Identifying the Least Common Subsumer Ontological Class

Ken Litkowski
CL Research
9208 Gue Road
Damascus, MD 20872 USA
ken@clres.com

Abstract

A common problem in computational linguistics is providing a semantic characterization of a set of lexical items. For the most part, research has focused on examining the semantic similarity of two items. When there are more than two items, it is not clear how to proceed. One approach might be to perform pairwise analysis on the set. Unfortunately, the computational complexity of this approach would quickly explode. We take a different approach. Semantic similarity is frequently computed within the context of a semantic hierarchy (usually WordNet), finding the node which is the least common subsumer. This approach involves identifying he path of each item to the root and finding where the two paths intersect. We build on this approach, noting first that a given set of lemmas activates only a subtree of a hierarchy. We describe an algorithm that identifies all the active paths and collapses the paths, finding stopping nodes that constitute ontological classes. We pick the path that covers the most lemmas. We present examples that have face validity.

In this paper, we consider the possibility of using ontologies (taxonomies, hierarchies) for characterizing the semantics of a set of lexical items. In section 1, we describe an algorithm that accomplishes this in four steps (assembling the ontology nodes, creating a tree of activated nodes, annotating the nodes with stopping pints, and collapsing the paths). The algorithm takes advantage of the fact that a given set of lexical items will activate only a portion of an ontology. In section 2, we present an example analysis based on one sense of one preposition that was used for development of the algorithm and its initial assessment. In section 3, we consider the possibility of using the results of this ontological analysis for enhanced semantic word sketches.

1. **Ontology Analysis**

The purpose of ontology analysis is to identify the primary semantic classes of a group of lemmas that have been tagged with a hypernym from a lexical resource. These tags are used as features in classification modeling and are essentially immediate hypernyms of the lemmas in the lexical resource. The use of hypernyms for measuring semantic relatedness stems from Wu & Palmer (1994), extended in Leacock & Chodorow (1998) and Pedersen et al. (2004), and pursued in many studies based on this early work. Wu & Palmer established the basic notion of a least common subsumer to measure semantic relatedness. We extend this notion to look at classes, rather than individual lexical items.

1.1. **Motivation**

One method of assigning a semantic class is to use the WordNet lexicographer file name (see, for example, McCarthy et al., (2015)). This is used as a feature in our classification modeling (Litkowski, 2014), but is of a very coarse granularity. When examining the set of immediate hypernyms triggered by the lemmas associated with a set of instances, one quickly gets the impression that some of the hypernyms are natural subsets of others. One would like to collapse these hypernyms to obtain a smaller subset that cover the most instances. Another strategy that has been used, in addition to the WordNet file names, is to obtain all hypernyms triggered by the lemmas. This strategy, however, produces an even coarser granularity. Thus, every noun will be identified as an *entity*, with prominent subclasses of *physical_entity* or *abstract_entity*; every verb will be identified as a species of *move* or *act* or *change*.

1.2. General approach

The hierarchies in the resources are trees. Each node consists of a unique name and a set of lemmas that trigger this node. A given lemma may trigger more than one node (corresponding to any polysemy of the lemma). The lemmas for a given set of instances will thus trigger some subset of nodes in the full tree. This subset will almost always be a small proportion of the nodes in the full tree. What we do is to construct a list of all paths from the immediate hypernym nodes to the root of the tree, with each node carrying the set of instances that have been immediately triggered and that have been triggered by any of its hyponyms. Thus, each nod in each path will record the set (union) of all instances by the immediate node and all its hyponyms. This acts as a weight of the significance of each path. The root node of all these paths will thus contain the union of all instances.

In general, it is the case that the higher nodes in each path will have no immediate triggers. Also, it is quite likely that not all of its children will be activated by a given set of lemmas. As a result, the set of paths that are generated will usually have several nodes that are not immediately triggered. We therefor look for the first node in each path that has active triggers and assess the relative frequency of each of these paths to identify the most significant one.

