Specific to General Learning via Least-General Generalizations

- We will describe the Golem learning algorithm, which is available for download and is much more practical than inverse resolution, but is based on specific-to-general search rather than general-to-specific.

- Golem attempts to solve the same learning problem as FOIL.

Given: B ~ background knowledge of ground atoms
P ~ positive examples
N ~ negativ examples

Find: H s.t. H is a definite clausal theory

$H \wedge B \models P$

$\forall n \in N,\ H \wedge B \not\models n$

Golem however does allow for the learned clauses to include function symbols.

— Furthermore Golem uses a similar outer loop as FOIL, where one rule is learned at a time and examples are removed at each step.

$R = \emptyset$
While $P \neq \emptyset$ do
    $r := \text{LearnRule}(P, N)$
    $P := P - \{e \in P : B \wedge r \models e\}$

— All of the work is done in LearnRule, ~~and~~ which has the goal of returning a definite clause that covers as many positives as posible and no negatives.

— Golem use a specific-to-general approach to implementing LearnRule (also called a bottom-up approah).

- Golem's LearnRule is based on the concept of least-general generalization (LGG) which we will now cover.

- Intuitively given two clauses C_1 and C_2 (you can think of these as rules) we can talk about the notion of the LGG of C_1 and C_2 which is a clause C that is more general than C_1 and C_2, but the least general such clause.

- Lets denote our generality relation as $\leq$.

- More formally $C = LGG(C_1, C_2)$ iff

  $C \geq C_1$ and $C \geq C_2$ and

  there is no C' s.t. $\underbrace{C' < C}_{\text{strictly less general}}$ and

  $C' \geq C_1$ and $C' \geq C_2$

- According to this definition an LGG of C_1 and C_2 covers all of the positive examples that C_1 and C_2, but as few additional ~~as p~~ ~~positives~~ examples as possible.

- One issue that arises in theory and practice is what definition of generality ($\leq$) we should use for the LGG process.

- Intuitively it would make sense to define generality in terms of entailment:

$$\boxed{C_1 \leq C_2 \quad \text{iff} \quad C_2 \models C_1}$$

Under this definition C_2 will cover all the positives that C_1 covers. To see this suppose

$C_1 \wedge B \models e$ and $C_1 \leq C_2$ then we get

$\cancel{C_1 \wedge B \models e \quad \text{iff} \quad C_2}$

$$C_2 \wedge B \models C_1 \wedge B \models e$$

as desired.

- However, using entailment as the notion of generality has 2 main problem:

  1) The LGG is not unique under entailment.
  2) Computing entailment ~~bewtwe~~ between clauses (even Horn ~~clases~~ clauses) is undecidable in general.

- For these reasons researchers have instead focused primarily on another notion of generality, which can be checked and has a unique LGG.

~~Definition:~~

- For the following we will think of clauses as being represented as sets of literals

  e.g. $C = a \wedge b \rightarrow c$

  is represented as

  $$C = \{\neg a, \neg b, c\}$$

Def: let C_1 and C_2 be two clauses. C_2 θ-subsumes C_1 iff there is a substitution θ such that $C_2\theta \subseteq C_1$

For example:

~~C_2 = daughter(x,Y)~~

C_2 : parent(Y,X) → daughter(x,Y)
C_1 : parent(Y,X) ∧ female(x) → daughter(x,Y)
C_3 : ~~parent(sophie,al~~
parent(alan, sophie) ∧ female(sophie) → daughter(soph, alan)

here we have that

C_2 θ-subsumes both C_1 and C_3
and
C_1 θ-subsumes C_3 (not vice versa)

~~θ-subsumption implies implication but~~

θ-subsumption is a sufficient but not necessary condition for entailment.

$\boxed{\text{Proposition: If } C_2 \ \theta\text{-subsumes } C_1 \text{ then } C_2 \models C_1}$

proof: Assume there is a θ s.t. $C_2\theta \subseteq C_1$.

To show $C_2 \models C_1$ assume that C_2 is true and then try to prove C_1.

