A SIMPLE ALGORITHM FOR IDENTIFYING ABBREVIATION DEFINITIONS IN BIOMEDICAL TEXT

ARIEL S. SCHWARTZ

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720
sariel@cs.berkeley.edu

MARTI A. HEARST

SIMS
University of California, Berkeley
Berkeley, CA 94720
hearst@sims.berkeley.edu

Abstract

The volume of biomedical text is growing at a fast rate, creating challenges for humans and computer systems alike. One of these challenges arises from the frequent use of novel abbreviations in these texts, thus requiring that biomedical lexical ontologies be continually updated. In this paper we show that the problem of identifying abbreviations' definitions can be solved with a much simpler algorithm than that proposed by other research efforts. The algorithm achieves 96% precision and 82% recall on a standard test collection, which is at least as good as existing approaches. It also achieves 95% precision and 82% recall on another, larger test set. A notable advantage of the algorithm is that, unlike other approaches, it does not require any training data.

1 Introduction

There has been an increased interest recently in techniques to automatically extract information from biomedical text, and particularly from MEDLINE abstracts.[3, 4, 7, 15] The size and growth rate of biomedical literature creates new challenges for researchers who need to keep up to date. One specific issue is the high rate at which new abbreviations are introduced in biomedical texts. Existing databases, ontologies, and dictionaries must be continually updated with new abbreviations and their definitions. In an attempt to help resolve the problem, new techniques have been introduced to automatically extract abbreviations and their definitions from MEDLINE abstracts.

In this paper we propose a new, simple, fast algorithm for extraction of abbreviations from biomedical text. The scope of the task addressed here is the same as the one described in Pustejovsky et al.:[14] identify <"short form", "long form"> pairs where there exists a mapping (of any kind) from characters in the short form to characters in the long form.[a]

[a] Throughout the paper we use the terms "short form" and "long form" interchangeably with "abbreviation" and "definition". We also use the term "short form" to indicate both abbreviations and acronyms, conflating these as have previous authors.

Many abbreviations in biomedical text follow a predictable pattern, in which the first letter of each word in the long form corresponds to one letter in the short form, as in *methyl methanesulfonate sulfate (MMS)*. However, there are many cases in which the correct match between the short form and long form requires words in the long form to be skipped, or matching of internal letters in long form words, as in *Gcn5-related N-acetyltransferase (GNAT)*. In this paper, we describe a very simple, fast algorithm for this problem that achieves both high recall and high precision.

2 Related Work

Pustejovsky et al.[13, 14] present a solution for identifying abbreviations based on hand-built regular expressions and syntactic information to identify boundaries of noun phrases. When a noun phrase is found to precede a short form enclosed in parentheses, each of the characters within the short form is matched in the long form. A score is assigned that corresponds to the number of non-stopwords in the long form divided by the number of characters in the short form. If the result is below a threshold of 1.5, then the match is accepted. This algorithm achieved 72% recall and 98% on "the gold standard," a small, publicly available evaluation corpus that this group created, working better than a similar algorithm that does not take syntax into account.[b]

Pustejovsky et al.[13] also summarize some drawbacks of other earlier pattern-based approaches, noting that the results of Taghva et al.[17] look good (98% precision and 93% recall on a different test set), but do not account for abbreviations whose letters may correspond to a character internal to a definition word, a common occurrence in biomedical text. They also find that the Acrophile algorithm of Larkey et al.[8] does not perform well on the gold standard.

Chang et al.[5] present an algorithm that uses linear regression on a pre-selected set of features, achieving 80% precision at a recall level of 83%, and 95% precision at 75% recall on the same evaluation collection (this increases to 82% recall and 99% precision on a corrected version).[c] Their algorithm uses dynamic programming to find potential alignments between short and long form, and uses the results of this to compute feature vectors for correctly identified definitions. They then use binary logistic regression to train a classifier on 1000 candidate pairs.

