LILI Interpreter Documentation

Release 1.0

Brandon Gottlob

July 08, 2015

CONTENTS

The primary goal of this project is to make LILI more intelligent when given voice commands in English. LILI currenty understands only a handful of commands, and these commands must be given the exact expected sentence structure. The LILI Interpreter makes it possible for LILI to understand an open vocabulary of words in more ambiguous sentence structures.

Contents:

CHAPTER ONE

WNTEST MODULE

class `semsim.wntest.`**`SemanticSimilarityResult`**(*unknown*, *known*, *unknown_synset*, *known_synset*, *unknown_definition*, *known_definition*, *sem_sim_score*)

This class is used to package results from semantic similarity tests. Each object of this class holds the result of a single unknown to known word mapping.

This class stores a variety of information for analysis purposes, including the synsets with the highest similarity score, their definitions, the semantic similarity score, and most importantly the known word that the unknown word will map to. This class is to be used for testing and analysis purposes to see how the semantic similarity measure may be improved. The only pieces of information important to the final result of the LILI interpreter is the known word that the unknown word is mapped to.

`unknown`

str

The unknown word that has been mapped to a known word

`known`

str

The known word that has been mapped to

`unknown_synset`

wn.Synset

The synset of the unknown word

`known_synset`

wn.Synset

The synset of the known word that has been mapped to

`unknown_defintion`

str

The definition of unknown_synset

`known_definition`

str

The definition of known_synset

`sem_sim_score`

number

The semantic similarity score between unknown_synset and known_synset

`semsim.wntest.`**`filter_results`**(*results_list*, *threshold*)

Returns a filtered list of the results of the given list based on the given semantic similarity score threshold

Given a list of *SemanticSimilarityResult* objects and a threshold value, filters the list removing result objects with semantic similarity scores less than the threshold. Returns the filtered list.

Parameters

- **`results_list`** (*list*) – The list of *SemanticSimilarityResult* objects to be filtered
- **`threshold`** (*number*) – The semantic similarity score threshold at which results with a score lower than this threshold will be removed from C{results_list}

Returns The filtered list of C{SemanticSimilarityResult} objects

Return type list

`semsim.wntest.`**`output_results`**(*results_list*, *output_filename*)

Prints the given list of `SemanticSimilarityResult` objects to a CSV file

Given a list of `SemanticSimilarityResult` objects and a .csv filename, writes the values of the result objects to specified file.

Parameters

- **`results_list`** (*list*) – The list of `SemanticSimilarityResult` objects to be printed to the CSV file
- **`output_filename`** (*str*) – The name of the CSV file to be written to

`semsim.wntest.`**`process_results`**(*results_list*)

Sorts a list of `SemanticSimilarityResult` objects in descsending order by semantic similarity score

Parameters **`results_list`** (*list*) – The list of `SemanticSimilarityResult` objects to be sorted

Returns The sorted list of `SemanticSimilarityResult` objects

Return type list

`semsim.wntest.`**`sem_sim_test2`**(*known_words_filename*, *unknown_words_filename*, ***kwargs*)

Tests semantic similarity mapping from unknown words to known words. Synsets of the unknown words are not known in advance and those of the known words are determined in advance.

Accepts a CSV file of known words paired with their assumed WordNet synset and a text file of unknown words (with one word per line). Each unknown word is matched up with the known word that it is most semantically similar to. “Known” words are words that LILI has been preprogrammed to recognize or respond to in some way, while “unknown” words are those that LILI does not understand by default. Semantic similarity measures are made using WordNet and attempt to allow LILI to understand an open vocabulary beyond the words and phrases it has been preprogrammed to respond to. This test returns a list of *SemanticSimilarityResult* objects to store the results of the test for each unknown word.

Parameters

- **`known_words_filename`** (*str*) – The filename of the CSV file containing known words paired with their assumed synsets
- **`unknown_words_filename`** (*str*) – The filename of the text file containing unknown words

Kwargs: pos (str): The part of speech of the words to be evaluated. Can have the values “verb”, “noun”, “adj”, or “adv”. If neither of these values are used or no value is provided, searching the synsets of the unknown word will not be filtered by part of speech, resulting in more processing time and potentially less accurate results

Returns The sorted list of `SemanticSimilarityResult` objects

Return type list

CHAPTER

TWO

INTERPRETER MODULE

`interpreter.interpreter.`**`binary_search_actions`**(*target*, *pool*)

Binary search to find a *target* word in a *pool* of possibilities.

This binary search is used to find a *target* word with fast performance even with a large search *pool*. It is implemented recursively to provide a simpler implementation. It assumes that the known actions list is sorted in descending (A to Z) order.

