Chris Dellario

MET CS 664

Project Design (Phase II)

6/27/2012

1. RESPONSE TO PHASE 1 COMMENTS  
  
Please see attached.

2. FINAL REQUIREMENTS

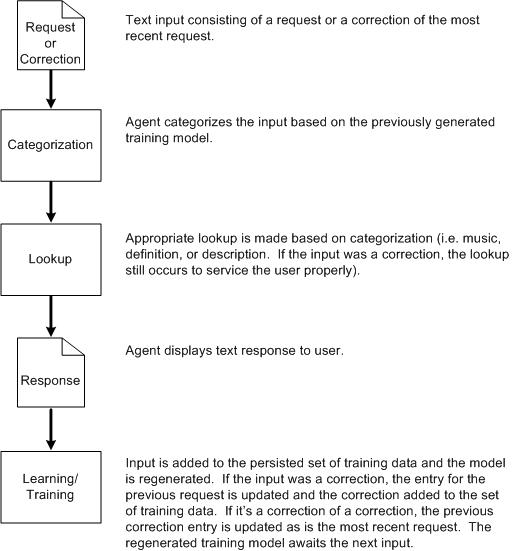
* D1F — Intelligent classification of input using natural language parsing into three basic categories. Note that in all three cases the system imposes one constraint on the input, that the word or phrase of interest (the one to be played, defined, or described) be wrapped in quotation marks (Please see section 4 below for details).
  + Request relating to music
  + Request for the definition of a term
  + Request for the encyclopedic description of a term
* D2F — Servicing of the requests
  + For music, the music library will consist of a text file listing songs and artists. Supported operations will consist of the following:
    - Play a song or artist. If the song or artist isn’t found in the music library then a message to that effect will be returned.
    - Pause something currently playing
  + For definitions, the agent will use the Stands4.com Dictionary Definition API[[1]](#footnote-1) to return a simple definition.
  + For descriptions, the agent will use the Wikipedia.org API[[2]](#footnote-2) to return the first paragraph of the content summary.
* D3F — Learning from feedback
  + The agent will use reinforcement-based learning, and will in turn refine its understanding of which verbs and other language elements relate to which types of requests. Requests that are not corrected will signal positive reinforcement while corrected input will indicate the opposite.
* N1L — To learn how to train the OpenNLP syntactic parser, providing a solution for determining the word or phrase of interest without wrapping it in quotes.
* N2F — Support for ambiguous input, such as a definition request that returns a list of possible terms from which the user then can choose.
* N3F — Support for elaborating on the output of the description such that more than the first paragraph of the content summary is returned.

3. DESIGN AND THEORY



Sequence of states per request

The system utilizes two artificial intelligence techniques, natural language processing (NLP) and reinforcement-based learning, in a cycle that categorizes a natural language request or correction, services it, and then learns from it. Through this cycle, the agent’s categorizations will change to suit the wishes of the user. For example, the initial set of training data I created assumes that a single-word request implies a dictionary lookup (e.g. “Zebra”), but through correction the system can learn to infer an encyclopedic lookup was intended.

