A Logical Framework for Template Creation and Information Extraction

David Corney*, Emma Byrne†, Bernard Buxton* and David Jones*
*Department of Computer Science, University College London
Gower Street, London WC1E 6BT, UK
Email: D.Corney@cs.ucl.ac.uk *or* D.Jones@cs.ucl.ac.uk
†The Business School, University of Greenwich
Old Royal Naval College, Park Row, London, SE10 9LS, UK
Email: E.L.Byrne@greenwich.ac.uk

***Abstract*— Information extraction is the process of automatically identifying facts of interest from text, and so transforming free text into structured data. The field has a history of applications analysing text from news and financial sources and more recently from biological research papers. Although much of this work has been successful, it has tended to be ad hoc. We propose a more formal basis from which to discuss information extraction. This will aid the identification of important issues within the field, allowing us to identify the questions to ask as well as to formulate some answers.**

It has long been recognised that there is a need to share resources between research groups in order to allow comparison of their different systems and to motivate and direct further research. We strongly feel that there is also a need to provide a *theoretical* framework within which text mining and information extraction systems can be described, compared and developed. Our framework will allow researchers to compare their methods as well as their results. We also believe that the framework will help to reveal new insights into information extraction and text mining practices.

One problem in many information extraction applications is the creation of templates, which are patterns used to identify information of interest. Within our framework, we describe formally what a template is, with definitions of words and documents, and other typical information extraction and text mining tasks, such as stemming and part of speech tagging, as well as information extraction itself. This formal approach enables us to show how common search algorithms could be used to create and optimise templates automatically, by means of sequences of overlapping templates, and we develop heuristics that make this search feasible.

I. Introduction

Information extraction (IE) [1] applications include analysing text from news and financial sources [2] and biological research papers [3], [4], [5]. Competitions such as MUC and TREC have been promoted as using real text sources to highlight problems in the real world. We strongly feel that there is also a need to provide a *theoretical* framework within which these information extraction systems can be described, compared and developed, by identifying key issues explicitly. The framework we present here will allow researchers to compare their methods as well as their results, and also provides new methods for template creation.

Information extraction is a diverse research area, but one common feature is the use of templates. A template is a pattern designed to identify "interesting" information to be extracted from documents, where "interesting" is relative to the user's intentions. An ideal template can be used to extract a large proportion of the interesting information available with only a little uninteresting information. Different types of templates exist, but in general, they can be thought of as regular expressions over words and the features of those words. Any sentence can be matched with a large number of templates, and many templates match a large number of sentences. This makes template creation a challenging problem.

Although it covers several key areas, this paper focuses on template creation. Currently, templates are typically designed by hand, which can be laborious and limits the rapid application of IE to new domains. There have been several attempts at automatic template creation [6], [7], and there are likely to be more in the future. To the best of our knowledge, no such system has demonstrated widespread applicability, but tend to be successful only within narrow domains. Some systems are effective, but require extensive annotation of a training set [8], which is also laborious.

One alternative to using templates is *co-occurrence analysis* [9]. This identifies pieces of text (typically sentences, abstracts or entire documents) that mention two entities, and assumes that this implies that the two entities are in some way related. Within our framework, this can be seen as a special case of a template, albeit a very simple one, as we show in Section II-C.

We propose treating template creation as a search problem of a kind familiar to the artificial intelligence community [10, ch. 3–4]. We define the space of candidate solutions (i.e. templates); a means of evaluating and comparing these candidate solutions; a means of generating new candidate solutions; and an algorithm for guiding the search (including starting and stopping). We expand these ideas in Section VI-B, where we show how to "grow" useful templates from given seed phrases.

However, in order to present our approach to template creation, we need to establish a suitable theoretical framework. Thus in Section II, we define the required concepts formally, moving from words and documents to templates and information extraction. In Section III, we describe how templates can

be ordered according to how specific or general they are, as a precursor to template creation and optimisation. In Section IV, we discuss how to modify a template to make it more general. In Section V, we give formal definitions of recall and precision within our framework. In Section VI, we discuss how they might be estimated in practice, and then discuss heuristic search algorithms and their feasibility, before a concluding discussion.

The framework itself is described in Sections II–V, with the remaining sections discussing possible implementations and extensions. A longer form of this paper, with more detailed examples and discussion is available as a technical report [11].

II. Basic Definitions

In this section, we define several terms culminating in a formal definition of information extraction templates.

