Annotating DBWorld Messages Using Domain Resources

James Chan Xiaolei Li Linus Wong

University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA

Abstract

The *DBWorld* mailing list is a primary channel of communication for database researchers around the world. Conference call-for-papers, job postings, book releases, and many other types of announcements are routinely posted in *DBWorld*. It has become an indispensable tool for researchers who wish to stay up-to-date with the research community. Unfortunately, the only means of distribution of the mailing list is through email or flat text copies on the web. In the world of hyperlinks and semantic data, this format is rather inadequate. A more useful format would be a fully annotated and linked version of the message. All persons' names, conference names, university names, and others would be linked to their appropriate homepages. This presentation would allow the reader to quickly access content on the web related to the message.

We propose to do exactly this task. Given a *DBWorld* message, we wish to annotate all relevant entities and link them to their appropriate websites (or potentially a set of related websites). To this end, we propose a system which will use many domain-specific resources and perform the proposed task. In this work, we will describe the components of our system and show experiment results.

1 Introduction

The database research community is one of the largest communities in computer science. There are hundreds, if not thousands, of conferences and journals devoted to the community. The number of researchers is probably several orders of magnitude beyond that. *DBLP* alone registers over 375,000 researchers. With such a large community, means of effective communication becomes very important. Conferences need to inform potential participants of deadlines in a timely manner, and schools need to inform recent graduates of open faculty positions.

For this purpose, the *DBWorld* mailing list was created. It is a primary channel of communication for database researchers around the world. Conference call-for-papers, job postings, book releases, and many other types of announcements are routinely posted in *DBWorld*. It has become an indispensable tool for researchers who wish to stay up-to-date with the community.

Unfortunately, the only means of distribution of the mailing list is through email or flat text copies on the web. In the world of hyperlinks and semantic data, this format is rather inadequate. If one wanted to explore the topic of the message further, he/she would have to explore separately using Google or whatever tools that are available. The message itself is not syntactically linked to objects on the web, even though it is semantically buried with entities. Instead, a much more useful format would a fully annotated and linked version of the message. All persons' names, conference names, university names, and others would be linked to their appropriate homepages. This presentation would allow the reader to quickly access content on the web related to the message.

We propose to do exactly this task. Given a *DBWorld* message, we wish to annotate all relevant entities and link them to their appropriate websites (or potentially a set of related websites). To this end, we propose a system named DBDebbie[1], which will use many domain-specific resources and perform the proposed task. Combining domain resources such as *DBLP* and *DBLife* with other features extracted from the message, we first perform named entity extraction, a well-studied problem in machine learning. Beyond this, we have a linker module which feedbacks to the classifier and links the found entities to appropriate webpages on the web.

The rest of the paper is organized as follows. In Section 2, we show the overall framework of DBDebbie. In Section 3, we describe the named entity extractor. The linker is discussed in Section 4. In Sections 5 and 6, we show experimental results and screenshots of the system. We present some discussions in Section 7 and conclude in Section 8.

CS498 AD project report. Fall, 2004.

[1] The name DBDebbie carries no deeper meaning. We chose it simply because it sounds cool.

2 Overall Framework

The DBDebbie system consists of two main modules: the named entity extractor and the entity web linker. Figure 1 shows the layout of the modules and also the flow of data through the framework.

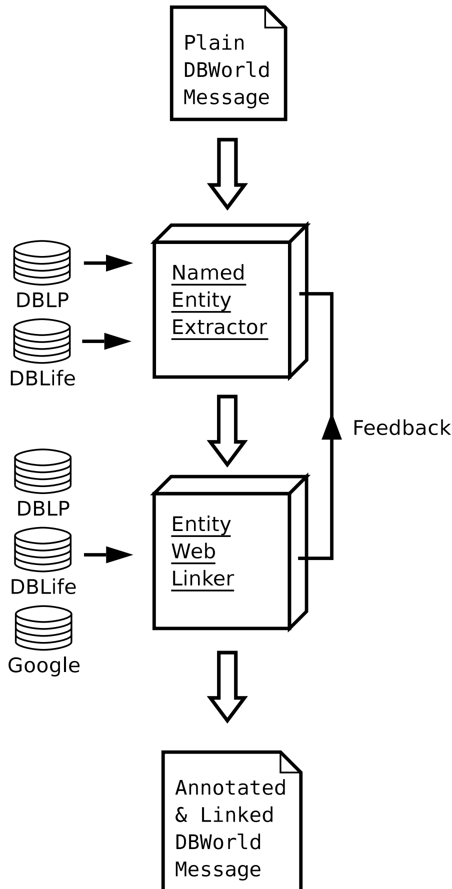

Figure 1: DBDebbie framework

The input message to the overall system is a simple flat text file and the output is an annotated version of the file in HTML, XML, or other desired format. The exact format of the output is not the focus of the system.

