CS 5350 Final Project Report

Vishay Vanjani, Alex Clemmer, *Godzillasaurus Rex*

December 9, 2011

1 Architecture of your IE system

Our system is a composite prediction system, with every slot predicted individually. Although we experimented with a number of different approaches, our final system used *AutoSlog-TS* for 3 of the 6 slots, a decision we made mainly on the strength of the algorithm's results for `test set 1`. Slots that were not covered using AutoSlog-TS were either not tractable for the algorithm, or we found a better solution, using some performance measure. The slots not covered were `incident`, `weapons`, and `perp_org`. `incident` prediction was handled with a custom pattern-matching system; `weapons` and `perp_org` were handled by a weighted histogram lookup.

1.1 Predicting the `Incident` Slot

We manually created around 50 patterns, linking each to an *incident type* (*e.g.*, "`attack`"). For each particular text, we scored each pattern, and output the incident type of the pattern that had the best score. Our default fallback was to simply output `attack`. The scoring of a pattern is a straightforward combination of the hand-annotated relevance of pattern and the frequency of pattern.

1.2 Predicting Slots Using AutoSlog-TS (`Perp_Ind`, `Targets`, and `Victim`)

We used AutoSlog-TS to automatically extract a lot of relevant patterns from the `dev` and `test1` sets, which we could then annotate as useful for a particular slot, or discard. We did this on a *per-slot* basis, and initially, we had a parallel system for every slot except the `Incident` slot (which we already had a good predictor for) and the `Perp_Org` slot. After manual review we found no relevant patterns for `weapons`.

The output of this system was 99 total patterns, which were then used at runtime to extract the context surrounding the set of probable possible answers. A summary of our this output is here (and you can see the extracted patterns in Appendix A & B):

Slot	Unique Patterns Extracted	Patterns after manual review
`Victims`	1021	45
`Targets`	1169	21
`Perp_Indiv`	521	33
`Weapons`	108	0
Total	**2819**	**99**

1.2.1 AutoSlog-TS Pipeline

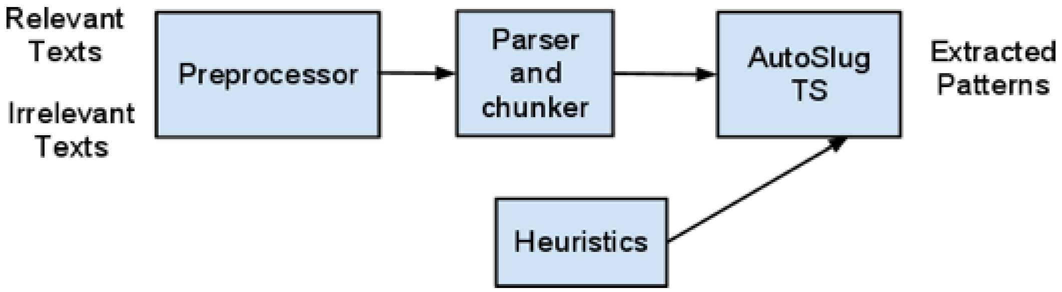
AutoSlog of course required a certain amount of scaffolding to work.

1. **Preprocessing:** Separate the header metacontent from the text, and things like that.

2. **Parsing and NP-chunking:** We parsed the main content of the page using the Stanford parser and used the parsed tree to build a NP chunked output.

3. **AutoSlog-TS:** We used regular expressions to look for heuristics in the chunked text and used the AutoSlog scoring mechanism to rank the extracted patterns.

Note that our pattern ranking criteria were: $relevance - rate \cdot \log_2(frequency)$. A pictoral summary of the process is here:



1.2.2 Pattern Extraction in AutoSlog-TS

We used a total of 13 heuristics for our system, 7 of these were for `Victim`, 4 were for `Target`, 2 were for `Perp_Indiv` and 1 was for `Weapon`. There was one overlapping heuristic between `Victim` and `Target`. As we saw in the table in section 1.2, this resulted in a total of 99 unique patterns.

One particular challenge was that we had to modify the heuristics given in the paper[1] to suit our parser, first, because our it did not distinguish between active voice and passive voice, and second, because it did not identify syntactic roles (*e.g.* subject, direct object, etc.) Of course, it is true that the dependency lists output by the parser could possibly be used to identify subjects and direct objects, but they were *prima facie* not very accurate, so we decided not to use them.

1.2.3 Pattern Matching in AutoSlog-TS

This step occurs at runtime, after we've extracted patterns from the text. Our job at this point is basically to find patterns in a given text, rank the patterns that are most likely to contain correct predictions for a given slot, and then strip out the extraneous information we've extracted along with the embedded answer. The process is as follows:

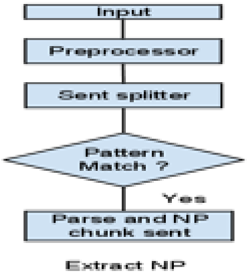
1. **Preprocessing:** Splits the file into multiple texts and splits each text into meta text and main text.

2. **Sentence-Splitting:** Splits the main text into different sentences using two patterns 1) dot followed by space or 2) dot followed by quotes.