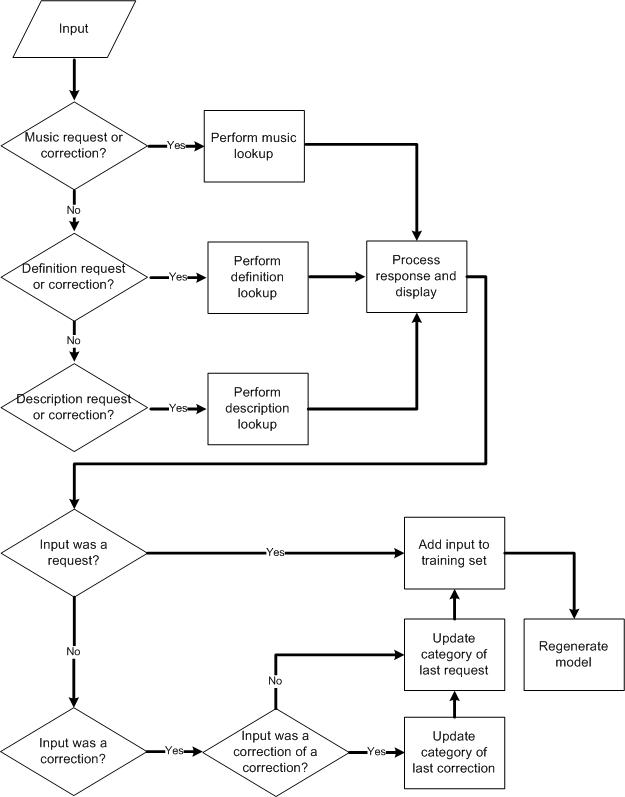


Data Flow of the agent

NLP systems categorize lexically by tagging words into their parts-of-speech, and then categorize groups of words into phrases to form syntactic categories that identify things like phrases and clauses[[3]](#footnote-3). These lexical and syntactic tag sets aren’t changeable (Please see information on the Penn Treebank tagset below). My agent needs a way to categorize at a sentence level and do so with categories I’ve defined, capabilities that the OpenNLP library provides (discussed in detail below). The agent begins with a model, a machine-readable file generated from a set of sample training sentences I created. To generate the model, the library uses the statistical analysis technique of maximum entropy to look for likely patterns in the relationships to parts-of-speech and their positions in the sentence[[4]](#footnote-4). For the categorization of the request types and correction inputs I created six categories, three for requests and three for their respective corrections:

|  |  |
| --- | --- |
| MusicCategory | MusicCorrectionCategory |
| DefinitionCategory | DefinitionCorrectionCategory |
| DescriptionCategory | DescriptionCorrectionCategory |

The process begins when a user enters input, a request or correction, to the system. The agent categorizes the input per the categories defined above and based on the result performs the appropriate lookup. The return from the lookup is parsed and displayed to the user. Next the system modified its set of training data, which contains all previous inputs. If the input was a request, it is added to the set. If the most recent input was a correction, the category of the most recent request is updated. And if the input was a correction of a correction, both the categories for the previous correction and most recent request are updated. Lastly, the model is regenerated based upon the newly updated training data.



With reinforcement-based learning the agent learns from a series of both positive and negative feedback[[5]](#footnote-5). In this case, the addition of the input to the training data set provides positive feedback, standardizing the relationship between the given input and its category. The update to the category following a corrective input serves as negative feedback.

4. TOOLS  
  
I opted to use tools for this project as my interest lay with a functional system capable of handling the three request types mentioned above. In preparing to design the system I evaluated several natural language processing libraries (and sub-libraries) and dictionary and encyclopedia services.

An enabler for my decision to leverage existing tools was the large number of natural language processing libraries I found in my initial investigation (please see the attached project proposal for details). From that set I chose two to evaluate: Learning Based Java (LBJ) and OpenNLP.

LBJ sports some impressive features but I discounted it in part because of the proprietary language it leverages. Having to learn to use its language and compiler added another set of tasks to an already long list, and having read about OpenNLP’s capabilities that additional investment didn’t appear worth the effort.

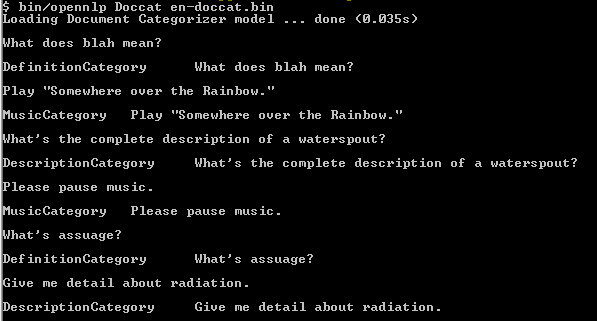
OpenNLP contains its own set of libraries which are suited to specific tasks. I decided to evaluate three and I investigated them in the following order: the Parts-of-Speech Tagger[[6]](#footnote-6), Parts-of-Speech Parser[[7]](#footnote-7), and Document Categorizer[[8]](#footnote-8). Each of these three libraries requires a pre-generated model to function, a model consisting of a text file containing hexadecimal formatted data. Each library comes with a training application used to create these models.

The Parts-of-Speech Tagger, or POSTagger, takes sentences as input and returns them with tags, indicating the part of speech, added to each word. OpenNLP uses a standard set of parts-of-speech tags called the Penn Treebank[[9]](#footnote-9) tagset, more extensive and granular than the parts of speech most of us learned in school.

I initially downloaded several training models[[10]](#footnote-10) from the OpenNLP site but found they made frequent mistakes with certain words, particularly with “please.” The addition of courtesy to any sentence tended to throw the tagger for a loop, with “please” placed at any point in the sentence resulting in it being incorrectly tagged as a verb rather than as an adverb. I subsequently created my own model by downloading a large training set[[11]](#footnote-11) formatted for use with the Parts-of-Speech Parser and reformatting it so as to be understood by the Tagger via a Perl script I wrote, and in my set of training data I corrected the usage of “please” along with a few other words.

