Implementing Inductive Concept Learning For Cooperative Query Answering

Maheen Bakhtyar[1⋆], Nam Dang[2⋆⋆], Katsumi Inoue[3], and Lena Wiese[4]

[1] Asian Inst. of Technology Bangkok, Thailand, Maheen.Bakhtyar@ait.asia
[2] University of Göttingen, Göttingen, Germany, lena.wiese@udo.edu
[3] National Inst. of Informatics, Tokyo, Japan, inoue@nii.ac.jp
[4] Tokyo Inst. of Technology, Tokyo, Japan, namd@de.cs.titech.ac.jp

Abstract. Generalization operators have long been studied in the area of Conceptual Inductive Learning (Michalski, 1983; De Raedt, 2010). We present an implementation of these learning operators in a prototype system for cooperative query answering. The implementation can however also be used as a usual concept learning mechanism for concepts described in first-order predicate logic. We sketch an extension of the generalization process by a ranking mechanism on answers for the case that some answers are not related to what user asked.

1 Introduction

Conceptual inductive learning is concerned with deriving a logical description of concepts (in a sense, a classification) for a given set of observations or examples; in induction, the resulting description is also called a hypothesis. Background knowledge can support the concept learning procedure. In his seminal paper on inductive learning, Michalski (1983) introduced and surveyed several learning operators that can be applied to a set of examples to obtain (that is, induce) a description of concepts; each concept subsumes (and hence describes) a subset of the examples. He further differentiates inductive learning into *concept acquisition* (where a set of examples must be classified into a predefined set of concepts) and *descriptive generalization* (where a set of observations must be classified into a newly generated and hence previously unknown set of concepts). In a similar vein, de Raedt (2010) emphasizes the importance of logic representations for learning processes as follows:

> "To decide whether a hypothesis would classify an example as positive, we need a notion of coverage.[...] In terms of logic, the example e is a logical consequence of the rule h, which we shall write as $h \models e$.

⋆ M.B. is currently a PhD student at AIT, Bangkok.
⋆⋆ N.D. is currently a Master student at TITech, Tokyo.

> This notion of coverage forms the basis for the theory of inductive reasoning[...]
> Especially important in this context is the notion of generality. One pattern is *more general* than another one if all examples that are covered by the latter pattern are also covered by the former pattern.[...] The generality relation is useful for inductive learning, because it can be used 1) to prune the search space, and 2) to guide the search towards the more promising parts of the space.[...] Using logical description languages for learning provides us not only with a very expressive and understandable representation, but also with an excellent theoretical foundation for the field. This becomes clear when looking at the generality relation. It turns out that the generality relation coincides with logical entailment.[...]"

In this paper we present the implementations of three logical generalization operators: Dropping Condition (DC), Anti-Instantiation (AI) and Goal Replacement (GR). The novelty of our approach lies in the fact that these operators are combined iteratively. In other words, several successive steps of generalization are applied and operators can be mixed. Our work is based on the soundness results of an optimized iteration that can be found in Inoue and Wiese (2011); the main result there is that it is sufficient to apply these three operators in a certain order: starting with GR applications, followed by DC applications and ending with AI applications (see Figure 1). This order is employed in our system when applying the three operators iteratively in a tree-like structure.

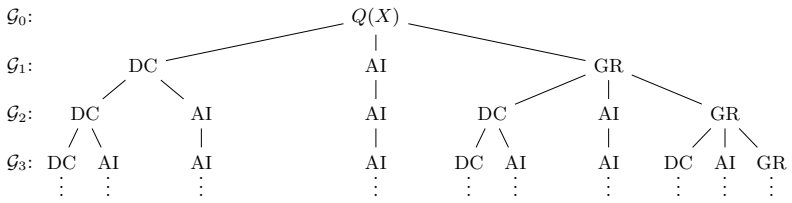

Fig. 1. Tree-shaped combination of DC, AI and GR

To the best of our knowledge, the iteration of these three operators is also novel when learning concepts from a set of examples. In this paper we present generalization as an application for a cooperative query answer system CoopQA: it applies generalization operators to *failing* queries which unsatisfactorily result in empty answers; by applying generalization operators we obtain a set of logically more general queries which might have more answers (called *informative* answers) than the original query. The implementation can however also be used in a traditional concept learning setting – that is, learning concepts by iterative generalization: as such we see the failing query as

an initial description of a concept (which however does not cover all positive examples contained in a knowledge base); we then use our tree-shaped generalization until all positive examples can be derived as informative answers to the more general query (and hence we have obtained a more general description of the initial concept).