The input to the named entity extractor module is again a flat text and the output is the file with all the potential named entities annotated. The types of entities we are concerned with include names, conferences, and universities. This annotated file is then fed as input into the linker module, which finds URLs or sets of URLs for each annotation. In finding these URLs, it will also make some refinements on the entities. The next two sections will discuss these modules in detail.

3 Named Entity Extractor

The first module in DBDebbie is to extract named entities (NE) in the document. This is an age-old problem in machine learning with many potential solutions. In DBDebbie, we did not use a general purpose NE extractor, because we had many domain resources. In our design process, we tried to leverage as many of these resources as possible. Using a full-blown Hidden Markov Model (HMM) or some other generative model might have been overkill for this problem. Instead, we chose to use a simpler classifier to determine if a phrase was an entity or not.

3.1 Candidate Phrase Generation

Candidate phrases are simply substrings of the message. They are generated by sliding windows of various sizes. In our implementation, we chose windows of sizes one through five. For example, suppose we have the following sentence.

```
I like blue more than green.
```

The generated candidate phrases of window size one would be {"`I`", "`like`", "`blue`", "`more`", "`than`", "`green`", "`.`"}. Window size two would be {"`I like`", "`like blue`", "`blue more`", "`more than`", "`than green`", "`green.`"}, and so forth with windows of sizes three, four, and five. Note that punctuations are treated as tokens as well. For a document of n tokens, this generation process would then generate approximately $5n$ phrases. We chose 5 as the maximum phrase length because we believe most entities are expressed with fewer than 5 tokens (*e.g.*, only 2 is needed for a name usually). Of course, having a bigger max would not hurt the system in terms of accuracy. It would only increase the running time.

3.2 Feature Generation

After the phrases are extracted from the document, we then generate a feature vector for each phrase. The vector has 20 features, which can be separated into 3 main categories: *DBLP*, *DBLife*, and other. Table 1 shows a summary of the features.

For features 1–6, we downloaded all the data available in *DBLP* and *DBLife*, and performed similarity comparisons offline. We computed a simple similarity score that examined permutations and combinations on a word level. We chose not to use a complicated similarity metric because using one would imply pairwise comparisons between the input and many strings in *DBLP* and *DBLife*. This would become too expensive in practice.

Similarity scores for DBLP conferences and journals are real valued and defined in the following way:

$$DBLPScore(s) = \frac{|s \bigcap S_{min}|}{max(|s|, |S_{min}|)}$$

where s is the token string and S_{min} is the minimal length partial match in the DBLP dictionary. Single letter tokens and common prepositions are not considered as partial matches, however they do still contribute to the length of the token string. This was done

No.	Value Type	Description
1	Number	Score of best match in *DBLP* name list
2	Number	Score of best match in *DBLP* conference list
3	Number	Score of best match in *DBLP* journal list
4	Number	Score of best match in *DBLife* name list
5	Number	Score of best match in *DBLife* conference list
6	Number	Score of best match in *DBLife* university list
7	Binary	If phrase is all words & punctuation
8	Binary	If all words are nouns
9	Binary	If all words are nouns, determiners, or prepositions
10	Binary	If all punctuations are periods
11	Binary	If all words are capitalized
12	Binary	If all letters are capitals
13	Binary	If there is at least one word in phrase
14	Binary	If there is at least one word of length > 1 in phrase
15	Binary	If there is a linebreak in phrase
16	Binary	If there is a university-indicating word in phrase
17	Number	Number of GATE-marked words in phrase
18	Number	Number of words in phrase that are in the top 3000 most frequent English word list
19	Number	Number of tokens in phrase
20	Number	Number of spaces in phrase

Table 1: DBDebbie feature vector

to prevent extremely common tokens from diluting the scores.