Assume C_2 is true.
~~Since we know that $C_2\theta \subseteq C_1$~~
~~We know~~
We know that for any θ that $C_2\theta$ is also true since variables in C_2 are universally quantified. In particular for the θ that demonstrates θ-subsumption $C_2\theta$ is true. Now C_1 is just $C_2\theta \cup \{C_1 - C_2\theta\}$ or equivalently we can get C_1 by adding disjuncts to $C_2\theta$, but C_1 must also be true becaus $A \vee B$ is true if A is true for any B. This shows that C_1 is true.

- The proposition show θ-subsumption is sufficient for entailment.

- The following shows it is not necessary.

$C_2 : P(x) \rightarrow P(f(x))$
$C_1 : P(x) \rightarrow P(f(f(x)))$

we have that

$C_2 \models C_1$ (why?)

but C_2 does not θ-subsume C_1.

~~But we have gained the~~

But lets go ahead and define our notion of generality in terms of θ-subsumption.

$$\boxed{C_1 \leq C_2 \text{ iff } C_2\ \theta\text{-subsumes } C_1}$$

Still C_2 covers all positives that C_1 does.

- With this notion we have gained a computable generality relation (requires finding a substitution).

- Also we have gained the property that LGG are unique wrt $\leq$. (unique upto variable renaming and certain other normalizations)

- So now the question is how to compute $LGG(C_1, C_2)$ under θ-subsumption.

- Intuitively the procedure is a type of "anti-unification" of C_1 and C_2. The procedure we will define works for any clause, definite or not, possibly involving function symbols.

- We will define the LGG of terms followed by the LGG of literals, followed by the LGG of 2 clauses.

LGG Definition for Literals

Terms:

$LGG(t,t) = t$ (identical terms)

$LGG(f(t_1,\ldots,t_n), f(s_1,\ldots,s_n)) = f(LGG(t_1,s_1),\ldots, LGG(t_n,s_n))$

If $t = f(t_1,\ldots,t_n)$ and $t' = g(s_1,\ldots,s_m)$ for $f \neq g$

$LGG(t,t') = V_{tt'}$, where $V_{tt'}$ is a new variable associated with tt'

$LGG(s,t) = V_{st}$, if s or t is a variable

Atoms:

$LGG(P(t_1,\ldots,t_n), P(s_1,\ldots,s_n)) = P(LGG(t_1,s_1),\ldots, LGG(t_n,s_n))$

$LGG(P(t_1,\ldots,t_n), Q(s_1,\ldots,s_m))$ is undefined when $Q \neq P$

Literals: (a literal is an atom or its negation)

- For positive literals L_1 and L_2 we get the LGG between atoms
- For $L_1 = \neg A_1$ and $L_2 = \neg A_2$ where A_1 and A_2 are atoms
  $LGG(L_1,L_2) = \neg LGG(A_1,A_2)$
- If L_1 and L_2 are of opposite polarity then $LGG(L_1,L_2)$ is undefined

---

Examples: $LGG(R(a,c), R(a,d)) = R(a, V_{cd})$

$LGG(R(c,f(c)), R(a,f(a))) = R(V_{ca}, f(V_{ca}))$

$LGG(R(c,f(c)), R(a,f(f(a)))) = R(V_{ca}, f(V_{cf(a)}))$

LGG for Clauses:

Suppose $C_1 = \{L_1, \ldots, L_n\}$
$C_2 = \{K_1, \ldots, K_m\}$
are 2 clauses with n and m literals each.

$$LGG(C_1, C_2) = \{ L' = LGG(L_i, K_j) \mid L_i \in C_1, K_j \in C_2, \text{ LGG}(L_i, K_j) \text{ is defined} \}$$

That is $LGG(C_1, C_2)$ is the disjunction of all pairwise LGGs between literals of C_1 and C_2 that are well defined.

