Extracting Enterprise Vocabularies Using Linked Open Data

Julian Dolby, Achille Fokoue, Aditya Kalyanpur, Edith Schonberg, and Kavitha Srinivas

IBM Watson Research Center,P.O.Box 704, Yorktown Heights, NY 10598, USA
dolby, achille, adityakal, ediths, ksrinivs@us.ibm.com

Abstract. A common vocabulary is vital to smooth business operation, yet codifying and maintaining an enterprise vocabulary is an arduous, manual task. We describe a process to automatically extract a domain specific vocabulary (terms and types) from unstructured data in the enterprise guided by term definitions in Linked Open Data (LOD). We validate our techniques by applying them to the IT (Information Technology) domain, taking 58 Gartner analyst reports and using two specific LOD sources – DBpedia and Freebase. We show initial findings that address the generalizability of these techniques for vocabulary extraction in new domains, such as the energy industry.
keywords: Linked Data, Vocabulary Extraction

1 Introduction

Most enterprises operate with their own domain-specific vocabularies. A vocabulary can be anything from a set of semantic definitions to a formal ontology. Vocabularies are necessary to work effectively in a global environment and to interact with customers. They facilitate common tasks, such as searching a product catalog or understanding general trends. However, building and maintaining a vocabulary manually is both time-consuming and error-prone.

This paper presents a process to fully automate the construction of a domain-specific vocabulary from unstructured documents in an enterprise. The constructed vocabulary is a set of terms and types, where we label each term by its type(s). We applied this process to the IT (information technology) domain, and automatically built an IT vocabulary taking 58 Gartner analyst reports as our input corpus. For this domain, we capture types such as *Distributed Computing Technology*, *Application Server*, and *Programming Language*, and use them to label terms like "Cloud Computing", "IBM WebSphere", and "SmallTalk".

Our approach is based on a simple observation: people searching for term definitions on the Web usually find answers in either a glossary or Wikipedia. We use LOD (Linked Open Data) as our source for domain-specific types. We decided to focus on two specific subsets of LOD as our reference data – DBpedia and Freebase. Both datasets derive from Wikipedia and thus have broad domain coverage. Also, both have type information, e.g., DBpedia associates entities with types from the YAGO and Wikipedia type hierarchies [1].

To perform vocabulary extraction, a key step is to determine what the relevant domain-specific types are for a particular corpus. Simply looking up corpus terms in LOD is not adequate since the terms may have different senses in LOD. Many or all of these senses may be unrelated to the domain. Therefore, we use statistical techniques to automatically isolate domain-specific types found in LOD. These types are then used to label corpus terms. This core vocabulary extraction process based on LOD is discussed in Section 2. For the Gartner case, this technique produced results with high precision (80%) but poor recall (23.8%).

To address this, we present two techniques to boost recall for vocabulary extraction. The first technique, presented in Section 3, directly improves recall by increasing the coverage in LOD. Numerous instances in DBpedia or Freebase have incomplete or no type information at all, which explains some of the poor recall. The second technique, presented in Section 4, improves recall by automated machine learning.

Our technique to improve coverage in LOD is a form of type inference based on the attributes of a particular instance, and based on fuzzy definitions of domains/ranges for these attributes. As an example, we can infer that an instance has a type *Company* if it has an attribute *revenue*, because statistically, across the DBpedia dataset, instances with the attribute *revenue* tend to have *Company* as a type. We performed this type inference on the entire LOD dataset, and then re-applied our core vocabulary extraction algorithm. Type inference improved recall to 37.6% without altering the precision.

Our technique to boost recall using machine learning relies on building statistical named entity recognition (NER) models automatically. We accomplished this by using seeds generated directly from LOD and exploiting structural information in both Wikipedia and DBpedia to generate high quality contextual patterns (features) for the model. The end result was an effective, general-purpose NER model that worked well across different corpora, i.e., it was trained on Wikipedia but applied to the domain-specific corpus, the IT analyst reports. Adding results from NER to our previous output gave us a net recall of 46% and precision of 78%.

We discuss strengths and limitations of our overall approach in Section 5. In particular, we describe initial findings that show the generalizability of the vocabulary extraction techniques in new domains.Finally, we discuss related work and conclusions in Section 6.

In summary, the main contributions of this paper are:

- We describe a process to automatically extract a domain specific vocabulary from unstructured data in the enterprise using information in LOD. We validate our techniques by applying them to a specific domain, the IT industry, with a precision of 78% and recall of 46%.
- We describe a set of techniques to boost recall in vocabulary extraction. The first improves the coverage of structured information in DBpedia and Freebase (two core pieces of LOD). By making LOD more robust, it becomes more useful for a range of applications, including our current task of extract-

ing vocabularies. The improved versions of LOD improved the recall of our vocabulary extraction process by 14% without affecting precision. The second technique improves recall by relying on techniques for automated named entity recognition, and this improves recall by an additional 8%.

2 Vocabulary Extraction

Our process extracts domain specific terms from an unstructured text corpus, and labels these terms with appropriate domain-specific types. This is a different problem than traditional NER. Off-the-shelf NERs are typically trained to recognize a fixed set of high-level types such as *Person*, *Organization*, *Location* etc. In our case, we need to discover the types for a specific domain, and use these discovered types to label terms in the corpus. We rely on sources outside the corpus, in particular LOD, since appropriate types may not even appear in the corpus.