DBLP Author scores were simply binary values based on whether the token string exactly matched a DBLP author or not. We also decided to added in rectified names; that is, names consisting of first and last name only. Thus, an author named "John F. Kennedy" could be matched with or without the middle initial.

Feature 6 in particular is not strictly based on entries within *DBLife*. Rather, we use the affiliations mentioned in that source as the initial dictionary, and augment this with additional universities derived from the Web through the use of GoogleSets to discover these new instances. We use pairs of universities from our initial dictionary to serve as seeds in a GoogleSet query, which returns new terms similar to the query terms. Although most results are relevant, we still must cull the result sets manually. This could easily be fully automated with a classifier similar to the one used for homepages in Section 4.

For features 7–20, many of them used orthographic features such as part of speech, capitalization, etc. These features were partially generated by GATE. GATE is an information extraction toolkit; we will discuss it in detail in Section 6. The university-indicating word list in feature 16 was hand-generated. It included words such as `University`, `Univ`, and `National`. The 3000 most frequent English words in feature 18 was found on the web. We also appended some handcrafted domain words to this list (*e.g.*, `database`, `xml`, `extraction`). For feature 17, GATE has some primitive NE extraction which labels countries, cities, phone numbers, etc. We simply counted them in the particular phrase.

3.3 Classification

Once the feature vectors are generated for every phrase, the problem becomes a traditional machine learning classification problem. We have a separate binary classifier for each entity type: name, conference, and university. They are trained independently and used in sequence. The classification process works as follows. First, we classify all phrases which are names. Then, we remove all the positive classifications and all phrases which overlap with them from the set of possible phrases. With the leftovers, we classify all phrases which are universities. Then, we prune the possible phrases again according to the classifications. And finally, we run the remaining windows through the conference classifier.

We chose this particular sequence of classification because of our own confidence in the classifiers. Also, it eliminates the problem of deciding which classifier to trust in case of overlapping classifications (*i.e.*, a single phrase being potentially classified positive by all 3 classifiers). Ideally, one should run the classifiers in parallel on all phrases and have some sort of confidence score to decide the final classification. A multi-class classifier would also work. In our implementation, we tried Naïve Bayes, Decision Tree, and Support Vector Machine (SVM).

To summarize, the NE extractor generates a set of candidate entity phrases from the document. For each phrase, a feature vector is constructed, which consists of lookups in *DBLP* and *DBLife* and other orthographic features. This vector is fed through three different classifiers in sequence and the corresponding will be labeled as a name, university, conference, or none. These labels are then passed on to the entity web linker module to be discussed in the next section.

4 Entity Web Linker

After the named entity extractor has annotated the entities in the document, the next step is then to link them to their appropriate websites on the internet. Again, we leverage the information available in *DBLP* and *DBLife* for this task. For every entity, we first perform a simple exact match in *DBLP* and *DBLife*. If a link is provided in there and does not return a 404 error in a browser, we choose that as the correct link. We believe this is the best choice because *DBLP* and *DBLife* are manually maintained systems.

For university and conference entities with no entries in *DBLP*or *DBLife*, we simply return the top Google result for a query consisting of the labeled entity phrase.

For name entities with no matches in *DBLP* or *DBLife*, we perform the following steps on each.

1. Fetch top n Google search results using the entity phrase as the query.
2. Generate feature vector for each resultant page.
3. Classify pages which are home pages.
4. For the positive classifications, compute similarity score between the page and the *DBWorld* collection.
5. Return the top k scoring pages.

Steps 1 and 5 are obvious. We discuss the rest in detail below.

4.1 Webpage Feature Vector

To represent web pages for our home page classifier to use, we convert each page into a feature vector of 50 elements derived from the URL and content of the page.

6 boolean features and 1 numerical feature are derived from the URL. The boolean features consider the presence or absence of: the words *cs* or *cse*, *edu*, *people*, *faculty*, a tilde, and the use of certain punctuation [?,+,=]. The numerical feature is the $\frac{\text{page depth}}{10}$ and capped at 1, *e.g.*, `http://www.somewhere.edu/~smith/` lies at depth 1 and therefore gets value 0.1.