This process bears a strong resemblance to one method used to assess semantic similarity between to words, i.e., finding the least common subsumer. When we have a reasonably sized set of lemmas, finding their least common subsumers is computationally intractable and would be very difficult to interpret intermediate steps. Our method is not computationally expensive and leads to the identification of what may be called least common subsumer ontological classes.

1.2.1. Step 1: Assembling the Set of Immediate Hypernym Nodes

The algorithm begins with a specification of the corpus, the preposition, the sense number, and the feature name that constitutes the immediate hypernym to be examined. This presumes that a feature file has been generated for the specified items. The feature file is read into a map giving the features for each sentence identifier; we iterate over the sentence identifiers to extract those of interest. We consider only those instances with the specified sense number, keeping a count of the number of such instances.

Not all instances will have the specified feature. For example, if the specified feature was generated only for nouns, several reasons may explain why no such feature was generated. The lemma may be a personal pronoun or gerund. The dependency parse may be in error, e.g., identifying the lemma as an adjective or adverb. Whatever the reason for a missing feature, we keep a count of the instances that have the desired feature. This count is the N upon which the relative frequencies are calculated.

The feature name is a code that consists of two parts: a word-finding rule and a feature-extraction rule. We search all the features generated for each sentence to find those that have this name. Each sentence may have 6,000 features. Depending on the lexical resource that has been used, there may several features in a sentence beginning with this feature name. Each such feature consists of the feature name and the value of the feature. We strip away the feature name to process just the feature values. These values constitute he immediate hypernym nodes of interest. We count the total number of features in the data. For each distinct feature value, we create a set of the instance identifiers; this set will be used in forming unions on the paths up the tree. We maintain a map of the distinct feature values, each with its set of instances. The number of entries in this map is the number of nodes in the hierarchy that have been activated or triggered.

1.2.2. **Step 2: Creating the Tree of Activated Nodes**

We need to transform the map of nodes from Step 1 into a tree of the activated nodes in the hierarchy specified by the lexical resource. We make use of a data structure for each node. This data structure consists of (1) a node name, (2) a node type, (3) a set of the instances that have been triggered at this node, (4) a set of all the instances triggered by this node and all hyponyms of this node, and (5) a list of all its immediate child nodes (i.e., its immediate hyponyms).

We begin by reading the lexical resource so that we can identify the parents of each node in the hierarchy of the lexical resource. We create a set of **nodesSeen** so that we can keep track of whether we have processed a node in the tree. We create an initial node with the name "root" and add this to what will become the tree of activated nodes, the **ontoMap**. We set the variable **lastNode** to this root node.

We then iterate over the set of nodes obtained from Step 1. For each **node** in this list, we obtain its list of its parents (i.e., its path all the way to the root of the tree) and iterate over the parent path, beginning at the end of the list. If **parent** is in **nodesSeen** (and thus has already been added to **ontoMap**), we simply set **lastNode** to this parent in **ontoMap**. If not, we first add it to **nodesSeen**. Then, we create a new node with **parent** as its name. We add this as a child to **lastNode**, reset **lastNode** to the newly created node, and put this new node into **ontoMap** under **parent**. To visualize this process in this iteration, we are ensuring that the path from the root to the current node exists in the growing tree (**ontoMap**).

After processing all parent nodes, and if **node** is not in **nodesSeen**, it is first added to **nodesSeen**, Then, we create a new node with **node** as its name and add its instance set to the set of instances triggered by this node. It is added as a child to **lastNode** and it is put into **ontoMap.** At this point, **lastNode** is reset to the root node to process the next **node**.

After processing all nodes in this manner, we have a tree of all the nodes in the hierarchy of the lexical resource that have been activated, either explicitly as a result of having instances or implicitly by reason of being in the path from such a node to the root.

