Bottom-Up Propositionalization

Stefan Kramer[1] and Eibe Frank[2]

[1] Institute for Computer Science, Machine Learning Lab
University Freiburg, Am Flughafen 17, D-79110 Freiburg/Br.
skramer@informatik.uni-freiburg.de
[2] Department of Computer Science
University of Waikato Hamilton, New Zealand
eibe@cs.waikato.ac.nz

Keywords: Propositionalization, Feature Construction, Constructive Induction, Relational Learning, Inductive Logic Programming, Support Vector Machines

Abstract. In this paper, we present a new method for propositionalization that works in a bottom-up, data-driven manner. It is tailored for biochemical databases, where the examples are 2-D descriptions of chemical compounds. The method generates all frequent fragments (i.e., linearly connected atoms) up to a user-specified length. A preliminary experiment in the domain of carcinogenicity prediction showed that bottom-up propositionalization is a promising approach to feature construction from relational data.

1 Introduction

One approach to handle relational learning problems is to transform them into propositional problems and subsequently apply propositional learning algorithms. This transformation requires the construction of features that capture relational properties of the learning examples. In this way, feature construction and model construction are decoupled. Such a transformation of a relational learning problem into a propositional one is called propositionalization. In this paper, we present a new approach to propositionalization

This paper is organized as follows: In Section 2 we propose a new method for propositionalization that proceeds in a data-driven manner. In Section 3, we present promising, preliminary results obtained with bottom-up propositionalization and support vector machines. Section 4 concludes the paper.

```
procedure fragment_generator_top_level

Input:
 molecules:                    examples to be transformed
Global:
 t:                            look-up table of local properties
 examples_for_fragment:        output ``by side-effect''

t ← create look-up table of local properties for each molecule
for each molecule m do
 for each heavy (non-hydrogen) atom a in m do
   e ← element(a)
   examples ← filter_examples(molecules,'e',t)
   depth-first(1,[a],'e',examples)
end procedure
```

Fig. 1. Pseudocode of the top-level of the fragment generator

2 Bottom-Up Propositionalization

In this section, we describe a new, bottom-up, data-driven method for propositionalization. The algorithm is particularly tailored for bio-chemical domains, where learning examples are 2-D descriptions of chemical compounds. The purpose of the algorithm is to discover *fragments* (i.e., linearly connected atoms) that occur frequently in the dataset. An example of a fragment would be *'o-s-c'*, meaning "an oxygen atom with a single bond to a sulfur atom with a single bond to a carbon atom". In contrast to fragments, we define *paths* as lists of concrete atom identifiers that match some fragment in a molecule. E.g., a path may be a list of (concrete) atoms $[a_1, a_2, a_3]$ that "instantiates" the fragment *'o-s-c'*.

We also distinguish between *generation* and *evaluation* of fragments. Generation is based on paths in individual molecules and computationally not very expensive. Evaluation means determining the set of molecules in which a fragment occurs. In contrast to generation, evaluation is computationally very expensive. So, our goal in algorithm design was to make excessive use of memory and hashing in order to avoid unnecessary, redundant evaluations.

Fig. 1 shows the pseudo-code of the top level of the algorithm. Basically, the top-level takes each heavy (non-hydrogen) atom in each molecule as a starting point and invokes the search for fragments for this atom. Besides (and before that), the top-level creates a look-up table of local properties of atoms and bonds. For each molecule, we store the elements and bond types (or bond orders) occurring in it. This is done for optimization purposes.

For each atom as a starting point, we invoke the depth-first search procedure as shown in Fig. 2. The arguments are the depth of search so far d, the current *path* in the molecule, the current *fragment* and the *examples* containing the current fragment. If d exceeds the maximum depth of search, search terminates and the algorithm backtracks. Otherwise, it loops through all refinements of the current *fragment*. *ref_path* denotes a path that is a refinement of *path*, *ref_fragment* denotes a refined *fragment*, and *ref_examples* denotes the set of examples that contain the refined fragment *ref_fragment*. Refinement steps are guided by concrete paths in molecules. E.g., path *'o-s-c'* may be refined yielding fragment *'o-s-c-c'*; this refined fragment is found by some extension of the path $[a_1, a_2, a_3]$ by one atom a_4, obtaining the new path $[a_1, a_2, a_3, a_4]$.

Next, the algorithm checks whether the newly obtained fragment is known to be infrequent. If this is the case, search terminates (in this branch) and the next refinement is tried. If not, the algorithm checks whether the refined fragment has already been evaluated. If it has already been evaluated, we retrieve the set of examples for the fragment and recursively invoke *depth-first*. Search continues in this case, since the fragment is not infrequent: Further refinements could lead to fragments that have not been evaluated, but still are occurring frequently.

In case a new, refined fragment has not been evaluated before, it will be evaluated then. After evaluation, the fragment is asserted as evaluated in the hash table. If the frequency of the fragment is large enough, we assert the examples (the set of examples containing the fragment) of the fragment to the hash table and the procedure is recursively invoked. If the frequency is below the user-specified threshold, the fragment is marked as infrequent in the hash table and search terminates in this branch.

The overall search strategy of the algorithm is depth-first, but this is really a detail of the approach. Search could be implemented in different ways as well. Actually, we chose depth-first just because this search strategy comes for free in Prolog. Note that we need two recursive calls to *depth-first* just for simplicity of presentation. Also note, that the algorithm primarily works by means of side effects. During search, it outputs the examples containing frequent fragments.

