

Integration of WordNet into a Topic Selection Routine

Jonathan Riehl

University of Chicago

CMSC 35900- 1: Topics in Artificial Intelligence

Introduction to Discourse and Dialogue

1. Introduction

WordNet is a lexical database of the English language that seeks to capture the relationships between words as a graph data structure [8]. In the WordNet graph, vertices represent word groupings of synonyms, while edges are labeled relations between either two specific words in a grouping or two synonym groupings. The richness of these relationships allow WordNet to encode information about the English language that is lexical (such as word morphology), syntactic (such as part of speech), and semantic (such as antonym groupings.) The amount of effort required to capture this much information has typically precluded its incorporation into conversational agents. However, with the public availability of WordNet, the WordNet project has assumed the cost of data entry, allowing other projects to freely benefit from its extensive information on the English language. This paper will investigate some of the ways the WordNet database may be integrated into conversational agents, and focus on one particular use of WordNet to extend an existing conversational agent.

Conversational agents are programs designed to engage humans in natural language conversations. One subgroup of conversational agents are chatter bots, which are so called because the agents typically lack any intentionality and have the primary goal of engaging users in superficial conversation as a form of amusement. Originating with Weizenbaum's Eliza [2], chatter bots have traditionally sought clever tricks to get the user to suspend disbelief without much investment in actual lexical, syntactic and semantic understanding of the user's input. Eliza serves to illustrate just how little infrastructure is required to accomplish the goal of parodying a Rogerian therapist. Eliza employed two primary tricks: a mapping from various pronouns to other pronouns (such as 'I' being transformed to 'you') that allowed the program to rephrase the user's statements as questions, and keyword activated stock phrases that gave the illusion of understanding.

These two tricks persist in modern chatter bot designs. The Loebner prize [3] winning ALICE program [1] uses an activation network that mirrors the keyword to canned response component of Eliza. ALICE employs keyword based patterns as the principle source of excitation, which are coupled with an internal state and used to determine state transitions in the bot's activation network. However, these bots require the hand generation of the semantic networks they use. This limits the perceived intelligence of the bot to reflect the amount of human effort that went into the design of

the bot. Ultimately it is impossible for a human designer to anticipate the full course of an arbitrary conversation. Even when a human user sticks to a topic that is encoded in the activation network, it is not uncommon for the bot to repeat itself after a few exchanges.

Other programs attempt to appear human by repeating previously observed inputs. These programs often avoid the overhead of hand coded data and parroting procedures by employing linguistic learning techniques such as Markov chains [7]. These bots learn a language model from user input and use this model to generate responses to user input. However, the language models employed by these designs very rarely encode more information than statistical relationships between lexical tokens. Correspondingly, these bots typically show little to no understanding of user input, and will often fail to generate syntactically correct output. A commonly used program that employs a lexical Markov chain is the MegaHAL chatter bot [5]. MegaHAL builds a forward and backward chaining Markov model of user input that has been tokenized into words. MegaHAL attempts to give the illusion of responsiveness by selecting a keyword from the previous input and using the keyword to generate a full utterance by simultaneously chaining forward and backward from the selected keyword.

The WordNet database can be used to benefit chatter bots in multiple ways. There are lexical benefits in that WordNet could be used to seed the bot's initial lexicon. Many of WordNet's string entries contain multiple words, and using these as primitive tokens would allow Markov chain based bots to encode longer and more meaningful sequences of tokens. WordNet includes a wealth of example utterances in its dictionary. These utterances could be added to a training corpus, giving a bot a much richer initial model of language. Finally, there are numerous semantic benefits. For example, by traversal of the synonym relationships, bots would be better able to resolve the meaning of an observed utterance and relate the utterance to other observed utterances.

This paper will now focus on how synonym relationships in the WordNet database were used in an extension of the MegaSAL bot implementation. Section 2 gives an overview of the MegaSAL design and implementation, comparing and contrasting MegaSAL with the bot it was modeled after, MegaHAL. Section 3 presents how WordNet was used to enhance the topic selection module in MegaSAL. Section 4 illustrates how the WordNet based topic tracking module performs in comparison to the previous topic tracker and a completely naive keyword selection routine. This paper concludes by speculating on the further WordNet based improvements that may be made to the topic tracker and the MegaSAL components.