Yeates et al.[19] examine acronyms in technical text. They address a more difficult problem than some other groups in that their test set includes instances that do not have distinct orthographic markers such as parentheses to indicate the

[b] There are some errors in the gold standard. The results reported by Pustejovsky et al.[13] are on a variation of the gold standard with some corrections, but the actual corrections made are not reported in the paper. Unfortunately, the corrections needed on the standard are not standardized.
[c] Personal communication, H. Schuetze.

proximity of a definition to an abbreviation (they report that only two thirds of the examples take this form). Their algorithm creates a code that indicates the distance of the definition words from the corresponding characters in the acronym, and uses compression to learn the associations. They compile a large test collection consisting of 1080 definitions; training on two thirds and testing on the remainder, reporting the results on a precision/recall curve.

Park and Byrd[12] present a rule-based algorithm for extraction of abbreviation definitions from general text. The algorithm creates rules on the fly that model how the short form can be translated into the long form. They create a set of five translation rules, a set of five rules for determining candidate long forms based on their length, and a set of six heuristics for determining which definition to choose if there are many potential candidates. These are: syntactic cues, rule priority, distance between definition and abbreviation, capitalization criteria, number of words in the definition, and number of stopwords in the definition. Rule priority is based on how often the rule has been applied in the past. They evaluate their algorithm on 177 abbreviations taken from engineering texts, achieving 98% precision and 95% recall. No mention is made of the size and nature of the training set, or whether it was distinct from the test set.

Yu et al.[21] present another rule-based algorithm for mapping abbreviations to their full forms in biomedical text. Their algorithm is similar to that of Park and Byrd. For a given short form, the algorithm extracts all the candidate long forms that start with the same character as the short form. The algorithm then tries to match the candidate long forms to the short form starting from the shortest long form, by iteratively applying 5 pattern-matching rules. The rules include heuristics such as prioritizing matching the first character of a word, allowing the use of internal letters only if the first letter of a word was matched, and so on. The algorithm was evaluated on a small collection of biomedical text containing 62 matching pairs, achieving 95% precision and 70% recall on average.

Adar[1] presents an algorithm that generates a set of paths through the window of text adjacent to an abbreviation (starting from the leftmost character), and scores these paths to find the most likely definition. Scoring rules used include “for every abbreviation character that occurs at the start of a definition word, add 1”, and “A bonus point is awarded for definitions that are immediately adjacent to the parenthesis”. After processing a large set of abbreviation-definition pairs, the results are clustered in order to identify spelling variants among the definitions. N-gram clustering is coupled with lookup into the MeSH hierarchy to further improve the clusters. Performance on a smaller subset of the gold standard yielded 85% recall and 94% precision; the author notes that 2 definitions identified by his algorithm should have been marked correct in the standard, resulting in a precision of 95%.[d]

[d] Results verified through personal communication with the author.

The work described in this paper arose because the authors found difficulties making the Park and Byrd algorithm work well on biomedical text. The rules it produces are very specific to the format of candidate abbreviations, and so many abbreviations were being represented by patterns that had not yet been encountered by the algorithm, and thus rule priority was not often applicable.

The approach closest to the one we present here is the algorithm of Yoshida et al.[20] Their algorithm assumes that the definition or the abbreviation occurs adjacent to parentheses, but their paper does not state how the length of candidate definitions is determined. Their algorithm scans words from the end of the abbreviation and candidate definition to the beginning, trying at each iteration to find a match for the substring of the abbreviation in the definition. The algorithm assumes that in order for a character from the abbreviation to be represented in the interior of a word in the definition, there must be a match of some other character from the abbreviation on the first letter of that word. In addition, characters that match in the interior of the word must either be adjacent to one another following that initial letter, or adjacent to one another following a syllable boundary. Each iteration of the algorithm requires a check to see if a subsequence can be properly formed according to these rules. They test this algorithm on a very large collection (they had an independent assessor evaluate more that 15,000 categorizations), achieving 97.5% precision and 95.5% recall.

