CS532, Winter 2010
Learning in First-Order Logic

Dr. Alan Fern, afern@cs.orst.edu

January 14, 2010

A key aspect of intelligence is the ability to learn knowledge over time. This problem is studied in the artificial intelligence sub-field of machine learning, for which we have an entire course (CS534). However, the vast majority of machine learning algorithm have been developed to learn knowledge that is inherently propositional, where the domain of interest is encoded in terms of a fixed set of variables. As we have argued in previous lectures, it appears critical for general AI systems to be able to explicitly represent and reason about objects and relations among objects, which was our primary motivation for studying first-order logic. Thus, it also seems critical to develop algorithms that can directly learn knowledge in a first-order representation. This section of the course will cover some of the fundamental approaches to learning first-order representations, many of which are quite practical and have been quite successful in important applications.

1 Introduction to First-Order Learning

Suppose that we want a computer to learn to determine whether or not a molecule is "sticky" (sticky is a property that I made up) based on its atomic structure. One approach would be to create a training set of molecules that are represented using first-order ground facts.

For example, two training instances might look like The first column of each training instance

$\text{molecule}(M_1) \wedge \text{atom}(M_1, A_{11}, \text{Carbon}) \wedge \cdots \wedge \text{Bond}(M_1, A_{11}, A_{16})$	$\text{sticky}(M_1)$
$mboxmolecule(M_2) \wedge \text{atom}(M_2, A_{21}, \text{Oxygen}) \wedge \cdots \wedge \text{Bond}(M_2, A_{21}, A_{23})$	$\neg\text{sticky}(M_2)$

provides a description of the two molecules denoted as M_1 and M_2. The second column of each instance indicates whether the molecule is sticky or not. Given a set of such examples, we would like to learn a formula $\phi(x)$ such that

$$\phi(x) \iff \text{sticky}(x) .$$

is consistent with the training instances. We call $\phi(x)$ a **definition** of the predicate sticky, and the formula

$$H = \{\phi(x) \iff \text{sticky}(x)\} .$$

a **hypothesis** for the predicate sticky.

In our example we might learn the following definition of sticky

$$\phi(x) = \exists\, a_1, a_2\ \ [\text{atom}(x, a_1, \text{Carbon}) \wedge \text{atom}(x, a_2, \text{Hydrogen}) \wedge \text{Bond}(x, a_1, a_2)]$$

which says that a molecule x is sticky iff it has a Carbon-Hydrogen bond. Note that $\phi(x)$ applies to any molecule regardless of its size, which is one of the advantages of learning a first-order representation.

While this example considered learning the definition of a single-arity predicate sticky, we can also consider learning definitions of higher arity predicates such as WillPurchase(x, y), which indicates that x will purchase item y. In such a case the learned formula $\phi(x, y)$ would depend on two variables.

2 A Logical Formulation of Learning

We will formulate the problem of learning logical formulas as finding a hypothesis formula H that satisfies

$$D \wedge B \wedge H \models L$$

where

D is a KB that describes the training examples.
For example, descriptions of all training molecules.

B is a KB that provides a set of background knowledge about the problem.
For example, we might provide in B a rule of the form, $\alpha(x) \Rightarrow \mathit{active}(x)$, which defines the "active" property of molecules. If a concept in B is related to the predicate to be learned, then it might be used in the learned definition.

L is a KB that provides the labels for all training examples.
For example, $\{\ \mathit{sticky}(M_1), \neg \mathit{sticky}(M_2), \ldots\ \}$.

That is – given the example descriptions and the background knowledge – the hypothesis should allow us to predict (or entail) the truth value of the target predicate for any instantiations of its variables (e.g. predict that $\mathit{sticky}(M_1)$ is true). In this sense we see that inferring, or learning, H is the inverse problem of deduction. In particular, rather than trying to produce facts that deductively follow from our knowledge, we wish to learn knowledge such that certain facts that we are given follow deductively from our current and learned knowledge. This process of inferring an H is sometimes called **inductive inference** as opposed to deductive inference.

As we will see later, learning algorithms will sometimes treat $D \wedge B$ as a single KB usually refereed to as background knowledge. So in this case we would just have $B \wedge H \models L$.