2. MegaSAL, Sister to MegaHAL

MegaSAL is a chatter bot based on the high level design of MegaHAL and implemented in the Python programming language [6] as a part of the IBot class framework (illustrated in Figure 1.) As previously described in Section 1, MegaHAL uses a forward and backward chaining Markov model of observed language to generate its utterances. MegaSAL retains this action for output generation, but attempts to distribute various elements of this design into a more modular and object oriented framework. This section begins with a description of the bot abstraction realized by the DefaultBot class, and specialized by the MegaSALBot class. Later subsections detail the associative array component and topic tracking components, both used by MegaSAL for output generation.

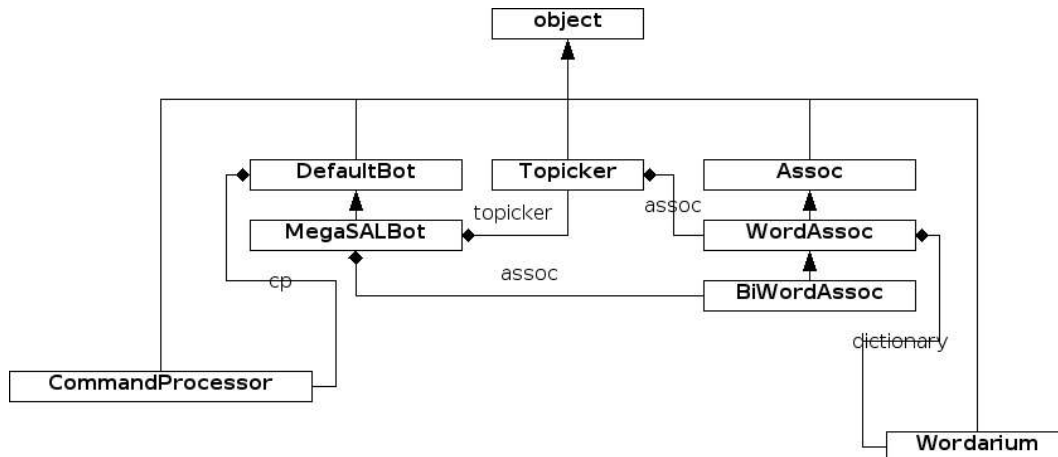


Figure 1: Overview of the IBot Class Hierarchy

2.1. The Bot Class Hierarchy

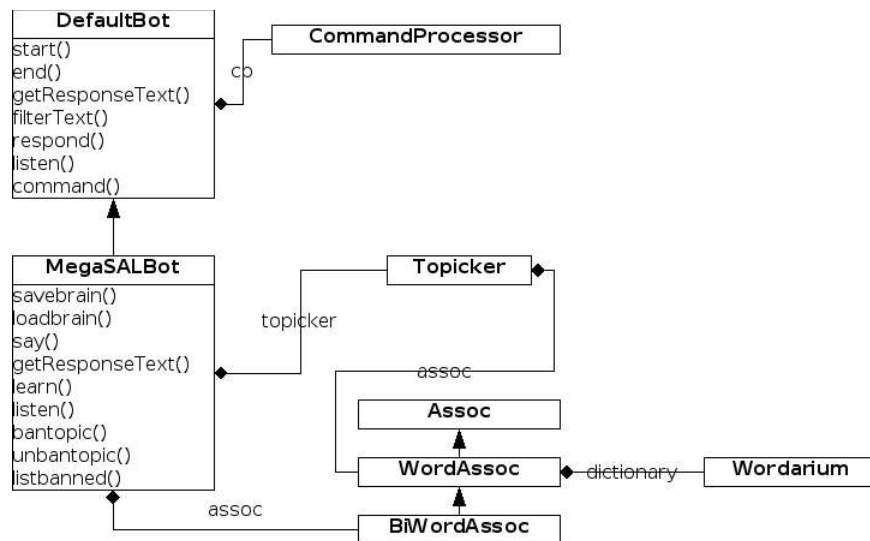


Figure 2: The DefaultBot and MegaSALBot Classes

The bot class hierarchy (shown in Figure 2) is designed to provide an abstraction of the functionality required in a conversational agent. The base class of this hierarchy is the DefaultBot class, which defines the basic set of operations used by the IBot framework to interact with the program's conversational component. The remaining IBot framework has been elided from both illustration and this paper, and deals with the

networking mechanics of bot interaction over Internet chat protocols. While other bot implementations inherit from the DefaultBot class, the second part of this section will focus on the class that implements the MegaSAL conversational agent, MegaSALBot.

