English Slot Filling with the Knowledge Resolver System

Hans Chalupsky
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA, USA
hans@isi.edu

Abstract

This paper describes the Knowledge Resolver system (KRes) and its performance on the TAC-KBP 2013 English Slot Filling task. KRes is a logic-based inference system aimed at improving statistical relation extraction by deduction and abduction inference towards the best document-level interpretation. For the 2013 evaluation we developed an initial KRes system that extracts a subset of seven TAC-KBP relations using manually constructed dependency patterns in concert with entity type and name-linking rules. For our baseline extraction engine we used the Blender Lab's KBP-Toolkit 1.5, which was also exploited at the front-end of KRes for its document indexing, selection and name expansion capabilities. Instead of trying to improve upon KBP-Toolkit results using inference, for this year we simply combined its results with those of KRes for our best system which landed us in the middle of the pack (only addressing 13 out of the 40 KBP slot types). We also report results for KRes relativized to the seven slot types it addressed which shows promise for future evaluations.

1 Introduction

This paper describes the Knowledge Resolver system (KRes) and its performance on the TAC-KBP 2013 English Slot Filling task. KRes is a logic-based inference system based on the PowerLoom knowledge representation and reasoning system, aimed at improving statistical relation extraction by linking extractions from across a section or whole document, and then using abduction to combine alternative extractions into the best document-level interpretation. We call this *story-level inference* which we have applied successfully for relation extraction and question answering in a sports domain (Chalupsky, 2012). We are currently generalizing this approach for more open domains such as the ones targeted by TAC-KBP.

This was our first participation in TAC-KBP and we are only partly along the way towards achieving our overall goal. For the 2013 evaluation we developed an initial KRes system that extracts a subset of seven TAC-KBP relations using manually constructed dependency patterns in concert with entity type and name-linking rules. We used Stanford's CoreNLP system for dependency parsing, coreference resolution and NER-typing as well as title lists we mined from Gigaword 4. For our baseline extraction engine we used the Blender Lab's KBP-Toolkit 1.5, which was also exploited at the front-end of KRes for its document indexing, selection and name expansion capabilities. Instead of trying to improve upon KBP-Toolkit results using inference, for this year we simply combined its results with those of KRes for our best system which landed us in the middle of the pack (only addressing 13 out of the 40 KBP slot types). Relativizing results to the slot types addressed by KRes shows a significantly improved picture and gives us promise for this approach for future evaluations if we cover a larger number of slots.

