm remains unaltered. Instance-based methods are appealing because each stored instance can be viewed as analogous to one cell in the tabular representation. The interpolation method of the instance-based learning algorithm then blends the value between these instances.

IBRL allows generalization of value across the state (or state–action) space. Unlike tabular representations it is capable of returning a value approximation for states (or state–action pairs) that have never been directly experienced by the system. This means that, in theory, fewer experiences are needed to learn a good approximation to the value function and, hence, a good control policy. IBRL also provides a more compact representation of the value function than a table does. This is especially important in problems with multi-dimensional continuous state spaces. A straightforward discretization of such a space results in an exponential number of table cells. This, in turn, leads to an exponential increase in the amount of training experiences needed to obtain a good approximation of the value function.

An additional benefit of IBRL over other value-function approximation techniques, such as artificial neural networks, is the ability to bound the predicted value of the approximation. This is important, since it allow us to retain some of the theoretical non-divergence results for tabular representations.

Structure of Learning System

IBRL can be used to approximate both the state value function and the state–action value function. For problems with discrete actions, it is common to store a separate value function for each action. For continuous actions, the (continuous) state and action vectors are often concatenated, and VFA is done over this combined domain. For clarity, we will discuss only the state value function here, although our comments apply equally well to the state–action value function.

The Basic Approach

IBRL uses an instance-based supervised learning algorithm to replace the tabular value function representation of common RL algorithms. It maintains a set of states, often called basis points, and their associated values, using them to provide a value-function approximation for the entire state space. These exemplar states can be obtained in a variety of ways, depending on the nature of the problem. The simplest approach is to sample, either regularly or randomly, from the state space. However, this approach can result in an unacceptably large number of instances, especially if the state space is large, or has high dimension. A better approach is to use states encountered by the learning agent as it follows trajectories in the state space. This allows the representational power of the approximation algorithm to be focused on areas of the space in which the learning agent is likely to be. This, too, can result in a large number of states, if the agent is long-lived. A final approach combines the previous two by sub-sampling from the observed states.

Each stored instance state has a value associated with it, and an instance-based supervised learning algorithm is used to calculate the value of all other states. While any instance-based algorithm can be used, kernel-based algorithms have proven to be popular. Algorithms such as locally weighted regression (Smart and Kaelbling 2000), and radial basis function networks (Kretchmar and Anderson 1997) are commonly seen in the literature. These algorithms make some implicit assumptions about the form of the value function and the underlying state space, which we discuss below. For a state s, the kernel-based value-function approximation $V(s)$ is

$$V(s) = \frac{1}{\eta} \sum_{i=1}^{n} \phi(s, s_i) V(s_i), \qquad (1)$$

where the s_i values are the n stored basis points, η is a normalizer,

$$\eta = \sum_{i=1}^{n} \phi(s, s_i), \qquad (2)$$

and ϕ is the kernel function. A common choice for ϕ is an exponential kernel,

$$\phi(s, t) = e^{\frac{(s-t)^2}{\sigma^2}}, \qquad (3)$$

where σ is the kernel bandwidth. The use of kernel-based approximation algorithms is well motivated, since they respect Gordon's non-divergence conditions (Gordon 1995), and also Szepesvári and Smart's convergence criteria (Szepesvári and Smart 2004).

As the agent gathers experience, the value approximations at each of the stored states and, optionally, the location and bandwidth of the states must be updated. Several techniques, often based on the temporal difference error, have been proposed, but the problem remains open. An alternative to on-line updates is a batch approach, which relies on storing the experiences generated by the RL agent, composing these into a discrete MDP, solving this MDP exactly, and then using supervised learning techniques on the states and their associated values. This approach is known as fitted value iteration (Szepesvári and Munos 2005).

Examples of IBRL Algorithms

Several IBRL algorithms have been reported in the literature. Kretchmar and Anderson (1997) presented one of the first IBRL algorithms. They used a radial basis function (RBF) network to approximate the state–action value function for the well-known mountain-car test domain. The temporal difference error of the value update is used to modify the weights, centers, and variances of the RBF units, although they noted that it was not particularly effective in producing good control policies.