Another important processing issue for abbreviations is disambiguation of multiple senses of the same short form. Pustejovsky et al.[13] describe an algorithm that yields abbreviation sense disambiguation accuracies of 98%, and Pakhomov[9] achieves accuracies of 89% on clinical records.

Yet another issue is normalization of different spellings of the same abbreviation. It is difficult to define what it means for two biomedical terms to refer to the same concept; Cohen et al.[6] provide one set of rules.

3 Methods and Implementation

3.1 Identifying Short Form and Long Form Candidates

The process of extracting abbreviations and their definitions from medical text is composed of two main tasks. The first is the extraction of <short-form, long-form> pair candidates from the text. The second task is identifying the correct long form from among the candidates in the sentence that surrounds the short form. Most approaches, including the one presented here, use a similar method for finding candidate pairs. Abbreviation candidates are determined by adjacency to parentheses.

The two cases are:

(i) long form ‘(‘ short form ‘)’

(ii) short form ‘(‘ long form ‘)’

In practice, most <short form, long form> pairs conform to pattern (i). Whenever the expression inside the parentheses includes more than two words, pattern (ii) is assumed, and a short form is searched for just before the left parenthesis (word boundaries are indicated by spaces). Short forms are considered valid candidates only if they consist of at most two words, their length is between two to ten characters, at least one of these characters is a letter, and the first character is alphanumeric. For simplicity, pattern (i) is assumed in the discussion below.

The next step is to identify candidates for the long form. The long form candidate must appear in the same sentence as the short form, and as in Park and Byrd[12], it should have no more than min(|A| + 5, |A| * 2) words, where |A| is the number of characters in the short form.

Although the algorithm of Park and Byrd allows for an offset between the short and long forms, we consider only long forms that are adjacent to the short form. For a given short form, a long form candidate is composed of contiguous words from the original text that include the word just before the short form.

3.2 Algorithm for Identifying Correct Long Forms

When the previous steps are completed there is a list of long form candidate words for the short form, and the task is to choose the right subset of words. Figure 1 presents the code that performs this task. The main idea is: starting from the end of both the short form and the long form, move right to left, trying find the shortest long form that matches the short form. Every character in the short form must match a character in the long form, and the matched characters in the long form must be in the same order as the characters in the short form. Any character in the long form can match a character in the short form, with one exception: the match of the character at the beginning of the short form must match a character in the initial position of the first (leftmost) word in the long form (this initial position can be the first letter of a word that is connected to other words by hyphens and other non-alphanumeric characters).

The implementation in Figure 1 uses two indices, *lIndex* for the long form, and *sIndex* for the short form. The two indices are initialized to point to the end of their respective strings. For each character *sIndex* points to, *lIndex* is decremented until a matching character is found. If *lIndex* reaches the beginning of the long form candidate list before *sIndex* does, the algorithm returns *null* (no match found).

```
/**
  Method findBestLongForm takes as input a short-form and a long-
  form candidate (a list of words) and returns the best long-form
  that matches the short-form, or null if no match is found.
**/
public String findBestLongForm(String shortForm, String longForm) {
  int sIndex;     // The index on the short form
  int lIndex;     // The index on the long form
  char currChar;  // The current character to match

  sIndex = shortForm.length() - 1;  // Set sIndex at the end of the
                                    // short form
  lIndex = longForm.length() - 1;   // Set lIndex at the end of the
                                    // long form
  for ( ; sIndex >= 0; sIndex--) {  // Scan the short form starting
                                    // from end to start
    // Store the next character to match. Ignore case
    currChar = Character.toLowerCase(shortForm.charAt(sIndex));
    // ignore non alphanumeric characters
    if (!Character.isLetterOrDigit(currChar))
      continue;
    // Decrease lIndex while current character in the long form
    // does not match the current character in the short form.
    // If the current character is the first character in the
    // short form, decrement lIndex until a matching character
   // is found at the beginning of a word in the long form.
    while (
      ((lIndex >= 0) &&
      (Character.toLowerCase(longForm.charAt(lIndex)) != currChar))
      ||
      ((sIndex == 0) && (lIndex > 0) &&
      (Character.isLetterOrDigit(longForm.charAt(lIndex - 1)))))
          lIndex--;
    // If no match was found in the long form for the current
    // character, return null (no match).
    if (lIndex < 0)
      return null;
    // A match was found for the current character. Move to the
    // next character in the long form.
    lIndex--;
  }
  // Find the beginning of the first word (in case the first
  // character matches the beginning of a hyphenated word).
  lIndex = longForm.lastIndexOf(" ", lIndex) + 1;
  // Return the best long form, the substring of the original
  // long form, starting from lIndex up to the end of the original
  // long form.
  return longForm.substring(lIndex);
}
```