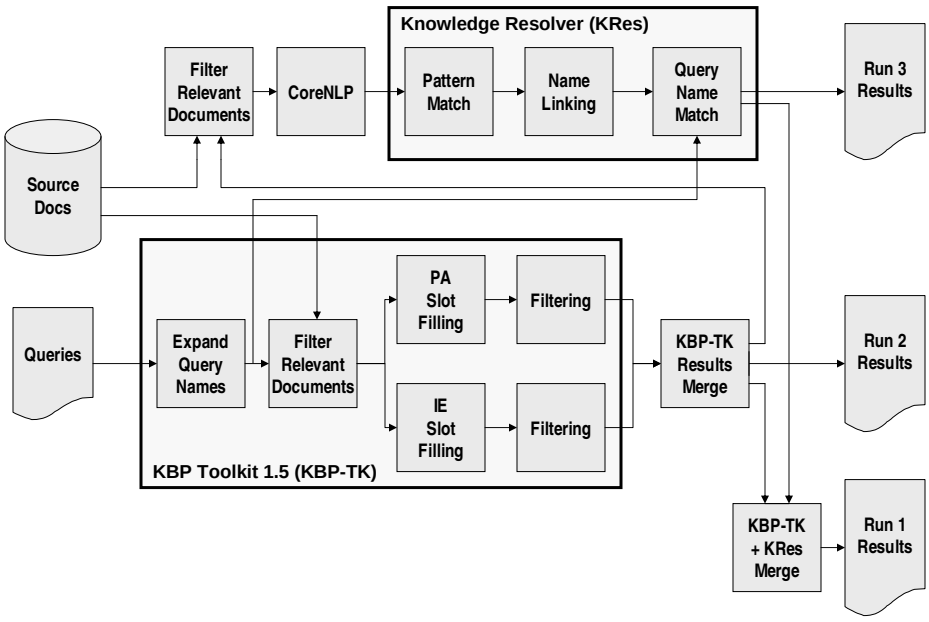

Figure 1: System architecture

2 Approach

Figure 1 shows the overall architecture of our TAC-KBP system. We have two mostly independent pipelines: (1) an extraction baseline built upon the KBP Toolkit 1.5 from the CUNY (now RPI) Blender Lab.[1] (2) The Knowledge Resolver pipeline which uses Stanford's CoreNLP 1.3 toolkit[2] for all core language processing tasks, and then applies dependency pattern matching and name linking as well as type inference to extract KBP slots. KRes exploits KBP-TK in two important ways: first, it is only applied to documents that are sources for any of the slots extracted by the KBP-TK which greatly reduces processing time and automatically focuses in on query-relevant documents. Second, it exploits KBP-TK's name and acronym expansion tool for its query name matching component. For our best Run 1 result, we combine the results of both pipelines in a simple, non-intelligent merge. We describe these systems in more detail below.

2.1 KBP Toolkit Pipeline

The KBP-TK provides tools for both entity linking and slot filling. A variant of the toolkit was used in the TAC-KBP 2010 evaluations (Chen et al., 2010). We only used its slot filling tools which are comprised of a pattern-based QA pipeline (labeled "PA Slot Filling" in Figure 1) derived from the OpenEphyra question answering system, and an IE-based pipeline (labeled "IE Slot Filling") using NYU's Jet system trained on ACE data (for relations that are similar but do not match exactly the TAC-KBP relation set).

The toolkit starts with a name-expansion step that processes the names of the evaluation queries and adds variants by dropping first name and/or middle initials, substituting nicknames and hypothesizing acronyms for organizations. The set of query names and their expansions is then fed to Lucene to filter potentially relevant documents from the source corpora. On the 2,000,000 news and web documents of the 2013 TAC-KBP source corpora,, these steps selected about 9,000 documents with potentially relevant matches. The filtered document set is then run independently through a pattern-based (PA) and an IE-based (IE) slot-filling pipeline. Both pipelines have respective filtering steps and produce a final results file each for query-relevant slots and their values.

We followed the manual included with the software and used the KBP-TK more or less out of

[1] http://nlp.cs.qc.cuny.edu/kbptoolkit-1.5.0.tar.gz

[2] http://nlp.stanford.edu/software/stanford-corenlp-full-2013-06-20.zip

the box without any retraining or reconfiguration. We had to make some minor adjustments to account for some changed slot names (e.g., map `per:member-of` and `per:employee-of` onto a single `per:employee-or-member-of`), and to compute the newly required document offsets for relation provenance. Unfortunately, our offset computation tured out to be incorrect, since it was done on improperly detagged documents where SGML tags were simply deleted instead of replaced by whitespace. When we caught the mistake, it was too late to change our submission. It is unclear how significantly our evaluation results were affected by this problem, since the severity depends on the length of a document.

As configured, KBP-TK extracts values for these eight slots:

```
per:cities_of_residence
per:countries_of_residence
per:statesorprovinces_of_residence
per:employee_or_member_of
per:spouse
per:title

org:member_of
org:top_members_employees
```

We did not perform any experiments to see whether opening this up to the full set would have been productive. Since KBP-TK is merely a baseline and our KRes pipeline currently only extracts a subset of the TAC-KBP slots, having a larger set of baseline slots unrelated to anything in the KRes pipeline would have not generated any useful insights.

To the best of our knowledge, the version of the KBP-TK we were using does not include a component to merge the results of its pattern-based and IE-based pipelines. We therefore built a simple merge algorithm that combines the results of both pipelines, ensures single-valued slots are only reported once, normalizes confidences of the two pipelines onto 0-1 intervals and computes document offsets for the arguments of a relation (with the caveat mentioned above that these offsets were based on incorrectly detagged documents). Offsets were recovered by matching slot argument strings to the context sentence provided by KBP-TK. This was slightly tricky for cases where a hypothesized query name acronym was used as the argument which then had to be linked to the full query name. When we failed to recover argument offsets for acronym arguments, we simply dropped the associated slot value from the result, since manual inspection on development data showed that those were generally incorrect.