The other 43 features are all boolean and extracted from the page contents. 2 features test for the phrase *home page* (with optional space) in the title and in any anchor text. The remaining 41 features test for particular keywords found to occur more often in people's homepages than non-homepages. To determine these words, we collect two equal sets of URLs, one set of valid homepages taken from *DBLife*, and the other set being non-homepages culled from Google queries on particular researcher names. Use of these particular non-homepages more accurately reflects the distribution of pages we expect to encounter in our Linker module. We then simply count the frequency of homepages containing each unique unique word, and subtract the frequency of these same words found in non-homepages. The top 41 words from this net set are taken as keyword features.

4.2 Home Page Classification

The home page classifier is trained using a set of positive and negative webpages in their feature vector representation. It is called on a named entity whenever that entity does not have an exact match in our domain sources, and the resulting pages classified as possible homepages or not. To simplify the classification process, we initially filter the Google results by removing all *DBLP* URLs (indicated by "`DBLP:`" in the title), `portal.acm.org` URLs, `Citeseer` URLs, and any URL with known non-text/html extensions, as we know none of these can be a person's homepage, yet they often show up in queries on researchers' names.

At this stage we would also like to minimize the number of false negatives, as false positives may be caught when calculating their similarity scores below, whereas false negatives would mean discarding the correct URL, guaranteeing no valid link despite the existence of one.

4.3 Similarity Score

If the home page classifier returns exactly one page, we take that as the answer. If it returns none, we link the entity to the Google search page. If it returns more than one, we calculate a score for every positively classified page and choose the top k.

The score is calculated by simply counting the occurrences of database words in the webpage. We hand crafted a specific list that includes words such as `database`, `mining`. A better approach to this scoring function would be a cosine similarity measure between the webpage and the message. Both the webpage and the message would be put into a vector space model with proper TFIDF heuristic weights.

4.4 Feedback to NE Extractor

In finding the websites of entities, we can further refine the named entities. This opportunity comes from an imperfection in the NE extractor module: because the candidate phrases come from overlapping sliding windows, it is possible for multiple entities to overlap. For example, suppose we have the following substring in a document.

... `John Doe Jane Smith` ...

Now suppose the extractor classified the following 3 phrases as people: "`John Doe`", "`Doe Jane`", "`Jane Smith`". In other words, the entities overlap and it is not clear how many or how to partition them. It would be incorrect to simply merge all of them since "`John Doe Jane Smith`" is not a real person.

Fortunately, the similarity score of websites can help us determine the correct partitioning. In this example, we perform three searches for the three phrases and compare the scores. The idea is that "`Doe Jane`" is unlikely to be a researcher and the corresponding website scores would be low. As a result, "`John Doe`" and "`Jane Smith`" would be discovered as the true researchers.

5 Performance Evaluation

We tested the classification performance of DBDebbie using *DBWorld* messages. At the heart of the NE extractors are the classifiers. With the D2K framework, we were able to use several existing classifiers with ease. We show recall, precision, and accuracy results of Naïve Bayes, C4.5 Decision Tree, and Support Vector Machine (SVM) for various entities. We used the libsvm package (C-SVC) with a linear kernel. The numbers shown are averages of 5-fold cross validation.

5.1 NE Classifiers

In Figures 2, 3, and 4, we show classification performance on researchers, universities, and conferences. We hand labeled 388 name, 111 university, and 66 conference positive examples. Adding negative examples, we trained with 1890 examples for names, 413 examples for universities, and 268 examples for conferences. As the graphs show, performance with researchers was best. University and conferences lagged behind. This is most likely a direct result of the lack of training examples. With so few training examples, we are also likely to overfit very easily.

As for comparisons between the different classifiers, we see that SVM is usually the strongest out of the three. C4.5 and Naïve Bayes are still competitive. One thing to note, we did not spend long in tuning the parameters of the classifiers. This is especially pertinent for SVM since it has many parameters. If one had time to carefully tune, classification performance would definitely be higher.

5.2 Home Page Classifiers

In Figure 5, we show the performance of our home page classifier. We took 348 positive examples from *DBLife*, and 268 negative examples from the result set that Google returned given various researcher names as queries, along with a few other random URLs.