We also note that, typically, domain-specific terms in a corpus are found using *tf-idf* scores. To the best of our knowledge, we have not seen type information being used as an additional dimension to filter domain-specific terms, and this is one of the differentiators of our approach.

Briefly, our process performs the following steps:

1. Extract a population of terms (noun phrases) from the domain corpus.
2. Extract a seed set of domain-specific terms from the corpus using traditional *tf-idf* scores.
3. Extract domain-specific types from LOD, using the seed terms from step 2.
4. Filter the set of all terms from step 1, based on the domain-specific types from step 3. The result is a set of domain-specific terms, labeled by their respective type(s). This allows certain relevant corpus terms that have low *tf-idf* scores to be selected based on their relevant domain-specific type.

The following subsections provide the details of each of these steps, and results for the IT domain.

2.1 Term Population

We extract the noun phrases from a domain corpus, using a standard NLP part-of-speech tagger (from OpenNLP[1]). These noun phrases comprise the population of all terms. For our case study, the domain corpus was 58 IT analyst reports from Gartner. The resulting term population extracted consisted of approximately 30,000 terms. To measure the precision and recall of our process, we created a gold standard from the term population. We randomly selected 1000 terms from the population, and four people judged whether each term in this sample was relevant to the domain. A term was considered relevant only if all four judges agreed. In the end, 10% of the sample terms were considered relevant to the IT domain. Therefore, we expect 3,000 terms in the population to be relevant.

[1] http://opennlp.sourceforge.net/

2.2 Domain-Specific Seed Terms

The next step is to extract an initial set of domain-specific terms from the corpus based on traditional *tf-idf* metrics. These terms are subsequently used to look up types in LOD, and form the basis for domain-specific type selection. For this task, it is more important to find a precise set of domain-specific terms, than to find all of the domain-specific terms. We use an off-the-shelf tool, GlossEx [2], that extracts words and noun phrases along with their frequency and domain-specificity. This associated data allows low-frequency, low-specificity terms to be filtered out. For our analyst report corpus, we selected proper noun phrases and common noun phrases with frequency ≥ 2 and with an appropriate tool-specific domain-specificity threshold. We obtained 1137 domain-specific terms from the Gartner IT reports.

2.3 Domain-Specific Types

Using the domain-specific seed terms, we discover a set of relevant interesting types from LOD. Our algorithm to discover domain-specific types is outlined in Table 1. The first step is to find a corresponding LOD entity for a domain-specific term. The most precise and direct way to do this is to encode the term directly as an LOD URI (i.e., by adding the DBpedia URL prefix) and check if it exists. This produced matching entities for 588 of the terms.

Ideally it should be possible to simply look up the type(s) of each of these entities and mark them as interesting. However, this does not work since a term can ultimately map to different types with different senses. For example, "Java" is a programming language and an island, and we may select the incorrect LOD entity and hence sense. Even if there is a single LOD entity for a term, it may not be the sense that is relevant for the domain. For example, the term "Fair Warning" is a software product, but there is only one type sense in LOD, which is `Category:MusicAlbum` (pointing to an album with the same name released by Van Halen).

To address this problem, we filter out uninteresting types using simple statistical information. We score LOD types based on the number of terms they match across all the documents in our corpus and filter out infrequent types (those whose frequency is below a pre-determined threshold). One issue here is the different kinds of type information in LOD – YAGO types from DBpedia, Freebase types and Categories that come from Wikipedia. We found that having separate frequency thresholds ($\alpha_Y, \alpha_F, \alpha_C$ resp. in Table 1) for each of the types produced better results.

Another issue we noticed with DBpedia is that several entities do not have any values in their *rdf:type* field, but had interesting type information in the *skos:subject* field. For example, the term "NetBIOS" has no *rdf:type*, though its *skos:subject* is mentioned as `Category:Middleware`. Hence, we take SKOS subject values into account as well when computing types for a seed term (step 4 of the algorithm). Also, DBpedia and Freebase come with different types for

the same instance, and we exploit these in step 4 to obtain additional related type information from Freebase.

Filtering also removes several low frequency types that are interesting, e.g., `yago:XMLParsers`. To address this issue, we consider low frequency types as interesting if they are subsumed by any of the high frequency types. For example, `yago:XMLParsers` is subsumed by `yago:Software` in the Yago type hierarchy, and is thus considered relevant as well (steps 6-8 of the algorithm).

