Named Entity Recognition and Normalization: A Domain-Specific Language Approach

Miguel Vazquez[1,*], Monica Chagoyen[2,3], and Alberto Pascual-Montano[2]

[1] Departamento de Ingeniería del Software e Inteligencia Artificial
miguel.vazquez@fdi.ucm.es
[2] Dpto. Arquitectura de Computadores, Universidad Complutense de Madrid, Madrid, Spain
[3] Biocomputing Unit, Centro Nacional de Biotecnología - CSIC, Madrid, Spain

Summary. We present, RNer, a tool that performs Named Entity Recognition and Normalization of gene and protein mentions on biomedical text. The tool we present not only offers a complete solution to the problem, but it does so by providing easily configurable framework, that abstracts the algorithmic details from the domain specific. Configuration and tuning for particular tasks is done using domain specific languages, clearer and more succinct, yet equally expressive that general purpose languages. An evaluation of the system is carried using the BioCreative datasets.

1 Introduction

Three factors have motivated the biomedical community to get interested in literature mining [10]. The first one is due to the recent high throughput techniques, like DNA microarrays, that, by involving large collections of entities, place bigger demands in the researcher's use of previous knowledge. The second factor is the fast pace at which scientific articles are being published, making it hard for researchers to keep up to date. Coupled with these is the trend followed by many editorials of making the articles available on-line, thus, making them readily available for analysis by software tools.

Named entity recognition (NER) is an important tool for literature mining, as it is found as an initial step in many information extraction applications [11]. Named entity recognition has been used in domains other than the biomedical, like news wire text. It is in the biomedical domain, however, where it presents the most challenges. Gene and protein names are the amongst the most common subjects for NER in the biomedical domain. They have several characteristics that makes them specially challenging. They are very numerous, in the order of millions, the number growing continuously, and the names used to refer to them are not necessarily standardized [7]. Furthermore, in many cases the names are the same across different genes or match some common English words, producing a significant amount of ambiguity [1]. These problems not only affect finding the mentions in the text, but also the subsequent task of identifying the actual entity

* Corresponding author.

J.M. Corchado et al. (Eds.): IWPACBB 2008, ASC 49, pp. 147–155, 2009.
springerlink.com

they refer to. This identification step is usually referred to as normalization, and is as much a challenging task as NER itself [2].

Several initiatives have dealt with the problem of NER in the biomedical context. We have studied closely the BioCreative competition [11, 2], which had several tasks relating to NER and normalization. We used some of the resources they provided to evaluate our system. As for the availability of tools for NER and normalization, we have found several freely available tools for the NER step alone. Some of them, like Abner and Banner [8, 6] we have studied in detail. At the time we did not find any available tool that implemented the normalization step, which motivated us to develop our own.

Examining the state of the art in NER reveals a strong trend towards the use of conditional random fields (CRF). These probabilistic models have proven to be appropriate for text labeling [5], which is how NER is usually approached. For NER, we use CRF to predict labels for a sequence of words given a set of features associated with each word. Most of the work on NER using CRF differs basically in the features used. In fact, the nature of the CRF algorithm reduces the problem to determining a suitable set of features. Most common features are orthographic: Does the word have any uppercase letter? Any digits? Is there a slash character in the word? Etc.

We developed a system, which we call RNer, for both NER and normalization that implemented a standard approach, but offered a practical way to be extended and configured. For NER, we implemented the same ideas from Abner and Banner, abstracted the core inner workings of the algorithm and committed all the details on creating the features to a domain specific language (DSL). For the normalization task we developed our own solution, based on some ideas from the BioCreative competition, and also separated all the configuration options to domain specific languages. The Ruby programming language was used in the implementations, mainly because its clear syntax and meta-programming possibilities makes defining DSL a very easy task.

The advantage of the use of DSLs is that it allows to integrate and test new ideas, for both NER and Normalization, using a more descriptive language than the original programming language. This makes adding new functionality closer to configuration than it is to programming, which has enabled us to produce a working tool with competitive performance with very little effort.

2 System Overview

This is an overview of the process we have implemented. Following sections will cover each aspect in more detail, but this section should be useful to gain perspective for further discussion.

The CRF model for NER is trained off-line. The training text is turned into a stream of words, with the words forming part of the entity mentions labeled accordingly. The model is trained so that, when a new stream of words comes along, it can assign the labels to determine mention boundaries.

