Inductive Logic Programming in Haskell - Progress Report

Marcus Peacock 0919274

26^{th} October 2012

Project Subject

Inductive Logic Programming is a research area at the intersection of machine learning and logic programming. My interest for this project lies in implementing an inductive learning system, or inductive learner, which presents plausible hypotheses which link together facts and observations in the form of relations.

Inductive Logic Framework

Inductive learners can either be interactive or empirical (with some systems being able to switch between these modes), where an empirical system deals with batches of data and interactive systems deal with information one piece at a time. My project will be focused on empirical learning, however, it will be able to deal with suggested working hypothesis, which is a feature of interactive learners.

Inductive learners start with a set of facts that are taken as truth which we call the background knowledge, a set of observations of a new relation which we call the positive examples and an optional set of observations where this new relation is false which we call the negative examples.

As an example the background knowledge could be the representation of a family tree using the parent, child, male, female and sibling relations and we are interested in learning the definition for the new relation 'uncle'. The background knowledge is the set of facts between literals that we know are true, such as: parent("dave", "jim"), sibling("joe", "dave") representing that "dave" is the parent of "jim" and "joe" is the sibling of "dave". An observation in this system might be uncle("joe", "jim"), representing that "joe" is the uncle of "jim". When it comes to negative examples there are two options; give a defined set of negative observations or have the system create the set of negative observations using the closed world assumption, that anything not observed to be true is false. The objective for this example is to come up with a relation that defines the uncle relation. Given enough information the system should be able to output the real world definition of uncle.

There are various complete Inductive Logic Systems such as FOIL [5], FOCL [2], GOLEM [4] and FORTE [4]. Each of these systems have been continually updated over the years to add more advanced features and implement freshly designed algorithms. The system I write will hopefully perform some of the basic functions of these systems, I am able to implement these features thanks to articles by S. Muggleton [3] and N. Lavrac and S. Dzeroski [1].

An empirical Inductive Logic system can be broken down in to three parts: information pre-processing, hypothesis construction and hypothesis post processing. The post processing can also be performed as part of the hypothesis construction. Hypothesis within my system will be in the form of a conjunction of positive or negated clauses.

Information pre-processing deals with handling imperfect data and generating negative examples if they are not given. There are various ways to generate negative examples but my implementation will use the negative examples it is given as a complete set or when given the empty set, generate every possible negative example using the closed world assumption. There should also be some way to determine if the input given is valid along with the ability to correct small errors typing errors in the information.

Hypothesis construction in my system works from general to specific. This part of the system can be seen as a graph search, the root node is usually the empty clause (it may also be a suggested working hypothesis) and the system generates all the possible clauses that can be added that would improve the hypothesis. This part of the system is the most difficult and time consuming as a hypothesis with just a few clauses can have a huge amount of possibilities to try, for this reason I will focus on attempting to only find clauses which truly improve the hypothesis and are not irrelevant. This is where hypothesis post-processing comes in, in order to create a suitably efficient system the post-processing must be done each time a clause is added, we need to remove all irrelevant and repeat possibilities as early and efficiently as possible.

Many Inductive Logic systems attempt to find a single most likely hypothesis as they will often work with incomplete data, this has pros and cons. As described by B. L. Richards and R. J. Mooney [6], these systems can get stuck in local maxima instead of finding optimal solutions, this leads to using beam search and probabilistic solutions. However, my focus will be on complete data and therefore instead of searching for a single solution within a certain probability threshold, I will begin by attempting to find every hypothesis that entails all the positive examples and none of the negatives. If a hypothesis is found with a certain number of clauses, there is no need to find one with more clauses as following the Occam's razor principle we should prefer the most simplistic solutions. Along the same lines we can also order these hypothesis by the number of variables they use, since a hypothesis that holds and mentions fewer variables is more simple and therefore preferable.

Implementation Progress

Eventually I hope to be able to deal with files as inputs of information, but for the time being I have encoded the examples straight into the system.

I decided that the best way to represent information was simply as a tuple, with the front of the tuple being the name of the fact and the back of the tuple being a list of the arguments or literals within that fact. At first I did not use the concept of negated clauses within my system, but I quickly decided that it would be easiest in the long run to encode whether a clause is negated within the clause itself, therefore my final representation is a 3-tuple with the first element being True or False to represent if the clause is positive or negative.

The background knowledge is represented as a set of clauses with the above format.

A relation is represented as a tuple also, the first element being the relation we are trying to find a hypothesis for and the second element being the set of clauses that are conjoined to make up the main part of the hypothesis. In a relation however, instead of using literals we use variables. Variables are represented with integers in order to make sure there is a solid distinction between literals and arguments.