Input: S_{dt} : set of domain-specific seed terms, threshold parameters $\alpha_Y, \alpha_F, \alpha_C$
Output: τ set of domain-specific types from LOD

(1) initialize potential type list $\tau_p \leftarrow \emptyset$
(2) for each domain-specific seed $T \in S_{dt}$
(3) encode T as a DBpedia URI U
(4) $\tau_p \leftarrow \tau_p \cup types(U)$
where $types(U)$ = values of prop. *rdf:type* for $U \cup$
values of prop. *skos:subject* for $U \cup$
'equivalent' types obtained via mappings to Freebase
(5) $\tau \leftarrow T \in \tau_p$ if
(freq(T) $\geq \alpha_Y$ and T is a *Yago* Type) or
(freq(T) $\geq \alpha_F$ and T is a *Freebase* Type) or
(freq(T) $\geq \alpha_C$ and T is a Wiki *Category*)
(6) for each type $X \in (\tau_p - \tau)$
(7) if there exists a type $Y \in \tau$ s.t. LOD $\models X \sqsubseteq Y$
(8) $\tau \leftarrow \tau \cup \{X\}$

Table 1. Extraction of Domain-Specific Types

We ran the type-discovery algorithm with 1137 seed terms and setting appropriate type-frequency thresholds ($\alpha_Y = 4$, $\alpha_F = 4$, $\alpha_C = 4$) based on manual inspection of highly frequent types in τ_p. This produced 170 interesting types. Upon manual inspection, we found the output to have extremely high precision (98%). As expected, there is a tradeoff between precision and recall – reducing the type-frequency thresholds increases the size of the output types but decreases its precision.

2.4 Filtered Terms and Types and Discussion of Initial Results

Once we have the set of domain-specific types, we revisit the entire population of corpus terms from Section 2.1. We find terms in this population that belong to one of the domain-specific types. This set is more comprehensive than the initial seed set in 2.2. Specifically, we select the terms from the population that are 'closely related' to entities in LOD which belong to at least one domain-specific type. In order to find 'closely related' entities, we perform a keyword search for each term over a database, populated with data from DBpedia and FreeBase, and indexed using Lucene. We select matches with a relevance score greater than 0.6. For example, searching for the term "WebSphere" over the Lucene index produces entity matches such as "IBM_WebSphere", whose corresponding *rdf:type*/*skos:subject* value belonged to one of our domain-specific types. Therefore, 'WebSphere' is selected as a domain-specific term.

This process resulted in 896 type-labeled terms using the 170 interesting types found in the previous step. We evaluated the precision of both the terms and their types by manually evaluating a 200 term sample. This gave us a precision of 80%. We also computed recall by taking into account our gold standard estimate and the precision, and found that recall was 23.8%.

Although the precision of our initial results was reasonable, our recall was poor. Our first approach to improving recall was to directly improve the coverage of types in LOD. The techniques developed are outlined in the next section.

3 Improving Coverage of LOD

In this section, we discuss techniques to add new knowledge to our reference LOD datasets – DBpedia and Freebase. For DBpedia, we used data dumps for DBpedia 3.1. For Freebase, we used the WEX data dumps from July [3].

Sometimes there are no types in this combined dataset, which is a problem when using the dataset to perform vocabulary extraction for any domain. We therefore enhanced the type information by linking instances and types across datasets, and by inferring new types for instances.

3.1 Adding Type Information from Linking

An obvious step to improve type coverage in Linked Open Data is to leverage the fact that DBpedia and Freebase might have different types for the same instance. We linked DBpedia instances to Freebase instances using their shared Wikipedia name. This technique allowed us match all 2.2 million Freebase instances except 4,946, mainly because of differences in Wikipedia versions between the two datasets.

Although linking instances achieves the aggregation of types across Freebase and DBpedia, the two type systems are still disconnected. It would be useful to know mappings between the two sets of types for vocabulary extraction. For instance, if we knew both that a type in DBpedia such as `yago:Byway102930645` maps to the corresponding Freebase type `freebase:/transportation/road`, and also knew that `yago:Byway102930645` is as an interesting domain specific type, then `freebase:/transportation/road` is likely an interesting type as well. DBpedia has 159,379 types and Freebase has a much smaller set of 4,158 types that are specified at a coarser level of granularity. We therefore used the relative frequency with which a given DBpedia type A co-occurs with a Freebase type B to drive the mapping. We considered a mapping valid if the conditional probability $p(FreebaseType|DBpediaType)$ was greater than .80. Manual inspection of a random sample of 110 pairings revealed that 88% mappings were correct. With this technique, we were able to map 91,558 DBpedia out of 152,696 DBpedia types to Freebase types (we excluded mappings which mapped to the freebase type /common/topic because its a top level type like `owl:Thing` or `yago:Entity`). In all, because a single DBpedia type can map to multiple Freebase types (e.g., `yago:InternetCompaniesEstablishedIn1996` is mapped to

`freebase:/business/company` and `freebase:/business/employer`), we had 140,063 mappings with the 80% threshold.

3.2 Type Inference

We propose a simple statistical technique to extract fuzzy domain and range restrictions for properties of an individual, and use these restrictions to perform type inference, as we describe below.

To illustrate how our technique works in DBpedia, take as an example the instance `yago:Ligier`, which is a French automobile maker that makes race cars. DBpedia has the types `yago:FormulaOneEntrants` and `yago:ReliantVehicles` as types, neither of which is a company. Yet, `dbpedia:Ligier` has properties specific to companies, such as `dbpedia-owl:Company#industry`, `dbpedia-owl:Company#parentCompany`, etc. If we had a predefined ontology, where `dbpedia-owl:Company#industry` had a domain of the type `yago:Company`, we could have used RDFS or OWL reasoning to infer that `yago:Ligier` is really a Company. However, manually defining domain and range restrictions is not an option, because DBpedia has 39,345 properties. We therefore used statistical techniques to define fuzzy notions of domains and ranges to perform type inference.