Figure 1 – Java Code for Finding the Best Long Form for a Given Short Form

Otherwise, each time a matching character is found, *sIndex* and *lIndex* are decremented. When *sIndex* is at the initial (leftmost) character of the short form, a match is considered only if it occurs at the beginning of a word in the long form. This is accomplished by decrementing *lIndex* until it reaches a non-alphanumerical character or reaches the beginning of the long form. Only then is the character it is pointing to checked for a match against the character *sIndex* points to (this allows for matches in the beginning of the long form just before hyphens). *lIndex* is then decremented until it reaches a space, or the beginning of the long form (whichever comes first), in order to include all the words that are connected, usually by hyphens, to the leftmost matched word in the long form. Finally, the algorithm returns the substring of the original long form, starting from *lIndex* up to the end of the original long form.

To increase precision, the algorithm discards long forms that are shorter than the short form, or that include the short form as one of the words in the long form.[e]

To illustrate the algorithm, consider the following pair <*HSF, Heat shock transcription factor*>. The algorithm starts by setting *sIndex* to point to the end of the short form (*HS**F***), and *lIndex* to point to the end of the long form (*facto**r***). It then decrements *lIndex* until a match is found (***f**actor*). *sIndex* is decremented by one (*H**S**F).* *lIndex* is decremented until a match is found (*tran**s**cription*). *sIndex* is decremented again (***H**SF*). Since *sIndex* now points to the beginning of the short form, the next match should be found at a beginning of a word in the long form. Therefore, *lIndex* is decremented until a valid match is found (***H**eat*). Note that another match was skipped (*s**h**ock*) because it was not in the beginning of a word. Also note, that although the algorithm did not match the second character correctly (*tran**s**cription* instead of ***s**hock*) it still found the right long form.

To illustrate when the algorithm might fail, consider the following example. <*TTF-1, Thyroid transcription factor 1*>. In this case the algorithm finds the following wrong match <*TTF-1, Thyroid **t**ranscript**i**on **f**actor **1***>. Our experiment results show that this kind of error is very rare.

The algorithm is based on the observation that it is very rare for the first character of the short form to match an internal letter of the long form. By adding the constraint that the first character of the short form matches the beginning of a word in the long form, together with the limitation on the length of the long form, the precision is increased by removing most of the false positives, without significantly reducing the recall. By contrast, adding additional constraints, as is done by most other algorithms, does not seem to help much in terms of precision, but can severely reduce the recall. To illustrate this point consider the results of Yu et al.[21], which is a similar algorithm to ours, but has additional constraints. While the precision of both algorithms is very similar, the recall of our algorithm is higher.

[e] This part of the algorithm is omitted from Figure 1.

4 Evaluation and Results

To evaluate the algorithm, 1000 MEDLINE abstracts were randomly selected from the results of a query on the term "yeast". These were then hand tagged, producing a list of 954 correct <short form, long form> pairs. The algorithm was also tested against a publicly available tagged corpus, the Medstract Gold Standard Evaluation Corpus,[22] which includes 168 <short form, long form> pairs.