2.1.1.The DefaultBot Class

In IBot terminology, the DefaultBot class and its subclasses define the bot persona. In implementation terms, this means that these classes are responsible for generating responses to utterances and other observations made while in a conversation. The basic interface of the DefaultBot class is composed of three operations: respond(), listen(), and command(). The listen() operation is used to input messages that are not observed, but not directed to the bot specifically. The command() operation is used to send administrative messages to the specific persona. The most important operation is the respond() operation which is responsible for responding to natural language input directed specifically to the bot. Currently, the default action of the respond() operation is to defer response generation to the getResponseText() operation, while providing a hook for the bot operator to choke bot output (which is often required when two chatter bots start talking to each other.) The DefaultBot is meant to only be a base class, and the base class implementation of getResponseText() simply repeats the text sent to it.

2.1.2.The MegaSALBot Class

The MegaSALBot class overrides the default respond() and getResponseText() operations. The listen() operation allows the MegaSAL persona to acquire new utterances and update its language model accordingly without requiring the bot to be directly addressed. The getResponseText() operation works through a series of subordinate operations. First, if the bot is configured to learn from the speaker, the persona updates its language model. Then, the persona selects a topic by asking its topic tracking and selection component to determine a seed keyword based on the topic tracker's current state and the input text (where the topic tracker is referenced by the topic attribute of the persona object). The persona will finally generate a reply by passing the topic keyword to its language model (as referenced by the assoc attribute of the persona object). The language model returns an utterance built by forward and backward chaining from the topic keyword. Below, the language modeling component and the topic tracking and selection component are both described in greater detail.

2.2.The Assoc Class Hierarchy

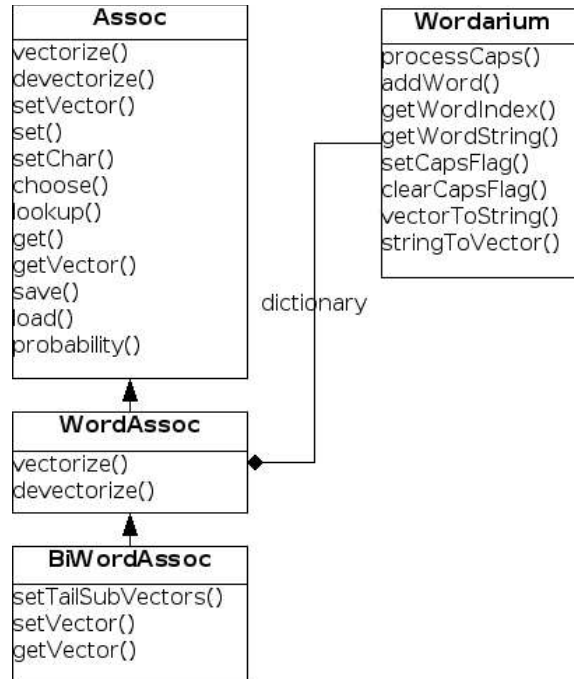


Figure 3: The Assoc Class Hierarchy

The Assoc class hierarchy (shown in Figure 3) implements a set of increasingly specialized associative arrays. The MegaSAL persona uses the most specific of these classes, BiWordAssoc, as its language model. This section describes the functionality of each of the Assoc classes, starting with the most general Assoc class and ending with the most specialized class, BiWordAssoc. Additionally, a later subsection describes the Wordarium utility class, which is required by the WordAssoc class and its subclass(es).