Our approach to type inference is based on correlating what properties an entity has with its explicit types. The idea is that if many instances of a particular type have a certain set of edges, then other entities with that same set of edges probably are instances of that type too. In performing this type of inference, we relied on the DBpedia to Freebase mappings we established in 3.1 to infer Freebase types. As discussed earlier, because Freebase types are specified at a coarser level of granularity, type inference is more robust for these types because of the larger sample size of instances.

More formally, we define the notion of a property implying a type based on the fraction of the given edge that pertain to instances with that property being greater than some threshold τ. Note that this notion applies to both subjects and objects of edges.

$$I_{subj}\left(p,t\right) \equiv \frac{\left|\{p(x,y)\,|x:t\}\right|}{\left|\{p(x,y)\,|\exists t_1 x:t_1\}\right|} > \tau \quad I_{obj}\left(p,t\right) \equiv \frac{\left|\{p(x,y)\,|y:t\}\right|}{\left|\{p(x,y)\,|\exists t_1 y:t_1\}\right|} > \tau$$

where $i:t$ is an `rdf:type` assertion between an instance i and a type t, and $p(i,x)$ is a role assertion which links instance i to instance x on a property p. This step can be thought of as inferring domains and ranges for properties, as was done in [4]; however, rather than use these types directly as such constraints, we use them in a voting scheme to infer types for subject and object instances.

Given the notion of a property implying a type, we define the notion of properties voting for a type, by which we simply mean how many of a given instance's properties imply a given type:

$$V\left(i,t\right) \equiv \left|\left\{p\,\middle|\,\left(\exists x\; p(i,x) \wedge I_{subj}\left(p,t\right)\right) \vee \left(\exists x\; p(x,i) \wedge I_{obj}\left(p,t\right)\right)\right\}\right|$$

We additionally define the notion of all edges that take part in voting, i.e. the number of edges that pertain to a given instance that imply any type:

$$V_{any}(i) \equiv \left|\left\{p \left| \exists t \left(\left(\exists x\ p(i,x) \wedge I_{subj}(p,t)\right) \vee \left(\exists x\ p(x,i) \wedge I_{obj}(p,t)\right)\right)\right\}\right|$$

Finally, given the notion of voting, we define the implied types of an instance simply as those types that receive the greatest number of votes from properties of that instance compared to the total number of properties of that instance that could vote for *any* type:

$$T(i) \equiv \left\{ t \left| \left(\forall_{t_1} V(i,t) \geq V(i,t_1)\right) \wedge \frac{V(i,t)}{V_{any}(i)} \geq \lambda \right.\right\}$$

We applied this technique to our data setting τ and λ to .5. We inferred types for 1.1M instances of the 2.1M instances in Linked Open Data. To help evaluate these types, we introduce a technique next to automatically detect when we might have inferred invalid types.

3.3 Evaluating type inferences

At its core, the detection of invalid type inference is based on the observation that if two types are known to be logically disjoint, such as *Person* and *Place*, and we infer a type that is disjoint with any of the explicitly asserted types of an instance, this constitutes an error in our type inference. In prior work, ontology reasoning has been used to automatically detect invalid type inference in text extraction [5]. However, extending this approach to Linked Open Data is not easy. Although YAGO types are organized in a hierarchy, there are no obvious levels in the hierarchy to insert disjoints. Freebase has no hierarchy at all: the type structure is completely flat.

Our approach therefore was to first statistically define a type hierarchy for Freebase, and then use that hierarchy to define disjoint classes to detect invalid type inferences.

Because Freebase has no type structure, each instance in Freebase is annotated with a flat set of types, in which more-general types occur along with less-general types. We use this fact to approximate the usual notion of supertype: a supertype Y by definition contains all the instances of its subtype X, and, in normal circumstances, thus we would expect to see the following, where P denotes probability: $P(i \in Y | i \in X) == 1$

Because the Freebase data is noisy, this probability will most likely be less than 1; to account for this, we can recast the above constraint as follows: $P(i \in Y | i \in X) > \tau$

Thus, $X \subset Y$ if instances of X are almost always instances of Y. This allows us to define groups of types that are related as subtypes; in particular, related groups are maximal groups that are closed under subtype, i.e. G is a group if and only if $\forall_{x,y} x \in G \wedge (y \subset x \vee x \subset y) \Rightarrow y \in G$

This definition gave us 78 groups, with τ set to .65. Of these, 26 groups were singletons, and the largest group had 409 types. We further manually grouped

the 78 groups by those that appeared to belong to the same domain; this gave us 35 larger groups of types, which covered 1,281 types out of 4,158 types. In practice, these groups were overwhelmingly disjoint, and we dropped the few types that occurred in multiple groups. The 35 groups of types were declared as pairwise disjoint (i.e., each type T in group A was declared disjoint from each type Q in group B).