On a corrected version of the gold standard, the algorithm identified 143 pairs. Out of these, 137 pairs were correct, resulting in a recall of 82% at precision of 96%. For comparison, the algorithm described in Chang et al.[5] achieved 83% recall at 80% precision, and that of Pustejovsky et al.[14] achieved 72% recall at 98% precision.[f]

Analysis of the 6 incorrect pairs reveals that in actuality, 2 of them are correct, but were overlooked by the creators of the gold standard.[g] The other 4 pairs are counted as incorrect, since they only partially matched the correct long form. For example, the algorithm found the pair <Pol II, polymerase II> instead of <Pol II, RNA polymerase II>. Allowing for reasonable partial matches, as was done in Chang et al.[5] and considering the 2 missing pairs as correct, the precision is increased to 99%, and the recall to 84%.

The algorithm missed 31 pairs: 9 (38%) pairs have skipped characters in the short form (e.g. <CNS1, cyclophilin seven suppressor>), 7 (23%) do not have any pattern match between the short form and long form (e.g. <5-HT, serotonin>), 5 (16%) have an out of order match (e.g. <ATN, anterior thalamus>), for 3 (10%) pairs the long form includes an additional words to the left of the match resulting in a partial match (e.g. <Pol I, RNA polymerase I>), 2 pairs have a short definition inside the parenthesis, 2 pairs have the long form inside parenthesis and the short form inside nested parenthesis, 1 pair has a comma in the long form, 1 pair has no parenthesis, and for 1 pair the algorithm found a wrong partial match (see the example at the end of section 3).

---

[f] It is important to note that because of the errors in the gold standard these results cannot be accurately compared. Each of these evaluations used its own interpretation of the standard, fixing different parts of it. In our evaluation, we followed the guidelines of Pustejovsky et al.[14] (received through personal communication), since they have developed and maintained the standard. We did not include here the results of Adar[1], since he used a subset of the original gold-standard with only 144 pairs, which did not include most of the pairs missed by the other algorithms.

[g] The two missing pair are <l'sc, lethal of scute>, and <cAMP, 3',5' cyclic adenosine monophosphate>. The second long form was only partially extracted by the algorithm (without the leading numbers).

On the larger test collection, the algorithms identified 827 pairs. Out of these, 785 pairs were correct, resulting in a recall of 82% at precision of 95%.[h] Analysis of the 42 incorrect pairs reveals that 17 are completely incorrect, 12 pairs have a matched long form that is a superset of the correct long form (this happens when the correct mapping includes unused characters, out of order mappings, first-character matches to internal letters, or when the first word of the correct long form is connected by hyphens to preceding words), 11 are partial matches where the extracted long form is a subset of the correct long form (this happens when the algorithm is able to match all the characters of the short form to a subset of the long form, and when the correct match does not start from the first word of the long form), and for 2 pairs the extracted long form includes left brackets that are not part of the correct long form.

Out of the 169 missed pairs, 70 (41%) have unused characters in the short form, 23 (14%) have an out of order match, 12 (7%) have first-character matches to internal letters, 12 (7%) have nested parentheses, 11 (7%) have some kind of transformation involved in their mapping (like 2D -> two-dimensional), 11 (7%) have partial matches (see above), 5 (3%) have a non-continuous long forms, 4 (2%) involve multiple concurrent definitions, 4 (2%) have a short form of only one character, and the rest of the pairs, 19 (11%), have miscellaneous issues.

5 An Alternative Algorithm

When we began to investigate the problem of abbreviation definition identification, we devised a much more complex algorithm than that presented here. This algorithm uses the representation of Park and Byrd[12] in combination with a variation on the decision lists algorithm, as applied by Yarowsky[18] to the lexical ambiguity resolution task. The algorithm makes use of training data to rank features that are combinations of matching rule transformations.

Space restrictions prevent detailed description of that algorithm (the interested reader should refer to Schwartz and Hearst[16] for a complete description of the algorithm). However, we found that it performed mildly better than our simple algorithm on both training sets, achieving for the gold standard 97% precision and 82% recall, which is a reduction in error of 17% over the simpler algorithm. For the larger test collection, it achieves 96% precision and 82% recall, which is an error reduction of 22% over the simpler algorithm.