The Normalization system is not trained like the NER, but rather, a data structure is generated from a lexicon file to help identify the mentions. The lexicon file contains each gene, along with the list of identifiers with which it is commonly referred to in the literature, usually in a per-organism basis. The Normalization works by matching the mentions found with the synonyms in the lexicon by progressively transforming each into simpler forms until a match, or matches, are made. The matches so found are then evaluated according to a set of rules to produce a match score. This match score is used to select the best match, or to reject them all, if none is found to be good enough. If two matches have the same score, the system selects the best match by comparing the overlap between the words in the text where the mention was found, and the text describing the gene in the correspondent Entrez GeneRIF entry.

3 Named Entity Recognition

The approach we follow for NER is based on that of Abner. We will describe briefly how it works. A more detailed description can be found in [5] or [9].

Finding mentions in text means determining the words that are likely to constitute a gene name. This is commonly done by labeling the words with their position relative to the mentions. The `IOB` schema that we use [8], has the label `B` mark the initial word on a mention, label `I` mark any other word in the mention, and label `O` is used for words that are outside of any mention. Named Entity Recognition is done by assigning these labels to a new sequence of words.

The labeling of a sequence of words using Conditional Random Fields is done by representing each word by a set of features, and using a probabilistic model to determine the sequence of labels that was most likely to have produce such a sequence of features. This model is built in an off-line, phase using example data, which, in our case, is the one provided as training for the BioCreative competition. Optionally, an specific NER model can be build for each organism by including in the training data the gene synonyms found in the organism lexicon.

Our system uses CRF++ [4] to build the model and produce the labels, as it provides bindings for Ruby. Abner and Banner use Mallet, a Java implementation of CRF with similar characteristics. The main difference between all three systems resides in the features used to represent each word. Abner defines a set of features based on morphological features, these features are further expanded in Banner, which includes semantic features as well. Our system abstracts all the feature generation from the rest of the system, and provides a DSL language to specify it. The features used to evaluate our system include those in Abner and Banner, excluding the syntactic based ones, as they slowed the system down considerably, and they did not seem to significantly enhance performance in our application.

We will now take a look at the DSL used to define the features. Figure 1 shows the definition of three of the features exemplifying the three ways to do this. The first one defines the feature `hasDigits` with a regular expression, so that

```
hasDigits     /\d/
prefix_3      /^(...)/
downcase      do |w| w.downcase end
```

Fig. 1. Feature declaration

if the word has a digit, then the feature is true. In the second case, `prefix_3` is defined with a regular expression capture, in this case the feature value is what ever is captured in by the parenthesis, the first three characters. The last example uses a code block, the result of which, the word in lower case, is assigned to the feature. These three features, along with 25 others, are described in the default configuration file, and can be easily over written, and extended. Regular expressions are used since they themselves are a DSL for string matching, and arguably the best way to capture the requisites of this particular domain.

The CRF tool that we use, CRF++ [4], has a particular characteristic of being able to consider a certain context around each word at any particular step, as opposed to just the current state and word features. This context may be defined in another section of our DSL, as shown in figure 2, which just states that features `special, token2, isPunctuation` and `isDelim` from words surrounding the current one in distance of up three should be considered. The features used here must have been defined in the previous DSL.

```
context_features   %w(special token2 isPunctuation isDelim)
context_window     %w(1 2 3 -1 -2 -3)
```

Fig. 2. CRF++ context

4 Normalization

Normalization is the name given to the process of identifying to what gene or protein a mention found in the text refers to. The BioCreative competition features some methods that do not separate NER from normalization. However the most common choice is to consider both process separate.

Our approach to Normalization is a 3 step pipeline. Mentions are assigned a number of candidate genes. The choice for each candidate is scored, filtered, and sorted; and the best is selected, if any survive the filtering. In the case that a draw still remains after scoring, it is resolved in a so called disambiguation step. Let us look at these three process in more detail: candidate matching, scoring and disambiguation.

Candidate matching is done using a ordered list of string matching functions, each one been more permissive than the previous one. The idea behind this setup is to find the best matches as soon as possible. This early stop avoids generating an unnecessary large list of candidates. However, functions at the end of the list should allow for lax enough matching, so that candidates are produced, even if unlikely; the following scoring step will use a more sophisticated method to

```
equal    do |w| [w] end
standard do |w| [w.downcase.split(/\s+/).sort.join("")] end
words    do |w|  w.scan(/[a-z]+/i)  end
```

Fig. 3. Declaration of name indexes

reevaluate their appropriateness. The matching functions are implemented using cue indexes. Each function turns a gene name into a list of cues, forming an index that associates each cue with the list of genes having the correspondent names. When a mention to a gene is found in the text, the system produces the cues for the first index, and checks if any matches are made; if not, it repeats the process for the next index. This is carried along until a match is made, or the last index also fails.