3 Terminology for First-Order Logic Learning

Throughout, we will assume that $Q(x)$ is the target predicate for which we are trying to learn a definition, where x may also correspond to a vector of variables $< x_1, \cdots, x_n >$ rather than just a single variable.

- If a hypothesis formula H says that $Q(x)$ is true for an example when $Q(x)$ is actually false, we say that the example is a **false positive** for H.
- Likewise, if a hypothesis formula H says that $Q(x)$ is false for an example, when $Q(x)$ is actually true, then the example is a **false negative**.

- Let $H1 = (\phi_1(x) \iff Q(x))$ and $H_2 = (\phi_2(x) \iff Q(x))$. We say that H_2 is a **generalization** of H_1, or equivalently, H_1 is a **specialization** of H_2, if it is true that,

$$\forall\, x\ [\phi_1(x) \Rightarrow \phi_2(x)]$$

  Sometimes this is written as $H1 < H2$. Intuitively if $H1 < H2$ then anytime that $H1$ predicts that $Q(o)$ is true for some object tuple o then so will $H2$.

- The **most specific hypothesis** according to the relation $<$ is $H = \textbf{false} \iff Q(x)$, which predicts that $Q(x)$ is false for all examples.

- The **most general hypothesis** according to the relation $<$ is $H = \textbf{true} \iff Q(x)$, which predicts that $Q(x)$ is true for all examples.

We will use the notions of generalization and specialization to structure the search space of hypothesis formulas. We may use two types of search operators:

Specialization operators take a hypothesis H as their input and then output a set of formulas $\{H_1, \cdots, H_n\}$ such that $H_i < H$.

Specialization operators are useful for dealing with false positives in the training data. That is, if our currently best guess hypothesis H commits false positives, then we can possibly correct some of those errors by changing our best guess to a specialization of H returned by the operator.

As an example consider the hypothesis,

$$P(x) \iff Q(x)\ .$$

A specialization operator might return

$$\begin{array}{ccc} P(x) \wedge R(x) & \iff & Q(x) \\ P(x) \wedge V(x) & \iff & Q(x) \\ \vdots & \vdots & \vdots \end{array}$$

each of which clearly are specialization since they make the definition of Q more strict.

Generalization operators take a hypothesis H as input and output a set of formulas $\{H_1, \cdots, H_n\}$ such that $H_i > H$.

Generalization operators are useful for dealing with false negatives.

As an example, consider the hypothesis

$$P(x) \wedge R(x) \iff Q(x)\ .$$

A generalization operator might return

$$\begin{array}{ccc} P(x) & \iff & Q(x) \\ R(x) & \iff & Q(x) \end{array}$$

You should be able to verify that these formulas are in fact generalizations.

There is a rich theory concerning specialization and generalization operators for logical languages. In this course we will simply show some concrete examples of such operators when we introduce our learning algorithms.

4 Learning as Search

Given a generalization and specialization operator we can cast learning a hypothesis formula as a search problem, where the goal is find a hypothesis that is consistent with the training examples, as formalized above. We will consider three different generic search approaches here and later we will give specific instances of the first two for learning first-order logic formulas.

4.1 General-to-Specific Search

Here the start node of the search space is the most general hypothesis

$$H_0 = \textbf{true} \iff Q(x)$$

which predicts that $Q(x)$ is true for all examples. This means that H will commit no false negatives, but it will commit false positives for all negative training examples (i.e. examples labeled $\neg Q(x)$).

We then use the specialization operator starting at H_0 in order to search through more specific hypotheses, in order to correct for the false positives. Note that in specializing a hypothesis H to H', H' might commit false negatives even if H does not. Thus, during our search we want to only consider specialization that do not commit false negatives; once a false negative is created, it is not possible to use specialization to correct the error. Section 5 introduces the FOIL learning system, based on this type of general-to-specific search.

4.2 Specific-to-General Search

This is just like general-to-specific search, only H_0 is taken to be the most specific hypothesis and we use a generalization operator in order to correct for false negatives.

4.3 Candidate Elimination

Candidate elimination is a learning algorithm that combines both general-to-specific and specific-to-general search. The algorithm is based on the idea of version spaces. Given a training set, the *version space* with respect to that training set is the set of all hypotheses that are consistent with the training data.

