Natural Language Processing With Prolog in the IBM *Watson* System

Adam Lally
IBM Thomas J. Watson Research Center

Paul Fodor
Stony Brook University

24 May 2011

On February 14-16, 2011, the IBM Watson question answering system won the Jeopardy! Man vs. Machine Challenge by defeating two former grand champions, Ken Jennings and Brad Rutter. To compete successfully at Jeopardy!, Watson had to answer complex natural language questions over an extremely broad domain of knowledge. Moreover, it had to compute an accurate confidence in its answers and to complete its processing in a very short amount of time.

The Question-Answering (QA) problem requires a machine to go beyond just matching keywords in documents, which is what a web-search engine does, and correctly interpret the question to figure out what is being asked. The QA system also needs to find the precise answer without requiring the aid of a human to read through the returned documents.

To address these challenges, the research team at IBM developed a software architecture called DeepQA, on which Watson is implemented. The DeepQA architecture assumes and pursues multiple interpretations of the question, generates many plausible answers or hypotheses, collects evidence for these hypotheses, and evaluates the evidence to determine if it supports or refutes those hypotheses [2]. Watson contains hundreds of different algorithms that evaluate evidence along different dimensions.

Watson utilizes Natural Language Processing (NLP) technology to interpret the question and extract key elements such as the answer type and relationships between entities. Also, NLP was used to analyze (prior to the competition) the vast amounts of unstructured text (encyclopedias, dictionaries, news articles, etc.) that may provide evidence in support of the answers to the questions. Some of Watson's algorithms evaluate whether the relationships between entities in the question match those in the evidence.

Watson's NLP begins by applying a parser [5] that converts each text sentence into a more structured form: a tree that shows both surface structure and deep, logical structure. For example, in the following example Jeopardy! question:

> *POETS & POETRY: He was a bank clerk in the Yukon before he published "Songs of a Sourdough" in 1907*

The output of the parser includes, among many other things, that “published” is a verb with base form (or *lemma)* “publish”, subject “he”, and object “Songs of a Sourdough”.

Next, Watson applies numerous detection rules that match patterns in the parse. These rules detect elements such as the *focus* of the question (the words that refer to the answer, in this case “he”), the *lexical answer types* (terms in the question or category that indicate what type of entity is being asked for, in this case “poet”), and the relationships between entities in either a question or a potential supporting passage.

We required a language in which we could conveniently express pattern matching rules over the parse trees and other annotations (such as named entity recognition results), and a technology that could execute these rules very efficiently. We found that Prolog was the ideal choice for the language due to its simplicity and expressiveness. The information in the parse is easily converted into Prolog facts, such as (the numbers representing unique identifiers for parse nodes):

```
lemma(1, "he").
partOfSpeech(1,pronoun).
lemma(2, "publish").
partOfSpeech(2,verb).
lemma(3, "Songs of a Sourdough").
partOfSpeech(3,noun).
subject(2,1).
object(2,3).
```

Such facts were consulted into a Prolog system and several rule sets were executed to detect the focus of the question, the lexical answer type and several relations between the elements of the parse. A simplified rule for detecting the `authorOf` relation can be written in Prolog as follows:

```
authorOf(Author,Composition) :-
    createVerb(Verb),
    subject(Verb,Author),
    author(Author),
    object(Verb,Composition),
    composition(Composition).

createVerb(Verb) :-
    partOfSpeech(Verb,verb),
    lemma(Verb,VerbLemma),
    member(VerbLemma, ["write", "publish",...]).
```

The `author` and `composition` predicates, not shown, apply constraints on the nodes (“he” and “Songs of a Sourdough”, respectively) to rule out nodes that are not valid fillers for the author and composition roles in the relation.

This rule, applied to the example, results in the new fact `authorOf(1,3)`, which is recorded and passed to downstream components in the Watson pipeline.

Now, assume that among the evidence that Watson gathered while attempting to answer the question is the text:

Songs of a Sourdough by Robert W. Service

This is phrased differently from the question and it would not match the example `authorOf` rule shown above. However, we have many other clauses of the `authorOf` relation that match different expressions of the same semantic relation, including one that applies in cases such as this, for example:

```
authorOf(Author,Composition) :-
    composition(Composition),
    argument(Composition,Preposition),
    lemma(Preposition, "by"),
    objectOfPreposition(Preposition,Author),
    author(Author).
```

Since both the question and text passage have a common relation, Watson can determine that the passage provides good support for the answer "Robert W. Service".

This is a very simple example that illustrates just one kind of pattern matching that Watson performs. Watson uses many different techniques for detecting and scoring the occurrence of concepts and relations in text including statistical and other rule-based methods.

In practice, natural language is complex and ambiguous, and the pattern matching rules that we require are therefore more complex than this simplified example. This problem is suited for solving with backtracking over pattern matching because we must check for lots of conditions over the parse and we want a query language where we can include/exclude these conditions depending on some context. The Prolog language is recognized to be an excellent solution for the problem of pattern matching and all problems that involve a depth-first search and backtracking [6, 1]. Although simple, the Prolog language is very expressive allowing recursive rules to represent reachability in parse trees and the operation of negation-as-failure to check the absence of conditions.

Prior to our decision to use Prolog for this task, we had implemented custom pattern matching frameworks over parses. These frameworks tend to end up replicating some of the features of Prolog but lack the full feature set of Prolog or the efficiency of a good Prolog implementation. Using Prolog for this task has significantly improved our productivity in developing new pattern matching rules and has delivered the execution efficiency necessary in order to be competitive in a Jeopardy! game.

More details on our use of NLP and Prolog in Watson are forthcoming [4, 3].

References

[1] Michael A. Covington. *Natural Language Processing for Prolog Programmers*. Prentice Hall, 1994.

[2] David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A. Kalyanpur, Adam Lally, J. William Murdock, Eric Nyberg, John Prager, Nico Schlaefer, and Chris Welty. Building Watson: An Overview of the DeepQA Project. *AI Magazine*, 31(3), 2010.

[3] Adam Lally, John Prager, Michael McCord, Branimir Boguraev, Siddharth Patwardhan, James Fan, Paul Fodor, and Jennifer Chu-Carroll. Question Analysis: How Watson Reads a Clue. *IBM Journal of Research and Development*, submitted.

[4] Michael McCord, Branimir Boguraev, John Prager, , and J. William Murdock. Parsing and Semantic Analysis in DeepQA. *IBM Journal of Research and Development*, submitted.

[5] Michael C. McCord. Using Slots and Modifiers in Logic Grammars for Natural Language. *Artificial Intelligence*, 18(3):327–367, 1982.

[6] Leon Sterling and Ehud Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques*. MIT Press, 2 edition, 1993.