Provisions have to taken that two equivalent fragments with atoms in reverse order (e.g., 'o-s-c' and 'c-s-o') are recognized as such. The obvious solution is to store and compare fragments only in a canonical order that has to be defined on fragments.

To summarize, the essential features of the algorithm are:

- *Frequency-based pruning* in the spirit of association rule mining. In this respect, our approach is related to Warmr [2]. One difference is that we make heavy use of memory for pruning (see the next point). Frequency-based pruning turned out to be very efficient for fragment generation. Further optimizations along the lines of the Apriori algorithm [1] are of course conceivable.
- Based on the observation that generation of fragments is relatively cheap, but evaluation is expensive, we make *heavy use of memory and hashing.* This is helpful for avoiding redundant computations as well as for pruning.
- We took a *bottom-up approach* to fragment generation. The idea is to generate only those fragments that really occur in examples. If a fragment occurs in an example, it is checked whether it is already evaluated on all other examples.
  Our fragment generator shares this bottom-up flavor with systems like Golem [6] and Progol [5]. Also, the underlying idea is similar to relational pathfinding [7].
- We used a *look-up table of local properties* for optimization purposes.

3 Results of the New Approach

We performed a preliminary experiment with the new approach in the domain of predicting carcinogenicity. The dataset comprises 337 examples and 31126 tuples. The default accuracy is 54%. Strong relational learners like S-CART [3][4] achieve an accuracy of about 65%, but these results are due to non-structural properties (mostly the so-called Ames test, a genotoxic test that is known to be correlated with carcinogenicity). Removing these strong, non-structural descriptors, S-CART produces a theory only four percent above the baseline accuracy (58%).

In the experiment, the fragment generator was set to search for fragments up to length 8. The frequency threshold (the minimum coverage parameter) was set to 5. The runtime of the fragment generator was about 10 minutes on a

```
procedure depth-first

Input:
  d:                                   depth of recursion so far
  path:                                path in some molecule
  fragment:                            fragment associated with path
  examples:                            examples containing fragment
Global:
  t:                                   look-up table of local properties
  examples_for_fragment :              output ‘‘by side-effect’’
  max_depth :                          parameter restricting depth of search
  min_coverage :                       required minimum frequency of fragments
  min_maj_coverage :                   required min. frequency of majority class

if (d > max − depth) then return
else
  new_d ← d + 1
  for each (ref_path, ref_fragment) ∈ refinements(path, fragment) do
    if ¬known_to_be_infrequent(ref_fragment)
    then if evaluated(ref_fragment)
         then get examples_for_fragment(ref_fragment, ref_examples)
              depth-first(new-d, ref_path, ref_fragment, ref_examples)
         else ref_examples′ ← filter_examples(examples, ref_fragment, t)
              ref_examples ← evaluate(ref_fragment, ref_examples′)
              assert(evaluated(ref_fragment))
              coverage ← |ref_examples|
              positive_coverage ← |positive(ref_examples)|
              negative_coverage ← |negative(ref_examples)|
              if coverage < min_coverage or
                 max(positive_coverage, negative_coverage) < min_maj_coverage
              then assert(known_to_be_infrequent(ref_fragment))
              else assert(examples_for_fragment(ref_fragment, ref_examples))
                   depth-first(new_d, ref_path, ref_fragment, ref_examples)
end procedure
```

Fig. 2. Pseudocode of the search algorithm employed by the generator

Linux PC with a Pentium II processor. The fragment generator identified 656 fragments as frequent, generating 11944 Prolog ground facts for the occurrence of these fragments in the examples. During search, 1813 fragments were marked as infrequent and remembered for pruning in search.

After propositionalization, we applied an algorithm for support vector machines to the transformed problem. We conjecture that support vector machines are particularly useful in the context of propositionalization, as they can deal with a large number of moderately significant features.

We applied the implementation of support vector machines in the WEKA workbench. Parameter E was set to 2 and C was set to 0.001. Since there is (currently) no method to determine these parameter values in a principled way, we determined the values of these parameters experimentally.

The model resulting from these settings included 297 support vectors. Most importantly, the predictive accuracies obtained from 10-fold cross-validations vary between 58% and 63%. Given that this has been achieved using only 2-D information (not even considering partial charges, let alone genotoxic tests), the result has to be regarded as very good. Although drawing general conclusions from this experiment seems unwarranted, we believe that this method, or rather, this combination of methods, is a promising alternative to existing approaches.

4 Conclusion

In this paper, we presented a new approach to propositionalization that works in a bottom-up, data-driven manner. It is tailored for bio-chemical databases where the examples are 2-D descriptions of chemical compounds. In an experiment, we have shown that the generation of all frequent fragments up to length 8 can be computed within a reasonable time, and that the predictive accuracy obtained using these fragments is satisfactory. However, further experiments will have to confirm that this is a viable approach to feature construction from relational background knowledge.

Acknowledgments

We would like to thank Kristian Kersting and the anonymous reviewer for commenting on an earlier version of the paper.

References

1. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. 1996.

2. L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.
3. S. Kramer. Structural regression trees. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, Menlo Park, CA, 1996. AAAI Press.
4. S. Kramer. *Relational Learning vs. Propositionalization: Investigations in Inductive Logic Programming and Propositional Machine Learning*. PhD thesis, Vienna University of Technology, Vienna, Austria, 1999.
5. S. Muggleton. Inverse Entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
6. S. Muggleton and C. Feng. Efficient induction of logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, London, U.K., 1992.
7. B.L. Richards and R.J. Mooney. Learning relations by pathfinding. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 50–55, Menlo Park, CA, 1992. AAAI Press.