In order to determine whether a relation holds we need to be able to decide if the relation covers all the positive and all the negatives. In order to do this we have to go through the positive examples, map the literals in the example with an integer argument and see if we can find a mapping to the rest of the variables such that all the clauses appear in the background knowledge, once this has been repeated for all the positive examples and they all hold, we can say this relation holds. Similarly for the negative examples a relation does not hold if it covers any of the negative examples. As the number of examples and variables increase it very quickly starts to take a lot longer to check that a relation covers the examples, which is yet another reason why we want to limit the amount of possibilities that are generated as potential clauses.

Implemented Example

To get a clearer understanding of the system I present the following simple example.

```
backno = [(True, "parent", ["ann", "mary"]),
            (True, "parent", ["ann", "tom"]),
            (True, "parent", ["tom", "eve"]),
            (True, "parent", ["tom", "ian"]),
            (True, "female", ["ann"]),
            (True, "female", ["mary"]),
            (True, "female", ["eve"])]

pEx = [(True, "daughter", ["mary", "ann"]),
        (True, "daughter", ["eve", "tom"])]
```

If we take backno as the set of facts that represent the background knowledge and pEx to be the set of observed facts about the relation we want to define. Starting with the empty relation:

```
rel = ((True, "daughter", [1,2]), [])
```

Representing the most general relation possible, that True entails daughter(1,2). In this example we will assume no negative observations are given, meaning we must generate all the possible negative examples, of which there are 23 but I wont list them here. Since this starting relation is valid for all the positive examples as well as the negative examples, we must start the search because we want to find the relation which covers no negative examples.

Every clauses that uses up to 3 variables is generated, we then filter these to find the clauses which cover the positive examples, leaving the following set:

```
[[(True,"parent",[2,1])],
    [(True,"parent",[2,3])],
    [(True,"female",[1])]]
```

Since none of these cover none of the negative examples the search continues, conjoining clauses with each member of this set until a relation is found that covers all the positive examples and none of the negatives. The relation found in more human readable format is:

Daughter(1,2) ← Female(1) AND Parent(2,1)

As expected this example runs very quickly, in a matter if milliseconds, and my project currently functions on larger examples, but it should give ea clear view of the goals of the system.

Moving Forward

At the moment I am using family trees as a domain for testing, as it provides human readable examples and it is easy to check if the system discovers the real world relation. So far the system can very quickly find hypothesis which use either none or a single 'undefined' variable, for example the relation mother(1,2) ¡- parent(1,2), female(1), where commas represent conjunction and uncle(1,2) ¡- parent(3,2), sibling(3,1). However, as the number of facts given as information decreases the number of valid relations that aren't the real world hypothesis increase, but the required time to find solutions and check if a solution holds increases with the amount of information given, this leads to the fact that finding algorithms that work quickly not only decreases the time taken to find solutions but also allows the use or larger data sets and can therefore find much less potential hypothesis.

It should be noted that currently my system can only determine whether hypothesis hold that do not include any negated clauses, since I started using family trees as my domain it was not a concern because family relationships don't include negated clauses, but in order for the system to be able to find

correct hypotheses in other domains I will need to implement that as part of the system.

One of the main difficulties is that it can take a long time to encode examples that can possibly result in correct answers. A family tree example with 14 people only using facts that relate siblings and parents could have 20 positive facts and over 400 negated facts, showing just how difficult it is to find a correct solution with limited information and how important it is that the system is as efficient as possible. Using an example of this size there are almost 10 different possible hypothesis for the uncle relation that are logically correct, but obviously only 1 of them is the real world definition.

Moving forward with the project I will work on refining the work I have already done and extending it to be able to deal with negated clauses. Once I am convinced that the system works as intended it will be able to test its efficiency on examples of different sizes and improve the complexity of relations it can discover in reasonable time.

Next term I will also work on structuring my code in more intelligent and concise ways, using resources such as Hoogle (http://www.haskell.org/hoogle/) to write more succinct functions and separate my work in to logic sections.

Revised Timetable

Date	Milestone
26th November 2012	Progress Report
10th January 2013	System functional with negated clauses
1st February 2013	Suitable examples written and tested
1st March 2013	Efficiency and readability work complete
4th March 2013	Project Presentation
25th April 2013	Final Report

References

[1] N. Lavrac and S. Dzeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, New York, 1991.

[2] C. A. Brunk M. J. Pazzani and G. Silverstein. A knowledge-intensive approach to relational concept learning. *Proceedings of the Eighth International Workshop on Machine Learning, pp. 432-436*, 1991.

[3] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4), 1991.

[4] S. Muggleton and C. Feng. Efficient induction of logic programs. *Proceedings of the First Conference on Algorithmic Learning Theory*, 1990.

[5] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning, 5:239-266*, 1990.

[6] B. L. Richards and R. J. Mooney. Learning relations by pathfinding. *Appears in Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92), pp.50-55, San Jose, CA*, 1992.