New York University KBP 2010 Slot-Filling System

Ralph Grishman and Bonan Min
Computer Science Dept.
New York University

grishman@cs.nyu.edu min@cs.nyu.edu

New York University fielded a small KBP (Knowledge Base Population) slot-filling system for the 2010 TAC (Text Analysis Conference) evaluation. Our goals were

- as one of the task coordinators (R.G.), to immerse ourselves in the details of the task
- to provide a high-precision (if low-recall) system for use in system combination with slot-filling systems being developed by the City University of New York (CUNY)
- to assess the effectiveness of pattern bootstrapping combined with manual review

System Description

Document retrieval

The entire text corpus was indexed using the Lucene document retrieval package[1]. As the first step in processing a query, Lucene was invoked to retrieve the documents containing the query name; a maximum of 500 documents were retrieved. No attempt was made to use the document associated with the name in the query in order to disambiguate possibly ambiguous names. As a minimal acknowledgement of possible name variation, if a person entity query name contained an initial or an organization entity query name contained a corporate suffix (“Corp.”, “Ltd.”, etc.) an alternate name was created omitting these items, and we retrieved documents with either the original or alternate name.

Document preprocessing

The retrieved documents are pre-processed using JET, the Java Extraction Toolkit, a large integrated package of language annotation tools[2] based on the Tipster Architecture.[3] The preprocessing steps include

- tokenization

[1] http://lucene.apache.org/java/docs/index.html
[2] http://cs.nyu.edu/grishman/jet/license.html
[3] http://cs.nyu.edu/grishman/tipster.html

- lexicon look-up coupled with part-of-speech tagging
- name tagging, using the ACE set of named-entity tags
- noun group and verb group chunking, coupled with a limited assignment of syntactic structure within the groups
- recognition of a few larger syntactic units, including in particular appositional structures
- recognition and normalization of time expressions based on the TIMEX2 standard
- coreference (for names, nouns, and pronouns)

In parallel, we ran the Stanford parser[4] and generated Stanford dependency trees; these were used for the pattern matching, described below.

Picking some low-hanging fruit

For a typical set of entities, the non-NIL slot fills are dominated by a few slot types, including *title*, *employee_of*, and *top_members/employees*. Furthermore, many instances of these and a number of other slots can be captured using only coreference and a few very local syntactic relations: pre-nominal titles ("General Smith"), nouns ("linguist Fred Smith"), names ("Ford president Smith"), possessives ("Ford's president Smith"), and *x of y* structures ("president of Ford"). Preprocessing already provided coreference and these relations. This allowed us to quickly hand-code rules for common cases for person entities of

- title
- employee_of
- family slots (parent, spouse, children, other_family)
- origin

and for organization entities of

- top_members/employees
- parent
- subsidiary

The basic strategy is to find all the mentions of the queried entity, and then find a KBP--relevant pattern linking one of those mentions to another mention. If that other mention is co-referential with a name, record that name as a slot value.

There is a special case for corporate titles: if a title is not explicitly linked to an organization (just "president", not "Ford president"), but the queried organization is the only one mentioned in the current or previous sentence, then the system assumes that it is an implicit reference to that organization.

We used a list of about 160 titles and occupations for the formal evaluation. This was subsequently expanded to 650 items by collecting and manually reviewing all

[4] http://nlp.stanford.edu/software/lex-parser.shtml

the Wikipedia infobox entries. The list distinguished top employees from other title entries.

Since the Jet coreference module already identifies coreferential name variants, including acronyms, we use this information directly for the *alternate_name* slot.

Pattern matcher

In addition to the approach just described, based on a small set of local syntactic relations, we implemented a more general pattern-matching procedure for identifying entity—slot value pairs. Each pattern is associated with a particular slot type.

We provided two types of patterns: word sequence patterns and dependency path patterns. A word sequence pattern is the middle context between an entity-value pair. For example, *", also known as*" is a suitable pattern for *alternate_name* slots*.* We use word sequence patterns for both entity-value and value-entity pairs. The latter are used for pairs in which the slot value precedes the entity mention. For example, the value-entity pattern "- based" can be used to identify the headquarters location of an organization entity ("New York-based General Electric"). A dependency pattern consists of alternating words and dependency labels, and matches the shortest dependency path between an entity-value pair. Dependency structures are generated using the Stanford dependency analyzer. For example: *"nsubjpass-1 known prep_as"* is a dependency pattern for entity type *person* and slot *alternative names* (*nsubjpass-1* represents an inverse arc (from dependent to head) labeled *nsubjpass* (passive subject)). We use breadth-first search to find the shortest path between entity-value pairs in the dependency representation. Having both types of patterns provides the generality of using syntactic structures (the ability to ignore intervening modifiers, for example) while still being able to capture some values despite syntactic analysis errors.