The candidate elimination algorithm is an incremental algorithm that maintains the version space as each example is processed. Starting with a version space V for the null set of examples (i.e. the set of all hypotheses), candidate elimination considers each training example e and removes those hypotheses from V which are inconsistent with e. For each negative example e^- (i.e. e^- is labeled $\neg Q(x)$), the algorithm removes all hypotheses from V that predict true for $Q(x)$. For each positive example e^+, the algorithm similarly removes hypotheses that predict $\neg Q(x)$. This iterative step can be stated more formally as

$$V_{i+1} = V_i - \{ \ H \mid H \in V_i \text{ s.t. } H \text{ is inconsistent with } e \ \}$$

After processing each example, all of the hypotheses in V will be consistent with the training data. If V contains a single hypothesis, then we can simply return it as the answer. If V still contains multiple hypotheses then we can return one of them at random, or return one according to some quality criterion (e.g. length), or return a combined hypothesis that somehow combines the

predictions of hypotheses in V. If the version space V becomes empty, then we return "failure," indicating that no hypotheses were found that are consistent with the data.

Generally this explicit procedure is not practical, since the space of possible hypotheses is very large or unbounded (e.g. the space of definite theories, or perhaps only size-bounded definite theories). Thus, the candidate elimination algorithm is generally implemented by maintaining the set V implicitly rather than explicitly, and using efficient algorithms for updating the implicit representation after observing a new example.

4.3.1 Implicit Version Space

One way to implicitly represent the version space V is by maintaining two sets of hypotheses S and G, the so called **most specific** and **most general** boundaries of the version space. S contains the set of most specific hypotheses that are consistent with the training examples, and G contains the most general hypotheses that are consistent with the training examples. That is,

$$\begin{aligned} S &= \left\{ H \mid \begin{array}{r} H \text{ is consistent with currently seen examples,} \\ \text{and there is no other such } H' \text{ s.t. } H' < H \end{array} \right\} \\ G &= \left\{ H \mid \begin{array}{r} H \text{ is consistent with currently seen examples,} \\ \text{and there is no other such } H' \text{ s.t. } H < H' \end{array} \right\} \end{aligned}$$

It is easily shown (see your book) that these sets completely define V; that is, the following property holds:

$$H \in V \iff \exists\, s, g\ (s \in S \wedge g \in G \wedge s < H < g)\ .$$

Given a current S and G representing the current version space and a new training example e, we can use specialization and generalization operators in order to update the sets. If all hypotheses in S and G are consistent with e then no changes are necessary. However, if a hypothesis s in S is inconsistent with e, then there are two possibilities:

1. e is a false positive for s. That is, s should predict negative but it predicts positive. This means that s is too general. But we know that no hypothesis more specific than s is consistent, so we simply remove s from the set S.

2. e is a false negative for s. In this case, s is too specific and needs to be generalized. We apply a generalization operator to s in order to get a set $\{s_1, \cdots, s_n\}$ to replace s. We remove any of the s_i that are more general than some other hypothesis already in S (i.e. we only want to keep most specific hypotheses).

Similar operations can be applied to the set G, see your book (pg. 685) for the details.

While the implicit version of the candidate elimination algorithm is much more tractable, it is not used often in practice. This is primarily due to the fact that for expressive hypothesis languages, the S and G sets can grow very large. However, in many cases it is possible to carefully design a hypothesis language, so that it contains useful hypotheses for a particular application and where the S and G sets can be maintained efficiently. Mostly though, the candidate elimination algorithm provides a nice conceptual framework for thinking about learning algorithms and for inspiring practical approximations.

5 Learning First-Order Definitions with FOIL

Foil (First-Order Inductive Learner) is a well known algorithm for learning first-order rules, in particular, first-order definite clauses, that define a target predicate.

5.1 Learning Dating Compatibility with FOIL

Suppose that we are working for a dating service (say HookMeUp.com). We have a database of customer information that includes, for example, facts about our customers obtained from a questionnaire. We also have many training examples of the relation Compatible(x, y), which is true of individuals x and y iff after a date they end up having a "successful relationship." Our boss asks us to come up with a set of rules based on this data that will allow us to predict whether x and y are compatible given their personal information. This is exactly the type of problem that Foil is intended to solve.

Lets make the example more concrete. Foil takes two inputs:

1. a set of background knowledge, which is simply a set of ground atoms defining the basic facts of our domain, and
2. a set of training examples which give a set of positive examples of the target predicate and negative examples of the target predicate.