2.2.1. The Assoc Class

The Assoc class implements a fixed depth associative array using the Python dictionary data type. The Python dictionary is used to map from an observed token to a data pair consisting of an observation count, and a nested associative array of depth equal to the current depth less one. Associative arrays of depth zero are represented by the empty dictionary. One benefit that is passed onto subclasses of the Assoc class is that the Python dictionary data structure allows associative arrays to capture statistics on sequences of arbitrary immutable Python objects.

To pass the flexibility of Python dictionary keys to subclasses, Assoc only requires subclasses to override two key class operations: vectorize() and devectorize(). The first operation transforms a string into a sequence of objects that can be encoded in the associative array. In this encoding scheme, the Python None object delimits the start and end of observed and generated vectors. By default, the Assoc class' vectorize() simply converts the input string into a sequence of characters, prefixed and terminated by the Python None object. The devectorize() operation is used to take vectors generated using the associative array and return strings, forming an inverse operation to the vectorize() method.

Once a string is translated to a vector, the setVector() operation updates the model. Conversely, the getVector() operation returns a random vector based on a weighted

traversal of the associative array. Bots that use the Assoc class or its subclasses update the language model by vectorizing string input and then calling setVector() (this is done by the set() operation.) Bots generate responses by feeding a seed object (which should be encoded in the array) to the get() operation, which then calls getVector() and devectorize() to return a string.

2.2.2.The WordAssoc Class

The WordAssoc class specializes the Assoc class by keeping a lexicon of observed words (and punctuation) and using indices into the lexicon as the observation objects. The WordAssoc class overrides the vectorize() operation by delegating tokenization of the string to the lexicon (implemented by the Wordarium class, described below.) WordAssoc also overrides the inverse operation, devectorize(), using the lexicon to translate a vector of indices into a string.

2.2.3.The BiWordAssoc Class

The BiWordAssoc class specializes the WordAssoc class by adding a backward chaining associative array. BiWordAssoc overrides the setVector() operation to not only update the forward chaining array, inherited from Assoc, but also trains the backward chaining array using the reverse input vector. The setTailSubVectors() method implements some additional machinery that is required to maintain the condition that any key in the top level forward array is also a key in the backward array. This prevents short utterances from not being fully encoded in the backward chaining array. Finally, BiWordAssoc extends the getVector() operation to backward chain from an input seed (if given) and prepend the result to the forward chaining traversal result.

2.2.4.The Wordarium Class

The Wordarium class implements a lexicon of observed words where a word is defined to be either punctuation, whitespace, or a series of characters that are neither punctuation or whitespace. WordAssoc instances reference Wordarium objects, which they use to enumerate and keep a lexicon of unique words. One of the differences between word segmentation in MegaHAL and MegaSAL is that MegaSAL encodes word case using a flag bit that is added to the word index. MegaHAL discards capitalization and recomputes capitalization (often incorrectly) based on perceived sentence boundaries. This difference means that MegaSAL encodes the words `the' and `The' as different entities whereas MegaHAL does not.

2.3.The Topicker Component

One of the major design differences between MegaHAL and MegaSAL involves how the two bots chose keywords for response generation. MegaHAL simply choses a random word from the input text with banned, uninteresting and nonsense words removed. Topic selection is indirect by virtue of how MegaHAL relies on its fast associative array implementation to generate multiple responses to the same input entry, constrained by some time limit. From the list of possible responses, MegaHAL selects the reply with the lowest probability of generation (meaning greatest information content according to its language model) [4]. This indirectly selects for keywords that have a lower probability.

In contrast, MegaSAL attempts to track the conversation topic and relate observed

input to the topics it is tracking. The Topicker class is the first implementation of a topic tracking system for MegaSAL and operates by keeping a cache of past seed keywords. When prompted for a seed, the Topicker object favors keywords from the input text that appear in the cache. Perhaps naively, the Topicker also favors words it has seen more frequently under the assumption that these are more useful seeds from an algorithmic perspective (having more in and out edges in the associative arrays, the bot would have more varied responses to these keywords). Under the information theoretic definition of information content being equal to the inverse log probability of an object, MegaSAL may be damning itself to less interesting or overly general responses. Conversely, MegaHAL's preference for rarity causes its syntax usage to degenerate rather easily in the presence of imperfect speakers.