Example:

$C_1 = \neg female(sophie) \vee \neg parent(alan, sophie) \vee daughter(sophie, alan)$
$C_2 = \neg female(jane) \vee \neg parent(bob, jane) \vee daughter(jane, Bob)$

$$LGG(C_1, C_2) = \neg female(V_{sj}) \vee \neg parent(V_{ab}, V_{sj}) \vee daughter(V_{sj}, V_{ab})$$
$$= female(x) \wedge parent(y, x) \rightarrow daughter(x, y)$$

$C_1 = \neg parent(alan, sophie) \vee grandfather(father(alan), sophie)$
$C_2 = \neg parent(xiaoli, sophie) \vee grandfather(father(xiaoli), sophie)$

$$LGG(C_1, C_2) = \neg parent(V_{ax}, sophie) \vee grandfather(father(V_{ax}), sophie)$$
$$= parent(X, sophie) \rightarrow grandfather(father(X), sophie)$$

- The LGG operator just described does indeed compute a least-general clause that generalizes the inputs.
- Recall that $C_1 \leq C_2$ means C_2 θ-subsumes C_1.

> **Theorem:** Let $C = LGG(C_1, C_2)$ be the clause computed by the above procedure. For any clause C', if $C_1 \leq C'$ and $C_2 \leq C'$ then $C \leq C'$.

- Furthermore it can be verified that $LGG(C_1, C_2)$ does θ-subsume both C_1 and C_2 as desired.

- Note that the LGG operator is commutative and associative

$$LGG(C_1, C_2) = LGG(C_2, C_1)$$

and

$$LGG(LGG(C_1, C_2), C_3) = LGG(C_1, LGG(C_2, C_3))$$

Also note that for any set of clauses $C = \{C_1, C_2, \ldots, C_n\}$ we can define the LGG for C to be the least general clause that ~~is a~~ is more general (or equivalent) to each C_i.

We can compute the LGG of a set of examples via pairwise LGG operations:

$$\cancel{LGG(\{C_1, \ldots, C_n\}) = LGG(C_1, \{}$$

$$\cancel{LGG(\{C_1, \ldots, C_n\}) = LGG(\{C_1, \ldots, C_{n-1}}$$

$$\boxed{LGG(\{C_1, \ldots, C_n\}) = LGG(LGG(\{C_1, \ldots, C_{n-1}\}), C_n)}$$

From above the order of pairwise LGGs does not matter.

- How can we use the LGG to solve our problem of learning a rule?

Given: B, P, N Return: a rule (i.e. definite clause) r such that for many $e \in P$ $B \wedge r \models e$ and for all $e \in N$, $B \wedge r \not\models e$.

- The high-level approach will first translate each positive example e into a very specific ground clause $S(e)$ s.t. $B \wedge S(e) \models e$. However, each ~~such~~ $C(e)$ will be so specific that it will not cover any other examples.

- Given the set of specific clauses $C = \{ S(e) : e \in P \}$ we will then try to find a subset C' of C such that the ~~clause~~ rule $r = LGG(C')$ covers many positives and no negatives.

- Lets now consider each of these steps.

Converting Positive Examples to Clauses

- Given a positive example e (e.g. $Q(a,b)$) we know that the clause $\to e$ is a clause that covers e. However this clause is quite general as it predicts $\to e$ regardless of B.
- We want to get a very specific clause that covers e relative to B.

- Recall that B is a conjunction (or set) of ground facts. A very specific clause that covers e relative to B is:

$$c = B \to e$$

Indeed $B \wedge c \models e$ but if we ~~generalize c, by rem add a~~ ~~tit~~ specialize c by adding facts not in B to the body then e will no longer be covered. So c is very specific.

- Thus in a strong sense

  $B \rightarrow e$ is the most specific clause relative to B that covers e.

- Thus, one choice would be to define $\varsigma(e)$ to be $B \rightarrow e$.

- However, $B \rightarrow e$ will have many facts in the body that are completely unrelated to e, meaning that the body is intuitively much bigger than it needs to be.

Example:

$$e = Rich(John)$$

$$B = \{HasNiceCar(John), HasGoodJob(John), HasNiceCar(Jane), HasGoodJob(Jane)\}$$

$B \rightarrow e$ includes facts about Jane in the body, which does not appear to be very useful. It seem more natural to limit the body to facts related to John.

In this case we would get
$HasNiceCar(John) \wedge HasGoodJob(John) \rightarrow Rich(John)$

- While the above is not strictly as specific as $B \rightarrow e$, it is most specific when we restrict the body to facts that are linked to the head by some chain of predicates.

- We will call such most specific clauses for e relative to B the saturation of e wrt B, denoted $s(e)$

- So the saturation is just $B' \rightarrow e$ where $B' \subseteq B$ contains all facts that are related somehow to e.

<u>Example:</u> $B = \{R(a,b), R(b,c), R(c,d), P(b), R(s,t), R(t,u), P(t)\}$

The saturation of $e = Q(a,d)$ is

$$R(a,b) \wedge \underbrace{R(b,c)} \wedge R(c,d) \wedge \underbrace{P(b)} \longrightarrow Q(a,d)$$

related to e
via chain of R's.

$R(s,t)$, $R(t,u)$, $P(t)$ not includes since they do not link to e.

- Now that we have the notion of saturation we can turn all of the positive examples into clauses. We will think of these as the most specific clauses that entail the examples with the background knowledge B.

- Given the saturations for e_1 and e_2 denoted $s(e_1)$ and $s(e_2)$ we know that $LGG(s(e_1), s(e_2)) = C_{12}$ covers both e_1 and e_2; i.e.

$$B \wedge C_{12} \models e_1 \wedge e_2$$ and is the least general clause that does so.

- We also know that if there is a negative example $n \in N$ s.t. $B \wedge C_{12} \models n$ then no ~~generaliza~~ generalization of both $s(e_1)$ and $s(e_2)$ can avoid covering n (since LGG is least).

- This suggests we search for the largest subset of saturated clauses ~~that~~ whose LGG covers no negatives. The resulting clause will cover many negatives and no positives.

- There are two challenges with this approach:
  1) Finding the best subset of saturated examples for which to compute LGG,
  2) Dealing with LGGs that get very large.

- We will first talk about (1) and inparticular describe how Golem solves this ~~task~~ problem.

- First, lets get some intuition about the problem.

- Recall that the overall GOLEM algorithm is aimed at learning a set of rules $r_1, \ldots, r_n$ that together cover all ~~exampt~~ positives. This intuitively means that we can think of the positive examples as being grouped according to the rule that will be used to cover it.

- For example, ~~pero~~ perhaps we have 4 pos examples: $\left\{ \begin{array}{l} sticky(m_1), sticky(m_2) \\ sticky(m_3), sticky(m_4) \end{array} \right\} = P$

and 1 neg. example $\{sticky(m_5)\} = N$

and m_1 and m_2 are sticky for a different reason than m_3 and m_4. That is, there are different rules for them.

Consider $B = \{P(m_1), P(m_2), Q(m_3), Q(m_4)\}$

the saturations are:

$$P(m_1) \rightarrow sticky(m_1) = S_1$$
$$P(m_2) \rightarrow sticky(m_2) = S_2$$
$$Q(m_3) \rightarrow sticky(m_3) = S_3$$
$$Q(m_4) \rightarrow sticky(m_4) = S_4$$

We can verify that

$$LGG(S_1, S_2) = P(x) \rightarrow sticky(x)$$
$$LGG(S_3, S_4) = Q(x) \rightarrow sticky(x)$$

which looks like a good set of rules for these example that do not cover negative examples.

But if we combine examples from different underlying rules we will over generalize:

$$LGG(S_1, S_3) = \rightarrow sticky(x)$$

which covers the negative example.

This examples shows that overgeneralization will often occur when we combine examples ~~from~~ whose underlying predictive rules differ.

So the rule learner is essentially trying to discover which sets of example belong to the same underlying rule and then generalize to get that rule.

- Now lets describe Golem's LearnRule:

- For clause C, example set E, and background facts B we define:

$$covers(C, B, E) = \{e \in E \mid B \wedge C \models e\}$$

i.e. the set of examples entaile by B and C.

- Our input to LearnRule is B, P, N and we want to output a clause/rule C that maximizes

$|covers(C, B, P)|$ under the constraint that $covers(C, B, N) = \emptyset$.

- Golem will first find a seed pair of examples to generalize (picking greedily according to coverage) and then greedily include one new ~~pair at a~~ example at a time into the generalization ~~until~~ ~~as~~ as long as no negatives are covered.

LearnRule(B, P, N)

$I = \{c = LGG(s(e), s(e')) \mid e \in P, e' \in P, \ covers(c, B, N) = \emptyset\}$

Repeat

$c^* = \arg\max_{c \in I} |covers(c, B, P)|$

$P = P - covers(c^*, B, P)$

$I = \{c = LGG(c^*, s(e)) \mid e \in P, \ covers(c, B, N) = \emptyset\}$

Until $I \neq \emptyset$

Return c^*

- The main complexity of the algorithm is the repeated LGG computations.

- How expensive are these computations?

- The time to compute $LGG(c_1, c_2)$ is $O(|c_1||c_2|)$ and the size of the resulting clause is also bounded by $|c_1||c_2|$.

- Unfortunately this shows that the size of an LGG resulting from multiple pairwise LGGs is only bounded by $|c_1||c_2|\cdots|c_n|$.

- This shows that the LGG can grow exponentially large in the # of clauses, and thus so can the time.
- This is bad news for ~~a practica~~ achieving a practical algorithm.

- Lets look at a good case and bad case for an LGG computation and then consider practical fixes.

Good/Ideal Case:

positives: $e_1 = Q(f(a),b)$, $e_2 = Q(f(c),d)$

$B = \{R(a,b), P(b), R(c,d), P(d)\}$

$S(e_1) = \{\neg R(a,b), \neg P(b), Q(f(a),b)\}$
$S(e_2) = \{\neg R(c,d), \neg P(d), Q(f(c),d)\}$

$$LGG(S(e_1), S(e_2)) = \{\neg R(V_{ac}, V_{bd}), \neg P(V_{bd}), Q(f(V_{ac}), V_{bd})\}$$
$$= (R(x,y) \wedge P(y)) \longrightarrow Q(f(x),y)$$
$$= C$$

- This clause is what we might expect.
- Suppose that we now attempt to generalize this clause with a new example that is inherently explained by a different rule.

$S(e_3) = \{\neg R(s,t), \neg N(u), Q(f(s), u)\}$

$LGG(C, S(e_3)) = \{\neg R(V_{xs}, V_{yt}), Q(f(V_{xs}), V_{yu})\}$

$= R(V_{xs}, V_{yt}) \rightarrow Q(f(V_{xs}), V_{yu})$

$= R(x, z) \rightarrow Q(f(x), y)$

- $S(e_3)$ has a different predictive pattern than $S(e_1)$ and $S(e_2)$ and as a result when we generalize we get the above clause ~~with~~ which is very likely to be overly general and cover negative example.

- Thus Golem would not consider ~~the step of adding in $S(e_3)$ an error and~~ adding $S(e_3)$ into this generalization.

Bad Case:

$e_1 = Q(a,c)$
$e_2 = Q(d,f)$

$B = \{ R(a,b), R(b,c), P(b), P(c)$
$R(d,e), R(e,f), P(e), P(f) \}$

$S(e_1) = \{ \neg R(a,b), \neg R(b,c), \neg P(b), \neg P(c), Q(a,c) \}$
$S(e_2) = \{ \neg R(d,e), \neg R(e,f), \neg P(e), \neg P(f), Q(d,f) \}$

$LGG(S(e_1), S(e_2)) = \{ \neg R(V_{ad}, V_{be}), \neg R(V_{ae}, V_{bf}),$
$\neg R(V_{bd}, V_{ce}), \neg R(V_{be}, V_{cf})$
$\neg P(V_{be}), \neg P(V_{bf})$
$\neg P(V_{ce}), \neg P(V_{cf})$
$Q(V_{ad}, V_{cf}) \}$

$= R(V_{ad}, V_{be}) \wedge R(V_{ae}, V_{bf}) \wedge R(V_{bd}, V_{ce}) \wedge R(V_{be}, V_{cf}) \wedge P(V_{be}) \wedge P(V_{bf})$
$\wedge P(V_{ce}) \wedge P(V_{cf}) \longrightarrow Q(V_{ad}, V_{cf})$

- We can see that this clause appears to contain alot of garbage in addition to what appears to be good patterns.
- For example the body literals $R(V_{ae}, V_{bf})$ and $R(V_{bd}, V_{ce})$ are not even linked to the head variables in any way while $R(V_{ad}, V_{be})$ and $R(V_{be}, V_{cf})$ are.

- In general full LGGs will produce many ~~arti~~ such excess patterns ~~and are~~ leading to the blowup in size of these clases.
- Note that when there is more background knowledge so that the saturations are larger, the problem becomes even worse.

- There are at least ~~two~~ three ways to deal with the blowup, each one being a way to prune the results of LGGs.

i) Remove logically redundant literals.

Note that in the above LGG whenever $R(V_{ad}, V_{be})$ is true then so is $R(V_{ae}, V_{bf})$. This means that we can remove $R(V_{ae}, V_{bf})$ from the clause and retain logical equivalence.

If we remove logically equivalent literals from the above we get.

$$R(V_{ad}, V_{be}) \wedge R(V_{be}, V_{cf}) \wedge P(V_{be}) \wedge P(V_{cf}) \rightarrow Q(V_{ad}, V_{cf})$$

$$\equiv$$

$$R(X,Z) \wedge R(Z,Y) \wedge P(Z) \wedge P(Y) \rightarrow Q(X,Y)$$

This clause looks correct as it captures the esential pattern in the two examples.

~~it is not always possible~~

But, in general testing for logically redundant literals requires expensive theorem proving, which is undecidable in the worst case.

Further, in many case the clauses can still be large after such pruning due to the fact that many unimportant patterns were captured. ~~WRT~~ Where importance is wrt ~~sepera~~ classifying positive from negative examples.

So in practice we usually consider simple logical simplification rules that can be checked quickly.

2) When the set of negative examples is sufficient, a very effective pruning rule is to remove a literal if after its removal no negatives are covered.

Golem has a greedy strategy for iteratively removing such literals. This is a very effective approach but relies on a good set of negative examples.

3) Language Restriction

Another approach is to restrict the types of clauses allowed and to prune clauses to stay within the restriction.

For example, one could limit the length of a "chain" of variables that connect to a head ~~viah~~ variable.

so $R(X,Y) \wedge R(Y,Z) \rightarrow Q(X,Z)$ might be allowed

but $R(X,Y) \wedge R(Y,Z) \wedge R(Z,V) \rightarrow Q(X,V)$ might not.

Golem has a special restriction called ij-determinacy, which we will not cover the details of.

- It is interesting to contrast FOIL and Golem, or more generally thes ~~sp~~ general-to-specific and specific-to-general approaches.

- In order to find complex rules, FOIL relies on a heuristic to guide it in the correct direction.

for example suppose a target rule is $R(x,v) \wedge R(v,w) \wedge R(w,y) \rightarrow Q(x,y)$

Such rules can be difficult for FOIL to find since the ~~heur~~ heuristic may not see that individual body atoms are useful by themselve.
such as $R(w,y)$ and $R(x,v)$

- In otherwords FOIL can sometimes fail due to shortsightedness. The situation can be improved some by increasing the amount of search (using non-greedy search), but there is a practical limit to this.

- In contrast Golem ~~starts with very detailed~~ will find ~~a~~ rules that capture all patterns that the examples have in common including ones that are not important and even logically redundant. So Golem can easily capture the pattern in the above rule, but the challenge is pruning down the rest of the patterns that are not important

- As you might guess people have considered ~~mer~~ combined appraches that, for example, will generate specific clauses via LGGs and generalize them via a FOIL-like process.