Parameters

- **`target`** (*str*) – The word to search for
- **`pool`** (*list*) – A list of tuples (*str*, *int*) - the first value of each tuple is the word to compare agains - the second value in each tuple is an index value that corresponds to a set of synonymous actions

Returns The index of the found word's corresponding set of synonymous actions - returns -1 if no match is found

Return type int

`interpreter.interpreter.`**`build_action_structures`**(*filename*)

Given a filename that contains a list of known main actions, generates the data structures needed to interpret commands.

Opens a file of known actions, where each action set is represented on one line, and each word contained in that action set is separated by a comma (if there are multiple words in that set). For each word in this file, a tuple is generated that contains that word along with the line number it is found on in the file (starting with 0). That line number value is the main action's index, which maps the word to it's set of synonymous actions. This tuple is then appended to the list of known actions. This list is sorted by A-Z order before it is returned to prepare it for binary search operations. A list of object extractor functions is created to correspond to the main action indices to be called later on. For each line, another function is added to the extractor function list. To determine the name of the extractor function to be added next, the first word in the action set (effectively the first word in the current line) is appended to the end of the string "object_dict_". Once the function name string is built, the function is retreived from the extractor module and is appended to the extractor function list.

Parameters **`filename`** (*str*) – The path (or filename if in the current directory) of the file that contains the list of known actions

Returns A tuple containing the two list structures needed to interpret sentences. The first list is the sorted list of known actions coupled with their aciton index values. The second list is a list of object extractor functions whose list indices correspond with the appropriate action index

Return type (list, list)

Note: This function only needs to run when the file is updated. It should not be run every time a new command needs to be interpreted.

`interpreter.interpreter.`**`extract_action`**(*sent*, *known_actions*)

Finds the main action that LILI can respond to given a tokenized command sentence.

The main action is most likely one of the first couple of words of the command, so the sentence is processed from first to last word. Each word in the sentence is searched for in an array of known actions. The first word that is found in the known actions array is determined to be the main action, and processing ends. A binary search algorithm is used to search for the word to keep processing time short even with large lists of known actions. A tuple that contains two values is returned. The first value is an index value that corresponds to the set of words that the found word maps to. The second value is an index value representing the position of the found word in the sentence.

Parameters

- **`sent`** (*list*) – A list containing the tokenized sentence
- **`known_actions`** (*list*) – A list of tuples, each containing the string of an action and an index value of the set of actions it corresponds to

Returns Tuple - first int is an index value that maps the found word to the other actions it is synonymous with - second int is an index value representing the position of the found word in the given sentence - Returns (-1, 0) if no main action is found

Return type (int, int)

`interpreter.interpreter.`**`generate_json`**(*action*, *object_dict*)

Returns a JSON string representation of an object dictionary with an entry for the main action's index.

First adds the main action and its index to the object dictionary, then uses the json module to dump the final dictionary into a JSON string.

Parameters

- **`action`** (*int*) – The index of the action that corresponds to its synonym set
- **`object_dict`** (*dict*) – An object dictionary to be converted to a JSON string

Returns A JSON string representation of the object dictionary and the main action index

Return type str

`interpreter.interpreter.`**`generate_object_dict`**(*sent*, *action_tuple*, *object_extractor_functions*)

Creates a dictionary of keywords from the sentence that are objects of the action to be taken.

Begins by part of speech tagging the sentence, then trimming the main action out of the sentence, so that it is not re-processed (depending on implementation details, the presence of the main action in the sentence may throw off results). Then calls upon the appropriate object extractor from the extractor module. The called object extractor is determined by the index value of the action that maps it to its set of synonymous actions.

Parameters

- **`sent`** (*list*) – A list containing the tokenized sentence
- **`action_tuple`** (*int, int*) – Tuple - first int is an index value that maps the found word to the other actions it is synonymous with - second int is an index value representing the position of the found word in the given sentence - Returns (-1, 0) if no main action is found - same as the return value for extract_action

Returns A dictionary that maps words from the sentence to types of objects that will be acted on

Return type dict

`interpreter.interpreter.`**`preprocess_text`**(*text*)

Preprocesses a raw text sentence. Currently only breaks it up into tokens.

Used to do any preprocessing on a sentence in raw text. Currently, the only preprocessing operation is tokenizing the sentence into individual words and punctuation marks.

Parameters **`sent`** (*str*) – A string containing a command sentence

Returns A list of preprocessed tokens from the sentence

Return type list

Note: Do not do any part of speech tagging in this function. It is a computation intensive task and may not be needed for all commands.