To evaluate the 1.1M inferred types better, we divided them into 3 categories: (i) *Verified* for the entities for which at least one of the inferred types is the same as an explicitly declared one (this category had 808,849 instances), (ii) *Additional inferred types* for the entities for which the inferred types were not disjoint with any existing type assertion, i.e. these denote additional inferred type assertions that helps improve coverage in DBpedia (this category had 279,407 instances), and finally (iii) *Invalid* for the entities for which at least one of the inferred types conflicted with an explicitly asserted type (this category had 6,874 instances).

To determine the accuracy of our inferred types, we took samples of the invalid and additional inferred types, and evaluated the precision for these categories with two random samples of 200 instances each. An instance was considered to be typed correctly if all the inferred types for an instance were correct. In the additional inferred types category, we typed 177 instances correctly, and 23 incorrectly. In the invalid category, we typed 21 instances correctly, and 179 wrong. Taking the overall results for all the categories into account, we achieved a net recall of 49.1% and an estimated precision of 95.8% accuracy. Note that we have the usual trade-off between precision and recall based on values for the parameters τ, λ, however we achieved a reasonably high F-score of 64.9. We added only the *Additional inferred types* into our version of Linked Open Data, and re-ran our vocabulary extraction. The results are described in the next section.

3.4 Results with improved LOD

Note that the previous execution of the vocabulary extraction algorithm (Table 1) using off-the-shelf LOD datasets produced 170 interesting types and 896 type-labeled terms in the output, with a precision of 80% and recall of 23.8%. We re-applied the algorithm with the newly discovered type assertions added to DBpedia and the new type mappings from DBpedia to Freebase (and the same input parameters as earlier) and discovered 188 interesting types and a net output of 1403 type-labeled terms. Manual inspection of a 200 term sample revealed that the precision was unaltered, and recall had increased to 37.6%, which validated our coverage enhancement techniques.

4 Improving Coverage using Statistical NER

Since LOD coverage is incomplete, the techniques described above do not produce a complete domain-specific vocabulary. From our gold standard, described in Section 2.1, we expect to find around 3K domain-specific terms, and the

output of the previous step is still quite short. In order to improve the coverage of our solution, we automatically build an NER model for domain-specific types.

The previous step produced a large number of interesting types (>170). These include YAGO types, Freebase types and Wikipedia Categories, which have related groups, such as, `yago:ComputerCompanies`, `freebase:venture_funded_company` and `Category:CompaniesEstablishedIn1888`, all conceptually subclasses of *Company*. Given the large number of closely related types, it does not make sense to build an NER model for each of the types. Instead we decided to look only at top-level types in the output (types that were not subsumed by any other). Furthermore, given the noise in the type-instance information in LOD, we decided to restrict ourselves to Yago types that have Wordnet sense ID's attached to them (e.g. `yago:Company108058098`), since they have a precise unambiguous meaning and their instances are more accurately represented in LOD. In our case, this yields five YAGO/Wordnet types: `yago:Company108058098`, `yago:Software106566077`, `yago:ProgrammingLanguage106898352`, `yago:Format106636806` and `yago:WebSite106359193`.

The process of building a statistical model to do NER is inspired by techniques described in systems such as Snowball [6] and PORE [7]. The basic methodology is the following – start with a set of training seed tuples, where each tuple is an <*instance, type*> pair; generate a set of 'textual patterns' (or features) from the context surrounding the instance in a text corpus; and build a model to learn the correlation between contextual patterns for an *instance* and its corresponding *type*. We combine the best ideas from both Snowball and PORE and make significant new additions (see the Related Work (Section 6) for a detailed comparison).

We could not afford to train the model on the IT corpus itself for two reasons: (i) lack of sufficient contextual data (we only had 58 reports), (ii) lack of adequate training seed tuples (even if we took the most precise term-type pairs generated in the previous section, it was not enough data to build a robust model). However, Wikipedia, combined with LOD, provides an excellent and viable alternative. This is because we can automatically obtain the training seed tuples from DBpedia, without being restricted to our domain-specific terms. We look for instances of the YAGO/Wordnet types in DBpedia, and find the context for these instances from the corresponding Wikipedia page.For our learning phase, we took either 1000 training seed instances per type or as many instances as were present in LOD (e.g., `yago:ProgrammingLanguage106898352` had only 206 instances). This gave us a total of 4679 seed instances across all five types.

A key differentiator in our solution is the kind of text patterns we generate (by patterns here, we mean a sequence of strings). For example, suppose we want to detect the type *Company*. The following text pattern [*X, acquired, Y*], where X, Y are proper nouns, serves as a potentially interesting pattern to infer that X is of type *Company*. However, a more selective pattern is the following: [X *acquired* <Company>]. Knowing that Y is of type *Company*, makes a stronger case for X to be a *Company*. Adding type-information to patterns produces more selective patterns.

