EXPERIMENTS IN IMPROVED METHODS FOR UNSUPERVISED RESOLUTION OF RELATIONS AND OBJECTS

Stephen Mayhew

Rose-Hulman Institute of Technology
mayhewsw@rose-hulman.edu

ABSTRACT

This paper has to do with relation extraction. This is probably best written after the rest of the paper is done.

Keywords— Natural Language Processing, Information Extraction, Relation Extraction, Coreference Resolution

1. INTRODUCTION

In this paper, I propose to implement a synonym resolution system for ReVerb, using much of the same ideas and infrastructure as RESOLVER. I propose to research methods, including mutual recursion, for improving recall, using RESOLVER as a baseline.

The field of Natural Language Processing has been moving forward slowly but steadily. As many of the lower level disciplines — such as part-of-speech tagging, and semantic parsing — are becoming more stable, some interest has shifted towards higher level projects.

One field in particular is of great importance: Information Extraction (IE). IE is useful primarily for tasks that involve reasoning over text, for instance, social media analysis.

Etzioni et. al. 2006 [1] have proposed a new era in IE, calling it Machine Reading (MR). MR is an ambitious task which seeks to give a computer full understanding of text by simple reading of text, just as humans do. Etzioni et. al. have suggested that MR become a new staple of the AI community, next to the age old giants of Machine Learning and Machine Translation.

Since that proposal, there has been considerable work towards this goal. In 2009, DARPA launched a 5-year Machine Reading program. Two of the standouts in this field are the KnowItAll project at University of Washington, and the Never-Ending Language Learner (NELL) [2] at Carnegie-Mellon University. These two projects have made great strides towards unsupervised holistic information extraction, but both still have a long way to go.

University of Washington has released the source code for a new relation extractor called ReVerb [3], which was built on and replaced the previous system, called TextRunner.

One major area of work is synonym resolution. ReVerb is able to extract information, but the next problem is being able to use that information. In particular, there are often relations returned which have different forms, but in reality map to the same underlying meaning. For the sake of actually using the knowledge, it is imperative that these relations are distilled into the smallest amount of useful knowledge possible. There is a system built on TextRunner called RESOLVER (2009) [4] which works to this very end. Likewise, there is a system built on NELL called ConceptResolver (2011) [5].

In [4], the future work discussion outlines possible directions for future research. The two major directions were left as avenues still to pursue: improvements in polysemy, and use of mutual recursion to improve recall.

Polysemy is the capacity for a word or relation to have multiple meanings. RESOLVER assumed that each word had exactly one meaning, which led to some shortcomings. Allowing words to have multiple meanings can make the synonym resolution task more difficult, but that is the nature of human language.

RESOLVER had a recall of 35%, which is very low. Yates 2009 [4] suggests experimentation with mutual recursion to improve recall. Mutual recursion is a bootstrapping approach that after performing clustering, uses those clusters to update the model, which in turn produces better clusters. Mutual recursion is not the only way to improve recall: there may be other more creative avenues.

2. EXPERIMENTAL SETUP

My system is written in Java. I started with the unmaintained Resolver Java code, and built my system from there. I reimplemented the main algorithm, but recycled the scoring code.

One large part of the work in [4] was the construction of an efficient algorithm for clustering. This algorithm is based on the idea that a pair of strings should only be considered for similarity if they have a similar context. My goal was not to improve this algorithm, but to improve on the results. Thus, I implemented the algorithm almost verbatim from [4]. The pseudocode is found in Figure 2.