Even with some corrections, though, the Tagger proved only somewhat useful. I learned that what I wanted was a tool that would provide insight into not only the part-of-speech but also which words formed the predicate. This led me to the Parts-of-Speech Parser, which includes not only word-level tags but also clause and phrase level ones. I ran tests using the single model available for download, but those tests failed to yield useful results because, I suspect, of the training data that informed the model. I attempted to create my own model using the training API but found its instructions too incomplete. For example, a required argument called a head-rule file isn’t described at all beyond a “todo” comment to document it. Without more information I can’t train the Parser, leading to the input constraint that terms of interest be wrapped in quotation marks. I’m pursuing the developer forums for further information.

My goal with these tools, though, was ultimately to classify the input requests, and here I found the Document Categorizer most helpful. Given a model it categorizes the inputted document into one of the pre-defined categories contained in that model. I created a model containing three categories for requests and three for corrective (i.e. negative reinforcement) statements, and found it surprisingly effective even given the tiny training sets (<60 examples each).



Sample categorization of requests shown through the document categorization command line interface

5. SCHEDULE

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Activity** | **Description** | **Status** |
| 6/4 | Project proposal | Submit proposal document. | Completed |
| 6/11 | Complete library selection | Investigate LBJ and OpenNLP and determine which to pursue | Completed |
| 6/18 | Service and tool selection | Finalize which services to call and which APIs to use. |  |
| 6/25 | Create foundational training model | Create a training set for the DocumentCategorizer that produces consistent and desirable results | Completed |
| 7/3 | Project Design | Submit design document. | Completed |
| 7/9 | Implement categorization and service calls | Use API instead of command line tool for calling the DocumentCategorizer and call services based on result. | Pending |
| 7/16 | Implement modification of the training set | Modify the training set based on the input and retrain the model | Pending |
| 7/23 | Implement parsing of results from services | Parse the results from the services. | Pending |
| 7/30 | Implementation | Attempt to train the OpenNLP Parser again. Try other alternatives. Cleanup code. Prepare for presentation. | Pending |
| 8/6 | Presentation | Submit and present the results of the project. | Pending |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **D** | **C-** | **C+** | **B-** | **B+** | **A** |  |
| **1. Clarity** | Disorganized or hard-to-understand | | Satisfactory but some parts of the submission are disorganized or hard to understand | Generally organized and clear | Very clear, organized and persuasive presentation of ideas and designs | Exceptionally clear, organized and persuasive presentation of ideas and designs | A |
| **2. Technical Soundness** | Little understanding of, or insight into material technically | | Some understanding of material technically | Overall understanding of much material technically | Very good overall understanding of technical material, with some real depth | Excellent, deep understanding of technical material and its inter-relationships | A |
| **3. Thorough-ness & Coverage** | Hardly covers any of the major relevant issues | | Covers some of the major relevant issues | Reasonable coverage of the major relevant areas | Thorough coverage of almost all of the major relevant issues | Exceptionally thorough coverage of all major relevant issues | A |
| **4. Relevance** | Mostly unfocused | Focus is off topic or on insubstantial or secondary issues | Only some of the content is meaningful and on topic | Most or all of the content is reasonably meaningful and on-topic | All of the content is reasonably meaningful and on-topic | All of the content is entirely relevant and meaningful | A |
| **5 Utilization of resources** | No useful use of notes, text(s), or Web with incorrect details or applicability | | Some useful use of notes, text(s), or Web with mostly correct details or applicability | Fairly good use of notes, text(s), or Web with correct details or applicability | Very good use of notes, text(s), or Web with correct details or applicability | Excellent use of notes, text(s), or Web with entirely correct details or applicability | A |
|  | Your grade: | 4.0 Excellent! |  |  |  |

1. <http://www.abbreviations.com/definitions_api.php> [↑](#footnote-ref-1)
2. <http://www.mediawiki.org/wiki/API:Main_page> [↑](#footnote-ref-2)
3. Russel & Norvig, p. 888. [↑](#footnote-ref-3)
4. http://www.cs.cmu.edu/afs/cs/user/aberger/www/html/tutorial/node2.html#SECTION00011000000000000000 [↑](#footnote-ref-4)
5. Russell & Norvig, p. 695. [↑](#footnote-ref-5)
6. http://opennlp.apache.org/documentation/1.5.2-incubating/manual/opennlp.html#tools.postagger [↑](#footnote-ref-6)
7. http://opennlp.apache.org/documentation/1.5.2-incubating/manual/opennlp.html#tools.parser [↑](#footnote-ref-7)
8. http://opennlp.apache.org/documentation/1.5.2-incubating/manual/opennlp.html#tools.doccat [↑](#footnote-ref-8)
9. http://www.comp.leeds.ac.uk/ccalas/tagsets/upenn.html [↑](#footnote-ref-9)
10. http://opennlp.sourceforge.net/models-1.5/ [↑](#footnote-ref-10)
11. http://stp.ling.uu.se/~nivre/master/penn02-21.trees [↑](#footnote-ref-11)