[h] The dataset was originally annotated by a graduate student in computational and biosciences. We furthered verified the data by comparing any questionable pairs against other occurrences of the same abbreviation in other abstracts, using the web site provided by Chang et al.[5] A pair extracted by the algorithm is considered correct only if it exactly matches a pair labeled in the dataset.

Because the simple algorithm is so much easier to implement and requires no training data, we recommend its use, in combination with checking the entire dataset for redundancy in definitions in order to further reduce the error rates.

6 Conclusions

In this paper we introduced a new algorithm for extracting abbreviations and their definitions from biomedical text. Although the algorithm is extremely simple, it is highly effective, and is less specific – and therefore less potentially brittle – than other approaches that use carefully crafted rules. Although we are staunch advocates of machine learning approaches for problems in computational linguistics, it seems that in the case of this particular problem, simpler is better. One can argue that the problem may vary across collections or languages, and so machine learning can help in these cases, but our experience with a machine learning approach to sentence boundary determination[11] suggests that most practitioners do not want to bother with labeling training data for relatively simple tasks.

Another advantage of the simplicity of the algorithm is its fast running time performance. The task of extracting the definition of an abbreviation, is a common pre-processing step of larger multi-layered text-mining tasks.[1, 10] Therefore, it is essential that this step be as efficient as possible. Since our algorithm needs to consider only one possible long form per short form, it is much faster than the alternative algorithms that first extract many possible long forms and then pick the best of them. To provide a rough comparison, using an IBM T21 laptop with a single CPU (800 MHz, 256 Mb RAM) running MS-Windows 2000, it takes our algorithm about 1 second to process 1000 abstracts, while the algorithm in Chang et al.[5] using a 5 processor Sun Enterprise E3500 server, processed only 25.5 abstracts per second. While our algorithm is clearly I/O bound (running time depends almost entirely on the time it takes to read the files from disk, and write the results back to the disk), the algorithm of Chang et al. seem to be heavily CPU bound.

The algorithm performs better or the same as the best results of other work, with the possible exception of that of Yoshida et al.[20] However, the main advantage of the proposed algorithm over the alternatives is its simplicity, and transparency. It was implemented with 260 lines of Java code and requires no training data to run. The Yoshida et al. algorithm is more complex, in that it requires a module for recognizing syllable boundaries, and it performs a substring check at each iteration of the loop.

Analysis of the errors produced indicates that further improvement of the algorithm requires the use of syntactic information, as suggested in Pustejovsky et al.[13] Shallow parsing of the text as a preprocessing step might help correct some of the errors inherent in the algorithm, by helping to identify the noun phrases near the

abbreviations. In addition, combining evidence from more than one MEDLINE abstract at a time, as was done in Adar[2], might also prove to be beneficial for increasing both precision and recall. Finally, the algorithm currently only considers candidate definitions when the abbreviation is enclosed in parentheses (and vice versa); finding all possible pairs is a more difficult problem and requires additional study.

Acknowledgements

We thank Barbara Englehardt for producing the marked-up data for the large test collection, and Jeff Chang and Hinrich Schuetze, Youngja Park and Roy Byrd, Jose Castaña and James Pustejovsky, and Eytan Adar for providing details about their data. This research was supported in part by a gift from Genentech, and in part by a grant from ARDA.