Definition 1: A *literal* λ is a word in the form of an ordered list of characters. We assume implicitly a fixed alphabet of characters.
Examples: "cat", "jumped", "2,5-dihydroxybenzoic".

Definition 2: A *document* d is a tuple (ordered list) of literals: $d =< \lambda_1, \lambda_2, \ldots, \lambda_{|d|} >$.
Examples: $d_1 =<$the, cat, sat, on, the, mat$>$, $d_2 =<$a, mouse, ran, up, the, clock$>$.

Definition 3: A *corpus* D is a set of documents: $D = \{d_1, d_2, \ldots, d_{|D|}\}$.
Example: $D_1 = \{d_1, d_2\}$.

Definition 4: A *lexicon* Λ is the set of all literals found in all documents in a corpus: $\Lambda_D = \{\lambda | \lambda \in d \text{ and } d \in D\}$.
Example: $\Lambda_{D_1} = \{$the, cat, sat, on, mat, a, mouse, ran, up, clock$\}$.

Every word has a set of attributes, such as its part-of-speech or its membership of a semantic class, some of which we now discuss. Although particular attributes are not a formal part of the framework, they are used in various illustrative examples throughout this paper.

It is common practice in information retrieval to index words according to their stem to improve the performance [12]. Similarly in information extraction, it is often helpful to identify words that share a common stem. Words may also belong to pre-defined semantic categories, such as gazetteers listing businesses, countries or proteins. In this framework, we are not concerned with the nature of such categories, but assume only that there exists some method for assigning such attributes to individual words. The role of each word in a sentence is defined by its *part of speech*, or lexical category, such as "singular common noun", "plural common noun" or "past tense verb". An implementation may limit this to exactly one label per word, based on the context of that word.

In regular expressions, a wildcard can "stand in" for a range of characters, and we use the same notion here to represent ranges of words. For example, we use the symbol '*' as the universal wildcard which can be replaced by any word in the lexicon, so every word has the attribute '*'. We also use the symbol '?' to represent any word *or no word at all*. We discuss these wildcards further in Section IV-B.

Other categories may be introduced to capture other attributes, such as orthography (e.g. upper case, lower case or mixed case), word length, language and so on. We could also treat punctuation symbols as literals if required, or as a separate category. However, the categories described above are sufficient to allow us to develop and demonstrate the framework.

Definition 5: A category κ is set of attributes of words of the same type. Common categories include "parts of speech" and "stems".
For convenience, we will label certain categories in these and subsequent examples. The choice of categories is not part of the framework but reflects categories likely to be used in a practical implementation. In particular, we use Λ to label the category "literals"; Π for "parts of speech" (lexical categories); Γ for "gazetteers" (semantic categories); Σ for "stems"; and Ω for "wildcards".

Example:
$\kappa_\Lambda = \{$the, cat, sat, on, mat, mouse, $\ldots\}$
$\kappa_\Sigma = \{$the_stem, cat_stem, sit_stem, on_stem, mat_stem$\ldots\}$
$\kappa_\Pi = \{$DT, NN, VBD, IN, $\ldots\}$
$\kappa_\Gamma = \{$FELINE, RODENT, ANIMAL, $\ldots\}$
$\kappa_\Omega = \{*, ?, \ldots\}$

We use the suffix '_stem' in stem labels to avoid confusing them with the corresponding literal. The part-of-speech labels follow the Penn Treebank tags [13]. Thus we use the symbol "DT" to represent determiners such as "the", "a" and "this"; "VB" to represent verbs in their base form, such as "sit" and "walk"; "VBD" to represent past-tense verbs, such as "sat" and "walked"; "NN" to represent common singular nouns, such as "cat" and "shed" and so on.

Definition 6: Let K be a set of categories of attributes. Each element κ of K is a single category of word attributes.
Example: $K_1 = \{\kappa_\Lambda, \kappa_\Sigma, \kappa_\Pi, \kappa_\Gamma, \kappa_\Omega\}$.

Definition 7: A term t is a value that an attribute may take, i.e. an element of a category of word attributes.
Examples:

$t_1 =$ cat, $t_2 =$ NN, $t_3 =$ FELINE, where $t_1 \in \kappa_\Lambda$, $t_2 \in \kappa_\Pi$, $t_3 \in \kappa_\Gamma$.