`interpreter.interpreter.`**`test_sent`**(*sent_text*)

Implements the order of execution (pipeline) of interpreting a sentence - utilized for checking test cases

CHAPTER

THREE

INTERPRETER MODULE DEMO

3.1 Initial Setup

```
move,go
turn,twist,rotate
stop
follow
talk,speak,tell
show,teach
```

Each line in this input file represents one *action set*. All *actions* in an set are recognized as having the same meaning and will be interpreted and executed by LILI in the same exact way. For example, *show* and *teach* are considered synonyms since they are in the same action set. Therefore the commands *Show me how to wash my hands* and *Teach me how to wash my hands* have the exact same meaning to the interpreter.

Each action set has an *action set index value*, which is calculated as the line number the set is found on minus 1.

- For example:
  - *[move, go]* has an action set index of 0
  - *[turn, twist, rotate]* has an action set index of 1

```
res = interp.build_action_structures("./source/known_actions.txt")
known_actions = res[0]
print known_actions

object_extractor_functions = res[1]
funcs = [func.__name__ for func in object_extractor_functions]
print funcs
```

The above code generates two important data structures in the interpretation task:

1. List of known actions
   - Each element is a tuple containing the action and its action set index
   - Each tuple takes the form (*action*, *action_set_index*)
     - The *action* is only used to find a specific word in the command
     - The *action_set_index* gives actual meaning to the action
2. List of *object extractor functions*
   - Pulled from the `extractor` module
   - There is one function per action set

- Each function corresponds a single action set
- The index of each function in the list is the same as its correspond action set's index
- The name of each function begins with `'object_dict\_` and the first action in its corresponding action set is appended to the end

```
[('follow', 3), ('go', 0), ('move', 0), ('rotate', 1), ('show', 5), ('speak', 4), ('stop', 2), ('tal
['object_dict_move', 'object_dict_turn', 'object_dict_stop', 'object_dict_follow', 'object_dict_talk
```

The first line of output shows the contents of the known actions list.

The second line of output shows the name of each function in the object extractor function list. The actual function is stored as an object in the list `object_extractor_functions`, but a list of the names are printed as output here for readability. Therefore, each individual function can be called later on by indexing that list like so: `object_extractor_functions[<index>](<params>)`.

Notice that the action set index of each known action corresponds to the appropriate object extractor. For example, the object extractor for *talk*, *speak*, and *tell* are in the action set with an index of 4 and their object extractor can be accessed at index 4 (the 5th element) of `object_extractor_functions`.

Also notice that the name of the extractor functions begin with `"object_dict\_"` and end with the first action of its corresponding action set. This concept is important to maintainability of the system. When new action sets are added to the input file, their indices are generated automatically, and the extractor functions are automatically placed in the correct order, as long as the first action of each set matches the suffix of the corresponding extractor function. Thus, the maintainer does not need to be concerned with the order of function declarations or lines in the input file.

3.2 Preprocessing and Action Detection

```
sent_text = "Teach me how to wash my hands"
sent = interp.preprocess_text(sent_text)
print sent
```

```
['Teach', 'me', 'how', 'to', 'wash', 'my', 'hands']
```

The above code simply preprocesses a raw text command, which currently only involves tokenizing it.

```
action_tuple = interp.extract_action(sent, known_actions)
print action_tuple
```

The above code searches through each token (from first to last) to see if any of the words match an action in the known actions list built in the *previous section*.

```
(5, 0)
```

The output of this search is a tuple containing the found action's set index along with its position in the sentence in the format (*action_set_index*, *position_in_sent*). The action detected in the given sentence *Show me how to wash my hands* is *show*, which is part of the action set with index 5, grouped with the action *teach*. Also, its position in the sentence is the first word, so its position index is 0. This function would yield the same exact output if it was given the sentence *Teach me how to wash my hands*.

3.3 Extracting Objects from the Command

```
object_dict = interp.generate_object_dict(sent, action_tuple, object_extractor_functions)

# Print dictionary values in same order every time
```

```
print "person: " + object_dict["person"]
print "show_action: " + object_dict["show_action"]
print "object: " + object_dict["object"]
print "video_title: " + object_dict["video_title"]
```

The above code extracts the objects from the command. The action set index determines which object extractor function will be applied and thus what set of rules are going to be used to extract the objects. Since the action is *teach*, the first action in its action set is *show*, and thus the appropriate object extractor function for this command is *`object_dict_show()`*, which is called by indexing the `object_extractor_functions` list inside of the *`generate_object_dict()`* function.

```
person: me
show_action: wash
object: hands
video_title: wash-hands
```

The above output simply displays the values of the objects along with their keys as stored in the resulting *object dictionary*, which is stored as a Python *dict*. The resulting object dictionary can be sent to another software component as a JSON string or a Python *dict* and the dictionary's content and semantic labels can used to determine which tasks LILI must complete.

See the *`extractor`* module for a full list of object extractor functions and the rules they utilize.