1.2.3. **Step 3: Annotating the Nodes**

At this point, we have the desired tree of active nodes, but the data structure associated with each node is not complete. We have entered the node name, the set of instances (if any) triggered at the node, and any children. We have not yet entered a type for the nodes nor the union of all instances in the node and its children. We will do so in a process of annotating the nodes in the tree, using a recursive function call beginning at the root. This function will basically iterate over the child nodes of the argument **node** and make tests on what is currently known about **node**.

As a first step, it is assumed that the argument **node** has triggering instances, setting **noInsts** to *false*. Another argument to the function is a Boolean **liveParent**, which indicates that the instant **node** has a parent somewhere in the path to the root of the tree that has triggering instances. Test 1: If **liveParent** Is true, the argument **node** is provisionally given a type equal to '#'; such a type will be used as a stopping criterion in identifying desired paths. If **liveParent** is *false* and **node** has no triggering instances, the type is set to 'x' and **noInsts** is set to *true*. Test 2: If the node does have triggering instances, the type is set to 'live'. (This test might override the first condition of Test 1, resetting '#' to 'live'.) Test 3: If the node is a terminal node (i.e., has no children, but assuredly has triggering instances), its instances are copied to the set of all instances for this node. We also return from this call to the function, since we know that it has no children. Test 4: If the node has been marked as 'live', **liveParent** is set to *true* so that any of its children that have no instances will be marked as '#'.

After the above tests, we iterate over the children of **node**. The first step of this iteration makes the recursive call with the child as the argument **node**. This will recurse all the way down to all terminal nodes of the tree. Importantly, this recursion will fill the all instances set of all the children. After the recursive call, we form the union of this node's all instances field with the all instances field of the child. After the iteration over the children, the all instances field will thus be the union of all the instances of all its children. In this way, we will populate all nodes in the tree with the set of instances covered by each node and all its children. The root node will then be exactly the set of all instances that have processed in the analysis.

During the iteration over the children of the argument node, we keep track of the maximum number of instances by any of the children. After the iteration, we can perform an additional test for setting the node type. We apply this test only if **noInsts** is *true* and this node does not have a live parent. If this node has more than one child and the maximum number of instances of any child is equal to the eventual size of the all instances set for the node, we can set the type to '#'. What this implies is that all the instances ascribed to one child have been moved up to the current node, and can thus be used as a stopping point in our selection of desired paths.

1.2.4. **Step 4: Identifying the Least Common Subsumer Classes**

At this point, we have a subtree of the hierarchy of the lexical resource that consists of all nodes that have been activated, either directly in 'live' nodes or in paths from such nodes to the root of the tree. All these nodes have been given a type, 'live', '#', or 'x'. Associated with each node is the set of all instances triggered by itself or all of its children. We are now ready to identify the least common subsumer classes, i.e., the deepest node in each path that covers all children. In general, nodes at the top of the subtree are not 'live'; this is because such nodes correspond to very general, all-embracing concepts with very few lemmas equally general. As a result, the subtree will have several paths from the top that have type 'x'. The objective of this step is to follow each path to some stopping point, which is generally the deepest node such its parents have no triggers. We can then evaluate the worth of each path as the size of the stopping point's set of all instances divided by the total number of instances under investigation (the N determined in Step 1, which is equal to the size of the all-instances set at the root). We do this in a recursive function that steps down the subtree, keeping track of the path from the root as we step down through the child nodes. The function call is started with a path consisting of just the root node.

The basic process of the recursive function is to iterate through the child nodes of the argument **node**, creating a new path by adding the child node to the argument **path**. The child node and the new path are the arguments to the recursive call during the iteration. There are several tests in this function for determining that we have reached a stopping point. When we reach a stopping point, we print the path, along with its relative frequency.