©2012 SWM

$E := \{e = (r, a, b) | (r, a, b)$ is an extracted assertion$\}$
$S := \{s | s$ appears as a relation or argument string in $E\}$ $Cluster := \{\}$
$Elements := \{\}$

1. For each $s \in S$:

   $Cluster[s] :=$ new cluster id
   $Elements[Cluster[s]] := \{s\}$

2. $Scores := \{\}$, $Index := \{\}$

3. For each $e = (r, a, b) \in E$:

   $property := (a, b)$
   $Index[property] := Index[property] \cup \{Cluster[r]\}$
   $property := (r, a)$
   $Index[property] := Index[property] \cup \{Cluster[b]\}$
   $property := (r, b)$
   $Index[property] := Index[property] \cup \{Cluster[a]\}$

4. For each property $p \in Index$: If $|Index[p]| <$ Max:

   For each pair $\{c_1, c_2\} \in Index[p]$:

   $Scores[\{c_1, c_2\}] :=$ similarity(c_1, c_2)

5. Repeat until no merges can be performed:

   Sort $Scores$
   $UsedClusters := \{\}$
   Repeat until $Scores$ is empty or top score $< Threshold$:

   $\{c_1, c_2\} := removeTopPair(Scores)$
   If neither c_1 nor c_2 is in $UsedClusters$:

   $Elements[c_1] := Elements[c_1] \cup Elements[c_2]$
   For each $e \in Elements[c_2]$:

   $Cluster[e] := c_1$

   delete c_2 from $Elements$
   $UsedClusters := UsedClusters \cup \{c_1, c_2\}$

   Repeat steps 2-4 to recalculate $Scores$

Fig. 1. Original clustering algorithm

That said, it is important to keep in mind that scalability is key for this kind of system because it is intended to work on massive data sets extracted from the web. Because of this, I will be careful to not decrease the performance substantially, unless I know it makes a substantial help.

I set it up so that the function in the pseudocode called $similarity$ is the main differentiation between the similarity metrics. Given a certain parameter, it would calculate and return the similarity using the chosen metric.

In step 5, it says to repeat until no merges can be performed. On almost no runs did this actually happen. This turns out to take so many merges that I usually just set a merge limit and stopped after so many merges. My number of choice was 4. [[I THINK I SHOWED THAT RUNNING MORE MERGES DIDN'T MAKE MUCH DIFFERENCE]].

I usually ran it for 4 merges. I would run the entire algorithm using different parameters for *Threshold*. I would run it in a loop, starting with *Threshold* at 0.1 and increasing it by 0.1 until it reached 1. This is a more or less comprehensive procedure because *similarity* function returns a probability, which is in the range [0,1].

I would typically store the similarity metric of choice in a different source file and just import it. This seemed to work the best.

[[RUNNING TIME METRICS]] have I sped it up/slowed it down considerably?

[[NUMBER OF LINES?]] I started out with 6441 lines of Java code from Yates, and wrote 1785 lines of my own code. I probably only use about 500 lines of code from Yates.

3. EXPERIMENTS AND RESULTS

My baseline results were the results from [4]. However, I found that I got significantly different results using the same methods, ESP and SSM. I believe that this can be explained by looking at the differences in data set. I do not know which data set Yates used, and I am reasonably sure that we used different sets, because we got such different results. As a result, I think it is not possible to compare our results directly. However, I think it is possible to compare improvement.

	Objects			Relations		
Model	Prec.	Rec.	F1	Prec.	Rec.	F1
ESP	0.56	0.41	0.47	0.79	0.33	0.47
SSM	0.62	0.53	0.57	0.85	0.25	0.39
RESOLVER	0.71	0.66	0.68	0.90	0.35	0.50

Fig. 2. Yates Data

3.1. SSM

I used a lot of good SSM metrics. More coming...

	Objects			Relations		
Model	Prec.	Rec.	F1	Prec.	Rec.	F1
SSM	0.93	0.53	0.67	0.59	0.32	0.42

Fig. 3. SSM Results

3.2. ESP

Results can be seen in Figure 5

	Objects			Relations		
Model	Prec.	Rec.	F1	Prec.	Rec.	F1
ESP	0.58	0.34	0.43	0.46	0.21	0.29

Fig. 5. ESP Results

3.3. Soundex

One experiment involved reducing each string to it's Soundex representation. Soundex is an algorithm that computes the phonetic representation of a string. It is used to find similarity between slightly misspelled words. The goal of this approach was to reduce noise in the data.