Definition 8: We define a *template element* T to be a set of terms belonging to a single category. Let $T = \{t_1, t_2, \ldots, t_n\}$, such that $t_i \in T$. Then $t_i \in \kappa \iff t_j \in \kappa,\ \forall t_j \in T$.
Examples:

$T_1 = \{$NN, VBD$\}$
$T_2 = \{$FELINE, RODENT, FLOOR_COVERING$\}$

The set $\{$NN, FELINE $\}$ is not a template element because "NN" and "FELINE" belong to different categories, namely κ_Π and κ_Γ respectively.

Definition 9: A *template* τ is a tuple of one or more template elements, $< T_1, T_2, \ldots, T_n >$, where $T_1 = \{t_{1,1}, t_{1,2}, \ldots,\}, T_2 = \{t_{2,1}, t_{2,2}, \ldots\}$ and so on. $|\tau|$ is the number of template elements in template τ, and is always greater than zero. Each template element T_i within a template consists of one or more terms of the same type.
Example:

$\tau_1 =< \{$the$\}, \{$FELINE, RODENT$\}, \{$VB, VBN$\} >$.

Definition 10: The *attributes* of a literal are the set of template elements defining the values of the literal in each category. We first define the set of attributes of a literal λ for a particular category κ as $\alpha(\lambda, \kappa) = \{T | \forall t \in T,\ t \in \kappa$ and λ has attribute $t\}$. The set of all attributes of a literal is the union of the attributes in each category: $\alpha(\lambda) = \bigcup_{\kappa \in K} \alpha(\lambda, \kappa)$. If a literal has no value for a particular category, then the category is omitted from the set α.
When we say "λ has attribute t", we assume that this relationship is defined outside of the framework. For example, there may be functions to assign a stem attribute to a word, or to assign a particular semantic category to any of a given list of words.

Example: α(cat) = {{cat}, {NN}, {FELINE, ANIMAL}, {cat_stem}}

In the case of α(the), the word "the" has a part-of-speech tag "DT" (determiner) and the obvious literal, but no gazetteer or stem entries. So $\alpha(\text{the}, \kappa_\Pi) = \{\text{DT}\}$, and $\alpha(\text{the}, \kappa_\Gamma)$ is undefined, and so $\alpha(\text{the}) = \{\{\text{the}\}, \{\text{DT}\}\}$.

The set of literal attributes Λ has the special property that every word has exactly one literal. Other categories in K may contain terms such that one literal may correspond to one or more terms, or to no term at all. For example, one literal may belong to more than one gazetteer, while another literal may belong to none. Therefore for any λ, $|\alpha(\lambda, \kappa_\Lambda)| = 1$. As a consequence, for any λ, $|\alpha(\lambda)| \geq 1$.

A. Membership of Terms

We now define the concept of membership to refer to the set of literals that share a particular attribute.

Definition 11: We define the members μ of a term t as being the set of all literals that share the attribute value defined by that term. I.e. $\mu(t) = \{\lambda | t \in \alpha(\lambda)\}$. Also, we define the members of a set of terms (such as a template element) as the union of the members of each term in the set: $\mu(\{t_1, t_2, \ldots, t_n\}) = \bigcup_{i=1}^{n} \mu(t_i)$
Examples:

μ(sit_stem) = {sit, sits, sat, sitting }.

μ(FELINE) = {cat, lion, tiger, ...}.

μ(RODENT) = {mouse, rat, hamster, ...}.

$\mu(\{\text{FELINE}, \text{RODENT}\})$ = {cat, lion, tiger, mouse, rat, hamster, ...}.

B. Information Extraction by Matching Document Fragments

Definition 12: We define a *fragment* of a document as being a tuple of successive literals taken from some document d. The function $f(d, a, b)$ returns a tuple of b words in order, from d, starting with the a^{th} word. Note that $|f| = b$. We say that $f \in d$.
Example: If d_1 =<the, cat, sat, on, the, mat >, then $f(d_1, 4, 3)$ =<on, the, mat>.