Smart and Kaelbling (2000) used locally weighted learning algorithms and a set of heuristic rules to approximate the state–action value function. A set of states, sampled from those experienced by the learning agent, were stored along with their associated values. One approximation was stored for each discrete action. Interpolation between these exemplars was done by locally weighted averaging or locally weighted regression, supplemented with heuristics to avoid extrapolation and over-estimation. Learning was done on-line, with new instances being added as the learning agent explored the state space. The algorithm was shown to be effective in practice, but offered no theoretical guarantees.

Ormoneit and Sen (2002) presented an offline kernel-based reinforcement-learning algorithm that stores experiences (s_i, a_i, r_i, s_i') as the instances, and uses these to approximate the state–action value function for problems with discrete actions. For a given state s and action a, the state–action value $Q(s, a)$ is approximated as

$$\hat{Q}(s, a) = \frac{1}{\eta_{s,a}} \sum_{i|a_i=a} \phi\left(\frac{d(s, s_i)}{\sigma}\right) \left[r_i + \gamma \max_{a'} \hat{Q}(s_i', a')\right], \qquad (4)$$

where ϕ is a kernel function, σ is the kernel bandwidth, γ is the RL discount factor, and $\eta_{s,a}$ is a normalizing term,

$$\eta_{s,a} = \sum_{i|a_i=a} \phi\left(\frac{d(s,s_i)}{\sigma}\right). \qquad (5)$$

They showed that, with enough basis points, this approximation converges to the true value function, under some reasonable assumptions. However, they provide no bound on the number of basis points needed to provide a good approximation to the value function.

Assumptions

IBRL makes a number of assumptions about the form of the value function, and the underlying state space. The main assumptions are that state similarity is well measure by (weighted) Euclidean distance. This implicity assumes that the underlying state space be metric, and is a topological disk. Essentially, this means that stattes that are close to each other in the state space have similar value. This is clearly not true for states between which the agent cannot move, such as those on the opposite sides of a thin wall. In this case, there is a discontinuity in the state space, introduced by the wall, which is not well modeled by the instance-based algorithm.

Instance-based function approximation algorithms assume that the function they model is smooth and continuous between the basis points. Any discontinuities in the function tend to get "smoothed out" in the approximation. This assumption is especially problematic for value-function approximation, since it allows value on one side of the discontinuity to affect the approximation on the other. If the location of the discontinuity is known, and we are able to allocate an arbitrary number of basis points, we can overcome this problem. However, in practical applications of RL, neither of these is feasible, and the problem of approximating the value function at or near discontinuities remains an open one.

Problems and Drawbacks

Although IBRL has been shown to be effective on a number of problems, it does have a number of drawbacks that remain unaddressed. Instance-based approximation algorithms are often expensive in terms of storage, especially for long-lived agents. Although the literature contains many techniques for editing the basis set of instance-based approximators, these techniques are generally for a supervised learning setting, where the utility of a particular edit can be easily evaluated. In the RL setting, we lack the ground truth available to supervised learning, making the evaluation of edits considerably more difficult. Additionally, as the number of basis points increases, so does the time needed to perform an approximation. This limitation is significant in the RL setting, since many such value predictions are needed on every step of the accompanying RL algorithm.

The value of a particular state, s, is calculated by blending the values from other nearby states, s_i. This is problematic if it is not possible to move from state s to each of the states s_i. The value of s should only be influenced by the value of states reachable from s, but this condition is not enforced by standard instance-based approximation algorithms. This leads to problems when modeling discontinuities in the value function, as noted above, and in situations where the system dynamics constrain the agent's motion, as in the case of a "one-way door" in the state space.

IBRL also suffers badly from the curse of dimen-sionality; the number of points needed to adequately represent the value function is exponential in the dimensionality of the state space. However, by using only states actually experienced by the learning agent, we can lessen the impact of this problem. By using only observed states, we are explicitly modeling the manifold over which the system state moves. This manifold is embedded in the full state space and, for many real-world problems, has a lower dimensionality than the full space. The Euclidean distance metric used by many instance-based algorithms will not accurately measure distance along this manifold. In practice, the manifold over which the system state moves will be locally Euclidean for problems with smooth, continuous dynamics. As a result, the assumptions of instance-based function approximators are valid locally and the approximations are of reasonable quality.

