Learning Novel Concepts in the Kinship Domain

Daniel M. Roy
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Abstract

This paper addresses the role that novel concepts play in learning good theories. To concretize the discussion, I use Hinton's kinship dataset as motivation throughout the paper. The standpoint taken in this paper is that the most compact theory that describes a set of examples is the preferred theory—an explicit Occam's Razor. The kinship dataset is a good test-bed for thinking about relational concept learning because it contains interesting patterns that will undoubtedly be part of a compact theory describing the examples. To begin with, I describe a very simple computational level theory for inductive theory learning in first-order logic that precisely states that the most compact theory is preferred. In addition, I illustrate the obvious result that predicate invention is a necessary part of any system striving for compact theories. I present derivations within the Inductive Logic Programming (ILP) framework that show how the intuitive theories of family trees can be learned. These results suggest that encoding regular equivalence directly into the training sets of ILP systems can improve learning performance. To investigate theories resulting from optimization, I devise an algorithm that works with a very strict language bias allowing all consistent rules to be entertained and explicitly optimized over for small datasets. The algorithm, which can be viewed as a special case implementation of ILP, is capable of learning a theory of kinship comparable in compactness to the intuitive theories humans use regularly. However, this alternative approach falls short as it is incapable of inventing the unary predicate *sex* to learn a more compact theory. Finally, I comment on the philosophical position of extreme nativism in light of the ability of these systems to invent primitive concepts not present in the training data.

Introduction

The core of the intuitive theory of kinship in western culture is the family tree, from which any number of queries about kinship relationships can be answered. Could a machine, presented with the kinship relationships between individuals in a family, learn the intuitive family tree representation?

This paper focuses heavily on a dataset introduced in Hinton (1986). In this dataset, a group of individuals are related by the following relations: *father, mother, husband, wife, son, daughter, brother, sister, uncle, aunt, nephew, niece.* The family tree representation efficiently encodes all of these relationships using a basis set composed of spousal relationships, parent/child relationships and the sex attribute.[1] To learn this theory, a machine would have to first invent the basis set and then redefine the existing relations in terms of this basis set.[2] How could a machine discover such a basis set?

According to my computational level theory, the basis set is not discovered at all. Rather, it is a byproduct of an optimization process that searches for the most compact theory that entails a set of examples. At the algorithmic level, the basis set could possibly be discovered through the process of local optimizations that lead to more compact theories.

While the computational approach is not computable in general because of the semi-decidability of first order logic, there has been great success at the algorithmic level in the field of Inductive Logic Programming (ILP). The problem ILP addresses is: learn a first-order logic theory that, together with provided background knowledge, logically entails a set of examples (Nienhuys-Cheng and de Wolf, 1997).

Using the ILP framework, it is possible to show how inverse resolution can devise all three of the basis set predicates that comprise the family tree representation. The most interesting result is the discovery of *sex* which requires that logical encodings of regular equivalence classes can be combined in an inverse resolution step to generate the new predicate.[3] This result suggests that explicitly encoding regular equivalence and other second-order properties of relational datasets may contribute to their learnability.

To investigate the computational level, I devise a special-case version of ILP that is optimized to use a very strict set of restrictions on the type of theories it can entertain. By trading expressibility for tractability, it is possible to explicitly optimize over the set of all possible rules for each relation individually. Unfortunately, optimizing across the relations is intractable. The resulting rules can be further compressed by using inverse resolution to invent new predicates that simplify existing ones. The resulting theory for the kinship domain is comparable in compactness to the family tree representation.

[1] The sex of the individual is often implicitly specified by the gender of the name.

[2] Personal communication and class notes of J. Tenenbaum (Tenenbaum, 2004)

[3] The regular equivalence classes for the kinship dataset are all pairs of generations and sex in the family tree White and Reitz (1983); Kemp et al. (2004).

Computational Level

Here is a short, computational account of inductive learning that explicitly prefers compact theories. An example in this framework shows why novel concepts necessarily arise in learning the most compact theory.

Essentially, given examples $\mathcal{E}$ and background knowledge $\mathcal{B}$, we are searching for the shortest first-order logic theory $\mathcal{T}$ such that $\mathcal{B} \wedge \mathcal{T} \models \mathcal{E}$. If we do not restrict the form of the theory $\mathcal{T}$ then this optimization problem is not computable as logical entailment is semi-decidable in full clausal logic (Nienhuys-Cheng and de Wolf, 1997). A possible variant of this optimization is to bound the number of steps required to prove entailment. For example, find the shortest theory that describes the kinship examples which takes fewer than n steps to prove entailment. If we restrict $\mathcal{T}$ to Horn clauses then we can guarantee decidability of entailment (Nienhuys-Cheng and de Wolf, 1997) and, therefore, we can solve the original optimization problem. The search through programs can be ordered by evaluating $\mathcal{B} \wedge \mathcal{T} \models \mathcal{E}$ for all programs of length one, then all programs of length two, and so on. The process is bounded above by the length of the input examples. The first such program that logically entails the examples is the most compact theory.

Of course, the time complexity of this optimization grows combinatorially with additional objects, examples, and predicates; Investigating non-trivial concept invention by direct optimization is clearly intractable.

The above computational theory is closely related to the Kolmogorov complexity. While inductive theory learning is concerned with developing the most compact theories that explain a set of examples, the Kolmogorov complexity is simply equal to the length of the most compact theory (up to an additive constant). More precisely, the Kolmogorov complexity of a set of examples, $K(E)$, satisfies the following inequality with respect to the length of the compact theory found by the above optimization, $K^*(E)$ (Grunwald and Vitanyi, 2004):

$$K(E) < K^*(E) + O(1)$$

A quick example shows how search for a compact theory necessarily involves the invention of new predicates. Consider Figure 1.

On the left of this figure are a set of examples. This example assumes a "closed world," which implies that if $E \not\models \psi$ then we can assume $\neg\psi$ (Nienhuys-Cheng and de Wolf, 1997). In a set of complete ground clauses, this means that any pair $\langle A, B \rangle$ not mentioned in some relation R implies $\neg R(A, B)$. In addition, the set of objects mentioned in the examples are distinct and no other objects exist. On the right is the most compact theory composed of Horn clauses.[4] The theory employs a predicate not present in the examples. This example proves that predicate invention is a necessary feature of optimal compression. Unfortunately, it is intractable to determine the optimal theory for examples of even moderate size by brute-force search. There have been several attempts to tackle this problem with more elegance. One such attempt is called Inductive Logic Programming (Nienhuys-Cheng and de Wolf, 1997).