One downside to this approach is that a misspelling such as "Lresident" instead of "President" will produce a different representation, whereas "Prisident" and "President" will map to the same string.

How often does this kind of error occur? How confident can I be that this is a good approach?

Metaphone is another phonetic representation algorithm, similar to Soundex.

	Objects			Relations		
Model	Prec.	Rec.	F1	Prec.	Rec.	F1
SSM + Soundex	0.93	0.52	0.67	0.54	0.31	0.39
ESP + Soundex	0.61	0.34	0.43	0.47	0.21	0.29

Fig. 7. Soundex Results

Be sure to reference Soundex and Metaphone

3.4. Mutual Recursion

I experimented with mutual recursion. This is how I did it: at the beginning of the code, every unique string is given a unique ID, which is it's cluster ID. This effectively ensures that every string starts out in it's own cluster. I started out by just more or less picking random numbers (started at 100 and moved up from there).

But! The new idea is to index each string with its Soundex representation. This will cause some clustering to happen at the very beginning, but I maintain that this is almost always desirable. Soundex should not merge many strings which are not

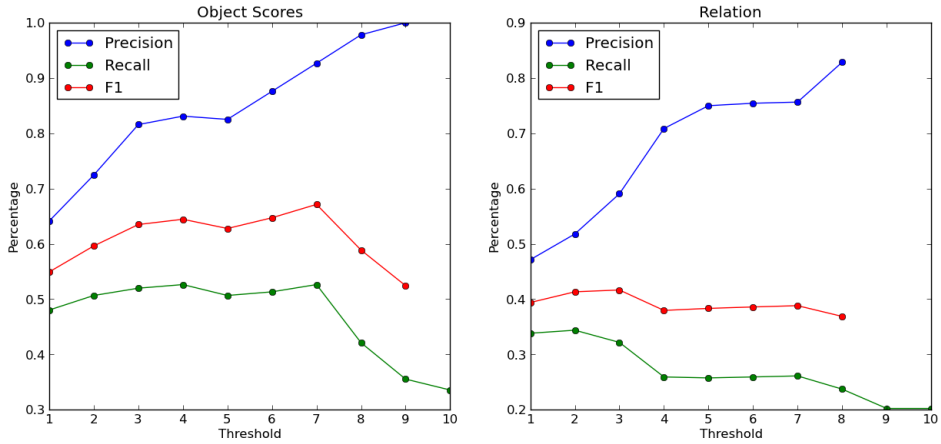

Fig. 4. SSM Results

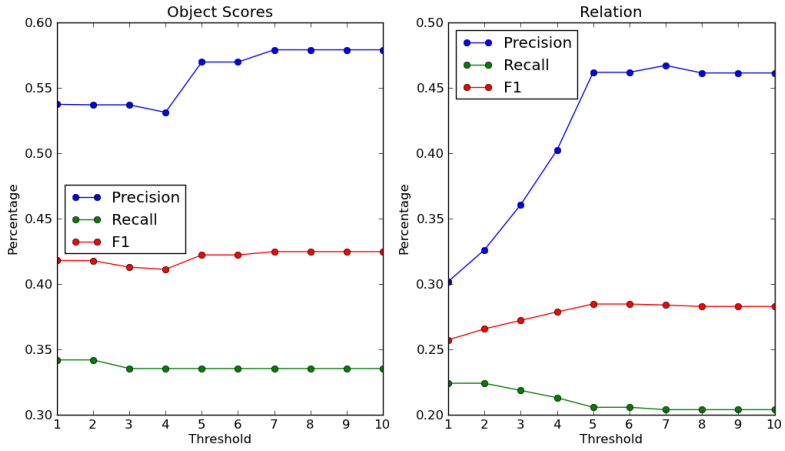

Fig. 6. ESP Results

only slightly different. It should correct spelling errors. Needs some experimentation as to whether it merges definitely different words.