As noted above, coreference analysis as part of document preprocessing identifies all mentions of the query entity; starting from each mention, the system then seeks to match all word sequence and dependency patterns. If the pattern matches and the value at the end of the pattern is of the correct type for the given slot (for example, an integer for *per:age* or a person name for *per:spouse*), it is accepted as a value for the slot, subject to the postprocessing described below.

Location slots in 2009 KBP were divided into city, state/province, and country slots in 2010. To handle this split, patterns were associated with generic 'location_of' slots (e.g., *location_of_birth, location_of_headquarters*). A postprocessing step then relabels the value with a specific 2010 slot name (e.g., *city_of_birth, state/province_of_birth, country_of_birth*) using a small gazetteer with the names of all countries and U.S. states; names not in the gazetteer are assumed to be cities.

Bootstrapping for pattern acquisition

We use a pattern bootstrapping procedure to grow the pattern set for extracting slot fillers. The bootstrapping procedure is similar to the one described in *Snowball* [Agichtein and Gravano 2000] except that our system is provided with a handful of seed patterns initially rather than seed pairs. After several iterations of bootstrapping, the system is able to generate hundreds of patterns that can provide wide coverage when these patterns are applied to extract slot fillers. The pattern bootstrapping procedure is illustrated in the following picture:

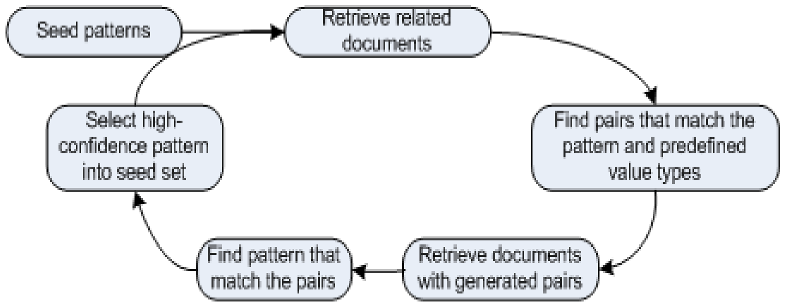
Seed patterns
Retrieve related documents
Find pairs that match the pattern and predefined value types
Retrieve documents with generated pairs
Find pattern that match the pairs
Select high-confidence pattern into seed set

Figure 1. Pattern Bootstrapping procedure

Our pattern bootstrapping procedure starts from a handful of high-precision seed patterns, and then we run the bootstrapping procedure (as shown in Figure 1) for several iterations to grow the pattern set. We use only word sequence patterns as the initial seed. For example, *", also known as"* is a word sequence pattern for an entity of type *person* and its value *alternative nam*es. By using high precision patterns rather than seed pairs, we're able to generate more pairs for training than using seeds pairs directly.

We use a ranking scheme similar to *Snowball* to approximately rank patterns and pairs for manual post-editing. Our goal is not necessarily to provide an accurate pattern-ranking algorithm but rather to give the editor a rough measure of how effective a pattern is for extracting correct slot fillers; then editors can manually select high confidence patterns for later iterations.

The confidence of a pattern is defined by

$$Conf(P) = \frac{P.positive}{P.positive + P.negative}$$

where $P.positive = \sum_{i=1}^{m} Conf(T_i)$. Here m is the number of times pattern P matches a pair that is in the seed set plus the number of times it matches a pair in the set of pairs extracted from previous iterations, and *Conf*(T_i) is the confidence of the pair. *P.negative = n·β*, where n is the number of times P matches a pair that doesn't appear in the correct pairs set, and β is the penalty factor for matching an incorrect pair (we set $\beta = 1.0$ for convenience).