In our example, the background knowledge will include all information about individuals in our company database:

$$\begin{array}{ccc} Age(JonSmith, 26), \; Height(JonSmith, 67in), \ldots, & MusicalTaste(JonSmith, Classical), \; MusicalTaste(JonSmith, Bluegrass), \ldots, & HairColorPreference(JonSmith, None), \ldots \\ \vdots & \vdots & \vdots \\ Age(MarthaGoo, 45), \ldots, & \ldots, & HairColorPreference(MarthaGoo, Red), \ldots \\ \vdots & \vdots & \vdots \end{array}$$

Foil assumes that any fact not included in the background knowledge is false; this assumption is commonly referred to as the **closed-world assumption**.

The training examples are divided into sets positive and negative cases:

Positive	**Negative**
$Compatible(JonSmith, MarthaGoo)$	$Compatible(JonSmith, EthelGoodman)$
$Compatible(NewellMarks, EsterMay)$	$\vdots$
$\vdots$	$\vdots$

These positive and negative examples, could have perhaps been collected based on actual feedback from the customers. Note that in the our previous formulation of logical learning we simply encoded negative examples using negation, e.g. $\neg$Compatible(JonSmith, EthelGoodman). However, since Foil does not explicitly do this, I will follow Foil's formalism.

Given the background knowledge and examples Foil will attempt to produce a set of rules that define the predicate Compatible. These rules will take the form of definite clauses where the

positive literal always involves the Compatible predicate—i.e. Compatible will always be at the head of the rules. For example FOIL might output a rule set such as,

$$
\begin{array}{rcl}
(Age(x,a_1) \wedge Age(y,a_2) \wedge DiffLessThan5(a_1,a_2) \wedge HairColor(a_1,c) \wedge HairColorPreference(a_2,c)) & \Rightarrow & Compatible(x,y) \\
\vdots & \vdots & \vdots \\
(LivesAt(x,l_1) \wedge LivesAt(x,l_2) \wedge ShortCommute(l_1,l_2) \wedge FoodPreference(x,f) \wedge FoodPreference(y,f) & \Rightarrow & Compatible(x,y) \\
\vdots & \vdots & \vdots
\end{array}
$$

5.2 An Overview of FOIL

If we denote the background knowledge by B, the positive examples by P, the negative examples by N, FOIL's learning objective is to find a set of rules H such that,

$$B \wedge H \models P$$

which states that we can infer positive examples from background knowledge, and

$$\forall\, n\ [n \in N \wedge \neg(B \wedge H \models n)]$$

which states that we do not infer any negative examples either. Note, FOIL groups information about the examples – and more general background information (e.g. distances between locations) – into the single KB named B.

When FOIL is unable to find a rule set that perfectly satisfies these conditions it will return a set that is as accurate as possible.

5.3 The FOIL Learning Algorithm

FOIL searches for a rule set using a commonly applied rule covering approach. Intuitively FOIL searches for one rule at a time, attempting to find rules that do not cover any negative examples but cover many positive examples (eventually all of them). After finding the first such rule, FOIL removes the positive examples covered by the rule from the training set, and then searches for an additional rule that covers many of the remaining positive examples, but still none of the negative examples. This process repeats until all positive examples have been removed from the training set. The following pseudo-code outlines this process.

```
R = {} // initialize ruleset to empty set

// continue until all positives are covered
WHILE not(empty(P))
    // learn a rule that covers many positives but no negatives
    r = LearnRule(P,N)

    // P' contains positive examples covered by r
    P' = {p | p \in P and B & r |= p}
    P = P - P'
    R = R + {r}
```

This algorithm is independent of the particular rule representation, and has been instantiated for many languages (logical and not). FOIL provides a first-order instantiation, which mainly entails defining the function LEARNRULE(P, N) which searches for a first-order rule that covers many positive examples but no (or very few) negative examples.

5.4 Learning FOIL Rules

The implementation of LEARNRULE is outlined via the pseudo-code in Figure 1.