Definition 13: A template *matches* a fragment of a document d if each successive template element in the template contains a term whose membership includes each successive word in the fragment. Given a fragment $f =< f_1, \ldots, f_n >$ and a template $\tau =< T_1, T_2, \ldots, T_n >$, we extend the membership function thus:

$$\mu(\tau, d) = \{f | f \in d \text{ and } \forall i \in 1 \ldots |\tau|, f_i \in \mu(T_i)\}$$

For a corpus D, the template matches the union of the template membership for each document: $\mu(\tau, D) = \bigcup_{d \in D} \mu(\tau, d)$.
This function returns a set of fragments, each of which consists of a tuple of literals that matches each successive element of the template τ, and each of which is found in a document in the corpus. Matching terms in this way is the core of information extraction. The words that are matched define the information that is to be extracted.
Example: Given a corpus D_1 (as defined after Definition 3) and a template $\tau_1 =<$ {DT}, {ANIMAL}, {VBD} >, then $\mu(\tau_1, D_1) =$ {<the, cat, sat>, <a, mouse, ran>}.

C. Co-occurrence Analysis

Co-occurrence analysis assumes that two entities in the same piece of text are related, without attempting a more sophisticated linguistic analysis of the text. In our framework, this can be represented by a template (or set of templates) that defines the two entities with a series of wildcards between them.

For example, suppose we use co-occurrence analysis to discover every sentence in a corpus that mentions two entities, as matched by template elements T_i and T_j. Let us assume that all sentences to be considered are finite with a maximum length of Q words. Then we could define two template $\tau_1 =< T_1, \ldots, T_i, \ldots, T_j, \ldots, T_Q >$ and $\tau_2 =< T_1, \ldots, T_j, \ldots, T_i, \ldots, T_Q >$. Two templates are required if we wish to allow for sentences with the two entities in different orders. We replace every template element except for T_i and T_j with the wildcard element '?' so as to match any sentence up to length Q that contains words that match our terms. With three or more entities, larger sets of templates may be required.

III. Term and Template Ordering

One motivation for creating this framework is to enable the use of common search algorithms for template creation. To do this effectively, we must define an *ordering* over the templates, which we can then use to develop practical search heuristics. A template that matches every possible fragment in a document is useless, as is one that matches no fragments at all. Somewhere between these two extremes lie useful templates that match the interesting fragments only, so the aim of template creation is to find a suitable trade-off between the generic and the specific. We therefore suggest that a useful ordering is one based on the number of fragments that a template is likely to match. We can use such an order to search across a range of templates and explore the trade-off. For unseen text, it is impossible to predict the amount of information to be extracted in advance, so instead, we develop a heuristic ordering that approximates it.

A. Superset Ordering of Terms and Template Elements

We define a specificity ordering over terms such that each term matches every word that its antecedent matches, along with zero or more extra words. We call this superset ordering.

Definition 14: Let $\geq_s$ be the ordering over superset specificity. Let t_1, t_2 be terms. If $\mu(t_1) \subseteq \mu(t_2)$ then $t_1 \geq_s t_2$, and we say that t_1 is at least as specific as t_2. If $\mu(t_1) \subset \mu(t_2)$ then $t_1 >_s t_2$, and we say that t_1 is more specific than t_2.
Example: Let FELINE and ANIMAL be two terms (specifically, gazetteers), such that μ(FELINE) = {cat, lion, tiger , ...} and μ(ANIMAL) = {antelope, cat, lion, tiger,..., zebra}. Then μ(FELINE) $\subset \mu$(ANIMAL) and so FELINE $>_s$ ANIMAL.

Some specificity orderings are dependent on the categories of the two terms. For example, if $t_1 \in \kappa_\Lambda$, then $\forall t_2 \in K,\ t_1 \geq_s t_2$. In other words, terms that represent literals are at least as specific as any other terms. Also, the wildcards '*' and '?' match every word, so we can say that wildcard terms are no more specific than any other term. I.e. if $t_1 \in \kappa_\Omega$, then $\forall\, t_2 \in K,\ t_1 \leq_s t_2$.

Definition 15: Let T_1 and T_2 be two template elements. We say that T_1 is at least as specific as T_2 if and only if every literal matched by any term in T_1 is also matched by some term in T_2. I.e. $T_1 \geq_s T_2 \iff \forall\lambda \in \mu(T_1),\ \lambda \in \mu(T_2)$. Similarly, $T_1 >_s T_2 \iff \forall\lambda \in \mu(T_1),\ \lambda \in \mu(T_2)$ and $\exists\lambda \in \mu(T_2)$ such that $\lambda \notin \mu(T_1)$.
Thus T_1 is more specific than T_2 if every literal matched by a term in T_1 is also matched by a term in T_2, and at least one literal is matched by a term in T_2 and not by a term in T_1.