6 Implementation

Much like our philosophy in designing the modules, we tried to leverage as much existing technology as possible. The overall DBDebbie system was built in the NCSA D2K environment. Specifically, we used the T2K libraries which offered modules for file I/O and other text-related tasks. T2K also has modules

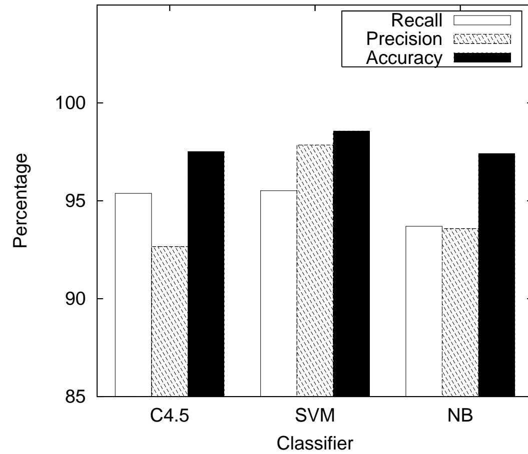

Figure 2: People NE Classifier Performance

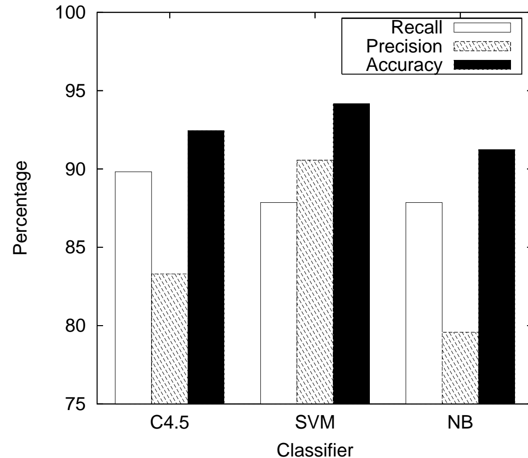

Figure 3: University NE Classifier Performance

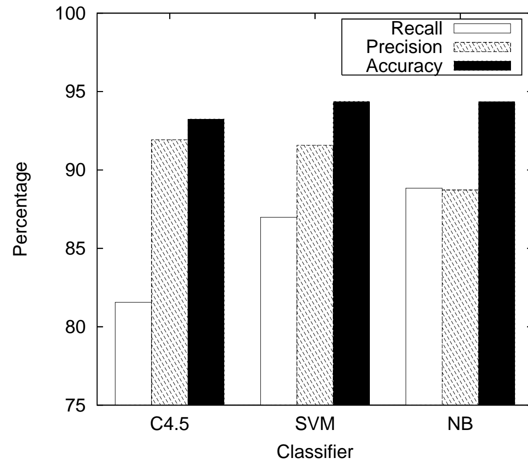

Figure 4: Conference NE Classifier Performance

which wrap around GATE [3]. As mentioned previ-

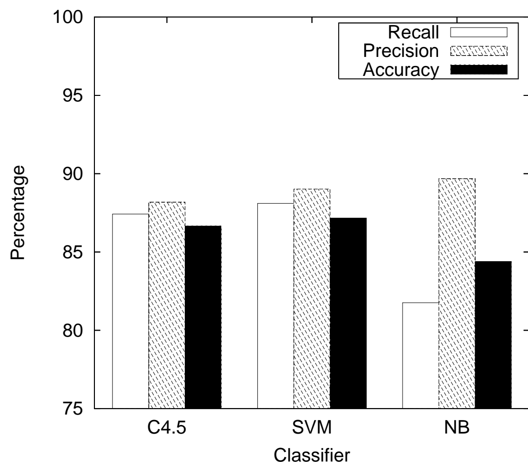

Figure 5: Home Page Classifier Performance

ously, GATE[2] is a complete natural language processing framework/architecture/system being built at the University of Sheffield, UK. It has been built and maintained for over eight years and offers modules for tokenization, part-of-speech tagging, primitive NE extraction, and more.

The overall framework of DBDebbie was coded in D2K using Java. Modules which needed to parse and read *DBLP* and *DBLife* were coded in either Python or Perl, which offered better text processing functionalities. We discuss them in some finer detail below.