The system only needs the cue generator functions, which are defined using a DSL like in figure 3. In this abridged example the first index function cue is the name or mention as-is; the second takes each word in the mention, sorts them, and joins them together in lowercase –this is a more accommodating cue; whereas the third index returns a list of cues composed of the words in the name or mention. Having in common any of the cues in the list will be enough to produce a match. For example, the mention `lactamase beta 2` would generate the cues `lactamase beta 2` for the *equal* index, `2betalactamase` for the *standard* index and `lactamase`, `beta`, and `2` for the *words* index.

The candidates generated in the previous process might be numerous, including plenty of false positives. To evaluate how good a match each candidate gene is, we compare each of the synonyms the gene has with the mention, generating similarity scores. The candidate gene is assigned the score of the best ranking name. The score is calculated as follows. Both mention and names are chunked down into tokens, and each token is assigned a token class. Each token class is then examined to evaluate the overlap in tokens. Each overlapping scenario is assigned a positive or negative score, and the total sum is the similarity score. The system uses a DSL to specify the token classes and another DSL to specify the cases and their scores.

The three examples in figure 4 illustrate the three ways to define the token types. The first one identifies a token as a roman numeral using a regular expression. If only the name of the type is given, as done in the second case, a default regular expression is attached, matching the same name, case insensitive, and with a possible "`s`" character at the end, to account for possible plurals (This is offered for convenience, an explicit regular expression could be used for cases that do not adhere to this pluralization rule). The last example uses a code block to make this check, in this particular case, it makes use of a hash of Greek letter

```
roman        /^[IV]+$/
promoter
greek        do |w| $greek[w.downcase] != nil end
```

Fig. 4. Token types

```
same.greek     5
miss.greek     -3
diff.promoter  -10
```

Fig. 5. Token comparison weights

names to check if the token is one of them. This, again, is done case insensitive. Tokens not assigned to any of the defined classes are assigned the class *other*. For example, `Lactamase Beta 2` would have a token of class *number*, which will be 2, a token of class *greek*, which will be `B`, and a token of class *other*, which will be `lactamase`. The tokens proposed in the application by default include 12 rules and 11 additional words.

Example figure 5 shows three ways to specify the scores that receive the different cases that could arise when comparing two token classes. The first states that if the same Greek letter names are found in the two, a value of 5 is added to the similarity measure. The second states that if the mention is missing a Greek letter, 3 is subtracted, and the last one states that if one of them has the token `promoter` and the other does not, 10 is subtracted. We have six operators: `same, distinct, common, diff, miss,` and `extra`. Meaning that they have the same tokens, no tokens in common, at least one in common, at least one different, the mention is missing one or the mention has an extra token. For example, the mention `Lactamase 2` and the name `Lactamase Beta 2` would have a common *other* token, a common *number* token, but the mention would be missing a *greek* token.

```
transform.roman do |t| [t[0].arabic, 'number'] end
compare.number do |l1,l2|
  val = 0
  val -= 4 if (l1 - l2).length >0 || (l2 - l1).length >0
  val -= 8 if l1[0]  != l2[0] &&  l1[0]
  val += 3 if l1[0]  == l2[0]
  val
end
```

Fig. 6. Advanced comparisons: Transformations and custom comparisons

In order to add more flexibility to this method, we have added two other operators: *transform*, used to change tokens from one type to another, and *compare*, that allows comparing a certain type of tokens using a code block. Figure 6 holds an example of both. The first one turns a roman to a number, allowing the number `1` to match `I` in roman form, for example. The second compares the numbers of mention and name in a finner grained manner.

We now have the ability to assign a similarity score between a mention and each of its candidate genes. For each candidate gene the best score amongst those from all it's known synonyms is selected. We use these score to rule out those candidates that score to low, and also to establish a ranking amongst those who score high enough. If there are several candidate genes containing the same

synonym (as it often occurs) they will score the same. In order to resolve the draws, we have a final step of disambiguation.

The disambiguation step is done by comparing the context in which the mention occurs, the paragraph for example, with the description of that gene in the Entrez Gene database. The comparison is made by finding the words in common between the text in the mentions context and the text in the gene description, and adding their word weights. The word weights are their inverse document frequency [3], calculated from a corpus of example text drawn from PubMed article abstracts.

