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

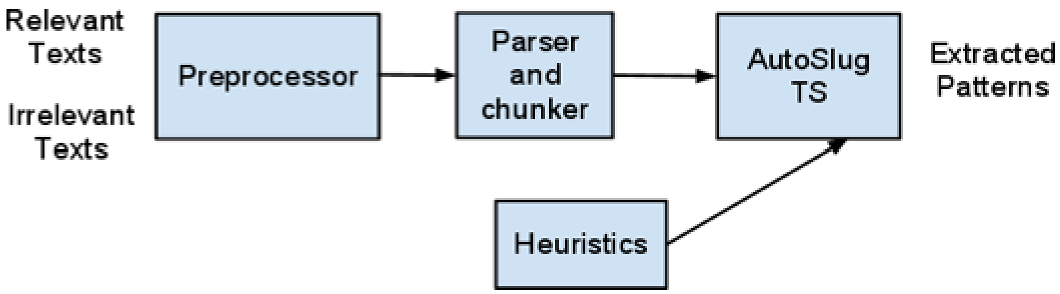
AutoSlog of course required a certain amount of scaffolding to work.

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:

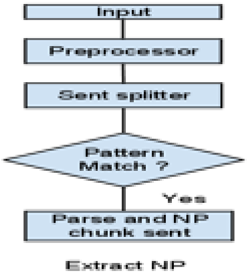
1. **Preprocessing:** Splits the file into multiple texts and splits each text into meta text and main text.

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

AutoSlog has the downside of being a very complicated slot with a number of different moving parts, and a significant pipeline, which makes it complicated to describe. In contrast, the `Weapon` slot is fairly simple: this is a weighted dictionary lookup. We find the weights using a cross-validated weight boosting scheme.

This is a viable approach because a very high percentage of the slots will accept an answer that is the `dev` set. Very close to all of the unique answers in the `test1` set are actually answers with more than one option, and almost always with at least one term we saw at least once in the `dev` set.

The question is how we know what the optimal weights are. The approach that made it into the final project works (roughly) as follows:

1. Begin with a hash table containing *key* : *value* pairs, where *key* is a word, and *value* is an integer. Look at dev set's answer key, and set simply count the number of times you encounter any answer. Record this in the hash table.

2. Initialize a model. Run the model over some subset of the output (we did this using 2/3rds of the output). If the learner gets something wrong, weight that feature negatively. If it gets it right, weight that feature positively.

3. Do this repeatedly until you have some number of trainers, each trained on overlapping parts of the dev set.

4. Average their feature vectors; this gives you a weighted histogram.

In the end, we simply hard-coded this weighted histogram into our submission. When it came time to predict the weapon, we adopted the following pipeline:

1. **Sentence-tokenize the document.**

2. **Look in the sentence for any of the words that were in the dev set answers.**

3. **Look in the histogram for the weight of each term found.**

4. **Output the term with the highest weight.**

2 Predicting the Perp_Org Slot

Perp_Org is slightly more complicated than the Weapon slot, but the basic concept is similar. The difference is that, in the previous case, we used simple dictionary lookups for the weapons themselves. In this case, we supplemented this weighted histogram not just with the counts of the organization, but with the counts of the verbs around it. More concretely:

1. Look at the dev set answer key; build a hash table containing perp org answers paired with the count of the number of times they appear (this is very similar to how we built the dictionary for the Weapon slot).

2. Now look at all the texts in the dev set. Sentence-tokenize each document, and then use NLTK's POS tagger to get POS annotations for every sentence. Pull all the verbs out and count how many times each occurrs in the same sentence as each particular word that was in the dev set answers. For example, if we have a word, *bomb*, and the verbs that occur around it are *explodes* and *kills*, then we count them. At the end, we have a count of the words that occur next to bomb.

We amplify the verb histogram using the algorithm described in the previous section. Then, using this histogram, we can construct the following algorithm:

1. **Sentence-tokenize the document.**

2. **Look in the sentence for any perp org that was also in the dev set.**

3. **If there exists at least one term, use NLTK's POS tagger to annotate that sentence; then pull all the verbs out. Look at the amplified counts of each verb, and sum them up.**

4. **Output the term that has the highest verb-weight sum.**

3 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.