Topicker class instances are cued to generate a topic via the `getTopic()` method, which returns a lexical index indicating the Topicker's guess at the most topical keyword in the input vector. The `getTopic()` operation first removes uninteresting and banned words from the input vector. Each word is then assigned a probability mass initially equal to its unigram probability. In the event that a word appears in the topic cache, the probability mass is incremented. The word with the highest probability mass is selected, and the cache is updated using a least recently used strategy. When the topic tracker can not determine a keyword from the input, it falls back to a random word in its cache, or when the cache is empty, a random word from the top level associative array. Topicker instances keep a reference to a `WordAssoc` instance (which is the super class of the `BiWordAssoc` class, used in MegaSAL) for the purpose of both looking up word length (via the lexicon associated with the language model) as well as calculating the unigram probability of a word.

3. Migration from Words to Semantic Categories

Section 2 described the existing MegaSAL design and implementation. This section will highlight the theory, design and implementation of an extension to the topic tracking component. In this extension, the WordNet database is exposed via a new Python module to a specialization of the Topicker class, the `SensePicker`. The `SensePicker`, shown in Figure 4, attempts to move away from tracking topic as a set of prior keywords. Instead of words, `SensePicker` tries to chose a word sense from the last observation, preferring word senses that relate to its cache of past sense selections, and use that sense to get a seed keyword. The following subsections detail the reasoning, design and details of both the WordNet module and the `SensePicker` class.

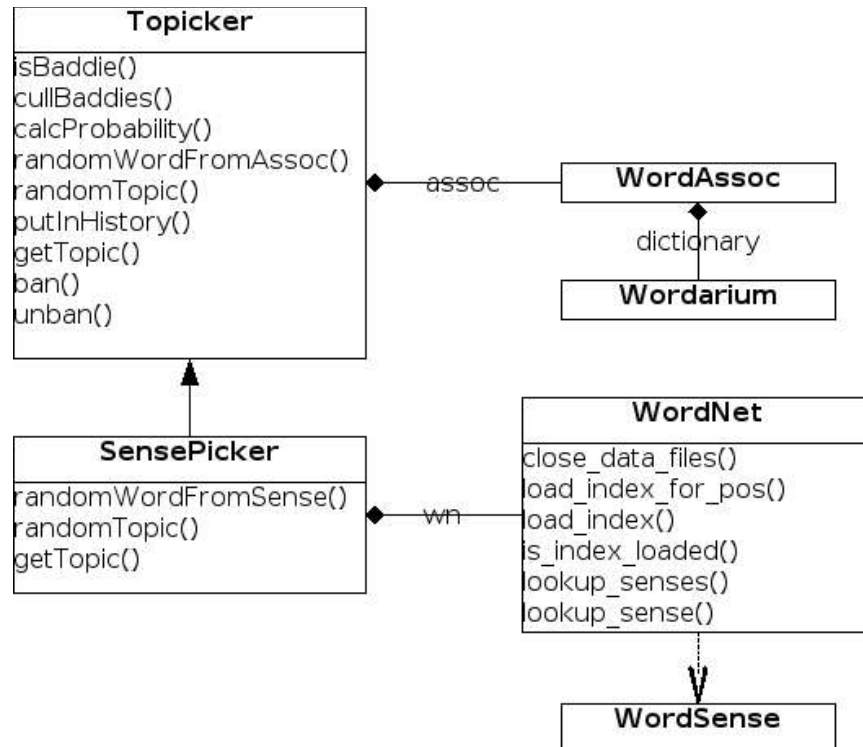


Figure 4: The Topicker Class Hierarchy

3.1. The WordNet Module

The WordNet database and graph data structure is exposed to users via two data sets. The first data set is an index that serves the role of a lexicon. At the most basic level, the index is a one to many map from words to synonym groupings (termed synsets in the WordNet literature), and is further partitioned according to part of speech. The second data set contains information specific to each of the synsets, including all the relationships between the synsets. The WordNet module retains this division by exposing two classes: the WordNet class and the WordSense class.