2.2 Knowledge Resolver Pipeline

Our main effort consisted of the development of the Knowledge Resolver system (or KRes). KRes is a logic-based inference system based on the PowerLoom knowledge representation and reasoning system,[3] aimed at improving statistical relation extraction through story-level inference. This process links extractions from across a section or whole document and uses abduction inference to evaluate alternative extractions towards the best document-level interpretation. See (Chalupsky, 2012) for an application of this type of inference in the domain of sports news. We are currently generalizing this approach for more open domains such as the ones targeted by TAC-KBP.

This was our first participation in TAC-KBP and we are only partly along the way towards achieving our overall goal. At this time the relation extractors available to us did not have the coverage and quality necessary to drive the document-level KRes inference. For this reason, we developed a small set of high-precision relation extractors that can support this inference in the future. The main approach taken was to detect relations based on manually formulated dependency tree patterns augmented with type information coming from CoreNLP, word lists (e.g., for titles) and specialized relation dictionaries (e.g., for family relations).

KRes uses the Stanford CoreNLP 1.3 toolkit[4] for all core language processing tasks such as tokenization, POS-tagging, sentence detection, NER-typing, dependency parsing and coreference resolution. CoreNLP annotations (such as sentences, mentions, NER-types, parse trees, etc.) are then translated into a logic-based data model for the PowerLoom knowledge representation and reasoning system. We use an extended version of PowerLoom (compared to the publicly released version), that im-

[3] http://www.isi.edu/isd/LOOM/PowerLoom/

[4] http://nlp.stanford.edu/software/stanford-corenlp-full-2013-06-20.zip

plements a variety of extensions relevant to NLP such as an extensive data model to represent text annotations, logic-based access to word lists and dictionaries, dependency tree matching, fuzzy string matching, various annotation translators, range indices for efficient annotation inclusion inference, and a number of other utilities. These extensions allow us to run the whole dependency pattern matching, inference and result generation process via the PowerLoom inference engine.

For TAC-KBP, KRes makes use of the KBP-TK document indexing and name expansion facilities. After the KBP-TK pipelines were run, KRes was only run on the subset of documents referenced in any of the KBP-TK results. This cuts down on processing time and also gives a better chance on selecting relevant documents only, since some of the query names are ambiguous. Of the about 9,000 documents selected in KBP-TK's initial document filtering step, only about 700 yielded results reported in KBP-TK's result files. Once relations are extracted, arguments are mapped onto query names in a name matching step that exploits the name variants generated by KBP-TK. These dependencies are depicted with two arrows from KBP-TK modules to KRes modules in Figure 1.

2.2.1 Dependency Pattern Matching

After processing a document through CoreNLP, a first phase would find potential relation matches based on various dependency patterns. Due to time constraints, we only developed patterns for seven of the TAC-KBP relations, age, family relationships and titles:

```
per:age
per:children
per:other_family
per:parents
per:siblings
per:spouse
per:title
```

These accounted for about one third of all answers in past evaluations and do not seem to be addressed very well by KBP-TK. To develop the requisite patterns, we collected relevant example sentences from the outputs of IBM's SIRE relation extractor (the outputs but not the extractor were available to us from participation in DARPA's MRP program). These were then manually inspected, corrected and augmented and used for pattern development. We also mined about 4,300 titles from Gigaword 4 using a simple pattern-based approach with manual inspection for quality control, and about 2,800 job titles from the CareerBuilder web site.

Dependency tree patterns were represented as PowerLoom list terms which were then interpreted by a pattern evaluation predicate. For example, the following pattern would match simple possessive constructs such as "John's father":

```
(listof dg-poss ^ family-relation-word)
```

Elements in the pattern can be dependency graph edge labels (such as `dg-poss`), named or unnamed PowerLoom relations (such as `family-relation-word` which accesses a small special-purpose dictionary for indicators of family relations), verbatim strings (such as prepositions) or the special root anchor constraint indicated by `^`. This pattern would then match the path from "John" to "father" in the following example dependency tree for "There he met John's father Frank":

```
(There
  RCMOD (met
          NSUBJ he
          XCOMP (Frank
                  NSUBJ (father
                          POSS John))))
```