Cross-References

▸ Curse of Dimensionality
▸ Instance-Based Learning
▸ Locally Weighted Learning
▸ Reinforcement Learning
▸ Value Function Approximation

Recommended Reading

Gordon GJ (1995) Stable function approximation in dynamic programming. In: Proceedings of the twelfth international conference on machine learning, Tahoe City, pp 261–268

Kretchmar RM, Anderson CW (1997) Comparison of CMACs and radial basis functions for local function approximators in reinforcement learning. In: International conference on neural networks, Houston, vol 2, pp 834–837

Ormoneit D, Sen Ś (2002) Kernel-based reinforcement learning. Mach Learn 49(2–3):161–178

Smart WD, Kaelbling LP (2000) Practical reinforcement learning in continuous spaces. In: Proceedings of the seventeenth international conference on machine learning (ICML 2000), Stanford, pp 903–910

Szepesvári C, Munos R (2005) Finite time bounds for sampling based fitted value iteration. In: Proceedings of the twenty-second international conference on machine learning (ICML 2005), Bonn, pp 880–887

Szepesvári C, Smart WD (2004) Interpolation-based Q-learning. In: Proceedings of the twenty-first international conference on machine learning (ICML 2004), Banff, pp 791–798

Intelligent Backtracking

Synonyms

Dependency directed backtracking

Definition

Intelligent backtracking is a general class of techniques used to enhance search and constraint satisfaction algorithms. Backtracking is a general mechanism in search where a problem solver encounters an unsolvable search state and backtracks to a previous search state that might be solvable. Intelligent backtracking mechanisms provide various ways of selecting the backtracking point based on past experience in a way that is likely to be fruitful.

Intent Recognition

▸ Inverse Reinforcement Learning

Internal Model Control

Synonyms

Certainty equivalence principle; Model-based control

Definition

Many advanced controllers for nonlinear systems require knowledge of the model of the dynamics of the system to be controlled. The system dynamics is often called an "internal model," and the resulting controller is model-based. If the model is not known, it can be learned with function approximation techniques. The learned model is subsequently used as if it were correct in order to synthesize a controller – the control literature calls this assumption the "certainty equivalence principle."

Interval Scale

An **interval** measurement scale ranks the data, and the differences between units of measure can be calculated by arithmetic. However, *zero* in the interval level of measurement means neither "nil" nor "nothing" as *zero* in arithmetic means. See ▸ Measurement Scales.

Inverse Entailment

Definition

Inverse entailment is a ▸ generality relation in ▸ inductive logic programming. More specifically, when learning from entailment using a background theory B, a hypothesis H covers an example e, relative to the background theory B if and only if $B \wedge H \models e$, that is, the background theory B and the hypothesis H together entail the example (see ▸ entailment). For instance, consider the background theory B:

```
bird :- blackbird.
bird :- ostrich.
```

and the hypothesis H:

```
flies :- bird.
```

Together $B \wedge H$ entail the example e:

```
flies :- blackbird, normal.
```

This can be decided through deductive inference. Now when learning from entailment in inductive logic programming, one starts from the example e and the background theory B, and the aim is to induce a rule H that together with B entails the example. Inverting entailment is based on the observation that $B \wedge H \models e$ is logically equivalent to $B \wedge \neg e \models \neg H$, which in turn can be used to compute a hypothesis H that will cover the example relative to the background theory. Indeed, the negation of the example is $\neg e$:

```
blackbird.
normal.
:-flies.
```

and together with B this entails $\neg H$:

```
bird.
:-flies.
```

The principle of inverse entailment is typically employed to compute the ▸ bottom clause, which is the most specific clause covering the example under entailment. It can be computed by generating the set of all facts (true and false) that are entailed by $B \wedge \neg e$ and negating the resulting formula $\neg H$.

Cross-References

▸ Bottom Clause
▸ Entailment
▸ Inductive Logic Programming
▸ Logic of Generality

Inverse Optimal Control

▸ Inverse Reinforcement Learning

Inverse Reinforcement Learning

Pieter Abbeel[1] and Andrew Y. Ng[2]
[1]EECS Department, UC Berkeley, Stanford, CA, USA
[2]Computer Science Department, Stanford University, Stanford, CA, USA
[3]Stanford University, Stanford, CA, USA

Synonyms

Intent recognition; Inverse optimal control; Plan recognition

Definition

Inverse reinforcement learning (inverse RL) considers the problem of extracting a reward function from observed (nearly) optimal behavior of an expert acting in an environment.