B. Ordering of Templates

We have discussed ordering of terms and of template elements. Here we generalise this to discuss entire templates. We want to be able to compare two templates, τ_1 and τ_2, and say which is more specific, i.e. which one matches fewer fragments. If τ_1 and τ_2 are related through superset ordering, then for a given set of documents D, this is testing if $\mu(\tau_1, D) \supset \mu(\tau_2, D)$ and therefore whether $|\mu(\tau_1, D)| > |\mu(\tau_2, D)|$. This depends on the corpus D and is potentially slow to evaluate, especially if D contains a large number of documents. Instead, we present an estimate of the relative specificity which is independent of D, and which will be useful when developing search heuristics.

If τ_1 and τ_2 contain the same number of sets of terms, then

$$\tau_1 \geq_s \tau_2 \iff T_{1,i} \geq_s T_{2,i} \quad i = 1 \ldots |\tau_1|,$$

and

$$\tau_1 >_s \tau_2 \iff T_{1,i} \geq_s T_{2,i} \quad i = 1 \ldots |\tau_1| \text{ and } T_{1,j} >_s T_{2,j}$$

for some j. We use $T_{n,i}$ to refer to the i^{th} element of template τ_n.

Also, if two templates are identical except that one is "missing" the first or last template element of the other, then the shorter of the two is less specific. I.e. if $\tau_1 =< T_1, T_2, \ldots, T_{n-1}, T_n >$, $\tau_2 =< T_1, T_2, \ldots T_{n-1} >$ and $\tau_3 =< T_2, \ldots, T_{n-1}, T_n >$ then $\tau_1 \geq_s \tau 2$ and $\tau_1 \geq_s \tau 3$.

Although these relationships do not provide a complete ordering over all templates, they do allow us to create and modify templates, and to develop useful search heuristics. We use this ordering to develop functions that create and modify templates in Section IV, and to develop methods to search efficiently for *good* templates in Section VI.

IV. Template Generalisation

Whether created manually or automatically, templates are usually based on examples of "interesting" phrases, typically identified by hand. These phrases are then used as "seeds" to help define more general templates, and in this work we assume that we are given some suitable phrases. We show how a wide range of templates can be created from each seed phrase. We first discuss how terms can be created and generalised, and then expand this to template creation and generalisation.

A. Creating and Modifying Template Elements

We now bring together several concepts discussed above, and define functions that create and generalise template elements.

Definition 16: Given a literal, we want to create a new template element, which is simply a set containing the literal. We define the trivial function initialise for this purpose: $\mathsf{initialise}(\lambda) = \{\lambda\}$.

Have created a template element, we can then modify it. We now define a function that modifies any given template element to produce a new set of template elements that is at least as general as the element given. This is based on the notion of superset ordering (Section III-A) in that the new template elements match a superset of the literals matched by the original template element. Furthermore, the new set of elements belongs to a specified category which is different from the category of the source element.

Definition 17: We define a function to create a more general set of template elements from a given template element, such that all of the terms in each new template element are members of a specified category. Given a template element $T = \{t_1, t_2, \ldots, t_{|T|}\}$ of category κ and given a target category $\kappa' \neq \kappa$, we create a set of template elements $\{T'_1, T'_2, \ldots, T'_n\}$ thus:

$$\begin{aligned}\mathsf{generalise}(T, \kappa') = \\ \{T' | T' = \{t'_1, t'_2, \ldots, t'_m\}, \text{ and } t'_1, t'_2, \ldots, t'_m \in \kappa', \text{ and} \\ \forall t'_i \in T',\ |\mu(t'_i) \cap \bigcup_{t \in T} \mu(t)| \geq 1 \text{ , and} \\ \bigcup_{t \in T} \mu(t) \subseteq \bigcup_{t' \in T'} \mu(t'), \text{ and there is no set } \{t'_p \ldots t'_q\} \\ \text{such that } \{t'_p \ldots t'_q\} \subset T' \text{ and } \bigcup_{t \in T} \mu(t) \subseteq \bigcup_{j=p}^{q} \mu(t'_j)\}.\end{aligned}$$

I.e. For each template element produced by $\mathsf{generalise}(T, \kappa')$, each term within that element belongs to category κ'; and each term within that element matches at least one literal matched by a term in T; and every literal matched by a term

or terms in T is matched by at least one term in the template element; and that no subset of terms exists that meets these two requirements. Note that $\bigcup_{t \in T} \mu(t)$ is the set of all literals that are members of terms in the original template element T, and that $\bigcup_{t' \in T'} \mu(t')$ is the set of all literals that are members of terms in the new template elements T'. Each new template element has to be different from the original template element, as they belong to different categories. This effectively ensures that the new element is *more* general than the original, and not just *as* general.