6.1 DBLP Scoring Implementation

Initially, we planned to query *DBLP* on the fly as we classified each window. However, we soon realized that this would create a tremendous performance bottleneck in our system, and instead opted to download local copies of *DBLP*'s conference, journal, and author lists. In addition, to prevent the dictionary files from being loaded every time we processed a document, the dictionary lookup engine has been implemented as a multi-threaded server that runs in the background. This was also necessary because D2k processes documents in a pipeline, and our server can take requests for feature processing and URL lookups simultaneously.

The Conference and Journal dictionaries have been implemented as nested dictionaries somewhat similar to tries, except search order over the tokens does not matter (*i.e.*, search order can be any permutation of the entry). The top level of the conference dictionary contains all words included in any of the *DBLP* listed conferences. Entries of this top level dictionary contain all words of conference names which have that entry in common. This indirection continues for a specified depth. To prevent combinatorial explosion, we limit the depth of indirection to 2 or 3, but this already reduces the number of pairwise substring matches needed to score input token string to an order of 10. Without this data structure, the substring search would have been pairwise across all *DBLP* conference names, which would have been prohibitively expensive. The journal dictionary is implemented similarly.

The author list is split into 3200 different dictionary files to speed up load times. Individual dictionary files are loaded on the fly, as needed for lookup. When adding entries to the author dictionary, we first clean up special characters (*i.e.*, ü, á, etc) and commas, and add each entry with and without the middle initials. In addition, *DBLP* denotes famous authors with a (*) next to their name, giving us an indication that we can browse one level deeper to extract their homepage. All relevant URLs are stored with associated conferences, journals, and authors for linking stage.

6.2 DBLife Scoring Implementation

The scoring module for *DBLife* lookups returned values in the range [0,1]. To support fast lookups into this dataset, we parse its XML files in Perl and store the researcher and conference entities in hashes. Universities are loaded in from an expanded dictionary derived from *DBLife* entries, as mentioned in Section 3.2. Exact matches as described below receive a score of 1. Note that because a period is considered a separate token, we associate the period with the word immediately to its left. If we cannot exactly match a candidate phrase against these entries, we fall back on a simple word-level similarity score. Each phrase receives a score for each of the possible entity labelings.

Conferences have two possibilities for exact matches. They are stored as a full name consisting of an acronym plus year, as well as just the acronym. Therefore any lookup that fully matches either the name+year combination or just the acronym is considered an exact match, and receives a score of 1. There are 53 conference entries in this inflated dictionary, *i.e.* including entries corresponding to the same conference but at different years.

Universities may also have one or two exact match entries. Universities that end in the word *University* have an additional entry in which that word is dropped. Therefore an exact match for a university requires identically matching either string should it have two entries, or just the one entry. There are 268 university entries in this inflated dictionary.

Researchers similarly may have 2 entries that a lookup could match to and be considered an exact match. We store their full name, *i.e.* First, Middle, and Last names as a single entry to match, as well as just their First+Last names. Therefore people with a middle name recorded in *DBLife* have 2 possible entries to exactly match against. We also store indices on just First names, Middle names, and Last names, used

[2]http://gate.ac.uk/

in the word-level similarity scores when exact matches fail. There are 364 unique researchers currently being monitored in *DBLife*, and 483 names in the inflated dictionary.

If no exact match is possible for a given entity class, we examine each word in the phrase, discarding punctuation. We do a lookup of each word in each class dictionary if the phrase was not an exact match for that class. So, a phrase such as "`VLDB Conference`" may not exactly match an entry, but "`VLDB`" would, and thus count towards the total number of words matched. For People, we perform a lookup in all 3 indices on First, Middle, and Last names. If there is an exact match between the word and an entry in any of these, we count this towards the total number of words matched. The score for the phrase then is $\frac{\text{total \# words matched}}{\text{total \# words}}$. We note that such a simple scoring function may make it appear as if a name is a full match if, for example, it is comprised of a First name from one person, and a Last name from another.

6.3 Screenshots

Figures 6 – 9 show some screenshots of our final implementation in D2K. Figure 6 shows the annotation viewer for showing the extracted named entities. Figures 7 and 8 show the decision tree viewer available in D2K. They allowed us to tune the features more carefully. Figure 9 shows the implementation in D2K for DBDebbie. And finally, we show a labeled HTML document in Figure 10.