Algorithmic Level

ILP is an algorithmic level approach to the above computational level theory. ILP searches through the space of theories guided by heuristics that seek out compact theories that are consistent with the examples. However, ILP systems make no guarantee of optimality. In particular, very few ILP systems available are capable of predicate invention, a necessary prerequisite for optimal theories as shown in the previous section.

Inventing New Predicates in ILP

Regardless of the performance of actual ILP systems on these problems, it is useful to ask whether these systems could, in principle, derive the expected relationships from the data. First, given the basis set, could an ILP system learn the intuitive theories? The PROGOL ILP system developed by Muggleton can learn a rule for *aunt* given *parent* and *sister* predicates and a few positive examples (Muggleton, 2004). That answered, could an ILP system learn the *parent* predicate on its own?

There are several methods in the ILP literature by which predicates can be invented (Nienhuys-Cheng and de Wolf, 1997, pg. 176) (Muggleton and Buntine, 1988). The most straightforward was developed in Muggleton and Buntine (1988) and is known as inverse resolution. Figure 2 shows the mechanism of intra-construction, one of two types of inverse resolution.

$$\frac{p \Leftarrow \psi, B, \phi \qquad\qquad\qquad p \Leftarrow \psi, C, \phi}{q \Leftarrow B \qquad p \Leftarrow \psi, q, \phi \qquad q \Leftarrow C}$$

Figure 2: Predicate invention via inverse resolution: intra-construction

Returning back to the idea of compression and deriving the most compact theory, how does the intra-construction rule affect the compactness of the resulting theory? By defining a metric $|\cdot|$ of the complexity of a term and assuming that the metric operates compositionally (the metric of a complex term is the sum of the metric of its parts), it can be shown that the intra-construction results in a more compact theory in the case that:

$$\begin{aligned} 2|p, \psi, \phi| + |B, C| &> |p, \psi, \phi| + 3|q| + |B, C| \\ |p, \psi, \phi| &> 3|q| \end{aligned} \tag{1}$$

Assuming that $|\cdot| = 1$, then p, ψ and ϕ need only contain 3 clauses between them to make this derivation more optimal than the antecedent.

Returning to the kinship example, Figure 3 is a derivation of the *parent* predicate using the above intra-construction rule. Essentially, the invention of the *parent* predicate is the result of a pressure to search for a compact theory. In the derivation

[4] The program being written to prove this (by explicitly searching) is unfinished. However, while I am not proof positive this is the most compact theory in Horn clauses, I am fairly certain.

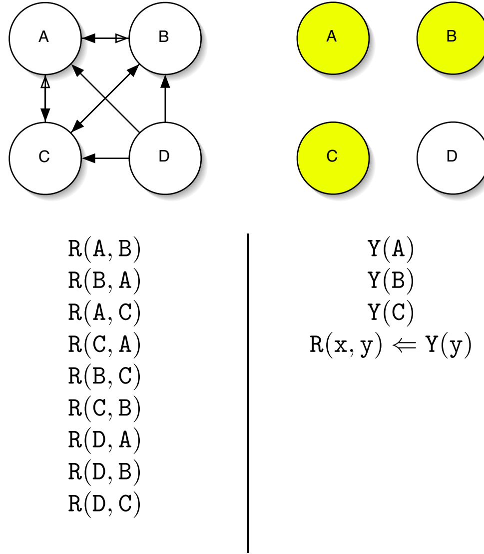


Figure 1: Learning Novel Concepts via Optimal Compression (Horn Clauses)

below, two rules for grandfather differ in that one describes a mother's father and the other a father's father. These differences are combined in an intra-construction derivation, creating the *parent* predicate.

$$
\begin{array}{rcl}
\text{grandfather}(x,y) & \Leftarrow & \text{mother}(z,y), \text{father}(x,z) \\
\text{grandfather}(x,y) & \Leftarrow & \text{father}(z,y), \text{father}(x,z) \\
\text{grandmother}(x,y) & \Leftarrow & \text{mother}(z,y), \text{mother}(x,z) \\
\text{grandmother}(x,y) & \Leftarrow & \text{father}(z,y), \text{mother}(x,z) \\
\hline
\text{parent}(x,y) & \Leftarrow & \text{mother}(x,y) \\
\text{parent}(x,y) & \Leftarrow & \text{father}(x,y) \\
\text{grandfather}(x,y) & \Leftarrow & \text{parent}(z,y), \text{father}(x,z) \\
\text{grandmother}(x,y) & \Leftarrow & \text{parent}(z,y), \text{mother}(x,z)
\end{array}
$$

Figure 3: Intra-construction derivation of *parent*

Figure 4, below, shows how the *spouse* predicate could be invented. Again, the invention of the *spouse* predicate aids to the compactness of the theory. In this example we have two rules that describe the set of positive examples of *mother-in-law*. They differ solely in whether the clause describes a husband's mother or a wife's mother. Via intra-construction, the differences between these two rules are merged into a new predicate, *spouse*, and the original rule is rewritten to use this new predicate.

$$
\begin{array}{rcl}
\text{mother-in-law}(x,y) & \Leftarrow & \text{wife}(z,y), \text{mother}(x,z) \\
\text{mother-in-law}(x,y) & \Leftarrow & \text{husband}(z,y), \text{mother}(x,z) \\
\text{father-in-law}(x,y) & \Leftarrow & \text{wife}(z,y), \text{father}(x,z) \\
\text{father-in-law}(x,y) & \Leftarrow & \text{husband}(z,y), \text{father}(x,z) \\
\hline
\text{spouse}(x,y) & \Leftarrow & \text{wife}(x,y) \\
\text{spouse}(x,y) & \Leftarrow & \text{husband}(x,y) \\
\text{mother-in-law}(x,y) & \Leftarrow & \text{spouse}(z,y), \text{mother}(x,z) \\
\text{father-in-law}(x,y) & \Leftarrow & \text{spouse}(z,y), \text{father}(x,z)
\end{array}
$$