Input: Sentence S containing training instance I, entities with Wikipedia URLs $WN_1..WN_k$; and complete Type-Outcome set for model OT
Output: Set CP of patterns (string sequences)
(1) Run S through OpenNLP POS tagger to get token sequence $TK : [.., < word, POS >, ..]$
(2) Remove tokens in TK where the word is an adverb , modifier or determiner
(3) Replace common nouns/verbs in TK with respective word *stems* using WordNet
(4) for each occurrence of pair $< I, POS(I) >$ in TK,
(where pos_i is position index of pair)
(5) $SPANS \leftarrow$ ExtractSpans(TK, pos_i)
(6) for each [start-pos, end-pos] $\in SPANS$
(7) $TK_{span} \leftarrow$ subsequence TK(start-pos, end-pos)
(8) $CP \leftarrow CP \cup$ word sequence in TK_{span}
(9) $CP \leftarrow CP \cup$ word sequence in TK_{span} replacing proper nouns/pronouns with resp. POS tag
(10) $CP \leftarrow CP \cup$ word sequence in TK_{span} replacing $WN_1..WN_k$ with corresponding types and their resp. super-types from LOD (provided that type is in OT)
(11) Remove adjectives (JJ) from TK
repeat (8)-(10) once
(Note: When generating patterns in steps (8)-(10), we replace training instance I by tagged variable 'X:POS(I)'

Subroutine: ExtractSpans(TK, pos_i)
(1) $SPANS \leftarrow \emptyset$
(2) for each j, $0 \leq j \leq (pos_i - 1)$
(3) if $TK(j)$.POS = *verb* or *noun*
(4) $SPANS \leftarrow SPANS \cup [j, pos_i]$
(5) for each j, $(pos_i + 1) \leq j \leq length(TK)$
(6) if $TK(j)$.POS = *verb* or *noun*
(7) $SPANS \leftarrow SPANS \cup [pos_i, j]$
(8) return SPANS

Table 2. Pattern Generation Algorithm

To add precise type information, we exploit the structure of Wikipedia and DBpedia. In Wikipedia, each entity-sense has a specific page, other Wikipedia entities mentioned on a page are typically hyperlinked to pages with the correct sense. E.g., the "Oracle_Corporation" page on Wikipedia has the sentence *"Oracle announces bid to buy BEA"*. In this sentence, the word *BEA* is hyperlinked to the 'BEA Systems' page on Wikipedia (as opposed to Bea, a village in Spain). Thus, using the hyperlinked Wikipedia URL as the key identifier for a particular entity sense, and obtaining type-information for the corresponding DBpedia URL, enables us to add precise type information to patterns. For the example sentence above, and the seed <*Oracle*, `yago:Company108058098`>, we generate the pattern [*X:NNP, announces, bid, to, buy,* <`yago:Company108058098`>] because "BEA_Systems" has the type <`yago:Company108058098`> (among others) in LOD, while X here is a variable representing the seed instance *Oracle*, and is tagged as a proper noun.

Moreover, not only do we substitute a named entity in a pattern by all its corresponding types in LOD, we add in super-type information, based on the type-hierarchy in LOD. We only focus on the YAGO/Wordnet types in the hierarchy which are the most precise. This generalization of patterns further helps improve recall of the model. Besides using type information, we also use a

stemmer/lemmatizer (using a Java WordNet API[2]) to generalize patterns, and a part-of-speech tagger to eliminate redundant words (e.g., determiners) and add POS information for proper nouns and pronouns in patterns. Details of our pattern generation is described in the algorithm in Table 2.

Type	Pattern
Company	[<NNP>, *be, acquire, by, X:NNP*]
Software	[<Company>, *release, version, of, X:NNP*]
ProgLang	[<Software>, *write, in, X:NNP*]
Format	[*encode, X:NNP*]
Website	[*X:NNP, forum*]

Table 3. Sample High Scoring Patterns

We use the text patterns as features to train a Naive Bayesian classifier that recognizes the concerned types. Finally, we repeat the recognition phase. Newly recognized term-type tuples are fed back into the system and used to rescore patterns taking in the new contexts, and also to generate new contexts for the remaining unrecognized terms by adding in type information. This process repeats until nothing changes. This feedback loop is effective, since we produce several patterns with type information in them, and these patterns are not applicable unless at least some terms in the context already have types assigned. For example, the sentence fragment *"IBM acquired Telelogic"* appears in our text corpus; initially, we detect that the term "IBM" has type *Company*, and feeding this information back to the system helps the machine recognize "Telelogic" is a *Company* as well (based on the pattern [<*Company*>, *acquired, X*] for Type(X):*Company*). We have to be careful during the feedback process since terms that have incorrectly recognized types, when fed back to the system, may propagate additional errors. To prevent this, we only feedback terms whose types have been recognized with a high degree of confidence ($Pr(T_i) > 0.81$). Some sample high scoring patterns captured by our model are shown in Table 3.

4.1 Evaluation of our NER Model

We evaluated our NER separately on Wikipedia data and the IT corpus. For the Wikipedia evaluation, we took our initial set of 4679 seed instances from LOD, and randomly selected 4179 instances for training and set aside the remaining 500 instances for evaluation. For evaluation on the IT corpus, we manually generated a gold-standard of 159 <*term, type*> pairs, by randomly selecting a sample of 200 pairs from the output of Section 2.4, and then manually fixing erroneous pairs. The results are shown in Table 4.

The table shows precision, recall and F-scores for our model over each of the domains, with and without the feedback loop implemented. The results are

[2] http://sourceforge.net/projects/jwordnet

Domain	No Feedback			With Feedback		
	Prec.	Rec.	F	Prec.	Rec.	F
Wikipedia	71.1	41.5	52.4	69.5	42.5	52.8
IT Corpus	76.4	38.6	51.3	76.5	52.3	62.1

Table 4. Evaluating our NER model

encouraging. While not near the performance of state-of-the-art NER's (which achieve F-scores in 90% range, e.g., [8]), there are several key points to keep in mind.

First, typical NERs detect a pre-defined set of types and are specially optimized for the types using a combination of hand-crafted patterns/rules and/or a large amount of manually annotated training data. We have taken a completely automated approach for both recognizing domain-specific types and generating training data and patterns, and our scores are comparable to, and in some cases even better than, similar approaches such as [6], [7]. The quality issues in LOD adversely affects our results, and thus the more we can improve the quality of LOD the better our results should be. Second, there is scope for improvement using a more robust classifier based on Support Vector Machines (SVMs).

Finally, the performance of our model across domain corpora is significant. The model, which is trained on Wikipedia and LOD and applied to IT corpus, performs comparably well without feedback, and substantially better with feedback (esp. recall). This indicates that the kind of patterns we learn on Wikipedia, using information from LOD, can be interesting and generic enough to be applicable across different domains. The performance improvement with feedback on the IT corpus was due to better uniformity in the writing style, and thus incorporating text-patterns for recognized terms in the feedback loop helped generate additional interesting domain-specific patterns.

4.2 Results with NER model

As a result of applying our NER model to the IT reports generated 381 new term-type pairs, which was added to the output of Section 3.4 to give us 1784 terms in all in our domain-specific vocabulary. Using the same evaluation process as earlier, we found that precision dropped a bit to 78%, but recall increased to 46%. Table 5 shows a sample of the extracted domain-specific vocabulary.

5 Discussion

Benefits of Using LOD: There are some direct benefits of using LOD that we should mention. First, having labeled domain-specific terms with appropriate types, a next logical step is to arrange the type labels in a hierarchy for classification purposes. Here, we can leverage the YAGO Wordnet hierarchy in DBpedia in addition to using our inferred type hierarchy from Freebase. Second, a way to

Software Developer
BMC Software, Fujitsu, IBM Software Group, Automattic, Apache Software Foundation,...
Telecommunications equipment vendors
Alcatel-Lucent, Avaya, SonicWALL, Lucent Technologies ...
Service-oriented business computing
Multitenancy, B2B Gateway, SaaS, SOA Governace, Cloud Computing
Content Management Systems
PHP-Nuke, OsCommerce, Enterprise Content Management, WordPress, Drupal
Software
Corep, Lotus Sametime Advanced, rhype, AIM Pro Business Edition, BPMT, Agilense ...
Programming language
Joomia, ABAP, COBOL, BASIC, ruby, Java
Java Plaform
NetBeans, JDevelopper, Java Software, Java ME, Oracle JDevelopper, ZAAP
Website
Twitter, Microsoft Live, Office Online, GMail, Second Life

Table 5. Sample IT Vocabulary Extracted

enrich the vocabulary is to capture relations between terms, e.g., the *developerOf* relation between the company *IBM* and the software *WebSphere*. Such relation information exists in LOD, making it possible to extract relevant relations between vocabulary terms. Third, because Wikipedia does cover a broad set of topics, our techniques can generalize to a new domain. For example, we conducted a preliminary experiment which suggests that our vocabulary extraction techniques can be generalized to the energy domain.

In our experiment, the input was a corpus of 102 news articles for the energy sector, drawn from various websites that specialize in news for the energy industry. The 102 news articles matched the 58 IT reports in terms of length (i.e., number of words). We started with a set of 1469 domain-specific terms drawn from GlossEx [2], and drew a sample of 500 terms. Of this sample, 204 terms were evaluated by four people to be relevant to the energy domain, leaving us with an estimate of 599 relevant terms in the overall sample. Our vocabulary extraction technique extracted 260 terms and types, of which 190 were correct terms and types (73% precision and recall of 32%). Sample terms and types we found in the energy sector included companies such as *Gazprom*, *Petrobras*, important people in energy such as *Kevin Walsh*, *Chris Skrebowski*, countries or regions relevant to energy such as *Sakhalin*, and *South Ossetia*, and terms such as *Tar sands*, *Bitumen*, *LNG*, and *Methane Hydrates*.

Limitations: Our current recall score is still less than 50% inspite of improving the coverage of information in our two LOD sources and using automated NER, both of which independently had reasonably good performance. Obviously, we could improve coverage by considering additional LOD sources or building more NER models. Yet another way to improve recall is to look at the general Web for additional type information. The idea, explored in previous work such as [9], is to capture *is-a* relations in text using Hearst Patterns encoded as queries to a Web search engine. For example, given the entity *IBM Websphere*, we can issue the following phrase query *"IBM Websphere is a"* to Google, parse the outputs looking for a noun phrase (NP) following the input query and use the NP as a basis for the type. There are two challenges here – recognizing type-phrases

that are conceptually similar (e.g., *'Application Server', "Web Server", 'Software Platform"*), which is typically done using clustering algorithms based on WordNet etc., and dealing with multiple type senses and figuring out the relevant type for a particular domain (e.g. *Java* could be a *Programming Language* or the *Island*). For the latter, we use our automatically discovered domain-specific types as filters. Initial results in using our type-discovery algorithm as an output filter to approaches such as [9] have been very promising – boosting recall to 75% for the Gartner case without altering precision, and we plan to further investigate this approach in the future.

6 Related Work and Conclusions

There are a number of attempts to define taxonomies from categories in Wikipedia, and map the classes to Wordnet (see [1], [10]), which address a different problem from inferring hierarchies from a relatively flat type structure like Freebase. Wu et al. [4] address the problem of creating a class hierarchy from Wikipedia infoboxes. Although their major focus is on detecting subsumption amongst these infobox classes, one aspect of their work is to infer ranges for infobox properties. For this, they examine what types of instances are referenced by these properties. This is related to what we do for type inference; however, they focus on inferring ranges for individual properties, whereas we use the domain and range information of all incident edges to infer types for instances themselves.

Research in NER has mainly focused on recognizing a fixed set of generic types, and little or no work has been done on recognizing a larger set of domain-specific types (as is our scenario). Alternately, there has been a lot of recent interest on relationship detection (e.g. Snowball [6], PORE [7]), and type detection can be seen as a special case of it (*is-a* relation). However, we differentiate ourselves in several ways. Like Snowball, we build text-patterns to represent the context. However, we obtain the appropriate training seeds automatically from LOD. Our patterns capture long-distance dependencies by not being limited to a fixed size context as in Snowball, and we add part-of-speech information to improve pattern quality. Also, both Snowball and PORE add type information to patterns, however Snowball uses an off-the-shelf NER, which suffers from granularity and PORE adds type information by looking at textual information on the Wikipedia page (e.g., Categories), which can be noisy and non-normative. As described in Section 4, our patterns contain precise type information (i.e. Yago Wordnet senses) from LOD for the precise-entity sense obtained by looking at Wikipedia URIs, and we generalize types by looking at the Yago-Wordnet type hierarchy. Finally, we have demonstrated that our patterns are generalizable across domains, a point not addressed in previous solutions.

On the broader problem of extracting semantic relationships from Wikipedia, Kylin's self-supervised learning techniques to extract a large number of attribute/value pairs from Wikipedia[11] has recently demonstrated very good results. Similar to our approach, the learning is performed on the structured information available in Wikipedia in the form of infoboxes. Our system differs

from Kylin in two important ways. First, our goal is to identify and extract only domain specific types, not all the values of the type attribute. Second, Kylin's technical approach assumes that the value of the attribute appears in the text used in the evaluation phase. In our two use cases, this assumption clearly does not hold for the type attribute. In fact, many of the discovered domain types were not mentioned in either the analyst reports or the energy articles.

In conclusion, we have shown that general-purpose structured information in Linked Open Data (LOD) combined with statistical analysis can be used for automated extraction of enterprise-specific vocabularies from text. We applied this idea to the IT domain looking at Gartner analyst reports, and automatically generated a vocabulary with 78% precision and 46% recall. As part of our solution, we have developed an algorithm to automatically detect domain-specific types for a corpus; a set of techniques to improve coverage of information in LOD. We have shown initial promising results for the Energy industy and are working on improving the solution coverage by leveraging the general Web.

References

1. Suchanek, F.M., Kasneci, G., Weikum, G.: Yago: A large ontology from wikipedia and wordnet. Web Semant. **6**(3) (2008) 203–217
2. Park, Y., Byrd, R.J., Boguraev, B.K.: Automatic glossary extraction: beyond terminology identification. In: Proceedings of the 19th international conference on Computational linguistics, Morristown, NJ, USA, Association for Computational Linguistics (2002) 1–7
3. Metaweb Technologies: Freebase data dumps. `http://download.freebase.com/datadumps/` (2008)
4. Wu, F., Weld, D.S.: Automatically refining the wikipedia infobox ontology. In: Proc. of 17th international conference on World Wide Web (WWW), New York, NY, USA, ACM (2008) 635–644
5. Welty, C., Murdock, J.W.: Towards knowledge acquisition from information extraction. In: Proc. of International Semantic Web Conf.(ISWC). (2006)
6. Agichtein, E., Gravano, L.: Snowball: Extracting relations from large plain-text collections. In: Proceedings of the Fifth ACM International Conference on Digital Libraries. (2000)
7. Wang, G., Yu, Y., Zhu, H.: Pore: Positive-only relation extraction from wikipedia text. In: Proc. of ISWC/ASWC2007), Busan, South Korea. Volume 4825 of LNCS., Berlin, Heidelberg, Springer Verlag (November 2007) 575–588
8. Chieu, H.L., Ng, H.T.: Named entity recognition with a maximum entropy approach. In: Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003, Morristown, NJ, USA, Association for Computational Linguistics (2003) 160–163
9. Cimiano, P., Staab, S.: Learning by googling. SIGKDD Explorations **6**(2) (DEC 2004) 24–34
10. Ponzetto, s., Strube, M.: Deriving a large scale taxonomy from wikipedia. In: Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI-07), Vancouver, B.C. (July 2007) 1440–1447
11. Wu, F., Hoffmann, R., Weld, D.S.: Information extraction from wikipedia: moving down the long tail. In: KDD. (2008) 731–739