5 Evaluation

The NER system comes with a set of default configurations, mostly copied from Abner, but with a few additions. The system performs fairly well with these defaults for gene mention recognition. We plugged our system into the BioCreative competition evaluation sets using these defaults. The results are shown in table 1, we downloaded Abner and Banner and performed the same tests. RNer, as we will call our system, seems to outperform Abner slightly. Banner performs significantly better than both, possibly because it uses syntactic information as features, which our default configuration did not include.

The Normalization step also comes with a default configuration. We show results for the BioCreative I task 1-B in tables 2 and 3. These results are obtained using only the default configuration, no specific tuning is performed for either organism, nor is the provided training data used in any way. We have performed the analysis using our NER system, as well as Abner and Banner. Our NER

Table 1. Results for the BioCreative I task 1-B

System	Precision	Recall	F-Measure
RNer	0.819	0.780	0.799
Abner	0.789	0.741	0.764
Banner	0.836	0.828	0.832

Table 2. Results for the yeast dataset of the BioCreative I task 1-B

NER	Precision	Recall	F-Measure
RNer	0.936	0.863	0.898
Abner	0.941	0.809	0.870
Banner	0.933	0.816	0.870

Table 3. Results for the mouse dataset of the BioCreative I task 1-B

NER	Precision	Recall	F-Measure
RNer	0.695	0.645	0.669
Abner	0.680	0.649	0.664
Banner	0.666	0.686	0.675

system outperforms Abner and Banner in the yeast dataset. Banner, however, outperforms both in the mouse dataset.

6 Discussion

Ruby, our implementation language, proved to be very useful due to its clear syntax and meta-programming possibilities. Ruby can also be compiled into Java bytecode and used in any Java framework. The result is a simple to use system that works out of the box but at the same time is flexible and easy to configure and adapt to other domains. Domain specific languages allow us to describe the specifics of the system in a more clear and succinct way.

While the system does not, for the time being, beat any record in performance, it shows competitive and promising results, even though no organism based tuning is performed and the training data for normalization, provided in BioCreative, is not being used.

7 Availability

This system was developed as part of another system called SENT, available at `http://sent.dacya.ucm.es`. Very likely, this system will be made available to public domain when SENT is published, along with the rest of the code that composes the SENT. In the meantime it will be provided free upon request.

Acknowledgements

This work has been partially funded by the Spanish grants BIO2007-67150-C03-02, S-Gen-0166/2006, CYTED-505PI0058, TIN2005-5619. APM acknowledges the support of the Spanish Ramón y Cajal program.

References

1. Chen, L., Liu, H., Friedman, C.: Gene name ambiguity of eukaryotic nomenclatures. Bioinformatics 21(2), 248–256 (2005)
2. Hirschman, L., Colosimo, M., Morgan, A., Yeh, A.: Overview of BioCreAtIvE task 1B: normalized gene lists. BMC Bioinformatics 6(1), 11 (2005)
3. Jones, K.S., et al.: A statistical interpretation of term specificity and its application in retrieval. Journal of Documentation 28(1), 11–21 (1972)
4. Kudo, T.: Crf++: Yet another crf toolkit (2005)
5. Lafferty, J.D., McCallum, A., Pereira, F.C.N.: Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In: Proceedings of the Eighteenth International Conference on Machine Learning table of contents, pp. 282–289 (2001)
6. Leaman, R., Gonzalez, G.: Banner: An Executable Survey Of Advance. In: Biomedical Named Entity Recognition. In: Pacific Symposium of Biocomputing (PSB) (2008)

7. Leser, U., Hakenberg, J.: What makes a gene name? Named entity recognition in the biomedical literature. Briefings in Bioinformatics 6(4), 357–369 (2005)
8. Settles, B.: Abner: an open source tool for automatically tagging genes, proteins and other entity names in text (2005)
9. Settles, B., Collier, N., Ruch, P., Nazarenko, A.: Biomedical Named Entity Recognition using Conditional Random Fields and Rich Feature Sets. In: COLING 2004 International Joint workshop on Natural Language Processing in Biomedicine and its Applications (NLPBA/BioNLP) 2004, pp. 107–110 (2004)
10. Shatkay, H., Feldman, R.: Mining the Biomedical Literature in the Genomic Era: An Overview. Journal of Computational Biology 10(6), 821–855 (2003)
11. Yeh, A., Morgan, A., Colosimo, M., Hirschman, L.: BioCreAtIvE task 1A: gene mention finding evaluation. BMC Bioinformatics 6, 1 (2005)