OpenWIMs Implementation

Jesse English • Benjamin Johnson • Benjamin Bengfort
http://www.openwims.org • openwims@gmail.com

definition: WIMs

"Weakly Inferred Meanings", a frame-based meaning representation derived from lightweight semantic analysis

Introduction

The OpenWIMs system is semantic text analysis engine that lives on a syntactic-semantic lexicon and expects as input a set of syntactic dependencies for the target text. OpenWIMs is an open-source, standards compliant implementation of a WIMs analyzer - the input, output, macrotheories and microtheories used comply to those defined by OpenWIMs.org.

This document will discuss at a high level the implementation of the OpenWIMs analyzer and its supporting lexicon and ontology.

Implementation

OpenWIMs is currently implemented in Java and conforms to the **lexical-constraints** macrotheory, with a deprecated implementation in Python. This document will discuss the Java implementation as the Python version needs to be rewritten to accommodate the current model, theory and knowledge.

The expected input of the system is an Annotation object, the primary output of Stanford's CoreNLP system. The annotation must have, at the minimum, tokenization, lemmatization, part of speech tagging and dependency graphs. Due to the nature of Stanford's implementation, having dependencies necessarily requires many other annotations to be found, which may be used by microtheories if available.

OpenWIMs utilizes a syntactic-semantic lexicon, anchored in an ontology, to disambiguate word senses and produce meaning representations. This is done in three (simplified) steps:

1. Every token is inspected independently - any tokens that contain roots in the lexicon will generate a frame. Words without roots in the lexicon have either (1) not been acquired or (2) do not have root status in our lexicon (such as most prepositions, do not have lexicon entries, but instead are used optionally in other lexical senses to modify meaning)
2. For each token with lexical senses, a single sense is selected by filtering each possible structure, resulting in a small set of possible (ambiguous) matches

    a. Filtering is done by inspecting the input dependencies with those defined in the lexical sense - any mismatch kicks the sense out of contention
    b. In the case of ties, any number of microtheories can be used to settle on a single sense; alternatively, depending on the *attitude* of the WIM, multiple structural matches may be returned
3. The selected structural match has an associated meaning representation - that meaning is then mapped into the word and any words linked to it by the structure, resulting in a semantic network represented by frames and relations

Lexicon

The OpenWIMs lexicon, originally derived from WordNet as a starting point for core lexical knowledge, is now manually maintained by the community. Below is a truncated description of the format of a lexical sense, and how it is used in processing.

Sense Name

Lexical senses names carry meaning, as well as organizational properties. Names are structured to represent the core ontological concept that the sense derives from, the root word the sense derives from, the part of speech the sense represents, and finally an index to avoid namespace collisions. As an example, the verb "hit", meaning *to strike*, might look like: **@strike:hit-v-1**.

In the OpenWIMs notation, ontological concepts are prefaced with an @; this sense's name should be interpreted as "meaning @strike, the verb whose root is 'hit'". When a sense is selected by the analyzer, the frame it generates will be an instance of the concept heading the sense's name.

Structures

Lexical senses contain any number of structures - these are syntactic patterns that represent the usage of the sense. Each structure itself contains a collection of dependency sets; only one structure need match the text for the sense to be a match. It is sufficient to say that a sense has *only one meaning* but may have *many different structural usages*.

Structures contain a collection of dependency sets; these sets are headed by a label (this label has no meaning and exists for human consumption only). Sets are, by default, optional to the structure. It can be specified that a set be mandatory (as in, the structure cannot be a match to a text if the set is not found), but this is the exception, not the rule. A sample set that details the usage of the preposition "with", meaning *using a tool to accomplish* is shown below:

```
+WITH-INSTRUMENT
prep(SELF, with{token='with'})
pobj(with, instrument{pos='NN', ont='@physical_entity'})
[SELF.instrument=instrument]
```

In this example, we can see the label of the set, **+WITH-INSTRUMENT**, and should be understood as "this sense can also include the dependency set based on the preposition 'with', meaning *using a tool to accomplish*".