4 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.

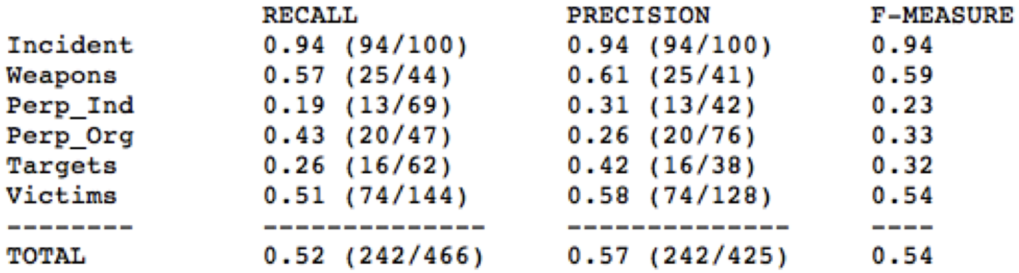
The remaining slots were `Perp_Org` and `Weapon`, both implemented by **Alex**. Additionally, Alex was responsible for integrating all the systems together, testing the holistic system, and tuning results of the holistic system. Further, Alex spent much of his time trying to produce viable alternatives 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). This mainly served to inform the direction of the project: if it was the case that one of the systems outperformed AutoSlog-TS in one of the slots, it may have been the case that AutoSlog-TS would have been abandoned; it is by virtue of the fact that this didn't happen that we submitted the system we did. Additionally, outlines of the approaches in the presentation slides, and, to a lesser extent, in the "originality" section.

5 Emphasis / Originality

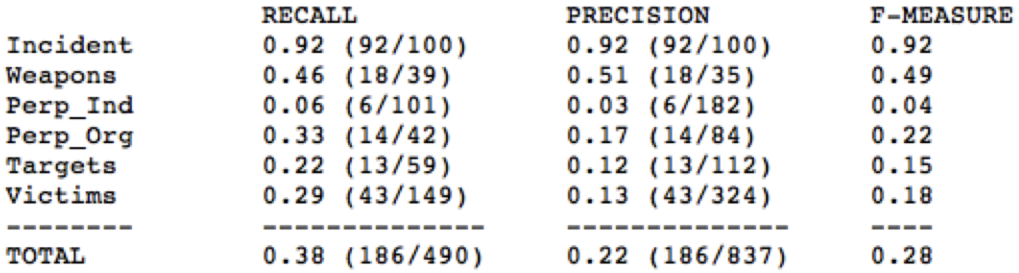
We started our project thinking that extracting the relevant patterns will be the most important part of the project, and because we did not want any bias in selecting of Patterns, we decided to go with the AutoSlug-TS approach. However we later found that the Post Processing phase was as important as 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.

6 Performance results and glass box analysis

Our results follow:



```
            RECALL            PRECISION         F-MEASURE
Incident    0.94 (94/100)     0.94 (94/100)     0.94
Weapons     0.57 (25/44)      0.61 (25/41)      0.59
Perp_Ind    0.19 (13/69)      0.31 (13/42)      0.23
Perp_Org    0.43 (20/47)      0.26 (20/76)      0.33
Targets     0.26 (16/62)      0.42 (16/38)      0.32
Victims     0.51 (74/144)     0.58 (74/128)     0.54
--------    --------------    --------------    ----
TOTAL       0.52 (242/466)    0.57 (242/425)    0.54
```



```
            RECALL            PRECISION         F-MEASURE
Incident    0.92 (92/100)     0.92 (92/100)     0.92
Weapons     0.46 (18/39)      0.51 (18/35)      0.49
Perp_Ind    0.06 (6/101)      0.03 (6/182)      0.04
Perp_Org    0.33 (14/42)      0.17 (14/84)      0.22
Targets     0.22 (13/59)      0.12 (13/112)     0.15
Victims     0.29 (43/149)     0.13 (43/324)     0.18
--------    --------------    --------------    ----
TOTAL       0.38 (186/490)    0.22 (186/837)    0.28
```

As you can see, we do significantly better on the test1 set than the test2 set. There are a number of reasons for this, but the main one is that AutoSlog-TS's postprocessing step was created exclusively for, and exclusively using, test1's data. This turns out to have caused the output to be very much overfitted, causing it to generalize poorly. Notice, for example, that the Weapon and Incident slots seem to have generalized quite nicely. Even the Perp_Org does well compared to the rest of the slots.