More formally, we concentrate on generalization of conjunctive queries that consist of a conjunction (written as $\wedge$) of literals l_i; a literal consists of a logical atom (a relation name with its parameters) and an optional negation symbol ($\neg$) in front. We write $Q(X) = l_1 \wedge \ldots \wedge l_n$ for a user's query where X is a free variable occurring in the l_i; if there is more than one free variable, we separate variables by commas. The free variables denote which values the user is looking for. Consider for example a hospital setting where a doctor asks for illnesses of patients. The query $Q(X) = \mathit{ill}(X, \mathit{flu}) \wedge \mathit{ill}(X, \mathit{cough})$ asks for all the names X of patients that suffer from both flu and cough. A query $Q(X)$ is sent to a knowledge base Σ (a set of logical formulas) and then evaluated in Σ by a function *ans* that returns a set of answers (a set of formulas that are logically implied by Σ); as we focus on the generalization of queries, we assume the *ans* function and an appropriate notion of logical truth given. Note that (in contrast to the usual connotation of the term) we also allow negative literals to appear in conjunctive queries; we just require Definition 1 below to be fulfilled while leaving a specific choice of the $\models$ operator open. Similarly, we do not put any particular syntactic restriction on Σ. However, one of the generalization operators scans Σ for "single-headed range-restricted rules" (SHRRR) which consists of a body part left of an implication arrow ($\rightarrow$) and a head part right of the implication arrow. The body of a SHRRR consists of a disjunction of literals whereas the head consists only of one single literal: $l_{i_1} \wedge \ldots \wedge l_{i_m} \rightarrow l'$; range-restriction requires that all variables that appear in the head literal also appear in one of the body literals; again, we also allow negative literals in the body and in the head. As a simple example for a SHRRR consider $\mathit{ill}(X, \mathit{flu}) \rightarrow \mathit{treat}(X, \mathit{medi})$ which describes that every patient suffering from flu is treated with a certain medicine.

CoopQA applies the following three operators to a conjunctive query (which – among others – can be found in the paper of Michalski (1983)):

Dropping Condition (DC) removes one conjunct from a query; applying DC to the example $Q(X)$ results in $\mathit{ill}(X, \mathit{flu})$ and $\mathit{ill}(X, \mathit{cough})$.

Anti-Instantiation (AI) replaces a constant (or a variable occurring at least twice) in $Q(X)$ with a new variable Y; $\mathit{ill}(Y, \mathit{flu}) \wedge \mathit{ill}(X, \mathit{cough})$, $\mathit{ill}(X, Y) \wedge \mathit{ill}(X, \mathit{cough})$ and $\mathit{ill}(X, \mathit{flu}) \wedge \mathit{ill}(X, Y)$ are results for the example $Q(X)$.

Goal Replacement (GR) takes a SHRRR from Σ, finds a substitution θ that maps the rule's body to some conjuncts in the query and replaces these conjuncts by the head (with θ applied); applying the example SHRRR to $Q(X)$ results in $\mathit{treat}(X, \mathit{medi}) \wedge \mathit{ill}(X, \mathit{cough})$.

These three operators all fulfill the following property of deductive generalization (which has already been used by Gaasterland et al (1992)) where ϕ is the input query and ψ is any possible output query:

Definition 1 (Deductive generalization wrt. knowledge base). *Let Σ be a knowledge base, $\phi(\mathbf{X})$ be a formula with a tuple $\mathbf{X}$ of free variables, and $\psi(\mathbf{X}, \mathbf{Y})$ be a formula with an additional tuple $\mathbf{Y}$ of free variables disjoint from $\mathbf{X}$. The formula $\psi(\mathbf{X}, \mathbf{Y})$ is a* deductive generalization *of $\phi(\mathbf{X})$, if it holds in Σ that the less general ϕ implies the more general ψ where for the free variables $\mathbf{X}$ (the ones that occur in ϕ and possibly in ψ) the universal closure and for free variables $\mathbf{Y}$ (the ones that occur in ψ only) the existential closure is taken:*

$$\Sigma \models \forall \mathbf{X} \exists \mathbf{Y} \left(\phi(\mathbf{X}) \rightarrow \psi(\mathbf{X}, \mathbf{Y}) \right)$$

In the following sections, we briefly present the implementation of the CoopQA system (Section 2) and show a preliminary evaluation of the performance overhead of iterating generalization operators (Section 3).