CHAPTER

FOUR

EXTRACTOR MODULE

interpreter.extractor.**is_noun**(*tag*)

Checks if a part of speech tag represents a noun. The tags "PRP" and those that begin with "NN" are considered nouns.

Parameters **tag** (*str*) – The part of speech tag to be checked

Returns True if the tag represents a noun, False if not

Return type bool

interpreter.extractor.**is_preposition**(*tag*)

Checks if a part of speech tag represents a preposition. The tags "TO" and "IN" considered to be prepositions.

Parameters **tag** (*str*) – The part of speech tag to be checked

Returns True if the tag represents a preposition, False if not

Return type bool

interpreter.extractor.**object_dict_follow**(*sent*)

Extracts objects out of a sentence that contains "follow" as its main action

Rules: 1. The first noun encountered is always the 'person' 2. The second noun (if included) is always the 'place'

Objects: 'person' - The person who will be followed 'place' - The location that the person will be followed to

Parameters **sent** (*list*) – A part of speech tagged list of tokens representing a sentence

Returns A dictionary mapping key words to semantic object categories

Return type dict

interpreter.extractor.**object_dict_move**(*sent*)

Extracts objects out of a sentence that contains "move" as its main action

Rules: 1. Only the first noun is detected as an object 1. If the first noun is preced by a preposition, then it is the 'place' 2. If the first noun is not preceded by a preposition, then it is the 'direction'

Objects: 'place' - A location to move to 'direction' - A direction to move in

Parameters **sent** (*list*) – A part of speech tagged list of tokens representing a sentence

Returns A dictionary mapping key words to semantic objects

Return type dict

interpreter.extractor.**object_dict_show**(*sent*)

Extracts objects out of a sentence that contains "show" as its main action

Rules: 1. The verb preceded by a "to" is the 'shown_action' 2. The noun that is not preceded by a determiner or "to" is the 'person' 3. The noun that is preceded either by a determiner or the 'shown_action' is the 'object'

Objects: 'shown_action' - The action that will be shown in a video 'person' - The person who will be shown the action or object 'object' - The object that is acted on in the video or a static object to be shown as a picture 'video_title' - The title of the video to be played - it is currently generated by concatentating the 'shown_action' with the 'object'

Parameters **`sent`** (*list*) – A part of speech tagged list of tokens representing a sentence

Returns A dictionary mapping key words to semantic objects

Return type dict

`interpreter.extractor.`**`object_dict_stop`**(*sent*)

Extracts objects out of a sentence that contains "stop" as its main action. Currently returns an empty dictionary.

Parameters **`sent`** (*list*) – A part of speech tagged list of tokens representing a sentence

Returns A dictionary mapping key words to semantic objects - currently returns an empty dictionary under all inputs

Return type dict

`interpreter.extractor.`**`object_dict_talk`**(*sent*)

Extracts objects out of a sentence that contains "talk" as its main action

Rules: 1. The noun to come after the word "about" is the 'topic' 2. The noun to come after any preposition that is not "about" is the 'person' 3. Other nouns will be tagged as 'unknown'

Objects: 'person' - The person to talk to 'topic' - The subject to talk about 'unknown' - The role of this noun is not known

Parameters **`sent`** (*list*) – A part of speech tagged list of tokens representing a sentence

Returns A dictionary mapping key words to semantic objects

Return type dict

`interpreter.extractor.`**`object_dict_turn`**(*sent*)

Extracts objects out of a sentence that contains "turn" as its main action. Currently uses the same rules as object_dict_move - see that method for more details

Parameters **`sent`** (*list*) – A part of speech tagged list of tokens representing a sentence

Returns A dictionary mapping key words to semantic object categories

Return type dict

CHAPTER

FIVE

TERMINOLOGY

action A verb that corresponds to a single task that LILI can complete. For example, the action *follow*, as in the command *"Follow me to the kitchen"*, tells LILI to follow someone.

action set A group of *actions* that have the same meaning to the interpreter. For example, the actions *turn*, *twist*, and *rotate* comprise an action set. These three actions can be used interchangeably in the same command. The commands *"Turn right"*, *"Twist right"*, and *"Rotate right"* will all be interpreted as the same command, all yielding the same output from LILI.

action set index A positive (or zero) integer value that uniquely identifies an *action set*.

object A key word that must be interpreted in order for LILI to perform the correct task. In the command *"Follow me"*, *follow* is the *action* and *me* is an object because LILI must know who to follow in order to do that task as the user intends.

object dictionary A dictionary, a set of key-value pairs, that maps a set of *objects* to semantic labels that correspond to the context and meaning of each object. In the command *"Follow me"*, *me* is stored as a value in an object dictionary whose key is *person*.

object extractor function A function that uses predetermined rules to extract *objects* from commands, tags the objects with semantic labels, and generates an *object dictionary*.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX