RC24684 (W0811-033) November 4, 2008
Computer Science

IBM Research Report

Extracting Enterprise Vocabulary Using Linked Open Data

Julian Dolby, Achille Fokoue, Aditya Kalyanpur,
Kavitha Srinivas, Edith Schonberg
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Extracting Enterprise Vocabularies Using Linked Open Data

Julian Dolby
dolby@us.ibm.com
Achille Fokoue
achille@us.ibm.com
Aditya Kalyanpur
adityakal@us.ibm.com
Kavitha Srinivas
ksrinivs@us.ibm.com
Edith Schonberg
ediths@us.ibm.com
IBM Research
19 Skyline Drive
Hawthorne, NY 10532

ABSTRACT

A common vocabulary is vital to smooth business operation, yet codifying and maintaining an enterprise vocabulary is an arduous, manual task. We present a fully automated process for creating an enterprise vocabulary, by extracting terms from a domain-specific corpus, and extracting their types from LOD (Linked Open Data). We applied this process to create a vocabulary for the IT industry, using 58 Gartner analyst reports as a corpus, and the LOD subset consisting of DBpedia and Freebase. We present novel techniques for linking, cleansing, and extending the types in this LOD subset, resulting in an improvement of 55% for our IT domain results. We further improved our results through NER over the corpus. Our NER training is completely automated, exploiting Wikipedia and DBpedia. Altogether, we achieved 46.3% recall and 78.1% precision.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing

General Terms

Linguistic processing, Machine learning

Keywords

Linked Open Data, DBpedia, Enterprise vocabulary

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 an intranet or product catalog, and understanding general trends. However, building and maintaining a vocabulary manually is both time-consuming and error-prone. Coverage is a continual problem. It is never clear whether the vocabulary terms will keep up with common search terms. Terminology is continually evolving, so that maintaining a vocabulary is a game of catch-up.

This paper presents a process to fully automate the construction of a domain-specific vocabulary. The constructed vocabulary is a set of terms and types, where we label each term by its type. We applied this process to the IT (information technology) domain. For example, we discover that *IBM* is a term, *Company* is a domain-specific type, and the type of *IBM* is *Company*.

Our approach is based on two simple observations. First, vocabulary terms are typically embedded in the documents of an enterprise. We use 58 IT analyst reports from Gartner as our corpus for extracting terms. Second, 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. This is attractive, since repositories such as DBpedia associate entities with types, such as those from the YAGO and Wikipedia type hierarchy [1].

As a separate goal, we were interested in how useful LOD is for real problems. LOD contains billions of RDF triples and is growing at a rapid pace. Since there is no data validation process, the quality of the data is an open question.

For the IT industry domain, the most relevant subset of LOD is DBpedia and Freebase. We therefore focused on these two datasets as our reference data. Both datasets derive from Wikipedia; DBpedia is primarily the result of extracting the structured parts of Wikipedia (such as infoboxes), and Freebase is largely manually edited. We found three technical challenges with this subset of LOD that we addressed to improve it as a source for our domain-specific type extraction:

- The linked data sets are not really linked. Although Freebase and DBpedia are derived from Wikipedia, the mapping from Freebase to DBpedia is not straightforward. We describe how we linked these sets in Section 2[1].
- Both datasets have a certain degree of noise. Given the size of the dataset in question, we need automated techniques to discover the noisy parts of the data. We

[1]Freebase and DBpedia became linked very recently. Our work is still valid because it characterizes the nature of this mapping.

WWW2009, April 20-24, 2009, Madrid, Spain.

describe a set of novel techniques to automatically discover noisy data in Section 3, which relies on semi-automated extraction of ontological constraints, and the use of reasoning for noise detection.

- The coverage of LOD is weak. Even when the basic terms are found in LOD, their types are often missing. For instance, about 580K instances out of 2.2M Freebase instances are missing any type, even after the mapping to DBpedia instances and types. We describe a set of statistical techniques to automatically extract domain and range constraints on DBpedia properties, and apply it to *infer* types for instances in LOD in Section 4.

Domain-specific terms may have different senses in LOD. Many or all of these senses may be unrelated to the domain. In the vocabulary extraction process, we use statistical techniques to isolate domain-specific types found in LOD. These types are used to label corpus terms. Our LOD cleansing techniques improved the recall of this process by 55%. We improved recall further by performing NER (Named Entity Recognition) to the corpus, using the domain-specific types. There are several novel features of our NER algorithm. We train and build a statistical model completely automatically, using seeds generated directly from LOD, we exploit structural information in both Wikipedia and DBpedia to generate high quality contextual patterns (features) for the model, and we build an effective, general-purpose NER model that works well across different corpora. Our model was trained on Wikipedia and applied to the domain-specific corpus, the IT analyst reports. Finally, we achieved 46.3% recall and 78.1% precision. The vocabulary extraction process is described in Section 5.

2. LINKING FREEBASE AND DBPEDIA

To exploit information from Freebase and DBpedia to build our IT vocabulary, the first step was to link these two datasets. For DBpedia, we used data dumps for DBpedia 3.1. For Freebase, we used the WEX data dumps from July [2]. This combined dataset consists of 137 million RDF triples for DBpedia, and 116 million RDF triples for Freebase.

2.1 Linking DBpedia and Freebase instances

To link the Freebase and DBpedia data, we converted the WEX names for Freebase instances (guids) into DBpedia URLs by encoding names as URLs with the prefix `dbpedia:` and searched for their corresponding DBpedia entities. We were unable to match 4,946 Freebase instances out of 2.2 million instances in the WEX file. Manual checks revealed that these Freebase instances indeed did not have corresponding DBpedia instances. A somewhat surprising discovery was that the instance mapping from Freebase to DBpedia was often not one-to-one. Freebase often groups instances together that are semantically the same. For example, Freebase has a single instance (or guid in Freebase's terms) that maps to 29 different names, all of which are characters from Atlas Shrugged. DBpedia represents each character as two different instances with redirects to each other (e.g., `dbpedia:Dick_McNamara`, `dbpedia:Characters_in_Atlas_Shrugged23Dick_McNamara`). Of 2.2M Freebase instances, 2.19M instances mapped to a single DBpedia instance. Of the remainder, 47,403 instances mapped to two DBpedia instances. At the extreme, a Freebase instance was mapped to 106 DBpedia instances. As described in Section 3, errors do exist in this mapping, and they also increase the noise in the data. The output of this step is to augment the LOD dataset with `owl:sameAs` edges between mapped DBpedia and Freebase instances.

2.2 Linking DBpedia Redirects

In DBpedia, resource redirects are common (e.g., `dbpedia:International_Business_Machines` redirects to `dbpedia:IBM`). Semantically, this often means that the two resources are the same (similar to `owl:sameAs`), see [3]. There are often multiple levels of redirects (e.g., `dbpedia:Desktop_Wiki` redirects to `dbpedia:Microsoft_Windows`, and numerous other resources such as `dbpedia:HTML` redirect to `dbpedia:Desktop_Wiki`).

We first tried to treat all resources in the transitive closure of all redirect edge as the same resource, following the semantics of `owl:sameAs`. Unfortunately, taking the transitive closure of `dbpedia:Microsoft_Windows` resulted in 44,955 resources, with disparate resources such as `dbpedia:yoga` and `dbpedia:Microsoft_Windows` mapped to the same entity. It appears that `property:redirect` is used in multiple ways: when an entity is the subject of a single redirect relation, it indicates that information pertaining to this entity is the object of the relation. But when an entity is the subject of more than one redirect, this appears to indicate a collection of related but distinct entities. We therefore decided to add `owl:sameAs` edges between instances that are connected by a singleton redirect edge. This process of including redirects introduces some noise, when the redirects are noisy in DBpedia.

2.3 Linking DBpedia and Freebase types

Freebase and DBpedia specify types at different levels of granularity. For instance, Freebase has broad types such as `freebase:/business/company`, whereas DBpedia has specific types such as `yago:FoodCompaniesEstablishedIn1994`, `yago:InternetCompaniesEstablishedIn1996`. Although DBpedia does have a hierarchy that links these specific types to higher level types, such as *Company*, through subclass relationships, these relationships are not always reliably present, and are sometimes incorrect.

We chose to establish a mapping from the finer granularity DBpedia types to coarser Freebase types. This allows us to automate noise detection later, as described in Section 3, and to check our type inference as described in Section 4. To do this mapping, we computed the relative frequency with which a given DBpedia type A co-occurs with a Freebase type B, for all instances that were typed with A; i.e., the conditional probability $p(FreebaseType|DBpediaType)$. We considered a mapping valid if this conditional probability 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` maps to `freebase:/business/company` and `freebase:/business/employer`), we had 140,063 mappings with the 80% threshold. If we adopted a stricter

threshold and restricted ourselves to those types with a conditional probability of 1.0, we mapped 71,594 types, and had 102,833 mappings. Table 1 shows sample selected mappings. We augmented the LOD dataset with `owl:equivalentClass` linkages for these mappings.

Low conditional probabilities in mappings were caused by: (a) instances not having any Freebase type at all, (b) instances with wrong Freebase types, (c) instances with wrong DBpedia types. Table 2 shows an example of (c) where the type `yago:Editor110044879` is mapped to people and software at the instance level, because `yago:LinuxTextEditors` and `yago:AmericanMagazineEditors` are subclasses of `yago:Editor110044879` in YAGO.

3. NOISE DETECTION

Noise in LOD comes from three different sources. First, the entity categories in Wikipedia are sometimes wrong. For example, `http://en.wikipedia.org/wiki/United_Farmers_of_Alberta` has an incorrect category *Federal political parties in Canada*. This noise gets propagated to DBpedia which converts these categories into YAGO types. Second, these YAGO types automatically mapped to higher level entities in WordNet [1] with a noise factor of 5%. Finally, the extraction process from Wikipedia to Freebase is noisy. For instance, `freebase:winnie-the-pooh` refers to the fictional character *Winnie the Pooh* as a person.

Our main focus was to reduce noise from the type assertions for entities in LOD. Manually checking every type assertion is not practical, but reasoning can potentially help automate noise detection by finding assertions that are *logically inconsistent*. This approach has been used in prior work to identify noise in text extraction [4]. The core idea is to define a set of constraints in an ontology (e.g., a set disjoint types such as Person, Place etc.), and use reasoning to identify the inconsistencies in the data at the points where the constraints are violated. For example, if an instance a is declared as both a *Person* and a *Place* then the reasoner would identify the constraint violation involving a because this would produce a logical inconsistency.

The problem with applying this approach to Web scale data like LOD is that it is impossible to manually define these constraints for the number of types and properties in DBpedia. DBpedia has 159,379 YAGO types and 39,345 properties present in the data, and Freebase has 4,158 types. Alternatively, using a type hierarchy, it is possible to specify disjoints at the level of "roots" in the hierarchy manually, even when the hierarchy has many classes. However, LOD does not have a well-defined, clean hierarchy. 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. It is impossible to determine which types are disjoint in Freebase, since there are 4,158 types.

Our approach to solving this problem is therefore to first infer a type hierarchy for Freebase, and then use that hierarchy to define disjoint classes to detect noise.

3.1 Inferring a Type Hierarchy

Our type inference technique is based on whether certain relationships between types exist or not. The types in Freebase typically fall into broad groups, e.g. `freebase:astronomy/dwarf_planet` and `freebase:astronomy/comet/` both denote objects in space, and most types that are disjoint from one are disjoint from both. Therefore, we could infer that neither of these types is a subtype of the other. Furthermore, `freebase:astronomy/celestial_object` appears to encompass both of these types, thus being essentially a supertype.

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, X does not contain all the instances of Y. Thus, in a sufficiently large set of data, we would expect to see the following, where P denotes probability:

$$P\,(i \in Y | i \in X\,) == 1$$
$$P\,(i \in X | i \in Y\,) << 1$$

Because the Freebase data is noisy, the first probability will most likely be less than 1; furthermore, because some instances of Y are instances of X, it is unclear how low the second probability will actually be. To account for this, we can recast the above constraints as follows:

$$P\,(i \in Y | i \in X\,) > \tau$$
$$P\,(i \in X | i \in Y\,) < P\,(i \in Y | i \in X\,)$$

Thus, $X \subset Y$ if instances of X are almost always instances of Y, and the other way round is less likely. Therefore, we define the following:

$$P\,(X \subset Y) \equiv$$
$$\begin{cases} P\,(i \in Y | i \in X\,) & , P\,(i \in Y | i \in X\,) > P\,(i \in X | i \in Y\,) \\ 0 & , otherwise \end{cases}$$

Finally, this allows us to estimate a set of "root" types $\mathfrak{R}$, which are types at the base of the hierarchy. We define our notion of disjointness at this this level of types. The root types are simply those that are very unlikely to be a subtype of anything:

$$\mathfrak{R} \equiv \{T \,|\forall_X P\,(X \subset T) < \epsilon\}$$

Using this procedure, we found 78 roots, with τ set to .65 and ϵ set to .01. Of these, 26 roots had no subclasses, and the largest root had 409 subclasses. The examples below show a sample of roots and sample "subclasses" we found for classes relevant to the IT domain (URLs have been abbreviated for brevity). Note that Freebase allows user added types. System types are separated from user types, which tend to be more idiosyncratic.

- *Software*:
  - System Types:
    `/computer/software`
    `/computer/web_browser`
  - User Types:
    `/window_manager`
    `/relational_database_software`
    `/revision_control_system`

We turn now to the issue of declaring disjoint classes, so we can automate the detection of inconsistencies in the data. Note that, in principle, the sets of types under each root need not be disjoint. In practice, we found that 54 classes were classified as subtypes of multiple roots.

DBpedia Type	Freebase Type	$\frac{\#\text{Both Types}}{\#\text{DBpedia Type}}$	$p\left(\frac{\text{FreebaseType}}{\text{DBpediaType}}\right)$
yago:BirdsOfMexico	freebase:../default_domain/bird	1009/1261	0.800159
yago:Byway102930645	freebase:/transportation/road	377/471	0.800425
yago:BirdsOfIndia	freebase:../default_domain/bird	730/912	0.800439
yago:BirdsOfGuatemala	freebase:../default_domain/bird	285/356	0.800562

Table 1: Sample mappings between Freebase and DBpedia

DBpedia Type	Freebase Type	$\frac{\#\text{Both Types}}{\#\text{DBpedia Type}}$	$p\left(\frac{\text{FreebaseType}}{\text{DBpediaType}}\right)$
yago:Editor110044879	freebase:/people/deceased_person	1495/4585	0.326063
yago:Editor110044879	freebase:/book/author	614/4585	0.133915
yago:Editor110044879	freebase:/computer/software	189/4585	0.041221
yago:Editor110044879	freebase:/film/editor	186/4585	0.040567
yago:Editor110044879	freebase:/film/writer	176/4585	0.038386

Table 2: Excluded mappings between Freebase and DBpedia

3.2 Determining Disjointness

Deciding whether or not two types must be disjoint is not a process we felt should be automated completely. At some point, the understanding that a *computer program* and a *person* are not the same thing must be allowed to override any amount of statistical data that claims that they might be. So our approach is to group types under a small set of roots using the hierarchy techniques above, then group together related sets of root types, and then decide manually which of the sets of roots must be disjoint from each other.

We need to group the roots into sets due to noise. For instance, it might seem that `/astronomy/asteroid` ought to be a subtype of `/astronomy/celestial_object`; however, while almost all appropriate types under `/astronomy/celestial_object` are indeed subtypes of `/astronomy/celestial_object` with very high probability, `/astronomy/asteroid` is not, based on the data in Freebase. We decided manually that types under the root `/astronomy/asteroid` should be grouped with types under the root `/astronomy/celestial_object`. We denote this as a relation $S(T_1, T_2)$, which is true when we have grouped T_1 and T_2 together.

After grouping the roots, we are left with only 35 groups of types, and we adopted the default that these sets of types should always be pairwise disjoint unless we decide otherwise. That is defined as follows (where R_i is a root, T_j is a type):

$$\mathfrak{D}(T_1, T_2) \equiv \exists R_1, R_2 \left(\begin{array}{l} \{R_1, R_2\} \subset \mathfrak{R} \wedge \\ T_1 \subset R_1 \wedge T_2 \subset R_2 \wedge \\ \neg S(R_1, R_2) \end{array} \right)$$

In practice, we needed 5 pairwise exceptions for types that are in fact used together heavily and our fully automatic technique did not capture completely. In particular, the `/location/location` types are often used in Freebase for things that have a location but are not really locations themselves. So, we had to decide that a location is allowed to co-occur with certain other root types: `time/event`, `astronomy/celestial_object` and `business/employer`. But even in such cases our hierarchy simplified the task since we only had to define such compatibility for the groups of roots we identified in the system.

3.3 Noise Detection Results

We applied reasoning to detect noise in type assertions in DBpedia. To summarize our previous step, we obtained an hierarchy of 78 root types, and we manually coalesced them into 35 groups of disjoint types. This hierarchy covered a total of 1281 types out of the 4,158 Freebase types overall in the data.

We found a total on 2,246 Freebase entities with a declaration of inconsistent types, and we examined a random sample of 201 of them to determine causes. We found that most of them were caused by inconsistent type declarations in Freebase and DBpedia, where inconsistency was determined manually, e.g., an entity is not allowed to be a film and a person. The breakdown of errors in typing in our sample are as follows:

1 was caused by an error in the type mapping from DBpedia to Freebase described in Section 2.3. Statistics suggested that recipients of the Polish military honor *Virtuti Militari* were all people, but in fact the city of Verdun received the honor, and that caused a clash between person and city.

5 were caused by errors in the mapping between Freebase and DBpedia instances. These result from Freebase instances were mapped to multiple DBpedia entities with conflicting types, as described in Section 2.1.

43 were the results of our disjoint types being too aggressive. Most of these came from two sources. First, some types, such as `/location` occur along with many other types, but we required them to be disjoint from most groups. Second, types that we took to be subtypes of `/people/person` based on statistics did in fact have significant numbers of other uses, like `/award/award_winner` being used in conjunction with business/company.

152 were the results of declared types that we decided were correctly deemed inconsistent.

We ran the noise detection algorithm to get rid of the problem of aggressive disjoint types, and now found 2,111 instances with noisy explicitly declared types. If we include supertypes from the YAGO hierarchy along with the explicitly declared types and check for noise, we find 18,750 instances with noisy declared types, which is approximately 1% of the data. Given that this is dramatically more than

without the hierarchy, and our evaluation showed our type mapping is very good, it is likely that many of these extra clashes are caused by erroneous supertype edges. This technique can be used to debug the YAGO hierachy in the future.

4. COVERAGE

The technique we use to improve coverage extracts a fuzzy domain and range restrictions for properties in DBpedia, and uses reasoning to infer types for instances[2]. 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 a number of properties that are rather 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 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 a fuzzy notion of domains and ranges automatically to infer types, as discussed in the next section.

4.1 Type Inference

Our approach to type inference is based on correlating what properties an entity has with what 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.

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.

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

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

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 [5]; 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:

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

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:

$$V_{any}(i) \equiv \left|\left\{p \;\middle|\; \exists t \left(\begin{array}{l}\exists x\ p(i,x) \wedge I_{subj}(p,t) \vee \\ \exists x\ p(x,i) \wedge I_{obj}(p,t)\end{array}\right)\right\}\right|$$

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:

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

We applied this technique to our data with the following conditions to infer new types:

- We set τ and λ to be .5
- Inferred implied types were included only if they were not disjoint, as defined in Section 3.

We divided the implied types into 4 categories, and report all our statistics in terms of Freebase types (to avoid the problem that many DBpedia instances are redirects of each other, and are basically synonyms for the same entity):

1. *Verified* for the entities for which at least one of the implied types is the same as an explicitly declared one. This category had 808,849 instances.
2. *Additional inferred types* for the entities for which the implied 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.
3. *Invalid* for the entities for which at least one of the implied types conflicted with an explicitly asserted type. This category had 6,874 instances.
4. *Inconsistent* for those entities for which we inferred multiple types that were themselves disjoint, we made no inference. We had 5,955 instances in this category.

To determine the accuracy of our implied 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. The results for the two categories are provided below. 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.

On closer inspection, we realized that our technique worked exceedingly well for certain types, and not others. To illustrate this, Table 3 shows the top 5 Freebase declared types for instances in each category.

As shown in Table 3, both the invalid and the inconsistent instances had common top types involving *radio stations* and *fictional characters*. For *radio stations*, we tended to infer that they were *locations* instead, and for *fictional characters*, we tended to infer that they were real people. Given the relatively small number of types that led to many of our wrong inferences, we could investigate these specific types to determine which specific properties are responsible and adjust our inference appropriately.

[2]We don't consider the Freebase properties because when Freebase properties are specified, the types are specified as well, and there is no opportunity to infer types based on properties

Category	
type URI	count
Inconsistent	
freebase:.../fictional_character	654
freebase:broadcast/radio_station	635
yago:Weapon104565375	385
yago:FictionalCharacter109587565	374
freebase:broadcast/broadcaster	291
Additional	
yago:Landmark108624891	7446
freebase:music/composition	3775
freebase:music/song	3771
yago:City108524735	3564
yago:Village108672738	3405
Verified	
freebase:people/person	256920
freebase:location/location	216326
freebase:location/dated_location	124885
yago:LivingPeople	119010
freebase:location/citytown	118394
Invalid	
freebase:broadcast/radio_station	2473
freebase:broadcast/broadcaster	1412
yago:RadioStation104044119	1094
freebase:.../fictional_character	948
freebase:people/person	557

Table 3: Top 5 Declared Types in each Category

5. VOCABULARY EXTRACTION

We now return to our original goal, to automatically generate a domain-specific vocabulary. Our process extracts domain specific terms from a domain-specific 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. For the IT domain, we would like to 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". We rely on sources outside the corpus, in particular LOD, since appropriate types may not even appear in the corpus.

Figure 1 illustrates our process for creating a vocabulary. Briefly, our process performs the following steps:

1. Extract a population of terms from the domain corpus. This is the set of all noun phrases.

2. Extract a seed set of domain-specific terms from the corpus.

3. Apply statistical techniques to 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 type.

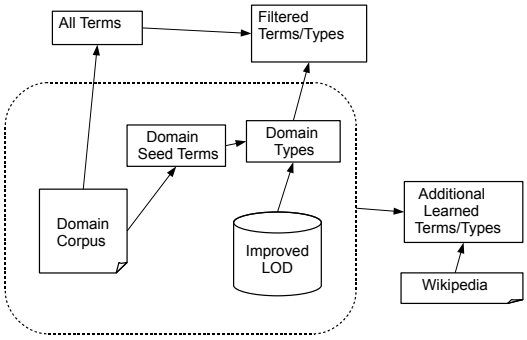

Figure 1: Vocabulary Extraction Process

5. Apply NER to the corpus to further improve coverage for the specific domain. The NER algorithm finds additional terms which belong to a subset of the types found in step 3.

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

5.1 Term Population

We extract the noun phrases from a domain corpus, using a standard NLP part-of-speech tagger (from OpenNLP[3]). These noun phrases comprise the population of all terms. For our case study, the domain corpus was 58 IT analyst reports from Gartner, available internally to IBM in the July timeframe. 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.

5.2 Domain-Specific Seed Terms

The next step is to extract an initial set of domain-specific terms from the corpus. These terms are subsequently used to look up types in LOD. 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, 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.

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

5.3 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 4. 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[4]) and check if it exists. This produced matching entities for 588 of the terms[5].

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 4) 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).

To further improve coverage in step 4, we also consider additional 'strongly related' types by looking at any associated Freebase-DBpedia type mappings and super-types from the type hierarchy.

Finally, note that filtering also removes several low frequency types that are interesting, e.g., `yago:XMLParsers`, `yago:ApplicationLayerProtocols`. 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).

We ran the type-discovery algorithm with 1137 seed terms. We did two separate runs. First, we ran it taking information from the original LOD, 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. The second time we ran it after improving the Freebase-DBpedia mappings, as described in Section 2, and improving the type hierarchy, as described in Section 3, again setting appropriate thresholds ($\alpha_Y = 12$, $\alpha_F = 12$, $\alpha_C = 4$) based on manual inspection. This produced 188 interesting types. Manually inspecting both sets of outputs showed very high precision ($> 90\%$) for each case.

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'/'super' types obtained via mappings and by looking at type hierarchy
(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 4: Extraction of Domain-Specific Types

5.4 Filtered Terms and Types

Once we have the set of domain-specific types, we revisit the entire population of corpus terms from Section 5.1. 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 5.2. 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.

We obtained two separate results at the end of this step as well, first considering LOD information as-is and taking the 170 interesting types found in the previous step. This produced 896 type-labeled terms. Then, we repeated the process taking the cleaned up version of LOD and the corresponding 188 interesting types found in the previous step. This gave us 1403 type-labeled terms, a substantial increase in the output (by approximately 500). A comparative evaluation of the precision/recall of the two outputs is discussed in Section 5.6.

5.5 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 5.1, 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

[4]Freebase naming convention, using *guid*s etc, makes it difficult to generate a potential URL for a term

[5]Alternately, we could do a keyword search for the term against LOD entity names and find closely-related matches but this risks matching unrelated entities, and thus adding a lot more noise

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, many of which are closely related, e.g., `yago:ComputerCompanies`, `freebase:venture_funded_company` and `Category:CompaniesEstablishedIn1888`, which are 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[6]. 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.

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. For example, 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 following pattern: [*X:NNP, announces, bid, to, buy,* <`yago:Company108058098`>] since "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

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 tagged token sequence $TK : [.., < word, POS >, ..]$
(2) Remove tokens in TK where the word is an adverb (RB), modifier (MD) or determiner (DT)
(3) Replace common nouns/verbs in TK with respective word *stems* using WordNet stemmer
(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 5: Pattern Generation Algorithm

[6]DBpedia naming convention provides a straightforward mapping to-and-from any DBpedia resource URL and the corresponding Wikipedia page URL

Scoring: Given type-outcomes $T_1..T_n$, and patterns $P_1...P_m$, compute a score for each pattern-type pair,

$$SC(P_i, T_j) = (n(P_i \cap T_j)/n(P_i)) \quad * \quad log_2(n(P_i \cap T_j))...$$
(Cond. Prob, using freq. n) X log(selectivity)
...(if $n(P_i) > 1$ and $n(P_i \cap T_j) > 0$)
(0 otherwise)

Decision Making: Given a set of patterns $IP_1..IP_o$ for an unrecognized named entity I
(1) Compute confidence $Conf(T_i)$ for each type T_i, $1 \leq i \leq n$
$Conf(T_i) = \sum_{1 \leq j \leq o} SC(IP_j, T_i)$
(2) Normalize $Conf(T_i)$ values into prob. dist. $\sum Pr(T_i) = 1$
(3) Guess type T_i if $Pr(T_i) >$ threshold (typically, .51)

Table 6: Scoring and Decision Making

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 7: Sample High Scoring Patterns

using type information, we also use a stemmer/lemmatizer (using a Java WordNet API[7]) 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 5.

After generating text patterns, the next key step is to score each pattern based on its *selectivity* (i.e., number of correct term-types pairs recognized) and *coverage* (quality of new term-type pairs detected). Assuming that the patterns are all independent, we compute a score based on the conditional probability of a pattern given a type from the frequency measures of positive and negative cases observed in the training data, and factor in selectivity based on frequency of positive cases. Also, to ensure that noisy patterns are not allowed in the model, we impose a pattern-frequency cutoff of 2. Finally, when making a decision, we take the sum of the pattern scores, normalize all type-outcome scores into a probability distribution and guess a type if it's normalized probability is greater than a certain threshold (.51 in our experiments). Details of our pattern scoring and decision-making scheme are in Table 6. Examples of highly selective patterns captured by our model are shown in Table 7.

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*). Note that 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 ($Pr(T_i) > 0.81$).

[7]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 8: Evaluating our NER model

5.5.1 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 5.4, and then manually fixing erroneous pairs. The results are shown in Table 8.

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 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. In this respect, 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 machine classifier based on MaxEnt or SVM.

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.

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 the previous step.

5.6 Evaluation of Complete Solution

The final output of our system is taken by aggregating the result of the procedure described in Section 5.4 which looks at LOD, and that generated by our NER model in Section

5.5. Since we did two experiments, first using the original LOD, and later with the cleansed version of LOD, we have two sets of results and we evaluated each separately.

For the original version of LOD, we found 1277 type-labeled terms in all (896 from LOD and 381 from NER). We evaluated the precision of both the terms and their types by manually evaluating a 200 term sample. This gave us a precision of 80.5%. We also computed recall by taking into account our gold standard estimate and the precision, and found that recall was 34.2%.

Taking the cleansed version of LOD, we found 1784 type-labeled terms in all (1403 from LOD and 381 from NER). The NER output was unchanged, since both experiments produced the same set of top-level YAGO/Wordnet types[8]. Using the same evaluation process, we found that precision dropped a bit to 78.1%, but recall increased substantially to 46.3%. This shows that our LOD cleansing improved our vocabulary extraction solution, without sacrificing accuracy.

6. RELATED WORK

There are a number of attempts to define taxonomies from categories in Wikipedia, and map the classes to Wordnet (see [1], [9]), which address a different problem from inferring hierarchies from a relatively flat type structure like Freebase. Wu et al. [5] 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.

Researchers have looked at extracting ontologies from text (see Hearst [10], KnowItAll [11]), however these techniques extract types appearing in the text itself, whereas we focus on obtaining type-information from LOD instead.

On the topic of NER, research 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 5.5, 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.

[8]Given that we now produce cleaner type-instance information from LOD, we plan to use it to build more NER models for types besides YAGO/Wordnet. Due to time constraints, we were unable to do so at the time of writing.

Acknowledgements

We would like to thank Jim Carolan, of the Strategic Technology Group at Gartner for his assistance and participation in the development of this process. His expertise within the IT research domain has helped direct and validate our findings.

7. 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] Metaweb Technologies: Freebase data dumps. `http://download.freebase.com/datadumps/` (2008)

[3] Wu, F., Weld, D.S.: Autonomously semantifying wikipedia. In: CIKM '07: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management, New York, NY, USA, ACM (2007) 41–50

[4] Welty, C., Murdock, J.W.: Towards knowledge acquisition from information extraction. In: Proc. of International Semantic Web Conf.(ISWC). (2006)

[5] 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

[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 (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] 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. (2007) 1440–1447

[10] Hearst, M.A.: Automatic acquisition of hyponyms from large text corpora. International Conference On Computational Linguistics (1992)

[11] Etzioni, O., Cafarella, M., Downey, D., Popescu, A.M., Shaked, T., Soderland, S., Weld, D.S., Yates, A.: Unsupervised named-entity extraction from the web: an experimental study. Artif. Intell. **165**(1) (2005) 91–134