Examples:

generalise($\{\text{cat}\}, \kappa_\Gamma$) = { {FELINE}, {ANIMAL} }

generalise({cat, dog}, κ_Γ) = { {FELINE, CANINE}, {ANIMAL} }

generalise({ {FELINE}, {ANIMAL} }, κ_Π) = { {NN, NNS } }

This last example holds if every member of the two gazetteers "ANIMAL" and "FELINE" are singular common nouns (NN) or plural common nouns (NNS).

Note also that generalise will return an empty set if no generalisation exists that matches all the required literals. Thus if the input set contains a literal that is not contained in any member of κ_x, then generalise$(T, \kappa_x) = \emptyset$.

B. Creating and Modifying Entire Templates

In the previous section, we defined the creation and generalisation of template elements. Templates are ordered lists of template elements (Definition 9), and we now apply the above concepts to create and modify templates. Given a seed phrase, in the form of a tuple of literals (i.e. a fragment), we can easily define a very specialised template that matches only that fragment. We can then modify this to increase its generality.

Definition 18: We extend the initialise function to create a specialised template from a fragment.

$$\text{initialise}(< \lambda_1, \lambda_2, \ldots, \lambda_n >) =< \{\lambda_1\}, \{\lambda_2\}, \ldots, \{\lambda_n\} > .$$

We define a new function that generalises any given template to create a new set of templates by modifying a single element of the template using the element generalisation function defined in Section IV-A. One template will be created for each possible generalisation of the specified template element.

Definition 19: Given the template $\tau =< T_1, T_2, \ldots, T_i, \ldots, T_n >$, then

$$\text{generalise}(\tau, \kappa, i) = \{\tau' | T_i' \in \text{generalise}(T_i, \kappa) \text{ and } \tau' =< T_1, T_2, \ldots, T_i', \ldots, T_n >\}$$

I.e. we replace the i^{th} template element with the result of its own generalisation.

Example: Let template τ_1 =<the, cat, sat>. Then generalise$(\tau_1, \kappa_\Gamma, 2)$ = {<the, FELINE, sat>, <the, ANIMAL, sat>}. In this case, generalise$(\tau_1, \kappa_\Gamma, 2)$ returns two templates because the second literal "cat" belongs to two gazetteers. In contrast, generalise$(\tau_1, \kappa_\Gamma, 1) = \emptyset$, because the first literal "the" does not belong to any gazetteer in the category κ_Γ.

So far, we have assumed that all templates are generated from a seed phrase by replacing literals in that phrase with other attributes. This restricts the templates created to be the same length as the seed phrase, and so will only match fragments of this fixed length. We can create templates which match longer (or shorter) fragments by introducing wildcards such as a "?" which matches any word *or no word at all*. The details are discussed in [11].

V. Measuring Template Quality

In this section, we define recall, precision and related terms within our framework. These measures are needed to guide the automatic search for templates discussed in the next section.

Definition 20: We define $I(D)$ as the set of interesting, relevant fragments contained in corpus D.

The ideal template would match these, and only these, fragments. This set is generally not known, as we discuss below, but it does allow us to define concepts such as "true positive" and the precision score. We now define a series of sets and measures with respect to a template τ and a corpus D.

Definition 21: True-positives TP$(\tau, D) = \mu(\tau, D) \cap I(D)$.

Definition 22: False-positives FP$(\tau, D) = \mu(\tau, D) \setminus I(D)$.

Definition 23:

$$\text{Recall r}(\tau, D) = \frac{|\mu(\tau, D) \cap I(D)|}{|I(D)|} = \frac{|TP|}{|I(D)|}.$$

Definition 24:

$$\text{Precision p}(\tau, D) = \frac{|\mu(\tau, D) \cap I(D)|}{|\mu(\tau, D)|} = \frac{|TP|}{|\mu(\tau, D)|}.$$

An ideal template would have $|TP(\tau, D)| = |I(D)|$ and $|FP(\tau, D)| = 0$. We therefore wish to maximise $|TP(\tau, D)|$ while minimising $|FP(\tau, D)|$, but there is typically a trade off between the two. This is an example of multi-objective optimisation. If we knew the relative value of true positives and the cost of false positives, then we could combine these into a single objective function, such as maximising $|TP(\tau, D)| - k \cdot |FP(\tau, D)|$. In practice, such a weighting is not usually available, but a number of evolutionary approaches have been successfully applied to similar problems [14], as we discuss further in Section VI-E.