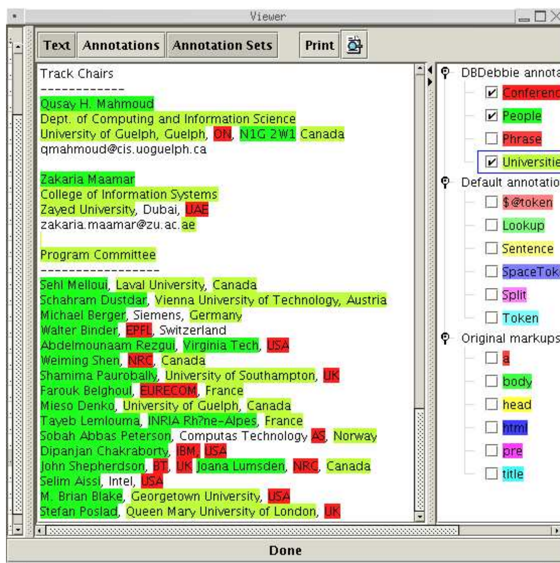

Figure 6: T2K Annotation GUI

7 Discussion

The value of a good dictionary for membership tests as features has been experimentally shown in many

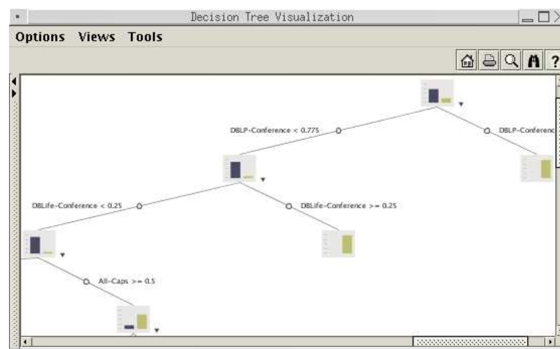

Figure 7: T2K Decision Tree Vis GUI for conferences

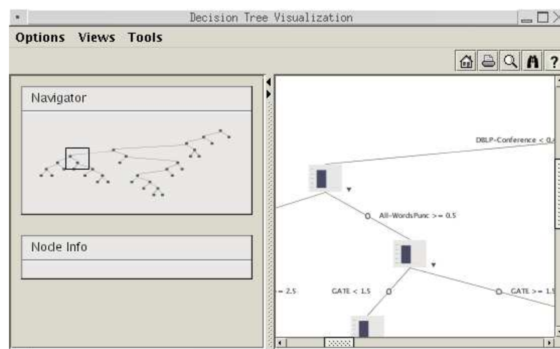

Figure 8: T2K Decision Tree Vis GUI for conferences

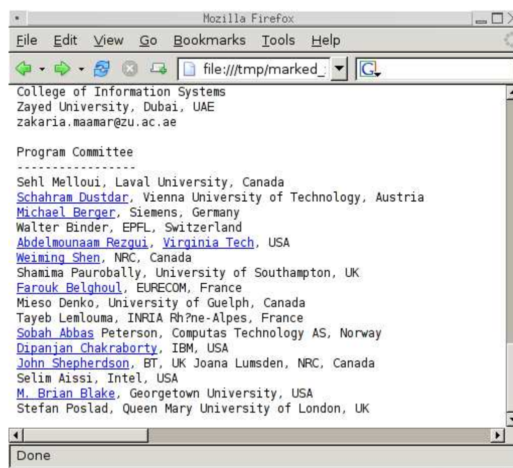

Figure 10: Exported HTML

named entity recognition (NER) tasks (*e.g.* [1]), and natural language tasks in general. The use of a good matching function is also important. Cohen, et al. [6] compare a number of string distance metrics for this purpose. Our own routines do not currently use any of these approaches due to limited time and complexity concerns regarding the number of candidate strings to compare against for every phrase in our sliding windows.