Here is a more complex age apposition patterns that would match phrases such as "Hussein Ali Awad, 30, ...". The `kappa` construct is used to define an unnamed PowerLoom relation which can be used in patterns to match more complex scenarios such as context tokens that are not directly part of a relation dependency path, in this case, a punctuation token following the age expression:

```
(listof dg-nn ^ dg-appos
        (kappa (?t ?p ?g)
          (and (possible-age-word ?t ?p ?g)
               (punctuation-token
                 (token-next-token ?t)))))
```

Using this approach, we defined ten patterns for age relations, eleven patterns for family relationships and nine for title relations. A set of leveled type constraints were used to restrict elements in these patterns. For example, an `organization-word` constraint would require the NER type of a token to be "organization". A

`plausible-organization-word` would relax this to also include NER types of "location". Finally, a `possible-organization-word` would allow NER types of "organization", "location", "misc", or "other" with the additional constraint that it had to be a noun-like construct.

CoreNLP only detects a small set of twelve NER types that does not include titles. To be able to detect title expressions, we mined about 4,300 titles from Gigaword 4 and about 2,800 job titles from the CareerBuilder web site. These were then normalized, stemmed and matched in a fairly relaxed way to extend this dictionary as much as possible. For example, if we had entries for "shop keeper" and "store owner" but not "shop owner", we would also match "shop owner" since it could be constructed by combining head and tail of two existing entries.

Documents were then pattern-matched with queries such as the one below, that retrieved all sentences and their dependency trees, matched them for patterns as done for age patterns below, and then asserted any matches so that they could be further analyzed and refined in subsequent steps:

```
(assert-from-query
 (retrieve all (?g ?p ?x ?y)
   (and (sentence-dependencies ?s ?g)
        (age-relation-pattern ?p)
        (dep-pattern-match ?g ?p ?x ?y)))
 :relation dep-pattern-match-result)
```

Similar materialized queries would handle family and title relations. A matched pattern does not necessarily mean the relation holds, since an argument might link to multiple candidates in a sentence via different patterns (this often happens for complex conjunctive lists). Subsequent steps described below would then select between such alternative matches by linking to names and/or selecting the closest candidate argument for a match.

2.2.2 Name Linking

Answers to TAC-KBP queries always involve at least one named argument, the query name, and possibly a second named argument for the slot value as is the case for family relations. Identifying names and linking them to relation arguments is therefore a central part of the task. We chose to separate relation detection and name linking into two separate phases. For example, when processing the sentence "After John left his wife, Susan, ..." we would first detect the spouse relation between "his" and "wife" and then link those arguments to their respective names "John" and "Susan" via pronoun coreference and an apposition pattern. Decoupling these steps instead of performing them in a single pattern match allows us to reuse the same mechanism across several patterns. It also makes the name linking step explicit which allows us to more easily perform additional inference when choosing between alternative candidate arguments.

Name links are established by linking a relation mention argument such as "wife" in the example above to the head token of a named mention of the appropriate NER type such as "person". Named mentions are detected by CoreNLP's deterministic coreference engine. We use those mentions but additional refine them by "tightening" them, since they might contain entire relative clauses or other descriptive phrases. In such cases we look for the mention head and then find the largest set of consecutive tokens around it that has the same NER type. Moreover, we detect additional hidden named mentions by looking for named token sequences with a non-trivial NER type that are not part of any known named mentions.

Given a relation argument x and the head n of a tight named mention, we look for a link between the two tokens as follows: we first check whether $x = n$, and, if that fails, for a connection via one of these dependency edges in this preference order: NN, APPOS, NSUBJ and DEP. Once a connection is found a name link is established which will prevent a name token to be linked to any other relation arguments in a sentence. In the next step, relative pronouns are resolved to any named referents via coreference links detected by CoreNLP. In the final step, we link personal pronouns to their referents via coreference. Given that we were focusing on family relations, and that CoreNLP's coreference engine has a weakness with overmerging named strings that have the same postfix (such as a last name shared by multiple family members), we restricted personal pronoun resolution to within a single sentence only, to avoid being hit by this problem (at the cost of some recall).

2.2.3 Slot Filling

In the final phase, we use the pattern matches and name links established in prior phases to extract TAC-KBP slot values for named mentions found in a document. We additionally perform some value normalization here (e.g., to normalize age values or handle multi-element arguments), and we map relation types such as "cousin" onto the appropriate TAC-KBP slot (such as `per:other_family`). We also perform some simple inference to handle inverse slots, e.g. infer `per:parents` from `per:children` and vice versa.