Motivation and Background

The motivation for inverse RL is twofold:

- For many RL applications, it is difficult to write down an explicit reward function specifying how different desiderata should be traded off exactly. In fact, engineers often spend significant effort tweaking the reward function such that the optimal policy corresponds to performing the task they have in mind. For example, consider the task of driving a car well. Various desiderata have to be traded off, such as speed, following distance, lane preference, frequency of lane changes, distance from the curb, etc. Specifying the reward function for the task of driving requires explicitly writing down the trade-off between these features.

Inverse RL algorithms provide an efficient solution to this problem in the apprenticeship learning setting – when an expert is available to demonstrate the task. Inverse RL algorithms exploit the fact that an expert demonstration implicitly encodes the reward function of the task at hand.

- Reinforcement learning and related frameworks are often used as computational models for animal and human learning (Watkins 1989; Schmajuk and Zanutto 1997; Touretzky and Saksida 1997). Such models are supported both by behavioral studies and by neurophysiological evidence that reinforcement learning occurs in bee foraging (Montague et al. 1995) and in songbird vocalization (Doya and Sejnowski 1995). It seems clear that in examining animal and human behavior, we must consider the reward function as an unknown to be ascertained through empirical investigation, particularly when dealing with multiatttribute reward functions. Consider, for example, that the bee might weigh nectar ingestion against flight distance, time, and risk from wind and predators. It is hard to see how one could determine the relative weights of these terms a priori. Similar considerations apply to human economic behavior, for example. Hence, inverse reinforcement learning is a fundamental problem of theoretical biology, econometrics, and other scientific disciplines that deal with reward-driven behavior.

Structure of the Learning System

Preliminaries and Notation

A Markov decision process (MDP) is a tuple $\langle S, A, T, \gamma, D, R\rangle$, where S is a finite set of states, A is a set of actions, $T = \{P_{sa}\}$ is a set of state-transition probabilities (here, P_{sa} is the state transition distribution upon taking action a in state s), $\gamma \in [0,1)$ is a discount factor, D is the distribution over states for time zero, and $R : S \mapsto \mathbb{R}$ is the reward function.

A policy π is a mapping from states to probability distributions over actions. We let Π denote the set of all stationary policies (We restrict attention to stationary policies, since it is well known that there exists a stationary policy that is optimal for infinite horizon MDPs.). The value of a policy π is given by

$$V(\pi) = \mathrm{e}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)|\pi\right].$$

The expectation is taken with respect to the random state sequence $s_0, s_1, s_2, \ldots$ drawn by starting from a state $s_0 \sim D$ and picking actions according to π.

Let $\mu_S(\pi)$ be the discounted distribution over states when acting according to the policy π. In particular, for a discrete state space, we have that $[\mu_S(\pi)](s) = \sum_{t=0}^{\infty} \gamma^t \mathrm{Prob}(s_t = s|\pi)$. (In the case of a continuous state space, we replace $\mathrm{Prob}(s_t = s|\pi)$ by the appropriate probability density function.) Then, we have that

$$V(\pi) = R^\top \mu_S(\pi).$$

Thus, the value of a policy π when starting from a state s_0 is linear in the reward function.

Often the reward function R can be represented more compactly. Let $\phi : S \to \mathbb{R}^n$ be a feature map. A typical assumption in inverse RL is to assume the reward function R is a linear combination of the features ϕ: $R(s) = w^\top \phi(s)$. Then, we have that the value of a policy π is linear in the reward function weights w:

$$\begin{aligned} V(\pi) &= E[\textstyle\sum_{t=0}^{\infty} \gamma^t R(s_t)|\pi] \\ &= E[\textstyle\sum_{t=0}^{\infty} \gamma^t w^\top \phi(s_t)|\pi] \\ &= w^\top E[\textstyle\sum_{t=0}^{\infty} \gamma^t \phi(s_t)|\pi] \\ &= w^\top \mu_\phi(\pi). \end{aligned} \tag{1}$$

Here, we used linearity of expectation to bring w outside of the expectation. The last equality defines the vector of *feature expectations* $\mu_\phi(\pi) = E\big[\sum_{t=0}^{\infty} \gamma^t \phi(s_t)|\pi\big]$.