To learn an individual rule, FOIL uses a general-to-specific search that starts with the most general rule. In our example, the most general rule is simply $Candidate(x, y)$, which predicts that all pairs are compatible. This rule hence covers all positive and negative examples, and needs to be specialized so that it no longer covers the negatives. We search for a good specialization of the most general rule using a greedy search through the space of candidate specializations.

```
LearnRule(P,N)
    r = Q(x1,...,xn)  // initialize r to most general rule
                      // that predicts Q is true for all tuples
    WHILE r covers some negative example
        // generate a set of specializations of r
        R' = SpecializationOperator(r)

        // score each specialization and select the best one
        r = selectBestRule(R',P,N)
```

Figure 1: Pseudo-code for FOIL's LEARNRULE(P, N) for the target predicate Q with arity n.

So we see that to learn a rule we must specify a specialization operator and a function/heuristic that scores the candidate specializations and selects the best one. It should be apparent that this rule learning algorithm amounts to a greedy heuristic search, where the search steps correspond to possible specializations and the heuristic function measures the fitness of a rule (relative to P and N). There have been numerous extensions to FOIL that consider alternative heuristics and more sophisticated search strategies (e.g. beam search).

5.5 FOIL Rule Specialization

FOIL uses a simple specialization operator which simply adds a single atom to the body of the current rule. For example, suppose that the only predicate besides Q was R. If we start with the most general rule $Q(x1, x2)$ the specialization operator might return the following set:[1]

$$
\begin{aligned}
R(x_1, x_2) &\Rightarrow Q(x_1, x_2) \\
R(x_2, x_1) &\Rightarrow Q(x_1, x_2) \\
R(x_1, x_3) &\Rightarrow Q(x_1, x_2) \\
R(x_3, x_1) &\Rightarrow Q(x_1, x_2) \\
R(x_3, x_2) &\Rightarrow Q(x_1, x_2)
\end{aligned}
$$

[1] Notice that x_3 is a new variable.

$$R(x_2, x_3) \quad \Rightarrow \quad Q(x_1, x_2)$$

Note that FOIL's specialization operator has the constraint that it will only consider adding atoms to the body that contain at least one variable already present in the rule being specialized.[2]

For example, FOIL would not consider the rule

$$R(x_3, x_4) \Rightarrow Q(x_1, x_2)$$

even though it is technically a specialization. This makes good sense, since the atom $R(x_3, x_4)$ places no constraints on the target predicate variables.

5.6 FOIL Rule Scoring

Given the above set of specializations, FOIL uses an information-theoretic heuristic to score the rules and select the best one. Suppose that FOIL selected the rule

$$R(x_3, x_2) \Rightarrow Q(x_1, x_2)$$

and that this rule still covers some negative examples.

The algorithm will then proceed to create specializations of this rule, possibly giving the following set of rules:[3]

$$\begin{array}{rcl}
R(x_1, x_2) \wedge R(x_3, x_2) & \Rightarrow & Q(x_1, x_2) \\
R(x_2, x_1) \wedge R(x_3, x_2) & \Rightarrow & Q(x_1, x_2) \\
R(x_3, x_1) \wedge R(x_3, x_2) & \Rightarrow & Q(x_1, x_2) \\
R(x_3, x_4) \wedge R(x_3, x_2) & \Rightarrow & Q(x_1, x_2) \\
\vdots & \vdots & \vdots \\
R(x_1, x_3) \wedge R(x_3, x_2) & \Rightarrow & Q(x_1, x_2) \\
R(x_1, x_2) \wedge R(x_3, x_2) & \Rightarrow & Q(x_1, x_2) \\
R(x_1, x_2) \wedge R(x_3, x_2) & \Rightarrow & Q(x_1, x_2)
\end{array}$$

Suppose now that FOIL selects the rule

$$R(x_1, x_3) \wedge R(x_3, x_2) \Rightarrow Q(x_1, x_2)$$

and that this rule no longer covers negative examples. At this point LEARNRULES(P, N) would return this rule and it would be added to the existing ruleset.

The information-theoretic heuristic used by FOIL is rather ad-hoc and we will not discuss it in detail. A number of extensions to FOIL have investigated alternative heuristics. All of these heuristics amount to different ways of rewarding high accuracy and high coverage of positive examples.

[2]There are more constraints, but the details are conceptually unimportant.

[3]Notice that x_4 is a new variable.