Finally, we process all slots found in a document and see whether they answer any of the given evaluation queries. For each named slot argument, we test whether it matches any of the query names or their variants as determined by KBP-TK's query name expansion tool via a Jaro fuzzy string match with a 0.9 threshold. If the argument name only had a single content token (e.g., a person's last name), we look for some additional match evidence to increase the probability of a correct match. For this we additionally test whether any of the variants associated with the query name contains a token different from the slot argument token that does appear somewhere in the document (such as a person's first name). Despite the simplicity of this mechanism, this seems to eliminate most spurious matches.

At this point we generate a KBP results file according to the constraints given by the slot and query definitions. Single-valued slots are given a single result value only. Since we do not have any confidence values at the moment, we choose one value at random. Ignored slots are suppressed. Otherwise, each relation mention found in a document that supported a slot value was output as a result. We did not address obviously redundant slot values resulting from this strategy which significantly impacted our precision score. In the evaluation section below, we show how a simple redundancy elimination scheme would improve our overall score.

3 Evaluation Results

Each run extracted slot fillers only from the TAC KBP 2013 source documents (only news and web corpora were processed, the discussion fora were ignored). No other external resources were used. Neither the KBP-TK nor the KRes pipeline runs did make use of the information in the reference knowledge base (the KBP-TK might have used the reference KB during the training and pattern learning phase).

We submitted three runs: a KBP-TK-only baseline (Run 2), a KRes-only baseline (Run 3) and a union of the two (Run 1). The Run 1 union was a simple non-intelligent merge similar to the merge of the PA and IE pipelines of KBP-TK. Results of these runs are summarized in Table 1.

Run ID	R	P	F1
KRes (Run 3)	0.088	0.277	0.133
KBP-TK (Run 2)	0.078	0.122	0.096
KBP-TK + KRes (Run 1)	0.150	0.157	0.153
KRes unique	0.087	0.490	0.148
KBP-TK unique	0.078	0.126	0.097
KBP-TK + KRes unique	0.146	0.190	0.166
KRes relative	0.318	0.490	0.386
KBP-TK relative	0.176	0.126	0.148
KBP-TK + KRes relative	0.273	0.190	0.224

Table 1: Evaluation Results

The first section of the table shows our official run results as submitted. Our best Run 1 lands us in the middle of all runs submitted this year. Note, that this system only extracts 13 out of the 40 TAC-KBP slot types. Despite the relatively low precision of KBP-TK, the combination of the two systems approximately doubles the individual system recalls due to the complementarity of their addressed slot types. Note that the results of KBP-TK were achieved by us (a third-party uninformed user) without any reconfiguration or tuning, therefore, they should not be used to assess the general performance characteristics of the toolkit.

The second section shows results after a simple literal duplicate removal, since we did not properly obey this value uniqueness constraint when constructing KRes and merged results. This shows a very significant jump in precision for KRes and slightly improved results for the other variants. Finally, the last section shows results relativized to the subset of slot types actually attempted by each system (these subsets are different for each variant, seven slots for KRes, eight for KBP-TK and thirteen for the combined system). These results are particularly relevant for KRes, since they were achieved

with only about ten dependency patterns per slot type which gives us promise for a significant upside once we expand the set of addressed slots. Also note that inexact matches such as "TV news man" instead of "news man" count as incorrect in the result computation due to the strict annotation guidelines for titles. Judging those more tolerantly would obviously improve results for all systems (not just ours).

Acknowledgment

This report is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory (AFRL) under agreement number FA8750-12-2-0342. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, AFRL or the U.S. Government.

References

H. Chalupsky. 2012. Story-level inference and gap filling to improve machine reading. In *Proceedings of the Twenty-Fifth International FLAIRS Conference*. AAAI Press.

Z. Chen, S. Tamang, A. Lee, X. Li, W. Lin, J. Artiles, M. Snover, M. Passantino, and H. Ji. 2010. CUNY-BLENDER TAC-KBP 2010 entity linking and slot filling system description. In *Proceedings of the 2010 Text Analytics Conference (TAC2010)*.