- Test 1: If a child node is a 'live' node, the new path is printed and we continue to the next child.
- Test 2: If a child node is an '#' node, the new path is printed and we continue to the next child.
- Test 3: If a child node does not have any empty children (i.e., all children have been triggered by at least one lemma) and the number of all instances of the child is equal to the number of all instances of the argument **node**, the path before adding the child is printed and we return from this call, without considering any further children. This test is actually made before Test 1 and Test 2.
- Test 4: If none of the preceding tests have fired, we recurse with the child node and the new path as arguments.
- Test 5: This test occurs prior to entering the iteration and indicates that we have no need to iterate through the children. If all children of the argument **node** are either 'live' or '#', we print the path and return from the function.
- Test 6: This test occurs prior to entering the iteration and indicates that we have no need to iterate through the children. This test requires that the argument **node** has more than one child and that the sum of the number of all instances of the children is equal to the number of all instances of the argument **node**, i.e., the child instances are disjoint and cover all the instances of **node**. In this case, we print the path and return from the function.

As indicated, when we print a path, we also print its relative frequency, i.e., the number of all instances in the final node of the path divided the number of instances in the root node. We select the path that has the highest relative frequency as identifying the best least common subsumer class, using the final node on this path as the class.

2. **Investigations Using the Algorithm**

The algorithm described in the previous section has been developed and tested using data from the Pattern Dictionary of English Prepositions (PDEP) (Litkowski, 2014). In particular, we select the set of instances tagged with a specific sense of a specific preposition from a specified subcorpus of PDEP. As the lexical resource, we have used the noun taxonomy as developed for the *Oxford Dictionary of English* (McCracken, 2004). This taxonomy has 2273 nodes on up to 12 levels, with an average of 46 objects (nouns or lemmas) per node. A given lemma may trigger multiple nodes in the taxonomy.

2.1. **Developmental Example**

We selected sense 14(5) ("expressing duration") of the preposition *over* from the TPP subcorpus of PDEP. This sense has 137 tagged instances, of which 125 have a total of 709 taxonomic features in 119 nodes. As an example, a common lemma in this set is *period*, which occurs 21 times as the object of the preposition; this word triggers 11 nodes in the taxonomy, including *workingperiod*, *era*, *symbol*, and *menstruation*.

The number of activated nodes means that there are 119 paths to be evaluated. These paths activated 222 of the 2273 nodes in the taxonomy. We first performed a manual collapsing of these paths and identified 16 paths with terminals that conformed to our intuitions about what should be least common subsumer nodes. We then developed and applied the algorithm in an effort to duplicate these intuitions. The following set of paths shows some of the results, in order of decreasing relative frequency:

- (0.864) root x->(125) abstraction x->(114) time 10(108)
- (0.456) root x->(125) abstraction x->(114) measure x(57)
- (0.296) root x->(125) group 13(37)
- …
- (0.192) root x->(125) naturalphenomenon 1(24)
- …
- (0.008) root x->(125) entity x->(50) object x->(37) naturalobject x->(1) organismpart x->(1) bodypart x(1)
- (0.008) root x->(125) mentalfeature x(1)

As can be seen, the top-ranked path conforms to our expectations that this sense of *over* expresses **time**. In the taxonomy, **time** is subdivided into two classes (**pointintime** and **period**). If we look up the word *duration* in the dictionary, we find that it is in the node **period**, and activates no other node; indeed, the 108 instances identified as being under **time** are also in the node **period**. Perhaps interestingly, the lower ranked paths also seem to hint at or express some components of meaning for this sense, namely, that the object expresses a **measure** and has some notion of being a **group**. The lower-ranked paths do not appear to be salient to the central meaning of this sense.

2.2. Generalizing to Other Senses and Lexical Resources

The algorithm is readily applied to examining the prepositional noun complements and noun governors of other senses in PDEP. Results from such ontological analyses do not seem as crisp as the developmental example, perhaps because **time** lemmas are somewhat distinctive. Notwithstanding, such analyses do appear to be meaningful, i.e., identifying salient components of meaning for a group of lemmas. Within the PDEP data, we can potentially examine the ontological characteristics of up to 6000 senses. Each such analysis can be accomplished in just a few seconds, so the challenge is to develop a systematic procedure for doing so.