Assembling a dictionary can be tedious if one is

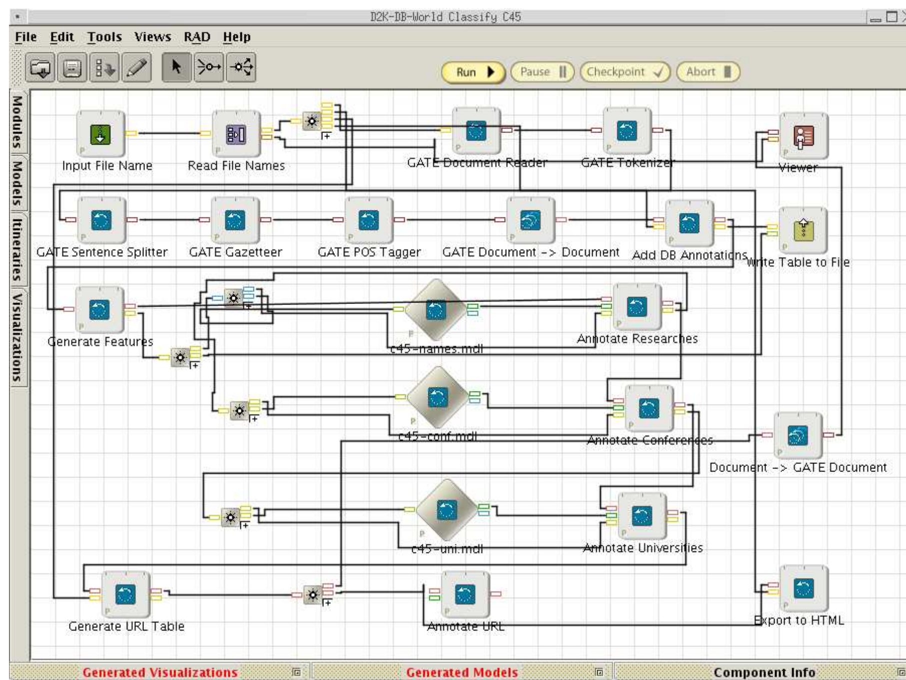

Figure 9: D2K GUI

not already present, and even one exists, it may not be comprehensive enough. McCallum and Li [5] have used GoogleSets to augment their dictionaries for named entity recognition (NER). They use it as an implementation of a method they call *WebListing*, in which seeds for lexicons are automatically extracted from labeled data along with their HTML formatting patterns. This information is coupled with a search engine to find other co-occurring terms with similar formatting to identify new items for their lexicon. We perform a similar step using GoogleSets during preprocessing to create a larger dictionary of universities, although it is straightforward to automate the expansion of our dictionary during the labeling task.

The previous paper was also the first application of Conditional Random Fields (CRFs) [4] to the NER task. CRFs are conditionally-trained undirected graphical models. They offer much flexibility in including arbitrary, overlapping, non-independent features into the model, unlike traditional generative models. The original work applied it to the part-of-speech tagging problem. There has since been much related research towards extending the basic model, automatic feature induction, and applications in other problem domains. Some of the features we use in our own classifiers are clearly not independent. CRFs may be a viable alternative to explore for our classifiers. It also may also provide a convenient way to associate a confidence score to our labelings, for possible use in our final output presented to the user [2].

8 Conclusion

In this project, we have attempted to construct a named entity recognizer and a entity web linker in the domain of *DBWorld* messages. In the process, we tried to use as much domain knowledge as possible, *e.g.*, *DBLP*, *DBLife*, Google, etc. Our preliminary results show that this approach could work well, especially with carefully tuned features and many more training examples.

References

[1] William W. Cohen and Sunita Sarawagi. Exploiting dictionaries in named entity extraction: Combining semi-markov extraction processes and data integration methods. In *KDD*, pages 89–98, 2004.

[2] Aron Culotta and Andrew McCallum. Confidence estimation for information extraction. In *Proceedings of Human Language Technology Conference and North American Chapter of the Association for Computational Linguistics (HLT-NAACL)*, 2004.

[3] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics*, 2002.

[4] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence

data. In *Proc. 18th International Conf. on Machine Learning*, pages 282–289, San Francisco, CA, 2001. Morgan Kaufmann.

[5] A. McCallum and W. Li. Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons. In *Proceedings of The Seventh Conference on Natural Language Learning (CoNLL-2003)*, 2003.

[6] Pradeep Ravikumar William W. Cohen and Stephen Fienberg. A comparison of string distance metrics for name-matching tasks. In *IIWeb*, pages 73–78, 2003.