We assume access to demonstrations by some expert. We denote the expert's policy by π^*. Specifically, we assume the ability to observe trajectories (state sequences) generated by the expert starting from $s_0 \sim D$ and taking actions according to π^*.

Characterization of the Inverse RL Solution Set

A reward function R is consistent with the policy π^* being optimal if and only if the value obtained when acting according to the policy π^* is at least as high as the value obtained when acting according to any other policy π, or equivalently:

$$U(\pi^*) \geq U(\pi) \quad \forall \pi \in \Pi. \tag{2}$$

Using the fact that $U(\pi) = R^\top \mu_S(\pi)$, we can equivalently write the conditions of Eq. (2) as a set of linear constraints on the reward function R:

$$R^\top \mu_S(\pi^*) \geq R^\top \mu_S(\pi) \quad \forall \pi \in \Pi. \tag{3}$$

The state distribution $\mu_S(\pi)$ does not depend on the reward function R. Thus, Eq. (3) is a set of linear constraints in the reward function, and we can use a linear program (LP) solver to find a reward function consistent with the policy π^* being optimal. Strictly speaking, Eq. (3) solves the inverse RL problem. However, to apply inverse RL in practice, the following three issues need to be addressed:

1. **Reward function ambiguity.** Typically a large set of reward functions satisfy all the constraints of Eq. (3). One such a reward function that satisfies all the constraints for any MDP is the all-zeros reward function (it is consistent with any policy being optimal). Clearly, the all-zeros reward function is not a desirable answer to the inverse RL problem. More generally, this observation suggests not all reward functions satisfying Eq. (3) are of equal interest and raises the question of how to recover reward functions that are of interest to the inverse RL problem.
2. **Statistical efficiency.** Often the state space is very large (or even infinite), and we do not have sufficiently many expert demonstrations available to accurately estimate $\mu(\cdot\,; \pi^*)$ from data.
3. **Computational efficiency.** The number of constraints in Eq. (3) is equal to the number of stationary policies $|\Pi|$ and grows quickly with the number of states and actions of the MDP. For finite-state-action MDPs, we have $|A|^{|S|}$ constraints. So, even for small state and action spaces, feeding all the constraints of Eq. (3) into an LP solver becomes quickly impractical. For continuous state-action spaces, the formulation of Eq. (3) has an infinite number of constraints, and thus using a standard LP solver to find a feasible reward function R is impossible.

In the following sections, we address these three issues.

Reward Function Ambiguity

As observed above, typically a large set of reward functions satisfy all the constraints of Eq. (3). To obtain a single reward function, it is natural to reformulate the inverse RL problem as an optimization problem. We describe one standard approach for disambiguation. Of course many other formulations as an optimization problem are possible.

Similar to common practice in support vector machines research, one can maximize the (soft) margin by which the policy π^* outperforms all other policies. As is common in structured prediction tasks ((see, e.g., Taskar et al. (2003)), one can require the margin by which the policy π^* outperforms another policy π to be larger when π differs more from π^*, as measured according to some function $h(\pi^*, \pi)$. The resulting formulation (Ratliff et al. 2006) is

$$\begin{aligned} \min_{R,\xi} \quad & \|R\|_2^2 + C\xi \\ \text{s.t. } & R^\top \mu_S(\pi^*) \geq R^\top \mu_S(\pi) + h(\pi^*, \pi) - \xi \\ & \forall \pi \in \Pi. \end{aligned} \tag{4}$$

For the resulting optimal reward function to correspond to a desirable solution to the inverse RL problem, it is important that the objective and the margin scaling encode the proper prior knowledge. If a sparse reward function is suggested by prior knowledge, then a 1-norm might be more appropriate in the objective. An example of a margin scaling function for a discrete MDP is the number of states in which the action prescribed by the policy π differs from the action prescribed by the expert policy π^*. If the expert has only been observed in a small number of states, then one could restrict attention to these states when evaluating this margin scaling function.

Another way of encoding prior knowledge is by restricting the reward function to belong to a certain functional class, for example, the set of functions linear in a specified set of features. This approach is very common and is also important for statistical efficiency. It will be explained in the next section.