5.7 FOIL Complexity

The main computational complexity of the FOIL algorithm is the generation and scoring of candidate specializations. Even with this simple example there were many possible candidates. When we have a large number of high arity predicates the number of candidate specializations grows very quickly. Regardless, FOIL is quite efficient compared to most techniques for learning first-order rulesets and often works well in practice.

5.8 FOIL Code

The code is freely available from Ross Quinlan (also the developer of C4.5 a popular decision tree algorithm) and quite easy to use.

http://www.rulequest.com/Personal/foil6.sh

6 Learning Logic via Inverse Resolution

As we noted earlier, inductive inference (i.e. learning) can be viewed as the inverse of deductive inference. It is natural then to consider developing a learning approach based on inverting deductive inference rules. Researchers in the field of **inductive logic programming** – a sub-field of machine learning that focuses on learning logic formulas – have studied this approach for the first-order resolution rule.

6.1 An Overview of Inverse Resolution

Suppose that we are given a clause C. We can then ask for two clauses C_1 and C_2 such that when resolved together yield C. If this is the case we say that C_1 and C_2 are the result of applying **inverse resolution** to C. Similarly, if we are given two formulas C and C_1, we can ask for a formula C_2, such that when resolved with C_1 yields C. Here again we say that C_2 is the result of applying inverse resolution to the **resolvent** C and **resolver** C_1.

As an example, consider the following clauses

$$C = Q(A) \quad \text{and} \quad C_1 = R(A, A)$$

One possible inverse resolution of these formulas is

$$C_2 = R(X, X) \Rightarrow Q(X)$$

since resolving C_1 with C_2 yields C.

In general, there can be many inverse resolutions (even infinitely many); for example, C_2 could have been chosen to be either of

$$\begin{array}{rcl} R(X,Y) & \Rightarrow & Q(X) \\ R(X,Y) & \Rightarrow & Q(Y) \end{array}$$

6.2 IR Learning

To see how inverse resolution might help us learn a hypothesis, recall our general learning objective which is to learn a hypothesis H such that

$$D \wedge B \wedge H \models L \ .$$

For any such hypothesis H and any label $Q(A)$ in L, we know that resolution can be used to derive a contradiction, given the formula $D \wedge B \wedge H \wedge \neg Q(A)$.

For example, if

$$D = \{R(A,A)\}, B = \{\}, H = \{R(x,x) \Rightarrow Q(x)\}$$

then we can resolve $\neg Q(A)$ with $R(x,x) \Rightarrow Q(x)$ to get the resolvent $\neg R(A,A)$, which resolves with $R(A,A)$ to give a derivation of false. Note that if we consider the resolution proof of false there is a corresponding sequence of inverse resolution steps starting at false which result in the hypothesis H.

This suggests a learning approach where we begin with the formulas

$$KB = D \wedge B \wedge \neg Q(A)$$

Starting with a contradiction, we apply inverse resolution operators – potentially using formulas in KB – until we generate a formula (or formula set) that satisfies some measure of fitness. The resulting formula(s) can then be taken to be part of a new hypothesis.

In the above example, starting from $false$ we apply inverse resolution to obtain the formula $\phi_1 = Q(A)$, which when resolved with $\neg Q(A)$ yields false. We can then apply inverse resolution to ϕ_1, perhaps resulting in $\phi_2 = R(x,x) \Rightarrow Q(x)$, which when resolved with $R(A,A)$ yields $\phi_1 = Q(A)$. At this point, we might be satisfied with ϕ_2 and take it to be our hypothesis (or at least part of the hypothesis). Given a correct inverse resolution operator we are guaranteed that ϕ_2 will lead to a resolution proof of $Q(A)$.

6.3 IR Complexity

Due to the potentially enormous number of possible inverse resolutions of a formula (or pair of formulas), the search space is highly intractable and we must rely on heuristics for guidance. In practice, algorithms based on inverse resolution have not been shown to be competitive with other approaches (e.g. FOIL-like approaches). Nevertheless, inverse resolution is conceptually interesting and theoretically represents a complete learning procedure for full clausal first-order logic.

7 First-Order Learning via Propositionalization

Perhaps the simplest and often effective approach to learning first-order formulas is to reduce the first-order learning problem to a propositional learning problem. The idea is to first compile the first-order learning problem into a propositional one, and then apply one of many possible propositional learning algorithms, e.g. a decision-tree learner. We then take the hypothesis returned by the propositional algorithm and **lift** it back to first-order form.