Figure 4: Intra-construction derivation of *spouse*

What about the *sex* predicate? Using only the predicates available in the system it is unclear how an intra-construction rule could produce a predicate whose meaning can be understood to represent the sex of an individual. Inverse resolution alone is insufficient. However, if the example dataset is augmented with predicates that describe the regular equivalence of the kinship tree, it then becomes clear how the *sex* predicate could be invented.[5]

In the derivation below, a binary predicate $REGE(x, c)$ asserts that an object x belongs to an equivalence class c. Using the same intra-construction rule used to derive *spouse* and *parent*, it now becomes clear how the *sex* predicate could be invented.

$$
\begin{array}{rcl}
\text{mother}(x,y) & \Leftarrow & \text{parent}(x,y), \text{REGE}(x,1) \\
\text{mother}(x,y) & \Leftarrow & \text{parent}(x,y), \text{REGE}(x,3) \\
\text{mother}(x,y) & \Leftarrow & \text{parent}(x,y), \text{REGE}(x,5) \\
\text{father}(x,y) & \Leftarrow & \text{parent}(x,y), \text{REGE}(x,2) \\
\text{father}(x,y) & \Leftarrow & \text{parent}(x,y), \text{REGE}(x,4) \\
\text{father}(x,y) & \Leftarrow & \text{parent}(x,y), \text{REGE}(x,6) \\
\hline
\text{female}(x,y) & \Leftarrow & \text{REGE}(x,1) \\
\text{female}(x,y) & \Leftarrow & \text{REGE}(x,3) \\
\text{female}(x,y) & \Leftarrow & \text{REGE}(x,5) \\
\text{male}(x,y) & \Leftarrow & \text{REGE}(x,2) \\
\text{male}(x,y) & \Leftarrow & \text{REGE}(x,4) \\
\text{male}(x,y) & \Leftarrow & \text{REGE}(x,6) \\
\text{mother}(x,y) & \Leftarrow & \text{parent}(x,y), \text{female}(x) \\
\text{father}(x,y) & \Leftarrow & \text{parent}(x,y), \text{male}(x)
\end{array}
$$

Figure 5: Intra-construction derivation of *male/female*

This result is interesting as it suggests that latent information in networks of relationships (like regular equivalence) could aid in the learning of relational data. Perhaps regular equivalence and similar meta-data that make explicit latent structure could act as "kernel tricks" for ILP systems, adding additional dimensions to the input data to make it easier to learn.

[5]If we were working within second-order logic, we could have the system recognize and report the regular equivalence classes automatically. This suggest that moving to the higher order logics may provide real advantages in learnability.

Graph Compression

In this section, an alternative to generic inductive logic programming is introduced that works by compressing graph representations of sets of ground binary relations. The system can be understood as performing a similar task as ILP with a language bias that restricts the form of the theories the system is capable of entertaining. The algorithm uses inverse resolution to invent new predicates when patterns exist that match the antecedent of the intra-construction rule.

Language Bias

Most ILP systems operate with a language bias that is either implicit (built into the system) or user-specified. The language bias can take the form of a restriction in the length of clauses, types of literals, or even grammars of allowable clauses. A language bias limits the search space of possible hypothesis. In this system, the examples presented to the system are positive examples of binary predicates. Negative examples are implied via a closed world assumption. Such a collection of binary predicates can be represented as a graph whose edges represent relations between objects.

The system learns theories of a very strict form which makes the compression problem very simple. The grammar of this subset of first-order clausal logic is shown as Figure 7. It should be explicitly noted that this language bias is highly specific to the kinship relationships we wish to learn. The resulting system is by no means intended to be considered a serious contribution to ILP. Instead, it is an attempt to think about the problems normally tackled by ILP from a standpoint more aligned with the (compression-based) computational theory outlined earlier. Because of the strict language bias, we are able to greedily optimize to find compact theories.

In summary, the language bias was chosen to simplify the compression algorithm. Regardless, the combination of predicate invention with this simple compression mechanism results in the formation of compact theories of kinship.

$$
\begin{array}{rcl}
R(A,B) & & \\
R(x,y) & \Leftarrow & R^1(n_1,x), \\
& & R^2(n_2,n_1), \\
& & \vdots \\
& & R^3(y,n_m), \\
& & x,n_1,...,n_m,y \text{ are distinct} \\
& & \forall i.R \neq R^i
\end{array}
$$

Figure 7: Language Bias

An interesting aspect of this language bias is that each rule can be at most the length of the number of objects because the grammar requires that the universally quantified variables be distinct. Therefore, there are a finite number of theories that can describe any finite set of positive examples. In addition, the grammar prevents recursive definitions by requiring that the head relation not exist in the body of the clause. These restrictions tradeoff expressibility for tractability. We use these simplifications to our advantage to allow some extent of explicit optimization in the process of devising compact theories.

Problem Description

This section describes the graph compression problem setup. We are given a graph $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$, where $\mathcal{N}$ is a set of nodes and $\mathcal{E}$ is a set of labeled edges $\mathcal{N} \times \mathcal{N} \times \mathbb{N}$. We are interested in finding a new graph $\mathcal{G}' = \langle \mathcal{N}, \mathcal{E}' \rangle$, where $\mathcal{E}' \subseteq \mathcal{E}$, and a set of rules $\mathscr{R}$ of the form in Figure 7 such that:

$$
\begin{array}{c}
\arg\min_{\mathscr{R}} \alpha \; |\mathscr{R}| + \beta |\mathcal{E}'| \\
\mathscr{R}^\infty(\mathcal{G}') \equiv_{\mathcal{G}} \mathcal{G}
\end{array}
$$

where $|\cdot|$ is a metric we use to measure complexity, and $R^\infty(\mathcal{G}')$ is the application of the rules to the graph until a fixed point is reached. The fixed point, $\mathscr{R}^\infty(\mathcal{G}')$, is equivalent to the original graph $\mathcal{G}$ once all edges whose labels do not exist in $\mathcal{G}$ have been removed.

Even this (much simpler) optimization problem is intractable (optimizing over the ordering of relations grows as $O(n!)$ where n is the number of relations.[6]). Therefore, we decompose the optimization into separate optimizations for each heuristic and greedily choose one heuristic at a time. Once chosen, the remaining relations are re-optimized separately. This process continues until all the relations are learned.