Remark When using inverse RL to help us specify a reward function for a given task based on an expert demonstration, it is not necessary to explicitly resolve the ambiguities in the reward function. In particular, one can probably perform as well as the expert without matching the expert's reward function. More details are given in Sect. "A Generative Approach to Inverse RL".

Statistical Efficiency

As formulated thus far, solving the inverse RL problem requires the knowledge (or accurate statistical estimates) of $\mu_S(\pi^*)$. For most practical problems, the number of states is large (or even infinite), and thus accurately estimating $\mu_S(\pi^*)$ requires a very large number of expert demonstrations. This (statistical) problem can be resolved by restricting the reward function to belong to a prespecified class of functions. The common approach is to assume the reward function R can be expressed as a linear combination of a known set of features. In particular, we have $R(s) = w^\top \phi(s)$. Using this assumption, we can use the expression for the value of the policy π from Eq. (1).

Rewriting Eq. (4), we now have the following constraints in the reward weights w:

$$\begin{aligned} \min_{w,\xi} \quad & \|w\|_2^2 + C\xi \\ \text{s.t. } & w^\top \mu_\phi(\pi^*) \geq w^\top \mu_\phi(\pi) + h(\pi^*, \pi) - \xi \\ & \forall \pi \in \Pi. \end{aligned} \tag{5}$$

This new formulation only requires estimates of the expected feature counts $\mu_\phi(\pi^*)$, rather than estimates of the distribution over the state space $\mu_S(\pi^*)$. Assuming the number of features is smaller than the number of states, this significantly reduces the number of expert demonstrations required.

Computational Efficiency

For concreteness, we will consider the formulation of Eq. (6). Although the number of variables is only equal to the number of features in the reward function, the number of constraints is very large (equal to the number of stationary policies). As a consequence, feeding the problem into a standard quadratic programming (QP) solver will not work.

Ratliff et al. (2006) suggested a formal computational approach to solving the inverse RL problem, using standard techniques from convex optimization, which provide convergence guarantees. More specifically, they used a subgradient method to optimize the following equivalent problem:

$$\min_{w,\xi} \|w\|_2^2 + C \max_{\pi \in \Pi} \left(w^\top \mu_\phi(\pi) + h(\pi^*, \pi) - w^\top \mu_\phi(\pi^*) \right). \tag{6}$$

In each iteration, to compute the subgradient, it is sufficient to find the optimal policy with respect to a reward function that is easily determined from the current reward weights w and the margin scaling function $h(\pi^*, \cdot)$. In more recent work, Ratliff et al. (2007) proposed a boosting algorithm to solve a formulation similar to Eq. (6), which also includes feature selection.

A Generative Approach to Inverse RL

Abbeel and Ng (2004) made the following observation, which resolves the ambiguity problem in a completely different way: if, for a policy π, we have that $\mu_\phi(\pi) = \mu_\phi(\pi^*)$, then the following holds:

$$U(\pi) = w^\top \mu_\phi(\pi) = w^\top \mu_\phi(\pi^*) = U(\pi^*),$$

no matter what the value of w is. Thus, to perform as well as the expert, it is sufficient to find a policy that attains the same expected feature counts μ_ϕ as the expert.

Abbeel and Ng provide an algorithm that finds a policy π satisfying $\mu_\phi(\pi) = \mu_\phi(\pi^*)$. The algorithm iterates over two steps: (i) Generate a reward function by solving a QP. (ii) Solve the MDP for the current reward function.

In contrast to the previously described inverse RL methods, which focus on merely recovering a reward function that could explain the expert's behavior, this inverse RL algorithm is shown to find a policy that performs at least as well as the expert. The algorithm is shown to converge in a polynomial number of iterations.

Apprenticeship Learning: Inverse RL Versus Imitation Learning

Inverse RL alleviates the need to specify a reward function for a given task when expert demonstrations are available. Alternatively, one could directly estimate the policy of the expert using standard a machine-learning algorithm, since it is simply a mapping from state to action. The latter approach, often referred to as imitation learning or behavioral cloning (links), has been successfully tested on a variety of tasks, including learning to fly in a fixed-wing flight simulator (Sammut et al. 1992) and learning to drive a car (Pomerleau 1989; Abbeel and Ng 2004).