3.1.1. The WordNet Class

Due to the overhead of processing and keeping a copy of the WordNet index in memory, the WordNet class makes several design decisions. First, users of the WordNet class are encouraged to use a WordNet singleton object, accessed via the class method, `get_instance()`. Second, the index file is not loaded when the class is first instantiated keeping construction and loading overhead decoupled. Finally, the index may be only partially loaded for a given part of speech using the `load_index_for_pos()` method. This last measure allows users to only load the parts of the index they may need, avoiding the overhead of loading all four index files in a single pass.

Words in the index may be accessed directly from a WordNet instance's `index` attribute, or via the `lookup_senses()` operation. The `lookup_senses()` operation returns a list of database coordinates for each synset referenced for the passed string. Database coordinates are coded as a pair. The first element in the coordinate pair is an absolute file position, and the second element is a part of speech tag. These uniquely identify each synset, and form the input type to the `lookup_sense()` operation.

3.1.2. The WordSense Class

Instances of the WordSense class are returned by the lookup_sense() method of the WordNet class. The WordSense class contains the data fields for a given synset as present in the data files. These fields include edges of the various synset relationships, a list of words in the synset, and the gloss text. Instances of the WordSense class are meant to be throw away objects and are not cached by the lookup_sense() operation since lookup time via a file seek is low. Notably, this may not be the case when large subgraphs need to be in memory at the same time.

3.2. The SensePicker Class

The SensePicker can be seen as a hybrid of the existing word based topic tracker and a more sophisticated (and completely unimplemented) system that unifies the WordNet index and bot lexicons, and moves synonym sets out into the language model. This hybridization is accomplished by translating a list of lexicon indices coupled with a LRU cache of previously used word senses into a probability distribution over word senses, selecting a random sense using the generated probability distribution, and finally selecting a random word from the sense, biased by unigram probability. This algorithm is used to override the existing getTopic() operation of the Topicper base class.

The word sense probability distribution is built by creating a map from WordSense instances to a probability mass. A synset's probability mass is determined by the following formula:

$$\begin{aligned} U &= \text{utterance vector, } C = \text{synset cache} \\ n(\text{synset}) &= \sum I(\text{synset} \in \text{senses}(U_i)) \\ m(\text{synset}) &= \sum I(\text{synset} = C_j) \\ \text{Pr}_{\text{mass}}(\text{synset}) &= 2^{n(\text{synset}) + m(\text{synset}) - 1}, \text{ if } n(\text{synset}) > 0 \\ \text{Pr}_{\text{mass}}(\text{synset}) &= 0, \text{ otherwise.} \end{aligned}$$

The sum of probability masses is used to bound a random variable and the result of the variable is in turn used to resolve upon a word sense. The SensePicker then defines a second probability distribution over the words that map into the chosen synset, and selects a word from that distribution. If the SensePicker is unable to find any words in the synset that are in the lexicon and not banned, then it selects a random sense from its cache, and picks a random word from the cached synset. A word sense is only added to or updated in the cache if the SensePicker was able to find a word in the synset. that was a valid seed keyword.

4. Analysis

This section provides a comparative analysis of the SensePicker component with both its predecessor, Topicper, and the MegaHAL bot. The analysis is based upon several test runs of the three components (each run used the same initial conditions and data.) The goal of each test was to provide some measure of topicality while retaining a basis in words, allowing the test metric to apply to the keyword selection based components. Section 4.1 describes the structure of the test data, test apparatus and the test metric, while Section 4.2 gives a summary of test outcomes with some brief analysis.

4.1. Test Setup

A tester input the test data in the form of a conversation the tester had with himself. The conversation amounted to a total of twelve utterances. For each utterance, the tester subjectively ranked each unique word in that utterance in terms of its importance to the utterance's topic. In cases where the tester deemed a word unrelated to the topic, the word was ranked based on how interesting the tester found that word to be with respect to the other words in that utterance. The experiment used the utterance text as the input to the topic selection routine of each component under test. The experiment then used the word ranking to define a metric, and applied the metric to the component's choice.