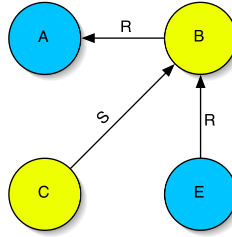
Greedy Implementation

The implementation of the graph compression optimization is written in MzScheme and is available in Appendix A. The input to the system is a graph representing the examples.

The algorithm first creates all valid rules for each of the relations in the graph (there are finitely many as explained above). This is done by creating a set of candidate rules for every positive instance of each relation. Candidate rules for a relation $\mathcal{E}_l(x, y)$ are created by enumerating every acyclic path between nodes x and y using only edges with labels $l' \neq l$. This candidate list is then pruned by verifying that no candidate rule implies a negative example. For each relation, there is a set of consistent candidate rules for each positive instance. These paths are then ordered according to a complexity measure that prefers short rules that explain the most positive examples. The final candidate rule for a relation is then formed by the disjunction of the best candidate rule for each positive instance. In practice, a single rule or disjunction of a few general rules describes the entire set of positive instances. By construction, the rule is consistent with the positive and negative examples.

At each iteration, a relation rule is chosen from the candidates by picking the most compact rule that conflicts with the fewest other rules (e.g. removing *sister* early in the process causes other rules to become much larger while removing

[6] Arguably this is not proven and could be solvable efficiently. However, there does not appear to be a problem decomposition that would lead to a dynamic programming solution.



$$\texttt{Hazel(A). Yellow(B). Yellow(C). Hazel(E).}$$
$$\texttt{R(B, A). S(C, B). R(E, B).}$$

Figure 6: Sample Graph and First-order Logic Ground Clause Equivalent

aunt has little affect on later rules). To prevent mutual recursion, all candidate rules using the selected relation are pruned. This process repeats until there are no consistent rules that describe the remaining relations. These relations are the *basis* relations, meaning that all the other relations are definable in terms of these relations.

These rules are then improved by looking for ways to combine disjunctions by predicate invention via inverse resolution. These new relations are instantiated as new edges in the graph. This process of generating the basis relations and looking for new ways to combine disjunctions continues until there are no candidate disjunctions remaining. At this point, a final basis set and derived set are generated. The basis set are ground clauses and the derived set are rules. By construction, the "application" of the rules to the set of ground clauses recreates the original graph.

Example: Kinship Data The graph compression algorithm was designed specifically with the kinship dataset in mind. In the kinship dataset, a group of individuals are described by a set of relations such as *mother, father, son, daughter, uncle, aunt, sister, etc.*. The usual representation of kinship relationships in western culture is the family tree. This representations is a very efficient way of representing the data (see Figure 8) and is most likely used because of this quality. The complete specification of every relationship in a sizable family tree will be much larger than the equivalent specification of the family tree and the set of rules to derive the remaining relations.

The kinship dataset contains certain patterns, each of which is addressed by a specific area in the graph compression algorithm. Kinship relationships are defined in terms of paths between individuals and the names of the steps in these paths. Our language bias matches this observation exactly and therefore we can expect to find theories that match our intuitive ones.

Walk-through The first pass through the kinship data results in the following rules:

```
basis set:
(husband mother wife son daughter)

rules (derived set):
((father ((mother husband) . 6))
 (sister ((father daughter) . 3))
 (brother ((father son) . 3))
 (niece
  ((husband sister daughter) . 1)
  ((sister daughter) . 1)
  ((brother daughter) . 1)
  ((wife brother daughter) . 1))
 (nephew ((niece brother) . 4))
 (aunt ((mother brother wife) . 2) ((father sister) . 2))
 (uncle ((aunt husband) . 4)))
```

The basis set is *husband, mother, wife, son, daughter*. Two of the disjunctions for the *niece* rule overlap. Intra-construction can derive a new predicate we know as *sibling* that is the disjunction of *sister* and *brother*. With the new predicate added to the graph, the rules are re-learned.

```
basis set:
(husband mother wife daughter)

derived set:
((son ((daughter sibling) . 6))
 (father ((mother husband) . 6))
 (sister ((father daughter) . 3))
 (brother ((father son) . 3))
 (niece
  ((husband sister daughter) . 1)
  ((sibling daughter) . 2)
  ((wife sibling daughter) . 1))
 (nephew ((niece brother) . 4))
 (aunt ((mother sibling wife) . 2) ((father sibling) . 2))
 (uncle ((aunt husband) . 4)))
```

No further simplifications can be found by applying intra-construction and so the basis/derived sets represents the final solution of the graph compression algorithm. Because the algorithm is able to learn only a very restricted set of theories, there is no way that the algorithm could learn the common family tree representation because this representation requires clauses not allowed by the language bias. In addition the graph compression mechanism cannot be extended to new language biases. However, the final basis set encodes the fundamental aspects of the family tree representation (*husband, mother, wife, daughter*), albeit less efficiently than one that includes a sex attribute and uses it to further compress the representation. Another disappointing aspect of these results is that the rules, though compact, do not match those generally used by humans to explain kinship relationships. For example, the uncle rule is simply "aunt's husband" which is technically correct with respect to this dataset, but not as intuitive as "parent's brother (or brother-in-law)." The reason the algorithm performs in this manner is that it greedily chooses the best rule at each iteration, largely ignoring the effect of these decisions on the overall optimality of the resulting theory. While the descriptions of the family relations are sometimes non-intuitive, they are as compact as those in the intuitive theory.[7]

[7] Personal communication and class notes of J. Tenenbaum (Tenenbaum, 2004)

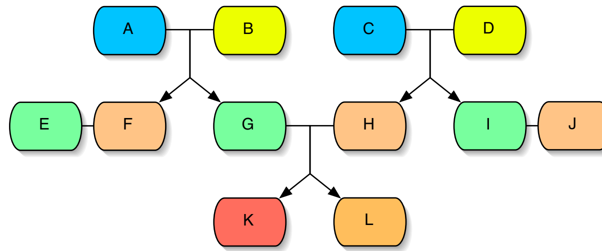


Figure 8: Efficient Family Tree Representation: Horizontal edges represent marriage, Vertical represent parent/child links. The nodes are colored according to their regular equivalence classes.