The patterns are ranked by

$$Conf_{R\log F}(P) = Conf(P) \times \log_2(P.positive)\,.$$

The confidence of new entity-value pairs (tuples) is computed by

$$Conf\,(T) = 1 - \prod_{i=0}^{|P|}(1 - Conf\,(P_i))\,.$$

The initial confidence values of all seed patterns are set to 1.0.

The accuracy of pattern confidence produced by *Conf(P)* depends on the completeness of the correct pair set for each slot. Given an incomplete set of correct pairs, this confidence function will only underestimate the confidence for some patterns for which we don't have enough pairs in the set of correct pairs. A higher-ranked pattern will still have a higher precision for matching entity-value pairs. Since our goal is to rank patterns for post-editing the editor only needs to manually inspect the low ranked patterns and remove some of the patterns that are not very accurate for extracting correct slot fillers. In practice, we found this ranking scheme provide good ranking for post-editing.

The bootstrapping procedure starts with 34 patterns (1 or 2 patterns for each slot). We run the bootstrapping procedure for 3 iterations. Then we manually inspect all extracted patterns and we keep 970 patterns that we believe to be highly accurate; of these, 443 patterns are word sequences patterns (326 patterns for entity on the left and 117 patterns for entity on the right) and 527 are dependency patterns. Following are a few examples:

Word sequence pattern (L*)	Word sequence pattern (R*)	Dependency patterns
, based in (org:location_of_ headquarters)	*- based* (org:location_of_ headquarters)	*nsubjpass-1 headquartered prep_in* (org:location_of_ headquarters)
was founded by (org:founded_by)	*, the founder of* (org:founded_by)	*nsubjpass-1 founded agent* (org:founded_by)
, died of (per:cause_of_death)	*that eventually killed* (per:cause_of_death)	*advmod-1 killed agent* (per:cause_of_death)

Table 1. Sample patterns produced by pattern bootstrapping.
(Slot name in parentheses.)
* L: entity name precedes value, R: entity name follows value.

Postprocessing

Information to fill a particular slot of an entity may appear in many documents about that entity and multiple times within a single document. Furthermore, the values may be different but equivalent, in which case only one should be reported.

In the case of single-valued slots, we prefer a value reported more than once over a value reported once (this reduces the chance of taking a value which is a typographical error), and otherwise we prefer the longer value (thus preferring a complete name over a partial one).

In the case of list-valued slots, two values *a* and *b* are considered redundant if

> *a* and *b* are identical after removing all blanks, hyphens, initials, and corporate suffixes, *or*
>
> one string is a substring of the other, *or*
>
> Levenshtein distance(*a, b*) <= 2

If two values are redundant, only one is retained, using the same preference rule applied to single-valued slots.

In addition, for list-valued slots we must suppress values which already appear in the reference Knowledge Base; for our system, this task was performed by a module provided by CUNY.

Results

The NYU system operating by itself got a precision of 54.3% and a F measure of 19.1% (Table 2). While simple in design, it met our goal of providing a high-precision component for system combination with CUNY systems. Two additional runs submitted by NYU (as well as two of the runs submitted by CUNY) involved system combination of the NYU system and three CUNY systems; these are summarized in Table 2 and described in the CUNY KBP paper (Chen et al. 2010).

	Precision	Recall	F
NYU1 (NYU system alone)	54.3%	11.6%	19.1%
NYU2 (CUNY-NYU system comb.)	20.0%	20.7%	20.3%
NYU3 (CUNY-NYU system comb.)	28.0%	26.0%	27.0%
CUNY-BLENDER2 (CUNY-NYU sys. comb.)	22.6%	24.1%	23.3%
CUNY-BLENDER3 (CUNY-NYU sys. comb.)	28.7%	27.9%	28.3%

Table 2. Official KBP slot-filling runs using NYU system.

References

Eugene Agichtein and Luis Gravano. Snowball: extracting relations from large plain-text collections. *DL '00: Proceedings of the fifth ACM Conference on Digital Libraries*, 2000, 85-94.

Zheng Chen, Suzanne Tamang, Adam Lee, Xiang Li, Wen-Pin Lin, Matthew Snover, Javier Artiles,Marissa Passantino and Heng Ji. 2010. CUNY-BLENDER TAC-KBP2010 Entity Linking and Slot Filling System Description. *Proc. Text Analysis Conference (TAC2010).*