2 Implementation Details

The focus of CoopQA lies on the efficient application of the underlying generalization operators. As CoopQA applies the generalization operators on the original query in a combined iterative fashion, the resulting queries may contain equivalent queries, which only differ in occurrences of variables or order of literals. CoopQA thus implements an equivalence checking mechanism to eliminate such duplicate queries. The most important step of query equivalence checking is finding substitutions between two queries. CoopQA tries to rearrange the literals in the two queries such that the substitutions can be obtained by mapping the variables in the two queries according to their positions. Instead of finding the substitution over the whole of the two queries, we segment the queries into *segments of pairwise equivalent literals* and find the correct ordering for each pair of corresponding segments. We now briefly sketch how each operator is implemented:

Dropping Condition (*DC*): For a query of length n (i.e., n literals in the query), n generalized queries are generated by dropping one literal. As this involves replicating the $n-1$ remaining literals, run-time complexity of DC is $O(n^2)$.

Anti-Instantiation (*AI*): If a query of length n contains M occurrences of constants and variables, at most M generalized queries (each of length n) are generated. Assume that the query was divided into r segments, then the literal affected by anti-instantiation has to be removed from its segment and the anti-instantiated literal with the new variable has to be placed into a (possibly different) segment; finding this segment by binary search requires $\log r$ time (which is less than n). Thus, run-time complexity of AI is $O(Mn)$.

Goal Replacement (*GR*): *GR*, as described in Section 1 (and in more detail in Inoue and Wiese (2011)), requires finding parts of the query that are subsumed by the body of a given SHRRR. The definition of subsumption is that a logical formula $R(X)$ subsumes $Q(Y)$ if there is a substitution θ

such that $R(X)\theta = Q(Y)$. Our implementation uses a generator that takes the query and the rule's body as input, and produces matchings of the body against the rule. We apply *relaxed* segmentation upon the two lists of literals (of the query and of the rule's body), which requires equivalent literals to only have the same predicate. For example, $q(X, Y)$ and $q(a, Z)$ are considered equivalent literals in this case. We then perform matching upon corresponding equivalent segments of the query against the segments of the rule's body to obtain a list of literals that are subsumed by the rule's body. Note that the segments in the query and the rule's body need not have the same size, as long as all the segments in the rule's body are found in the query. The matching literals are then replaced in the query by the rule's head, which already has the substitution applied.

The worst case scenario of GR is when both the query and the rule contain only one segment of (relaxed) equivalent literals. Given a query of size n, a rule with the body of size k ($n \geq k$), the number of possible matchings of k out of n literals from the query against the rule is $\binom{n}{k}$. For each combination of k literals, we have to perform permutation to find the correct ordering to determine the substitution against the rule body. Thus, in the worst case, the cost of finding the substitution for each combination is $k!$. Hence, complexity of finding all matchings of the query against the rule's body is $O(k!\binom{n}{k})$. In general, for a rule with a body of s segments, each of length k_i, and a query with s corresponding segments, each of length n_i, complexity is $O(\sum_{i=1}^{s}(k_i!\binom{n_i}{k_i}))$.

3 Performance Evaluation

We present some benchmarking and performance evaluation results of our implementation of the generalization operators. We focus on the cost of the tree-shaped generalization process.