VI. Searching for Good Templates

In VI-B we will discuss the development of search algorithms, but first we need a practical way to estimate true-positive and false-positive scores.

A. Estimating True- and False-positive Scores

As noted earlier, we do not know which fragments are interesting *a priori*, and so definitions 21–24 cannot be used directly in calculating the recall or precision of a template. Instead, we need something that we can measure in practice, and which should approximate the "ideal" values above.

Suppose that we had a set of documents such that every fragment was labelled as either "interesting" or "not interesting". Then we could use standard supervised learning algorithms to construct useful templates and directly measure

the number of true positives, false positives and so on, to find an optimal template. This could then be used to find further information in the same field. However, while a small number of such labelled corpora do exist (e.g. [15]), they are for a few very precisely defined application areas, and so of little general use, as they cannot be used to aid IE in other application areas. Annotating documents in this way is very time consuming for a domain expert, and one aim of information extraction is to reduce the time and effort required to find relevant information.

Suppose instead that we have a set of "positive" documents, each of which is believed to contain some interesting information, and also a set of "neutral" documents, each of which may or may not contain relevant information. I.e. we have no prior knowledge about relevant information in neutral documents. Such sets are easier to define, for example by using information retrieval methods. We can then compare the proportion of information retrieved from neutral and from positive documents to evaluate a template. We assume that a "good" template will retrieve more information from positive documents than from neutral documents, even if we don't know in advance which pieces of information are useful, or how much useful information exists in any particular document.

We divide a corpus D into two sets, namely the set of positive documents, D_+, and the set of neutral documents, D_N, such that $D = D_+ \cup D_N$ and $D_+ \cap D_N = \emptyset$. We now use these sets of documents to define estimates of the numbers of true-positive fragments and false-positive fragments matched by a template τ.

Definition 25: We define an estimated true-positive set $\widehat{TP}(\tau, D) = \mu(\tau, D_+)$,

Definition 26: We define an estimated false-positive set $\widehat{FP}(\tau, D) = \mu(\tau, D_N)$.

Although these estimates are too crude to allow estimation of precision and recall scores, they are sufficient to guide the search for useful templates.

Let us consider some other properties of templates generated using superset generalisation. Suppose we have two templates, τ_1 and τ_2. If $\tau_1 >_s \tau_2$, then $\mu(\tau_1, D) \subset \mu(\tau_2, D)$. This relative specificity relation holds for any set of documents, so if one template matches fewer fragments than another in one corpus, then it will in any other corpus as well. Thus given two corpora D_a and D_b:

$$|\mu(\tau_1, D_a)| < |\mu(\tau_2, D_a)| \implies |\mu(\tau_1, D_b)| \leq |\mu(\tau_2, D_b)|.$$

We defined terms such as "true positive" and "false positive" above. Now we can say that if $\tau_1 >_s \tau_2$, then the number of true positives returned by τ_1 is no more than the number returned by τ_2, and equivalently for other scores:

$$|\text{TP}(\tau_1, D)| \leq |\text{TP}(\tau_2, D)|$$

and

$$|\text{FP}(\tau_1, D)| \leq |\text{FP}(\tau_2, D)|.$$

The equivalent inequalities also hold for the estimates defined above. I.e. if $\tau_1 >_s \tau_2$, then

$$|\widehat{\text{TP}}(\tau_1, D)| \leq |\widehat{\text{TP}}(\tau_2, D)|$$

and

$$|\widehat{\text{FP}}(\tau_1, D)| \leq |\widehat{\text{FP}}(\tau_2, D)|.$$

As these relationships hold for any set of documents D, we can predict some properties of templates without fully evaluating them. This property will be useful in developing heuristic search methods, as we discuss in Section VI-C.

B. Search Algorithms

As stated in the introduction, heuristic searching requires definitions of candidate solutions; a means to generate and evaluate candidate solutions; and a suitable search algorithm. We have now defined the candidate solutions (i.e. templates, Definition 9) and means to generate (Section IV) and evaluate them (using estimates of true positive and false positive scores given in Definitions 25–26). We now turn to the search algorithms themselves.