Sets consist of any number of dependency maps, and any number of applied meanings. Dependency maps are used to confirm the use of a sense (by validating the structure's constraints against the input text) and meanings are used to produce the final WIMs output.

Dependency Maps

Dependency maps consist of four distinct elements:

1. The type - e.g., `nsubj`
2. The governor - a variable, or the keyword `SELF` - meaning the token that is being disambiguated currently
3. The dependent - a variable, forming the set of valid values that a governor can take, including `SELF`
4. Selectional constraints - a set of strict limitations applied to the dependent that must be adhered to for the structure to match:
    a. `token` - the token must be the word provided
    b. `pos` - the token must be interpreted in this context as the part of speech provided
    c. `ont` - the token must have at least one valid meaning (amongst all its senses) that is a match (e.g., if a constraint demands `ont=@human`, then **@worker:postman-n-1** will match as @worker is-a @human, but **@feline:cat-n-1** will not match)
    d. `micro` - the supplied microtheory (annotated with a `#`, as in `#date`) must match

Continuing with the example above, we see two dependencies, both of which must exist for the set to match.

```
+WITH-INSTRUMENT
prep(SELF, with{token='with'})
pobj(with, instrument{pos='NN', ont='@physical_entity'})
[SELF.instrument=instrument]
```

The first dependency expects a `prep`, where the governor is denoted as `SELF` and the dependent as `with`. Dependents are *always* variables - the usage of `with` here is a

convenience, and does not in and of itself imply anything about the token, only that whatever token matches will now be *referred to internally as* `with`.

This variable has, unsurprisingly, been chosen to reflect the actual token that should be mapped. Dependents can have any number of selectional constraints applied to them - these take several forms, in the case of `with`, the only constraint applied is that the actual token be the word "with".

This first dependency can be read as "a prepositional dependency, whose governor is the token in question, and whose dependent is the word 'with'".

```
+WITH-INSTRUMENT
prep(SELF, with{token='with'})
pobj(with, instrument{pos='NN', ont='@physical_entity'})
[SELF.instrument=instrument]
```

The second dependency expects a `pobj`, with the governor denoted as `with` - meaning the variable that was mapped in the first dependency. The dependent is a new variable, named `instrument` and containing two constraints: the first is that the token's part of speech be a noun (NN representing the set of all nouns) and the second being that the token have at least one interpretable meaning as a @physical_entity.

This dependency can be read as "a prepositional object whose governor is the token previously mapped to the variable `with`, and whose dependent is a noun that must have at least one possible meaning interpretation of @physical_entity".

Applied Meanings

Applied meanings contain three elements:

1. The domain that will have the meaning applied to - this must be a valid variable defined (and mapped) in the dependencies, or can be the keyword `SELF`
2. The property - this is the property on the domain that will be assigned a value
3. The range - this is the filler of the property on the domain, and must be either a valid variable defined in the dependencies, the keyword `SELF`, or a literal value

```
+WITH-INSTRUMENT
prep(SELF, with{token='with'})
pobj(with, instrument{pos='NN', ont='@physical_entity'})
[SELF.instrument=instrument]
```

Continuing with the above example, the last line defines the meaning this dependency set contributes to the sense, if the set is a match. The left hand side assigns to a frame (represented by the token's variable that it derives from) a value to its property (represented by

the dot notation) being the value of the variable found on the right hand side.

In this case, the meaning can be read as “assign to the frame derived from this sense a value to the property *instrument* being the token that has been mapped to the variable `instrument`”.

Ontology

The OpenWIMs ontology, originally derived from WordNet as a starting point for core ontological knowledge, is now manually maintained by the community.

Currently, the ontology serves only as a hierarchy to map lexical senses to and to use to disambiguate via the ont selectional constraint. In the future, it is the intent that the OpenWIMs ontology grow to include acquired knowledge of concept interactions (which can then be leveraged to make more detailed selectional constraints).