What's really dragging down the average is the recall on Perp_Indiv; when you just take out that one slot, it bumps our score by more than 4%, from 0.28 to 0.32. This indicates that perhaps we're not doing as bad as it seems, *prima facie*, because this is not even the worst slot—the worst slot by far is Victim, which produces 324 predictions for 149 slots. This is a consistent trend: in each of the AutoSlog-TS results (Perp_Ind, Targets, and Victims), we make *at least* twice as many predictions as there are actual answers. One simple modification we could do to improve our score would be to simply throttle the number of predictions we make in each slot, particularly the Victim slot. The effect of this is profound: consider if we had gotten a modest precision rate of 30% (rather than the absurd 13% we did get), we would have boosted our total score to 0.37.

It's not hard to show convincingly that there are a number of fairly simple things that we could have done to increase our results *without* affecting the actual learning algorithm. This is comforting to know, since it indicates that the things we learned during this project are not necessarily completely discounted by the abysmal performance on test2. While we did make at least one huge mistake, it is not true that our approach was completely hopeless.

7 Surprise Factor

1. Among 4 different parsers, we could barely get coherent output. To chunk NPs it was much more effective to word-tokenize, apply simple POS-tags using the default NLTK tagger, 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.

8 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.

APPENDIX A

This appendix lists all the heuristics we used along with regex patterns. All of these Heuristics were taken from the AutoSlog-TS publication.

Slot	Heuristic	Regex
victim	NP VBD VBN	`(\[[^/]*?\])\/NP\s+(\w+?)\/VBD\s+(\w+?)\/VBN`
victim	NP VBD NN	`(\[[^/]*?\])\/NP\s+(\w+?)\/VBD\s+(\[.*?\])\/NP\s+`
victim	VBD NP	`\s+(\w+?)\/VBD\s+(\[[^/]*?\])\/NP`
victim	VB(all) TO VB NP	`\s+(\w+?)\/VB\w{0,1}\s+(\w+?)\/TO\s+(\w+?)\/VB\s+(\[`
victim	VBG NP	`\s+(\w+?)\/VBG\s+(\[[^/]*?\])\/NP`
victim	NN VBD NP	`(\[[^/]*?\])\/NP\s+(\w+?)\/VBD\s+(\[[^/]*?\])\/NP`
victim	NN IN NP	`(\[[^/]+?\])\/NP\s+(\w+?)\/IN\s+(\[[^/]*?\])\/NP`
Perp_I	NP VB(all)	`(\[[^/]*?\])\/NP\s+(\w+?)\/VB\w{0,1}`
Perp_I	NP VBD TO VB	`(\[[^/]*?\])\/NP\s+(\w+?)\/VBD\s+(\w+?)\/TO\s+(\w+?)\`
Weapon	VB IN NP	`\s+(\w+?)\/VB\w{0,1}\s+(\w+?)\/IN\s+(\[[^/]*?\])\/NP`
Target	VB(all) TO VB NP	`\s+(\w+?)\/VB\w{0,1}\s+(\w+?)\/TO\s+(\w+?)\/VB\s+(\[`
Target	VBD NP	`\s+(\w+?)\/VB\w{0,1}\s+(\[[^/]*?\])\/NP`
Target	NN IN NP	`(\[[^/]+?\])\/NP\s+(\w+?)\/IN\s+(\[[^/]*?\])\/NP`
Target	VBD VB(all) prep NP	`\s+(\w+?)\/VBD\s+(\w+?)\/VB\w{0,1}\s+(\w+?)\/IN\s+(\`