Philosophy

One of the central questions in philosophy of mind and cognitive science is how humans generate new concepts. Extreme nativists, like Jerry Fodor, deny this is even possible, believing instead that we are born with every concept we use in life and that learning is simply the process of recalling these innate concepts (Laurence and Margolis, 2002). His thesis relies on the assumption that the primitive concepts that compose complex concepts are themselves indivisible and cannot be defined in terms of smaller parts. However, his position can be undermined if we can show how new primitive concepts—not defined in terms of other concepts—can be learned. Are humans born with the concept of *object*? Showing that such a concept is learnable would not only be a fantastic result but would answer a question philosophers have wrestled with for millennia.

Most attempts to define concept learning avoid the issues raised by Fodor by relying on operational definitions of how concepts are learned; Concepts are the *processes* that emerge when a mechanism interacts with its environment. This operational approach describes several empiricist attempts to show how concepts can be bootstrapped from experience. Gary Drescher's *Schema Mechanism*, based on his interpretation of Piaget's work with young children, is one such attempt.

In essence, this work describes how a computational mechanism can learn a theory for a relational dataset by constructing new relations and redefining the original relations in terms of these new concepts in such a way that the resulting theory is more compact than the original. One argument is that such a mechanism is learning new concepts because these new relations are not defined in terms of existing concepts.

The only possibly novel concept learned in the kinship dataset is that of sex. Both *parent* and *spouse*, while not present in the original dataset, are complex concepts formed by the disjunction of simpler concepts. The *sex* predicate is unique in that it is a new concept that appears seemingly from nowhere, helping to define the kinship dataset. To understand whether sex is a novel, primitive concept, we must consider how the *sex* predicate arises at both the algorithmic and computational levels.

At the algorithmic level (ILP), it appears that the *sex* predicate is, in fact, not novel as it is the result of an inverse resolution step that combines latent information described by the regular equivalence of the dataset. Just as *parent* and *spouse* are complex concepts defined in terms of simpler ones, *sex* is a complex concept formed by the disjunction of all objects is the female equivalence classes.[8]

At the computational level the opposite seems to be true. As if by magic, new predicates are invented that result in the most compact theory that entails a set of examples. In the kinship case, we could imagine that the most compact theory uses a representation similar to that of the family tree and a set of rules that define the remaining relationships. At the computational level, these new concepts appear automatically. However, our inability to explain why and when novel concepts appear should not bestow upon those concepts any sort of special status. If the *sex* predicate arises from the latent structure in the dataset, then the *sex* predicate is not truly novel because the latent structure is already present in the examples and need only be squeezed out. Therefore, the concept of sex is already present in the data and the work presented in this paper does not undermine Fodor's nativism. However, even if we were to assume the position of nativism, because there is no efficient way of finding these new concepts, intelligence may in fact be regarded as the ability to efficiently devise compact theories, in which case, the fact that everything is already known in a strict theoretical sense is of little consequence.

Conclusion

A computational approach that optimally compresses a set of examples necessarily requires the invention of predicates. Discovering these predicates is intractable at the computational level. However, at the algorithmic level, with methods such as those employed by ILP, we can discover predicates useful for compressing a theory by using inverse resolution. For the kinship domain it was shown how an ILP system could in principle learn the three fundamental predicates that comprise the family tree representation. While these predicates are novel in the sense that they are not strictly present in

[8] In the kinship theory presented in class, the *parent, spouse* and *sex* predicates are the basis set in which the remaining of the relationships are defined. This basis set could have as easily been *husband, father* and *sex*. However, while rearranging the ground clauses and rules such that the new predicates are the ground clauses (the basis set) does elevate the status of these predicates to primitive concepts, it similarly demotes the status of the original predicates to that of complex concepts. I have primarily concerned myself with how these novel concepts can be learned. Once they are invented, rearrangements can be made to optimize further. I believe that explaining how they arise is the most important aspect. Simply inventing new predicates is not a serious algorithmic level theory as it entails an impossibly large expansion in the search space.

the examples, their invention requires that they be defined in terms of primitive concepts. In the case of the *sex* predicate this requires that we augment the system with explicit representations of the regular equivalence of the examples. Finally, because the predicates arise from inverse resolution, they are not truly novel, primitive concepts and thus do not undermine the philosophical position of extreme nativism.

Bibliography

R. Cilibrasi and P. Vitanyi. Clustering by compression. *Submitted to IEEE Trans. Infomat. Th.*, 2004.

J. Feldman. Minimization of boolean complexity in human concept learning. *Nature*, 407:63–633, 2000.

P. Grunwald and P. Vitanyi. Shannon Information and Kolmogorov Complexity. *Submitted to IEEE Trans. Infomat. Th.*, 2004.

G. E. Hinton. Learning distributed representations of concepts. In *Proc. Ann. Conf. of the Cognitive Science Society*, volume 1, 1986.

C. Kemp, T. L. Griffiths, and J. B. Tenenbaum. Discovering latent classes in relational data. *MIT AI Lab Memo*, 19, 2004.

S. Laurence and E. Margolis. Radical concept nativism. *Cognition*, 86:22–55, 2002.

D. Marr. Artificial intelligence: A personal view. *Artificial Intelligence*, 9:37–48, 1977.

S. Muggleton. Progol software. `http://www.doc.ic.ac.uk/~shm/progol.html`, December 2004.

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

S. Muggleton and L. D. Raedt. Inductive logic programming: Theory and methods. *J. Log. Program.*, 19/20:629–679, 1994.

S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228. February 1997.

D. Page and A. Srinivasan. ILP: a short look back and a longer look forward. *Journal of Machine Learning Research*, 4:415–430, 2003.