The imitation learning approach can be expected to be successful whenever the policy class to be considered can be learned efficiently from data. In contrast, the inverse RL approach relies on having a reward function that can be estimated efficiently from data.

Cross-References

▶ Apprenticeship Learning
▶ Reinforcement Learning
▶ Reward Shaping

Recommended Reading

Abbeel P, Ng AY (2004) Apprenticeship learning via inverse reinforcement learning. In: Proceedings of ICML, Alberta

Doya K, Sejnowski T (1995) A novel reinforcement model of birdsong vocalization learning. Neural Inf Process Syst 7:101

Montague PR, Dayan P, Person C, Sejnowski TJ (1995) Bee foragin in uncertain environments using predictive hebbian learning. Nature 377(6551):725–728

Pomerleau D (1989) Alvinn: an autonomous land vehicle in a neural network. In: NIPS 1, Denver

Ratliff N, Bagnell J, Zinkevich M (2006) Maximum margin planning. In: Proceedings of ICML, Pittsburgh

Ratliff N, Bradley D, Bagnell J, Chestnutt J (2007) Boosting structured prediction for imitation learning. Neural Inf Process Syst 19:1153–1160

Sammut C, Hurst S, Kedzier D, Michie D (1992) Learning to fly. In: Proceedings of ICML, Aberdeen

Schmajuk NA, Zanutto BS (1997) Escape, avoidance, and imitation. Adapt Behav 6:63–129

Taskar B, Guestrin C, Koller D (2003) Max-margin markov networks. In: Neural information processing systems conference (NIPS03), Vancouver

Touretzky DS, Saksida LM (1997) Operant conditioning in skinnerbots. Adapt Behav 5:219–47

Watkins CJ (1989) Models of delayed reinforcement learning. Ph.D. thesis, Psychology Department, Cambridge University

Inverse Resolution

Definition

Inverse resolution is, as the name indicates, a rule that inverts resolution. This follows the idea of induction as the inverse of deduction formulated in the ▶ logic of generality. The resolution rule is the best-known deductive inference rule, used in many theorem provers and logic programming systems. ▶ Resolution starts from two ▶ clauses and derives the resolvent, a clause that is entailed by the two clauses. This can be graphically represented using the following schema (for propositional logic).

$$\frac{h \leftarrow g, a_1, \ldots, a_n \text{ and } g \leftarrow b_1, \ldots, b_m}{h \leftarrow b_1, \ldots, b_m, a_1, \ldots, a_n}.$$

Inverse resolution operators, such as *absorption* (17) and *identification* (17), invert this process.

To this aim, they typically assume the resolvent is given together with *one* of the original clauses and then derive the missing clause. This leads to the following two operators, which start from the clauses below and induce the clause above the line.

$$\frac{h \leftarrow g, a_1, \ldots, a_n \text{ and } g \leftarrow b_1, \ldots, b_m}{h \leftarrow b_1, \ldots, b_m, a_1, \ldots, a_n \text{ and } g \leftarrow b_1, \ldots, b_m},$$

$$\frac{h \leftarrow g, a_1, \ldots, a_n \text{ and } g \leftarrow b_1, \ldots, b_m}{h \leftarrow b_1, \ldots, b_m, a_1, \ldots, a_n \text{ and } h \leftarrow g, a_1, \ldots, a_n}$$

The operators are shown here only for the propositional case, as the first order case is more involved as it requires one to deal with substitutions as well as inverse substitutions.

As one example, consider the clauses

1. flies :- bird, normal.
2. bird :- blackbird.
3. flies :- blackbird, normal.

Here, (3) is the resolvent of (1) and (2). Furthermore, starting from (3) and (2), the absorption operator would generate (1), and starting from (3) and (1), the identification operator would generate (2).

Cross-References

▶ First-Order Logic
▶ Logic of Generality

Is More General Than

▶ Logic of Generality

Is More Specific Than

▶ Logic of Generality

Isotonic Calibration

▶ Classifier Calibration

Item

▶ Instance

Item Space

▶ Instance Space

Iterative Algorithm

▶ K-Medoids Clustering

Iterative Classification

▶ Collective Classification

Iterative Computation

▶ K-Means Clustering