$E := \{e = (r, a, b) | (r, a, b)$ is an extracted assertion$\}$
$S := \{s | s$ appears as a relation or argument string in $E\}$ $Cluster := \{\}$
$Elements := \{\}$

1. For each $s \in S$:

   $Cluster[s] :=$ id: Soundex representation of s
   $Elements[Cluster[s]] := \{s\}$

2. $Scores := \{\},\ Index := \{\}$

3. For each $e = (r, a, b) \in E$:

   // The major change for mutual recursion is here
   $propIndex := Cluster(a) + Cluster(b)$
   $Index[propIndex] := Index[propIndex] \cup \{Cluster[r]\}$
   $propIndex := Cluster(r) + Cluster(a)$
   $Index[propIndex] := Index[propIndex] \cup \{Cluster[b]\}$
   $propIndex := Cluster(r) + Cluster(b)$
   $Index[propIndex] := Index[propIndex] \cup \{Cluster[a]\}$

4. For each property $p \in Index$: If $|Index[p]| <$ Max:

   For each pair $\{c_1, c_2\} \in Index[p]$:

   $Scores[\{c_1, c_2\}] :=$ similarity(c_1, c_2)

5. Repeat until no merges can be performed:

   Sort $Scores$
   $UsedClusters := \{\}$
   Repeat until $Scores$ is empty or top score $< Threshold$:

   $\{c_1, c_2\} := removeTopPair(Scores)$
   If neither c_1 nor c_2 is in $UsedClusters$:

   $Elements[c_1] := Elements[c_1] \cup Elements[c_2]$
   For each $e \in Elements[c_2]$:

   $Cluster[e] := c_1$

   delete c_2 from $Elements$
   $UsedClusters := UsedClusters \cup \{c_1, c_2\}$

   Repeat steps 2-4 to recalculate $Scores$

Fig. 8. Algorithm with Mutual Recursion

Further, when indexing strings for the Index data structure, previously it had been done with properties. The problem with this was that once two strings had been merged, the property picking didn't reflect that. That is, if two strings s_1 and s_2 had properties (s_a, s_b) and (s_a, s_c) respectively, and a run showed that s_b and s_c were synonymous (and were therefore merged), then the properties should be merged. This was not happening in the original code.

My solution is change the key value for Index data structure. Where it used to be the actual property, I will change it a concatenation of the Cluster IDs for the 2 strings in the property. This way, given two properties (s_a, s_b) and (s_c, s_d), these will be the same key in Index if and only iff s_1 has the same cluster ID as s_c and s_b has the same cluster ID as s_d.

One problem with this: Soundex is not supposed to be an ID, it is supposed to be a result of an algorithm given a string as input. My algorithm will cause many strings to be given ID's which are soundex strings, but definitely do not apply. For example, "Environmental Protection Agency" and "EPA" should definitely be clustered, which means they will have the same cluster ID (which is a Soundex string). However, they clearly should not share a Soundex string. This is not a major problem, but it may cause some confusion when interpreting intermediate results.

	Objects			Relations		
Model	Prec.	Rec.	F1	Prec.	Rec.	F1
Mutrec ESP	0.80	0.35	0.49	0.43	0.21	0.28
Mutrec SSM	0.61	0.40	0.48	0.50	0.32	0.39

Fig. 9. Mutual Recursion Table

NOTE: extra merges made a significant difference here. What if it merged all the way?

3.5. WordNet Similarity

First thing to note about WordNet is that it can only be used for relations. The proper nouns which comprise the object parts of the extractions are much less likely to be found in WordNet.

Wordnet. [6]

I also used WordNet similarity metrics. Using a WordNet Similarity implementation, from the java package edu.sussex.nlp.jws (no citation available).

The first thing to be said is that my WordNet experiments were only for relations. WordNet does not contain proper nouns. Thus, it does not make sense to compare objects using WordNet because very few of them would be likely to actually be in there.