3. **Pattern Matching:** Uses regular expressions to match the extracted patterns for victims, targets, and perpetrators in a given sentence. For our purposes, the forward patterns were the patters in which the relevant NP occured at the beginning (e.g "`<NP> WAS\s*[\w]*\s*MURDERED`"). In contrast, backward patterns are patterns in which the relevant NP occurs at the end (e.g. "`MURDERED <NP>`). We looked for forward patterns first and then for backward patterns, if a forward pattern matched then the backward pattern containing the same verb is not considered. For example, if "`WAS\s*[\w]*\s*MURDERED`" matched then we did not look for "`MURDERED <NP>`".

4. **Parsing and NP Chunking:** If a sentence matches a particular pattern then we extract the relevant NP (i.e., for forward patterns we extract NP occurring just before the pattern and for backward patterns we extract the NP occurring after the pattern).

A pictoral representation of this process is here:



1.2.4 Post Proessing in AutoSlog-TS

This modules goal was to filter through the list of NPs returned by previous step before choosing the correct answers for each slot. We wrote methods that performed the following tasks in order to complete this module.

1. **Extracting Multiple Answers from an NP:** We found that in some cases the extracted NP contained multiple answers separated by and, accompanied by, etc. We chose to look for such words and split the NP appropriately.

2. **Removing Synonyms:** We had a hard coded list of synonyms for this task.

3. **Overlapping Answers Removal:** We often found that some of our answers overlapped (e.g., Hector Oqueli and Oqueli). In these cases, we simply chose the largest of the two strings.

4. **Appositive Removal:** Our answers for the victim slot usually contained appositives like "Obama, President of the United States". So we maintained a list of relevant appositives and removed these if they were a part of the NP.

5. **Flag List:** We maintain flag list for all 3 of the slots. Flag list marks NPs that should be removed from consideration. For example, a victim flag list contained these words like *headquarters* and *embassy*. If such an NP in the list was reported, we simply removed it.

6. **Redundant word List:** This is a list of unnecessary words like verbs, weekdays, and prepositions that may be found alongside the answer, but which destroy the correctness of our answers.

7. **One word List:** This is a list of words which on their own could not be answers to any of the slots. Examples include words like *headquarters*, *urban*, *salvador*, and so on. These words were summarily removed from any answers.

1.3 Predicting the `Weapon` slot

2 External resources

Our final project uses NLTK and the Stanford Parser. However, we also at various points in the project used the Berkeley Parser, the Collins Parser, and the Stanford NER tagger, and both the lexicalized and normal Stanford PCFG parsers. NLTK was used for sentence and word tokenization, in addition to pos tagging and chunking.

3 Contribution of each team member

For a given slot, we ended up deploying the system that was most effective on the `test1` set. Although we performed many experiments, the `Victim`, `Perp_Indiv`, and `Target` slots all ended up having an AutoSlog-TS backend, which was implemented almost entirely by **Vishay**. He was responsible for the pre- and post-processing of the AutoSlog system, as well as almost all of the internals. Finally, he implemented the initial `Incident` predictor, which was responsible largely for our score in the dry run.

The remaining slots were `Perp_Org` and `Weapon`, both implemented by **Alex**. Additionally, Alex was responsible for integrating all the systems together, testing all the systems, and tuning the output. Further, Alex spent much of his time trying to produce viable alternatives to AutoSlog-TS, and although you won't see them in the final system, you can still see the code in the repository (especially if you sift through the commits in the .git file). Outlines of the approaches in the presentation slides, and, to a lesser extent, in the "originality" section.

4 Emphasis / Originality

We started our project thinking that extracting the relevant patterns will be the most important part of the project, and because we did not want any bias in selecting of Patterns, we decided to go with the AutoSlug-TS approach. However we later found that the Post Processing phase was as important as the Pattern Selection phase. We were extracting a lot of irrelevant NPs and the few relevant NPs that we were extracting were filled with redundant words. Our most original module was the Post processor (and Weapns ????) because we tried a lot of unique things in that module. However it was also the least robust part of our project.

5 Performance results and glass box analysis

6 Surprise Factor

1. Among 4 different parsers, we could barely get coherent output. To chunk NPs it was much more effective to word-tokenize, and then use NLTK's builtin NP chunker to find NPs.

2. We had not anticipated that the post processing module to be as complicated as it was. It was really difficult to sift through all the NPs extracted and get the correct answers.

3. Simple techniques worked better. Most people who had mined their answers from `dev` set did really well, even for general slots like `Victim`, `Target`, and `Perpet_Indiv`.

7 Successes, regrets, and lessons learned

Our success is straightforward: we were best-in-class on the `test1` set. This is undisputable. We also tried out a huge number of techniques and technologies, and were even successful in producing competetive results in other approaches that we tried.

That said, our results on the `test2` set were very bad, mainly because our post processing step was manually developed using only the `test2` data. This caused massive overfitting, and in retrospect, there are a number of different ways this could have been avoided:

1. Building the system on the `dev` set alone, and using the test set *only* for tuning.

2. Building an ML-based post processor.

3. Using an NER tagger plus a dictionary (for nameless victimsm *e.g.*, people, child, wife, etc.) to validate the answers for the victim slot.

This is a big lesson not really because we didn't know about overfitting before (we did), but because it makes the lesson of overfitting *very* concrete: avoid anything that even *looks* like overfitting.