CS151 Project Report: Movie Q&A Chatbot

Lara Bagdasarian(lbagdas) Emma Zhong(emmazjy)

URL to Run: https://swish.swi-prolog.org/p/movie_chatbot.pl; File Submission Details: **readme.pdf**

Problem

Our chatbot parses a user-input movie-oriented question entered in natural language and formulates a natural language response sourced from a small database containing movie information. In the case of an ambiguous query, the chatbot will prompt the user to fill in the missing information in order to retrieve the most precise answer. The key tasks associated with implementing such a chatbot are tokenizing a natural language query, parsing it into key terms, retrieving relevant data from a database, and outputting a meaningful response that draws attention to missing information in a query with missing fields. The three categories (attributes), from which users can ask questions about and provide info regarding are movies, directors, and stars.

A Logic Programming Solution

We used a logic programming approach to handle all aspects[1] of our chatbot's functionality. Logic programming served our needs by providing a paradigm in which logical relationships (such as grammatical rules) can be easily encoded and link subproblems in a readable way. Such rule-based organization of subproblems is particularly well suited to structuring natural language queries. Since our chatbot is tasked to answer queries about a particular subject-matter with strongly associated key-words (e.g. "movie", "film", "director", etc.), we were able to anticipate common grammatical constructions and a limited vocabulary with which we can expect users to enter a question and formulate them into logical rules. Using Prolog to formalize these grammatical components through our implementation of a definite clause grammar (DCG) allows us to utilize hard-coded movie jargon like several common words to refer to a movie that we can anticipate usage of as well as common relative locations of question substructures. In this way we were able to ultimately identify the necessary information in a query in order to obtain meaningful data from a database in a way that was also easy to update when we thought of new question constructions, since such an update usually involved updating only a submodule instead of modifying all related code. Furthermore, our rule-based formulation of grammar rules gave us easy access to missing information, which we used to prompt users to input information that would disambiguate queries with the potential for multiple interpretations.

Challenges

One obstacle that we encountered related to using logic programming for all aspects of our program had to do with updating parameters in the heads of our logic programming rules. Our function heads often held many parameters at higher levels, which came from when we found ourselves often needing to revise our code at a lower submodule, such as at maintaining a suffix to keep track of number (i.e. singular or plural) at the lowest-level stages of parsing, we needed to update all higher-level functions which needed access to this information (in the case of our singular/plural example, output-level functions also need access to this information to determine whether to output a single result or multiple, which means all intermediate functions must hold a "Suffix" variable in their heads). Thus, plotting out what information would be critical at higher-level tasks when working on lower levels of our program became crucial.

Performance and difficulties with converting between Prolog and Epilog

The implementation of some functions in the parsing procedure requires scanning through the entire input list. It can potentially become a bottleneck for more complicated queries with long entries. While testing,

[1] We chose to write a Python script to parse the CSV file containing all the movie data into Prolog readable format due to the fact that logic programming is not suitable for parsing large files. By default our database contains data of 1000 movies, each with one director and three stars. With our Python script, it is easy to extend our database to a larger volume.

we noticed that the Epilog version of our code is substantially slower than the Prolog version. This slowdown might be due to the fact that we implemented in Epilog a lot of the Prolog built-in functions, including `split_string` and `nth1`. When implementing our Epilog version of our project, we found ourselves "deconstructing" Epilog built-in functions such as when implementing our tokenization functionality in Epilog, string splitting was done in a roundabout way that used the Epilog built-in functions inefficiently such as converting our query into a list of symbols and then back to a list of strings. In Prolog we were used to having character-level control over strings which we utilized in our string functions. If we were to have originally implemented our project in Epilog originally, we would have utilized Epilog built-in functions in a more natural way that we suspect might have sped up program runtime.

Design of Parsing Functionality

Parsing data into Prolog

The ultimate goal of parsing a query from the user is to obtain the values (or lack thereof) of three query features essential to retrieving results: question attribute (QAttr) or the Attribute which is being requested, provided attribute (PAttr), or the Attribute corresponding to the information provided, and the information provided (ProvidTerm), which must be contained in full or in part (in the case of names) in our database. Our higher-level parsing patterns are written in terms of how these modules (and a question word (qWordPat) module) aid in recognizing the structure of a sentence and where the other module might be held in relation to one another) relate in sequence and also handle select exceptions which were difficult to formalize according to this framework (e.g. trailing "in").

Supported query inputs

As there are many ways to inquire about movies, we have selected a subset of grammatical constructions that we allow users to enter their queries in and will outline their common features below using the following terminology. A user-input query must conform to the following patterns:

Question words (QWordPat) (+ denotes a suffix may be added): "give+", "who+", "what+", "show+"
Attribute instances (QAttr, PAttr): movie, director, star
Number: plural ending ("s") of QWordPat yields multiple results and non-plural ending chooses single random
Provided info from database (ProvidTerm): full element from database or just last name if actor or director

Pattern	Example	Notes
[QWordPat] [QAttr] [suffix determining number] [ProvidTerm] [PAttr]	*What movie has Quentin Tarantino directed?*	ProvidTerm has to come right after QAttr or right before PAttr
[QWordPat] [QAttr] [suffix determining number] [PAttr] [ProvidTerm]	*Show movies that are directed by Quentin Tarantino*	
[QWordPat] [PAttr] [ProvidTerm]	*Who starred in The Hateful Eight?*	ProvidTerm has to be movie title
[QWordPat] [ProvidTerm] [PAttr]	*Who did The Hateful Eight have as director?*	ProvidTerm has to be movie title
[QWordPat] [ProvidTerm as star pattern] "in"	*What was Quentin Tarantino in?*	'Movie' implied as QAttr

Future Steps

An additional feature that we believe would be useful in future iterations is the option for a user to request "why" a result was given. Such a feature would communicate the understood question and provided info and what grammatical construction we believe was used to provide insight into how the query was parsed and consequently hints as to how a user could improve their queries. Another extension would be to support more patterns and identifying more keywords(such as "both", "not") in our parsing procedure, we would be able to support complex queries like "What movies did Johnny Depp and Tom Cruise both star in?". This addition would also require an extension of our query interface to support set operations including union, intersection and complement.