The difficulty of this approach was the strictness of WordNet. If it didn't have the word in question, it will not return anything. The data is noisy: some relations contain nothing but stopwords, others contain many stopwords and include irrelevant non-verbs ('always', 'after', 'through'). One large problem was how to know which word to take from the relation for comparison.

Another large problem is lemmatizing the words so there is a standard for comparison. Another problem with this is that the lemmatization is not always an easy decision. Perhaps a relation actually consists of 3 words ("get out of"). How to know which words are stopwords and which words are actually useful in the phrase? In "born in" the word "in" is not very important.

For the three relations "pulls," "got out of," "is pulling," we want them to be cleaned so they look like: "pull," "get out of," "pull." So cleaning up is a difficult job.

There is still plenty of work to be done here. Getting the data clean is the biggest problem. A large number of words and phrases were not found in the data.

Could also look into checking objects. "President Grover Cleveland," vs. "Governor Grover Cleveland" or something.

	Objects			Relations		
Model	Prec.	Rec.	F1	Prec.	Rec.	F1
WNsim (JWS)	0.93	0.53	0.67	0.55	0.37	0.44
WNsim (JWNL)	0.93	0.53	0.67	0.45	0.33	0.38

Fig. 10. WordNet Results

Be sure to reference JWS and JWNL

4. CONCLUSIONS

All told, nothing worked very well, but WordNet Sim got the best results.

5. FUTURE WORK

Other ideas include checking if a substring is in the other string. Checking of type of data: 1941 and 1939 are dates, and likely should be clustered, if similar?

Comparing objects against acronyms. Environmental Protection Agency vs. EPA.

My question: if items are originally grouped together by their properties, does that mean that similar objects with radically different properties will never be clustered?

Look into metaphone, and other phonetic type similarity metrics.

Use of WordNet for objects also.

5.1. Data Set

I used a data set which I got from Dr. Alex Yates. He said it was the same as he had used in his work, but I don't think it was. I say this because I got different results when using the same methods: ESP, SSM, etc.

I believe that the choice of data set makes a significant difference in the results. For example, a data set whose clusters tend to be very similar strings but be found in a wide variety of contexts would get a high score from SSM. Likewise, a data set whose clusters are very different on an SSM scale, but which all

come from very similar contexts, are likely to score high with ESP.

Ultimately, the best model data set is one which accurately models real world data, such as the data which comes out of ReVerb, or TextRunner. It would be a very valuable exercise to analyze the data from ReVerb, and come up with statistics about its makeup.

In my original system, I would get inconsistent results from two identical runs. That is, same data set, same code, different results. These results could be as much as a 5% difference in F1 measure. I found that this had to do with my extensive use of HashSets in the implementation. I changed these all to TreeSets, and the results became consistent. But this begs an important question: does ordering matter? Does this mean there are other ways to order the data that can give better results? I claim that this is unimportant. It is very hard to say what is the "best result" anyway because of the fallibility of the data set. I believe that what is most important is the amount of change between results. For this reason, consistency in results is more important than absolute accuracy (which is not even well-defined).

6. REFERENCES

[1] Oren Etzioni, Michele Banko, and Michael Cafarella, "Machine reading," *AAAI*, 2006.

[2] B. Kisiel B. Settles E.R. Hruschka Jr. A. Carlson, J. Betteridge and T.M. Mitchell, "Toward an architecture for never-ending language learning.," in *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2010.

[3] Anthony Fader, Stephen Soderland, and Oren Etzioni, "Identifying relations for open information extraction," *EMNLP*, 2011.

[4] Alexander Yates and Oren Etzioni, "Unsupervised methods for determining object and relation synonyms on the web," *Journal of Artificial Intelligence Research*, vol. 34, pp. 255–296, 2009.

[5] J. Krishnamurthy and T.M. Mitchell, "Which noun phrases denote which concepts?," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2011.

[6] George A. Miller, "Wordnet: A lexical database for english," *Communications of ACM*, vol. 38, No.11, pp. 39–41, 1995.