J. B. Tenenbaum. Personal communications, November 2004.

D. R. White and K. P. Reitz. Graph and semigroup homomorphisms social networks. 5:193–234, 1983.

A Graph Compression Code

kinship.scm

```
(define kinship-examples
  '(
    (father Christopher Arthur)
    (father Christopher Victoria)
    (father Andrew James)
    (father Andrew Jennifer)
    (father James Colin)
    (father James Charlotte)

    (mother Penelope Arthur)
    (mother Penelope Victoria)
    (mother Christine James)
    (mother Christine Jennifer)
    (mother Victoria Colin)
    (mother Victoria Charlotte)

    (parent Christopher Arthur)
    (parent Christopher Victoria)
    (parent Andrew James)
    (parent Andrew Jennifer)
    (parent James Colin)
    (parent James Charlotte)
    (parent Penelope Arthur)
    (parent Penelope Victoria)
    (parent Christine James)
    (parent Christine Jennifer)
    (parent Victoria Colin)
    (parent Victoria Charlotte)

    (husband Christopher Penelope)
    (husband Andrew Christine)
    (husband Arthur Margaret)
    (husband James Victoria)
    (husband Charles Jennifer)

    (wife Penelope Christopher)
    (wife Christine Andrew)
    (wife Margaret Arthur)
    (wife Victoria James)
    (wife Jennifer Charles)

    (spouse Christopher Penelope)
    (spouse Andrew Christine)
    (spouse Arthur Margaret)
    (spouse James Victoria)
    (spouse Charles Jennifer)
    (spouse Penelope Christopher)
    (spouse Christine Andrew)
    (spouse Margaret Arthur)
    (spouse Victoria James)
    (spouse Jennifer Charles)

    (son Arthur Christopher)
    (son Arthur Penelope)
    (son James Andrew)
    (son James Christine)
    (son Colin Victoria)
    (son Colin James)

    (daughter Victoria Christopher)
    (daughter Victoria Penelope)
    (daughter Jennifer Andrew)
    (daughter Jennifer Christine)
    (daughter Charlotte Victoria)
    (daughter Charlotte James)

    (brother Arthur Victoria)
    (brother James Jennifer)
    (brother Colin Charlotte)

    (sister Victoria Arthur)
    (sister Jennifer James)
    (sister Charlotte Colin)

    (sibling Arthur Victoria)
    (sibling James Jennifer)
    (sibling Colin Charlotte)
    (sibling Victoria Arthur)
    (sibling Jennifer James)
    (sibling Charlotte Colin)

    (uncle Arthur Colin)
    (uncle Charles Colin)
    (uncle Arthur Charlotte)
    (uncle Charles Charlotte)

    (aunt Jennifer Colin)
    (aunt Margaret Colin)
    (aunt Jennifer Charlotte)
    (aunt Margaret Charlotte)

    (nephew Colin Arthur)
    (nephew Colin Jennifer)
    (nephew Colin Margaret)
    (nephew Colin Charles)

    (niece Charlotte Arthur)
    (niece Charlotte Jennifer)
    (niece Charlotte Margaret)
    (niece Charlotte Charles)))

(define (kinship-regular-equivalence)
  '((christopher 1)
    (andrew 1)
    (penelope 2)
    (christine 2)
    (margaret 3)
    (victoria 3)
    (jennifer 3)
    (arthur 4)
    (james 4)
    (charles 4)
    (colin 5)
```

```
    (charlotte 6)))

(define first-literal car)
(define remaining-literals cdr)
(define get-relation car)
(define get-objects cdr)

(define (remove-last lst)
  (reverse (cdr (reverse lst))))

(define (length< l1 l2)
  (< (length l1) (length l2)))

(define (first-n lst n)
  (cond ((or (<= n 0) (not (pair? lst))) '())
        (else
         (cons (car lst) (first-n (cdr lst) (- n 1))))))

(define (union . l)
  (let loop ((l l)
             (b (list)))
    (if (null? l)
        b
        (loop (cdr l)
              (let loop ((a (car l))
                         (b b))
                (if (null? a)
                    b
                    (if (member (car a) b)
                        (loop (cdr a) b)
                        (loop (cdr a) (cons (car a) b)))))))))

(define (setdiff a b)
  (filter (lambda (aval) (not (member aval b))) a))

(define (intersect a . bs)
  (let loop ((a a)
             (bs bs))
  (cond ((null? bs) a)
        ((null? a) a)
        (else
         (loop
          (filter (lambda (aval) (member aval (car bs))) a)
          (cdr bs))))))

(define (extract extractor lst)
  (let loop ((lst lst)
             (items '()))
    (if (null? lst)
        items
        (loop (cdr lst)
              (union (extractor (car lst))
                     items)))))

(define (extract-relations examples)
  (extract (lambda (literal) (list (get-relation literal))) examples))

(define (extract-objects examples)
  (extract get-objects examples))

(define (query-examples examples queryliteral)
  (if (null? examples)
      #f
      (let ((literal (first-literal examples)))
        (or (equal? literal queryliteral)
            (query-examples (remaining-literals examples)
                            queryliteral)))))

(define (build-graph relations relnames objects objnames adjgraphs examples coloring)
  (list relations relnames objects objnames adjgraphs examples coloring))

(define (vector-select v lst)
  (let loop ((i (vector-length v))
             (lst lst)
             (result '()))
    (if (> i 0)
        (loop (+ i 1)
              (cdr lst)
              (if (= 1 (vector-ref v (- i 1)))
                  (cons (car lst) result)
                  result))
        result)))

(define (find lst)
  (let loop ((i 0)
             (result '())
             (lst lst))
    (if (null? lst)
        (reverse result)
        (loop (+ i 1)
              (if (car lst)
                  (cons i result)
                  result)
              (cdr lst)))))

(define (index-of i lst)
  (find (map (lambda (q) (equal? i q)) lst)))

(define (first-index-of i lst)
  (car (index-of i lst)))

(define identity (lambda (x) x))

(define graph-relations car)
(define graph-relnames cadr)
(define graph-objects caddr)
(define graph-objnames cadddr)
(define graph-adjgraphs (lambda (graph) (cadddr (cdr graph))))
(define graph-examples (lambda (graph) (cadddr (cddr graph))))
(define graph-coloring (lambda (graph) (cadddr (cdddr graph))))
(define (graph-select-relation graph relation)
  (vector-ref (graph-adjgraphs graph) relation))
(define (graph-get-all-edges graph relation)
  (let ((relgraph (graph-select-relation graph relation)))
    (apply append
           (map (lambda (obj1)
                  (map (lambda (obj2)
                         (list obj1 obj2))
                       (find (vector->list (vector-ref relgraph obj1)))))
```

```
                (build-list (vector-length relgraph) identity)))))
(define (graph-get-edges graph relation node)
  (let ((relgraph (graph-select-relation graph relation)))
    (find (vector->list (vector-ref relgraph node)))))
(define edge-start-node car)
(define edge-end-node cadr)
(define (path-last-node path) (caar path))
(define (graph-query-edge graph relation obj1 obj2)
  (vector-ref (vector-ref (vector-ref (graph-adjgraphs graph) relation) obj1) obj2))
(define (graph-list-edges graph relation obj1)
  (find (vector->list (vector-ref (vector-ref (graph-adjgraphs graph) relation) obj1))))

(define (parse-examples examples coloring)
  ; take a list of examples, extract the relations, generate graphs and return
  ; lists of objects, the graphs, etc
  (let* ((relations (extract-relations examples))
         (relnames (map symbol->string relations))
         (objects (extract-objects examples))
         (objnames (map symbol->string objects))
         (adjgraphs (build-vector
                     (length relations)
                     (lambda (relation)
                       (build-vector
                        (length objects)
                        (lambda (obj1)
                          (build-vector