References

1. L. A. Adamic, D. Wilkinson, B. A. Huberman, and E. Adar, "A Literature Based Method for Identifying Gene-Disease Connections" *Proceedings of the 2002 IEEE Computer Society Bioinformatics Conference*, Stanford, CA, August 2002.
2. E. Adar, "S-RAD: A Simple and Robust Abbreviation Dictionary" *HP Laboratories Technical Report*, September 2002.
3. M.A. Andrade et al. "Automatic annotation for biological sequences by extraction of keywords from MEDLINE abstracts. Development of a prototype system." *ISMB 1997*. **5**: 25-32.
4. C. Blasche et al. "Automatic extraction of biological information from scientific text: protein-preotein interactions" *ISMB 1999*. **7**: 60-67.
5. J.T. Chang, H Schütze, and R.B. Altman, "Creating an Online Dictionary of Abbreviations from MEDLINE" *JAMIA*, to appear.
6. B. Cohen, A. E. Dolbey, G. K. Acquaah-Mensah, and L. Hunter, "Contrast and variability in gene names" *Proceedings of the ACL Workshop on Natural Language Processing in the Biomedical Domain*, Philadelphia, PA, July 2002.
7. M. Craven, and J Kumlien, "Constructing Biological knowledge Bases by Extracting information from Text Sources" *Proceedings of the 7th International Conference on Intelligent Systems for Molecular Biology*, 77-86, Heidelberg, Germany. AAAI Press.
8. L.S. Larkey, P. Ogilvie, M.A. Price, and B. Tamilio, "Acrophile: an automated acronym extractor and server" *Proceedings of the ACM Fifth International Conference on Digital Libraries*, DL '00, Dallas TX, May 2000.

9. S. V. Pakhomov, “Semi-supervised Maximum Entropy-based Approach to Acronym and Abbreviation Normalization in Medical Texts” *Proceedings of ACL 2002*, Philadelphia, PA, July 2002.
10. M. Palakal, M. Stephens, S. Mukhopadhyay, R. Raje, and S. Rhodes, “A Multi-level Text Mining Method to Extract Biological Relationships” *Proceedings of the 2002 IEEE Computer Society Bioinformatics Conference*, Stanford, CA, August 2002.
11. D. Palmer, and M. Hearst, “Adaptive Multilingual Sentence Boundary Disambiguation” *Computational Linguistics*, 23 (2), 241-267, June 1997.
12. Y. Park, and R.J. Byrd, “Hybrid Text Mining for Finding Abbreviations and Their Definitions” *Proceedings of the 2001 Conference on Empirical Methods in Natural Language Processing*, Pittsburgh, PA, June 2001: 126-133
13. J. Pustejovsky, J. Castaño, B. Cochran, M. Kotecki, M. Morrell, and A. Rumshisky, “Extraction and Disambiguation of Acronym-Meaning Pairs in Medline” unpublished manuscript, 2001.
14. J. Pustejovsky et al. “Automation Extraction of Acronym-Meaning Pairs from Medline Databases” *Medinfo* 2001;10 (Pt 1): 371-375.
15. T.C. Rindflesch et al. “Extracting Molecular Binding Relationships from Biomedical Text” *Proceeding of the ANLP-NAACL 2000*, pages 188-195 Association for Computational Linguistics, Seattle, WA, 2000.
16. A. Schwartz, and M. Hearst, “A Rule Based Algorithm for Identifying Abbreviation Definitions in Biomedical Text Using Decision Lists” *University of California, Berkeley, Technical Report*, to appear.
17. K. Taghva, and J. Gilbreth, “Recognizing Acronyms and their Definitions” *IJDAR 1* (4), 191-198, 1999.
18. D. Yarowsky, “Decision Lists for Lexical Ambiguity Resolution: Application to Accent Restoration in Spanish and French” *Proceedings of the ACL, 1994*: 88-95
19. S. Yeates, D. Bainbridge, and I. H. Witten, “Using compression to identify acronyms in text” *in Data Compression Conference*, 2000.
20. M. Yoshida, K. Fukuda, and T. Takagi, “PNAD-CSS: a workbench for constructing a protein name abbreviation dictionary” *Bioinformatics*, 16 (2), 2000.
21. H. Yu, G. Hripcsak, and C. Friedman, “Mapping abbreviations to full forms in biomedical articles” *J Am Med Inform Assoc 2002*; 9(3): 262-272.
22. http://www.medstract.org/gold-standard.html