The algorithm can be extended to other lexical resources, such as WordNet, FrameNet, and VerbNet, each of which also has hierarchical data. The depth of these hierarchies varies. For example, verb hierarchies may be much shallower, so these will have shorter paths. To apply the algorithm with these resources is relatively straightforward. In the PDEP data, immediate hypernyms have been entered for

WordNet, and also in a preliminary way for the other resources. However, we have not yet implemented the path construction for these resources. An important aspect of implementing ontological analysis is that it cannot be accomplished in the first pass over the data, i.e., in the phase that generates the features. Instead, it requires a separate pass over the data once after the lemmas and immediate hypernyms have been identified.

2.3. Outstanding Issues with Algorithm

As indicated for the developmental example, 119 nodes in the ontology were activated, leading to 119 paths for the analysis. With the manual analysis, we identified 16 paths as intuitively correct. However, as implemented, the algorithm was only able to reduce the number of paths to 34. We experimented with several ways of collapsing the path set. For example, in the list shown in section 2.1, there were four **measure** paths, which were combined manually in showing the results. In general, there is a trade-off in specifying the criteria for stopping the path collapsing, where some rules lead to very truncated paths and others lead to overly expansive paths. Further work is needed to achieve the correct balance.

While the listing of paths is generally correct and matching intuition, they may be difficult to interpret. The final node in each path is somewhat opaque and is not perspicuous as to what lies below the node. Thus, it is not clear that **time** has two children, of which **period** is the dominate child and also covers all the instances covered by **time**. Also, although the algorithm prints out the number of instances that are covered by the path, the list of lemmas is not shown, so the range of coverage is not immediately seen.

In examining other prepositions and senses, the stopping rules frequently seem to stop too short, giving only one or two nodes. In many of these cases, it may be desirable to show more nodes, particularly the ones that are more frequent. In the example sense, the node **group** is not subdivided further. As can be seen, this node is directly triggered by 13 instances, a third of all its instances; an additional 21 instances are triggered at its child **category**, with the remaining three instances triggered by other children of **group**. While this node is not dominant in the overall analysis of the example sense, it does suggest that a tinge of meaning contributed to this sense of *over* by this class.

3. Enhanced Semantic Word Sketches

McCarthy et al. demonstrated the value of using the WordNet lexicographer file names in enhancing word sketched. These file names are somewhat coarse in granularity. We suggest that the least common subsumer classes may provide a finer-grained level of granularity below this level and above the level of the lemmas. The method outlined in the algorithm acts to group similar lemmas. Moreover, while we have focused on preposition complements and governors, this algorithm would also be applicable to the subjects and objects of verbs.

References

Claudia Leacock and Martin Chodorow. 1998. Combining Local Context and WordNet Similarity for Word Sense Identification. In Christiane Fellbaum (ed.), *WordNet: An Electronic Lexical Database*. Cambridge, Mass. MIT Press.

Ken Litkowski. 2014. Pattern Dictionary of English Prepositions. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*. Baltimore, Maryland, USA, pp. 1274-83.

Diana McCarthy, Adam Kilgarriff, Milos Jakubicek, and Siva Reddy.2015. Semantic Word Sketches. *Corpus Linguistics 2015*. Lancaster University.

James McCracken. 2004. Rebuilding the Oxford Dictionary of English as a Semantic Network. In: *COLING 2004 Proceedings of the Workshop on Enhancing and Using Electronic Dictionaries*. Geneva, Switzerland, pp. 69-72.

Ted Pedersen, Siddharth Patwardhan, and Jason Michelizzi. 2004. WordNet::Similarity:- Measuring the Relatedness of Concepts.. In *Demonstration Papers at HLT-NAACL 2004*. Boston, Massachusetts, USA, pp. 38-41.

Zhibiao Wu and Martha Palmer. 1994. Verb Semantics and Lexical Selection. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*. Las Cruces, New Mexico, USA, pp. 133-38.