When faced with a first-order learning problem, this is often a good approach to begin with. The main questions are how to compile to propositional form (often called **propositionalization**) and how to lift the resulting hypothesis. While there are numerous solutions to both steps, these notes give just one example.

7.1 Propositionalization of F-O Logic

Consider the learning setting of FOIL. We are given a set of background facts B, a set of positive examples P of some 2-arity target predicate $Q(x, y)$, and a set of negative examples of Q. For simplicity assume that the only predicates contained in B are $R(x, y)$ and $P(x)$.[4]

The main idea of propositionalization is to create a set of features that capture relational properties of and between the arguments of the target predicate. Here, we will assume that each feature is represented by a first-order conjunction with free variables x and y. Some examples might include

$$\begin{array}{rcl} F_1(x,y) & = & P(x) \\ F_2(x,y) & = & P(y) \\ F_3(x,y) & = & R(x,y) \\ F_4(x,y) & = & R(y,x) \\ F_5(x,y) & = & R(x,z) \wedge R(z,y) \\ \vdots & \vdots & \vdots \end{array}$$

Given a particular positive training example $Q(CAT, DOG)$, where CAT and DOG are constants in B, we now have that each feature is either true or false under the substitution $\theta = \{x/CAT, y/DOG\}$ relative to the background knowledge B. For example, $F_1(CAT, DOG)$ is true iff $P(CAT)$ is in B.

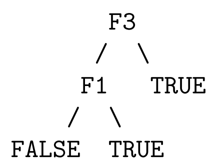
Thus given a set of n first-order features, such as the above, we can generate a propositional training example with n binary features $F_1, \cdots, F_n$. For our positive training example $Q(CAT, DOG)$ we would get

$$+ < F_1 = F_1(CAT, DOG), \cdots, F_n = F_n(CAT, DOG) > \ .$$

Likewise for a negative example $Q(DOG, CAT)$ we would generate

$$- < F_1 = F_1(DOG, CAT), \cdots, F_n = F_n(DOG, CAT) > \ .$$

This propositional training set can then be given to any learner that handles boolean input attributes. For example, we might learn a decision tree,



which indicates that:

1. if the first feature is true then the classification should be true, otherwise
2. if the third feature is true the classification should be true, otherwise
3. the classification should be false.

[4] This discussion can be trivially generalized to higher arity target predicates.

7.2 Lifting Propositional Hypotheses

We can turn this tree into the following set of rules:

$$\begin{aligned} F_3 &\Rightarrow true \\ \neg F_3 \wedge F_1 &\Rightarrow true \\ \neg F_3 \wedge \neg F_1 &\Rightarrow false \end{aligned}$$

Recall that our features all depend on the arguments to our target predicate x and y and that the tree output is intended to predict the truth value of $Q(x, y)$. This means that we can lift these rules to first-order form as follows,

$$\begin{aligned} F_3(x, y) &\Rightarrow Q(x, y) \\ \neg F_3(x, y) \wedge F_1(x, y) &\Rightarrow Q(x, y) \\ \neg F_3(x, y) \wedge \neg F_1(x, y) &\Rightarrow \neg Q(x, y) \end{aligned}$$

Substituting the definition of the features, we get

$$\begin{aligned} R(x, y) &\Rightarrow Q(x, y) \\ \neg R(x, y) \wedge P(x) &\Rightarrow Q(x, y) \\ \neg R(x, y) \wedge \neg P(x) &\Rightarrow \neg Q(x, y) \end{aligned}$$

as our lifted first-order hypothesis. It should be clear that this hypothesis will make an error on a first-order example iff the propositional tree makes an error on the corresponding propositional example.

7.3 Propositionalization Complexity

The main open issue with this propositionalization approach is how to select the set of first-order features. Perhaps most commonly, this is done by requiring that the features be conjunctions, then limiting the number of literals and the number of variables that the conjunctions may contain. Given these bounds, there are a finite number of features, and we can simply use all of them.

Of course, for even modest bounds, the number of features can be quite large. In these cases it is important to use learning algorithms that are robust to large numbers of features, most of which might be only weakly relevant. More sophisticated approaches to propositionalization use more intelligent mechanisms that enumerate only features that are considered to be useful according to some heuristic.