Our benchmark suite generates a random knowledge base and query which consists large set of a combinations of illnesses and treatments from our med-

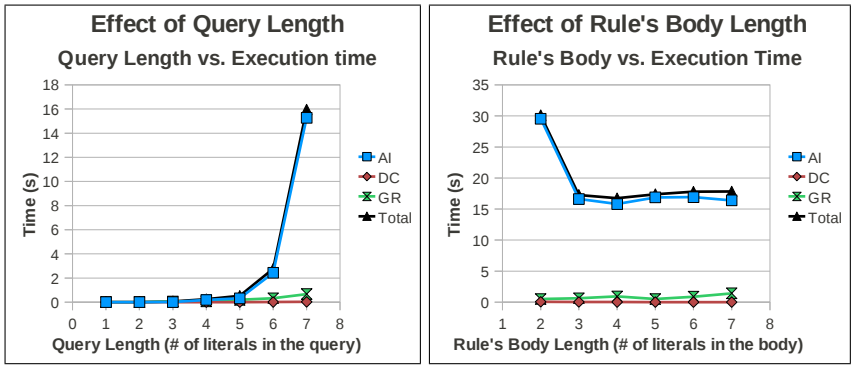

Fig. 2. Effects of query length and rule body length

ical example. We chose this synthetic data set because goal replacement has to capture the semantic dependencies in the data expressed by rules; so far we could not find any real-world benchmark with appropriate rules for goal replacement.

We analyze the effects on execution time by varying various parameters such as query length, number of SHRRRs in the knowledge base and the length of rule bodies. The runtime graphs show the time spent on each of the three operators alone as well as their summed total execution time. The tests were run on a PC under Linux 2.6 with a 64 bit Ubuntu 10.04 using Java 1.6 with OpenJDK Runtime Environment 6. The PC had 8 GB of main memory and a 3.0 Ghz Quad Core processor.

We first analyze how query length affects execution time of generalization. Figure 2 shows the total time taken by each operator when running CoopQA for a certain query length (that is, number of literals). We observe an increase in the execution time of AI operator after query length 5 as potentially more AI operations are possible; in case of DC and GR operation, effect of query length on the execution time is negligible. Analyzing the effect of number of rules contained in the knowledge base shows that the total execution time increases with the number of rules as shown in Figure 3. Closely investigating each operator reveals that there is no significant change in execution time in case of AI. In case of DC it is again negligible; however, we observe a linear increase in execution time in case of GR and that is because of increased matching and replacement. Lastly, there is no effect on DC and GR execution

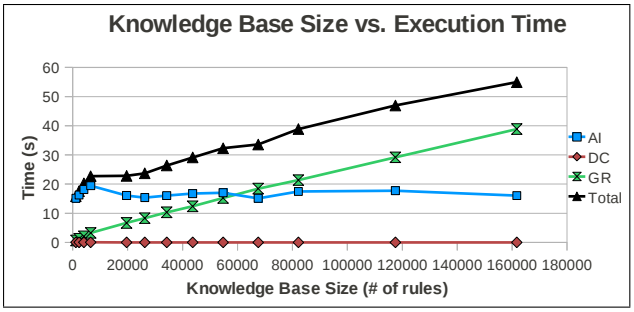

Fig. 3. Effects of number of rules

time when rule bodies are longer (see Figure **??**). Yet, time decreases for AI. This is due to the replaced part of the query: a long rule body makes the query shorter by replacing a longer part of the query with one literal; hence less AI operations are possible.

Profiling our code using VisualVM[5] shows that AI operation takes up 48.4% of the total execution time; the main cost lies in performing equivalence

[5] http://visualvm.java.net/

checking to detect duplicated queries when a new one is generated (taking up to 33.6% of total execution time). *GR*'s main bottleneck is to replace matched literals to generate queries, not the matching against rules itself.