The test apparatus used three topic selection components. The first component was an instrumented MegaHAL. The tester modified the MegaHAL code to record the keyword used for the response that had the maximum surprise measure, and had the MegaHAL component return that keyword instead of its normal response. The second and third components were fragments of the MegaSALBot, one using a Topicker instance, and the other using a SensePicker instance for topic selection. Both the Topicker and SensePicker had separate language models, but each model was of the same level of associativity. For each test, each component used the exact same training data for its language model and the same keyword ban list. For each test utterance, the apparatus allowed the component to update its language model from the utterance before selecting a keyword for response generation. Finally, the apparatus recorded the result of the test metric as applied to the component's keyword selection.

The test metric is defined by the following equation, where R is the keyword ranking, and the senses function is a map from a string to the set of synsets that WordNet maps that word to:

$$\begin{aligned} \text{testMetric}(kw) &= 1.0 - (i / |R|), \text{ where } kw = R_i \\ \text{testMetric}(kw) &= 1.0 - (i / |R|), \text{ where } \text{senses}(kw) \cap \text{senses}(R_i) \neq \emptyset \\ \text{testMetric}(kw) &= 0.0, \text{ otherwise} \end{aligned}$$

The equation is meant to measure how interesting the tester would have found the component's keyword selection to be. The first condition is used when there was an exact string match between the keyword chosen and a keyword in the ranking. The second condition is based on the intersection of the input keyword's senses and a ranked keyword's senses being non-empty. The metric evaluates to zero when no trivial relationship can be drawn between the selection and any word in the current ranking.

4.2. Test Results

The following table gives the best test outcome, the worst test outcome and the average test outcome for all three units over a series of ten test runs. Note that when run from identical initial conditions, the Topicker's algorithm is deterministic with respect to keyword selection from the current utterance, but not from the list of prior topics (which this metric fails to measure, as discussed below.)

<i>Bot/Topic Selector</i>	<i>Best</i>	<i>Worst</i>	<i>Average</i>
MegaHAL	7.97	7.17	7.64
MegaSAL Topicker	7.87	7.87	7.87
MegaSAL SensePicker	10.14	8.02	9.44
Best possible = 12, Worst possible = 0			

While ten test runs are hardly enough to provide statistically significant bounds on the results, the outcome range seems tight enough to make some generalizations. First, it appears that the addition of a topic state to MegaSAL has little to no impact on performance in comparison to MegaHAL. However, it should also be noted that the test metric only measures keyword interest with respect to the last utterance. This metric fails to reward the possibly interesting result of the Topicker and SensePicker attempting to return to a previous utterance's keyword selection (and ideally topic.) Second, the SensePicker appears to outperform both the Topicker and MegaHAL routines on the test metric. Third, there is an apparent increase of outcome range (possibly impacting statistical variance) over the MegaHAL results, which implies that the SensePicker might not provide as consistent a user experience as MegaHAL.

5. Conclusions

Based on the metrics used for the project, it appears that there is a benefit to using semantic networks for topic selection. While no qualitative data is available yet, it is hoped that by using information from WordNet, the MegaSAL bot will provide a user experience that improves the illusion of topicality, cohesion among sets of generated utterances, and natural language understanding. Certainly, if prior word rankings were factored into the test metric, both the Topicker and SensePicker components would only stand to improve their scores with respect to the MegaHAL design. Furthermore, the apparent improvement between the Topicker and SensePicker components, implies that user experience with generative chatter bots could be further enhanced by moving the WordNet data out into the language model.

6. Works Cited

- [1]A.L.I.C.E. AI Foundation. <http://www.alicebot.org/>
- [2]`ELIZA.' From Wikipedia. <http://en.wikipedia.org/wiki/ELIZA>
- [3]Hutchens, Jason. `How Megahal Works.'
<http://megahal.aliioth.debian.org/How.html>
- [4]Loebner, Hugh. `Loebner Prize Home Page.'
<http://www.loebner.net/Prizef/loebner-prize.html>
- [5]MegaHAL. <http://megahal.aliioth.debian.org/>
- [6]Python Language Website. <http://www.python.org/>
- [7]Weisstein, Eric W. `Markov Chain.' From *MathWorld*- - A Wolfram Web Resource. <http://mathworld.wolfram.com/MarkovChain.html>
- [8]WordNet. <http://www.cogsci.princeton.edu/~wn/>