                           (length objects)
                           (lambda (obj2)
                             (query-examples examples
                                             (list (list-ref relations relation)
                                                   (list-ref objects obj1)
                                                   (list-ref objects obj2))))))))))
         (coloring (lambda (object)
                     (if (symbol? object)
                         (cadr (assoc object coloring))
                         (cadr (assoc (list-ref relations object) coloring))))))
    (build-graph relations relnames objects objnames adjgraphs examples coloring)))

(define kinship-graph (parse-examples kinship-examples kinship-regular-equivalence))

(require (lib "list.ss" "srfi/1"))

(define (complexity rules)
  (let loop ((complexity 0)
             (rules rules))
    (if (null? rules)
        complexity
        (let ((rule (car rules)))
          (loop (+ complexity
                   (apply + 1 (map (lambda (subrule) (length (car subrule))) (cdr rule))))
                (cdr rules))))))

(define (reachable-set graph relationpath startnode)
  (let loop ((queue (list (list startnode)))
             (relationpath relationpath))
    (if (or (null? relationpath) (null? queue))
        (map car queue) ; grab end nodes
        (let ((relation (car relationpath)))
          (loop (append-map (lambda (partial-path)
                              (let ((lastnode (car partial-path)))
                                (filter pair?
                                        (map (lambda (node)
                                               (if (member node partial-path)
                                                   '()
                                                   (cons node partial-path)))
                                             (graph-list-edges graph relation lastnode)))))
                            queue)
                (cdr relationpath))))))

(define (trim-invalid graph relation revrelationpaths)
  ; remove rules that imply edges that are negative
  ;(printf "trimming... ")
  (map (lambda (revrules)
         (filter (lambda (revrule)
                   (let ((rule (reverse revrule)))
                     (let loop ((startnode (- (length (graph-objects graph)) 1)))
                       ;(printf "checking ~a (~a) ~n" startnode (map (lambda (relation) (list-ref (graph-relations graph) relation)) rule))
                       (if (< startnode 0)
                           #t ; rule ok!
                           (let ((reachable (reachable-set graph rule startnode))
                                 (realreachable (graph-list-edges graph relation startnode)))
                             ;(printf "reachable ~a~nreal reach ~a~n~n" reachable realreachable)
                             ; need to make sure that all of positives are in reachable
                             ; need to make sure that none of negatives are in reachable
                             (and (null? (setdiff reachable
                                                  realreachable))
                                  (loop (- startnode 1))))))))
                 revrules))
       revrelationpaths))

(define (findrelationpaths graph relation disallowedrelations maxsize)
  (let ((allowable (setdiff (build-list (length (graph-relations graph)) identity)
                            (union (list relation) disallowedrelations))))
    (let loop ((fcpaths '())
               (edges (graph-get-all-edges graph relation)))
      (printf "~n")
      (if (null? edges)
          fcpaths
          (let ((edge (car edges)))
            ;(printf "learning about ~a(~a,~a)~n"
            ;        (list-ref (graph-relnames graph) relation)
            ;        (list-ref (graph-objnames graph) (edge-start-node edge))
            ;        (list-ref (graph-objnames graph) (edge-end-node edge)))
    (loop (cons
                   (compress-paths (findpaths graph
                                              allowable
                                              (edge-start-node edge)
                                              (edge-end-node edge)
                                              maxsize))
                   fcpaths)
  (cdr edges)))))))

(define (choose-covering setofrelationpaths)
  ; one idea is to calculate covering-size for each relationpath and then greedily choose
  ; the largest covering until the entire set is covered... i tihnk there is a optimum way
  ; of doing this using dynamic programming (!!)
  (let ((setofrelationpaths (map (lambda (relationpaths)
                                   (map (lambda (rp)
                                          (cons rp (apply +
```

```
                                                          (map (lambda (rps)
                                                                 (if (member rp rps) 1 0))
                                                               setofrelationpaths))))
                                        relationpaths))
                                 setofrelationpaths)))
    (let ((sorted-rps (map (lambda (rps)
                             (quicksort rps (lambda (rp1 rp2)
                                              (or (> (cdr rp1) (cdr rp2))
                                                  (and (= (cdr rp1) (cdr rp2))
                                                       (length< (car rp1) (car rp2)))))))
                           setofrelationpaths)))
      (if (member '() sorted-rps)
          'no-rule
          (union (map car sorted-rps))))))

(define (pick-optimal-covering sorted-rps)
  (if (member '() sorted-rps)
      'no-rule
      (union (map car sorted-rps))))

(define (create-order setofrelationpaths)
  (printf "ordering... ")
  ; one idea is to calculate covering-size for each relationpath and then greedily choose
  ; the largest covering until the entire set is covered... i tihnk there is a optimum way
  ; of doing this using dynamic programming (!!)
  (let ((setofrelationpaths (map (lambda (relationpaths)
                                   (map (lambda (rp)
                                          (cons rp (apply +
                                                          (map (lambda (rps)
                                                                 (if (member rp rps) 1 0))
                                                               setofrelationpaths))))
                                        relationpaths))
                                 setofrelationpaths)))
    (let ((sorted-rps (map (lambda (rps)
                             (quicksort rps (lambda (rp1 rp2)
                                              (or (> (cdr rp1) (cdr rp2))
                                                  (and (= (cdr rp1) (cdr rp2))
                                                       (length< (car rp1) (car rp2)))))))
                           setofrelationpaths)))
      sorted-rps)))

(define (build-rule-from-paths graph dop)
  (if (member '() dop)
      'no-rule
      (map (lambda (conjunction)
             (cons
              (map (lambda (relation)
                     (list-ref (graph-relations graph) relation))
                   (car conjunction))
              (cdr conjunction)))
           dop)))

(define (findpaths graph allowable startnode endnode maxsize)
  (printf "finding paths... ")
  (let loop ((queue (list (list (list startnode -1))))
             (successpaths '())
             (count 0))
    ;(printf "queue = ~a~n" queue)
    ;(printf "~a " (length queue))
    (if (or (null? queue) (null? maxsize) (>= count maxsize))
        successpaths
        (let* ((path (car queue))
               (queue (cdr queue))
               (lastnode (path-last-node path)))
          ;(display path)
          ;(display lastnode)(newline)
          (if (equal? lastnode endnode)
              (loop queue
                    (cons path successpaths)
                    (+ count 1))
              (loop (append queue
                            (append-map
                             (lambda (relationgraph)
                               (filter-map
                                (lambda (object-node)
                                  (if (and (not (member object-node (map car path)))
                                           (graph-query-edge graph relationgraph lastnode object-node))
                                      (begin
                                        ;(printf "new path! ~a~n" (cons (list object-node relationgraph) path))
                                        (cons (list object-node relationgraph) path))
                                      #f))
                                (build-list (length (graph-objects graph)) identity)))
                             allowable))
                    successpaths
                    count))))))

(define (compress-paths paths)
  (printf "compressing... ")
  (let loop ((paths paths)
             (cpaths '()))
    (if (null? paths)
        cpaths
        (let ((path (remove-last (map cadr (car paths))))
              (paths (cdr paths)))
          (if (member path cpaths)
              (loop paths cpaths)
              (loop paths (cons path cpaths)))))))

(define (remove-nth lst n)
  (cond ((null? lst) lst)
        ((<= n 0) (cdr lst))
        (else
         (cons (car lst) (remove-nth (cdr lst) (- n 1))))))

(define (generate-allrules graph allowable donotdefine)
  (if (file-exists? "allrules.output")
      (printf "FILE ALREADY EXISTS!")
      (let* ((disallowedrelations (setdiff (graph-relations graph) allowable))
             (allrules (map (lambda (relation)
                              (let* ((junk (printf "~ngenerating allrules for ~a" relation))
                                     (relationindex (first-index-of relation (graph-relations graph)))
                                     (disallowedrelations (map (lambda (relation) (first-index-of relation (graph-relations graph)))
                                                               disallowedrelations))
                                     (relationpaths (findrelationpaths graph relationindex disallowedrelations 100))
                                     ;(relationpaths (map (lambda (lst) (first-n lst 100)) relationpaths))
                                     (relationpaths (trim-invalid graph relationindex relationpaths))
                                     (relationpaths (create-order relationpaths)))
                                relationpaths))
                            (setdiff (graph-relations graph) donotdefine))))
```

```
        (with-output-to-file "allrules.output" (lambda () (write allrules))))))

(define (optcompress graph donotdefine elimination-order)
  (let* ((allrules (with-input-from-file "allrules.output" (lambda () (read)))))
    (let elimloop ((rules (list))
                   (allrules allrules)
                   (relations (setdiff (graph-relations graph) donotdefine))
                   (elimination-order elimination-order))
      (printf "finding new rule~n")
      ;(pretty-print rules)
      (if (null? allrules)
          rules
          (let* ((coverings (map (lambda (relationpaths)
                                   (pick-optimal-covering relationpaths))
                                 allrules))
                 (possiblerules (map (lambda (rule relation)
                                       (cons relation rule))
                                     coverings relations))
                 (possiblerules (filter (lambda (x) (not (equal? (cdr x) 'no-rule)))
                                        possiblerules))
                 (possiblerules (map (lambda (rule)
                                       (cons (apply + (map (lambda (otherrule)
                                                             (if (ormap (lambda (subrule)
                                                                          (member (first-index-of (car rule) (graph-relations graph))
                                                                                  (car subrule)))
                                                                        (cdr otherrule))
                                                                 1 0))
                                                           possiblerules))
                                             rule))
                                     possiblerules))
                 (sortedrules (quicksort possiblerules
                                         (lambda (r1 r2)
                                           (cond ((< (car r1) (car r2)) #t)
                                                 ((> (car r1) (car r2)) #f)
                                                 ((< (length (cddr r1)) (length (cddr r2))) #t)
                                                 ((> (length (cddr r1)) (length (cddr r2))) #f)
                                                 (else
                                                  (< (apply max (map cdr (cddr r1)))
                                                     (apply max (map cdr (cddr r2)))))))))
                 (junk (pretty-print sortedrules))
                 (sortedrules (map cdr sortedrules)))

            (if (null? sortedrules)
                rules
                (let* ((newrule (if (null? elimination-order)
                                    (first sortedrules)
                                    (car (filter (lambda (rule) (equal? (car rule) (car elimination-order))) sortedrules))))
                       (rel (car newrule)))
                  (elimloop
                   (cons (cons (car newrule) (build-rule-from-paths graph (cdr newrule))) rules)
                   (map (lambda (posexamples)
                          (map (lambda (posexample)
                                 (filter (lambda (rule)
                                           (not (member (first-index-of rel (graph-relations graph)) (car rule))))
                                         posexample))
                               posexamples))
                        (remove-nth allrules (first-index-of rel relations)))
                   (setdiff relations (list rel))
                   (if (null? elimination-order)
                       elimination-order
                       (cdr elimination-order)))))))))

; DO NOT DEFINE
(define do-not-define '(parent spouse sibling))

; ALLOWABLE RELATIONS
;(define allowablerelations (graph-relations kinship-graph))
(define allowablerelations (setdiff (graph-relations kinship-graph) '(parent spouse)))
;(define allowablerelations '(husband mother wife son daughter))
;(define allowablerelations '(husband parent wife spouse son daughter))

; GENERATE ALL RULES and COMPRESS
(generate-allrules kinship-graph allowablerelations do-not-define)
(define kinship-rules (optcompress kinship-graph do-not-define '(uncle))) (pretty-print kinship-rules)

;(define derived-set (setdiff (graph-relations kinship-graph) (map caar kinships-rules)))

(complexity kinship-rules)
(setdiff (graph-relations kinship-graph) (append (map car kinship-rules) do-not-define))
```