4 Discussion and Conclusion

The CoopQA system uses three generalization operators that were widely used for inductive learning of concepts; it applies them in a cooperative query answering system in order to discover information for a user which might be related to his query intention. In contrast to other approaches using *DC*, *AI* and *GR*, tree-shaped generalization profits from an efficient combination of the three operators. The presented implementation shows favorable performance of the generalization operators. Future and ongoing work in the CoopQA system is covering the important issue of answer relevance which we discuss briefly: some answers might be generalized "too much" and are "too far away" from the user's query intention; a relevance ranking for answers can provide the user with the most useful answers while disregarding the irrelevant ones. In particular, a *threshold* value for answer ranks can be specified to return only the most relevant answers to the user and an aggregation of ranks reflecting the iteration of operators must be defined. Dropping Conditions and Goal Replacement are purely syntactic operators that do not introduce new variables. A relevance ranking for them can be achieved by assigning the answers to generalized queries a penalty for the dropped or replaced conditions. In contrast, anti-instantiation leads to the introduction of a new variable. Recall from Section 1 the example query $Q(X) = \mathit{ill}(X, \mathit{flu}) \wedge \mathit{ill}(X, \mathit{cough})$ that asks for patients suffering from flu and cough at the same time. Applying AI on the constant cough leads to the generalized query $\mathit{ill}(X, \mathit{flu}) \wedge \mathit{ill}(X, Y)$ where the condition of cough is relaxed and any other disease would be matched to the new variable Y. These diseases might be very dissimilar to cough and hence totally irrelevant from the point of view of the original query $Q(X)$. A more intelligent version of the AI operator can hence rank the answer to the generalized query regarding their *similarity to the original query*. Here we have to differentiate between the case that a constant was anti-instantiated, and the case that a variable was anti-instantiated. More precisely, within a single application of the AI operator, we can assign each new (more general) answer ans_j the rank value rank_j, where the rank is calculated as follows:

- if Y is the anti-instantiation of a constant c (like cough in our example), we obtain the similarity between the value of Y in answer ans_j (written as $\mathit{val}_j(Y)$) and the original constant c; that is, $\mathit{rank}_j = \mathit{sim}(\mathit{val}_j(Y), c)$.
- if Y is the anti-instantiation of a variable (like in the generalized query $\mathit{ill}(Y, \mathit{flu}) \wedge \mathit{ill}(X, \mathit{cough})$ where different patients X and Y are allowed), we obtain the similarity between the value of Y in ans_j and the value of the variable (say, X) which is anti-instantiated by Y in the same answer; that is, $\mathit{rank}_j = \mathit{sim}(\mathit{val}_j(Y), \mathit{val}_j(X))$.

Let us assume we have predefined similarities $sim(bronchitis, cough) = 0.9$ and $sim(brokenLeg, cough) = 0.1$ for our example. An answer containing a patient with both flu and bronchitis would then be ranked high with 0.9; whereas an answer containing a patient with both flu and broken leg would be ranked low with 0.1. Such similarities can be based on a taxonomy of words (or an ontology); in our example we would need a medical taxonomy relating several diseases in a hierarchical manner. Several notions of distance in a taxonomy of words can be used (e.g., Shin et al (2007)) to define a similarity between each two words in the taxonomy (e.g., Wu and Palmer (1994)). When the AI operator is applied repeatedly, similarities should be computed for each replaced constant or variable; these single similarities can then be combined into a rank for example by taking their *weighted sum*. To make the system more adaptive to user behavior, the taxonomy used for the similarities can be revised at runtime (with an approach as described in Nikitina et al (2012)).

References

DE RAEDT, L. (2010): About Knowledge and Inference in Logical and Relational Learning. In *Advances in Machine Learning II*, pp. 143–153, Springer.

GAASTERLAND, T., GODFREY, P. and MINKER, J. (1992): Relaxation as a platform for cooperative answering. *Journal of Intelligent Information Systems, 1(3/4)*, pp. 293–321.

INOUE, K. and WIESE, L. (2011): Generalizing conjunctive queries for informative answers. In *9th International Conference on Flexible Query Answering Systems*. Lecture Notes in Artificial Intelligence, vol. 7022, pp. 1–12, Springer.

MICHALSKI, R. S. (1983): A Theory and Methodolgy of Inductive Learning. In *Machine Learning: An Artificial Intelligence Approach*, pp. 111–161, TIOGA Publishing.

MUSLEA, I. (2004): Machine learning for online query relaxation. In *Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 246–255, ACM.

NIKITINA, N., RUDOLPH, S. and GLIMM, B. (2012): Interactive ontology revision. In *Journal of Web Semantics 12*, pp. 118–130.

SHIN, M. K., HUH, S.-Y. and LEE, W. (2007): Providing ranked cooperative query answers using the metricized knowledge abstraction hierarchy. In *Expert Systems with Applications, 32(2)*, pp. 469–484.

WU, Z. and PALMER, M. (1994): Verb Semantics and Lexical Selection. In *32nd Annual Meeting of the Association for Computational Linguistics*, pp. 133–138, Morgan Kaufmann Publishers.