APPENDIX B

This Appendix Contains all the patterns we used for Extraction System. Most of these patterns were learnt using the AutoSlog-TS, and some of them were added(or modified) manually.

```
VICTIM

<NP> DIED INSTANTANEOUSLY
<NP> WAS\s*[\w]*\s*ASSASSINATED
```

```
<NP> WAS\s*[\w]*\s*INJURED
<NP> WAS\s*[\w]*\s*KIDNAPPED
<NP> WAS\s*[\w]*\s*KILLED
<NP> WAS\s*[\w]*\s*MURDERED
<NP> WAS\s*[\w]*\s*WOUNDED
<NP> WERE\s*[\w]*\s*KILLED
<NP> WERE\s*[\w]*\s*KIDNAPPED
<NP> WERE\s*[\w]*\s*ABDUCTED
<NP> WERE\s*[\w]*\s*ASSASINATED
<NP> WERE\s*[\w]*\s*INJURED
<NP> WERE\s*[\w]*\s*MURDERED
<NP> WERE\s*[\w]*\s*WOUNDED
<NP> WERE\s*[\w]*\s*SHOT
<NP> WAS\s*[\w]*\s*SHOT
ASSASSINATION OF  <NP>
ASSASSINATIONS OF  <NP>
ASSASSINS OF  <NP>
ATTACK ON  <NP>
ATTACKED  <NP>
BLOOD OF  <NP>
BODIES OF  <NP>
BURIAL OF  <NP>
DEATH OF   <NP>
DEATHS OF <NP>
DISAPPEARANCE OF <NP>
ELIMINATED <NP>
FUNERAL OF <NP>
FUNERALS OF <NP>
KIDNAPPED <NP>
KIDNAPPING OF <NP>
KIDNAPPINGS OF <NP>
KILLED <NP>
KILLING <NP>
KILLING OF <NP>
LIVES OF <NP>
LOSS OF <NP>
MACHINEGUNNED <NP>
MASSACRE OF <NP>
MURDER OF <NP>
MURDERED <NP>
MURDERERS OF <NP>
MURDERS OF <NP>
WOUNDED  <NP>

PERP_INDIV

<NP>  ASSASSINATED
<NP> ATTACKED
```

<NP>BLEW
<NP> BOMBED
<NP> BURNED
<NP> COMMITTED
<NP> DETONATED
<NP> FIGHTERS
<NP> FIRED
<NP> HURLED
<NP> INFLICTED
<NP> KIDNAPPED
<NP> KILLED
<NP> MACHINEGUNNED
<NP> SABOTAGED
WAS KIDNAPPED BY <NP>
WERE KIDNAPPED BY <NP>
WERE ABDUCTED BY <NP>
WAS ABDUCTED BY <NP>
WAS ATTACKED BY <NP>
WAS KILLED BY <NP>
WERE KILLED BY <NP>
WAS MURDERED BY <NP>
WERE MURDERED BY <NP>
WAS SHOT BY <NP>
WERE SHOT BY <NP>
WAS ASSASSINATED BY <NP>
WERE ASSASSINATED BY <NP>
WAS INJURED BY <NP>
WERE INJURED BY <NP>
WERE WOUNDED BY <NP>
WAS WOUNDED BY <NP>
WERE DETONATED BY <NP>

TARGET

AN ATTACK ON <NP>
ATTACK ON <NP>
ATTACKED <NP>
ATTACKS ON <NP>
BOMBED <NP>
BOMB ATTACK ON <NP>
BOMB IN <NP>
BOMBING <NP>
BURNED <NP>
BURNING <NP>
DAMAGED <NP>
DESTROYED <NP>
DESTRUCTION OF <NP>
DOWNED <NP>

```
DOWNING OF <NP>
DYNAMITE ATTACK ON <NP>
FIRED <NP>
HIT <NP>
MACHINEGUNNED <NP>
SABOTAGED <NP>
SET FIRE TO <NP>
```

Perp_Org (We got these patterns while reviewing the other slots. we did not get time to implement these)

```
GUERRILLAS OF
MEMBERS OF
ACTIVE MEMBER OF
SPOKESMAN OF
```