In all cases, we assume that we are given a seed fragment f. The root node of the search corresponds to a template consisting of a tuple of template elements, each containing a single literal: $\tau_{root} = \mathsf{initialise}(f)$ (Definition 18). From this, we can modify each element in the template by a single application of the $\mathsf{generalise}(\tau, \kappa, i)$ function (Definition 19). We can estimate the number of true positives and false positives of each of these new templates, and then decide which template to explore and modify next. The exact number of templates produced at each stage depends on the words themselves, because each $\mathsf{generalise}(\tau, \kappa, i)$ function will return 0, 1 or more templates (see Definition 17 and the accompanying discussion). We must also decide when to terminate the search, as we do not know *a priori* the quality of the best possible template, as we are assuming that we do not have a fully labelled corpus.

A simple exhaustive search method would be to start with a literal template created from the seed phrase using the initialise function. For each element in this template, we then apply the generalisation function, using each category in turn, so that each application generates a new template. If every literal has exactly $|\alpha|$ attributes, then a seed phrase of $|f|$ literals has $|\alpha|^{|f|}$ possible templates. Thus a 20-word seed phrase consisting of literals having 5 attributes will be matched by $5^{20} \approx 10^{14}$ templates. In practice, some words will have fewer attributes (e.g. words that don't appear in any gazetteers) and some will have more. We therefore need to introduce heuristics to make the search feasible, unless the seed fragment is very short.

C. Feeding Knowledge Forward

After estimating the numbers of true positives and false positives for any template, we can place a lower bound on those values for all templates that are derived using the generalise function. This is because we know that the derived

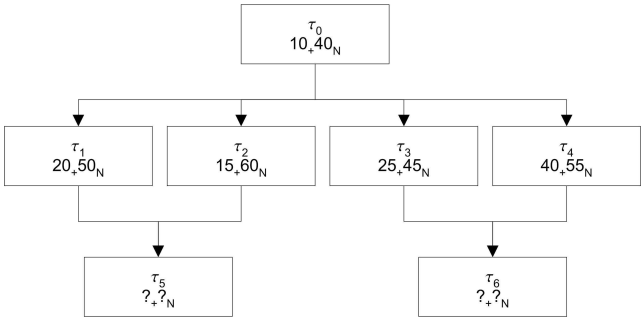


Fig. 1. Part of a search tree. Each node is a template with the number of true positives (x_+) and false positives (y_N) shown, with a "?" for unknown values. Each child node can be created by superset generalisation from any of its parents.

templates will match a superset of the fragments matched by the ancestor template (Section VI-A).

For example, suppose we have a template τ_1 and we evaluate this and find it matches 20 positive fragments and 50 neutral fragments, i.e. $|\mu(\tau_1, D_+)| = 20$ and $|\mu(\tau_1, D_N)| = 50$. If we then modify τ_1 using superset generalisation to create τ_2, then we know that $\mu(\tau_1, D) \subseteq \mu(\tau_2, D)$ for any corpus D. We therefore know that τ_2 matches *at least* 20 positive fragments and *at least* 50 neutral fragments, from D_+ and D_N respectively. Furthermore, any other templates derived using superset generalisation from τ_1 or from τ_2 will also match at least those numbers.

Therefore, as we carry out a search, we can calculate lower bounds on the estimates of true positives and false positives for a wide range of templates without the computational expense of evaluating each template. Instead, we can just choose the best template from the range available, if we use a best-first search. By "best" here, we might mean the template with the (estimated) most true positives, which will tend to produce templates with a high recall, though possibly with a low precision. If instead we choose the template that matches the least false-positive fragments then we will tend to produce templates with a high precision, though possibly with a low recall. Which of these options is more appropriate depends on the task at hand.

Every template has at least one parent, because they are created using superset generalisation; but most templates can be created in more than one way, from several parent templates. Therefore, many templates will have more than one parent. Consider the partial search graph shown in Figure 1. Here, templates $\tau_0 - \tau_4$ have been evaluated, and the number of true positives and false positives are shown for each. For example, τ_1 matches 20 fragments from D_+ and 50 from D_N, and is therefore assumed to match 20 true positives and 50 false positives. At this stage of the search, the decision to be made is which node to evaluate next: τ_5 or τ_6? We can feed forward the facts that the two parent templates of τ_5, τ_1 and τ_2 have 20 and 15 true positives respectively, and that therefore τ_5 must have at least 20 true positives. Similarly, it must have at least 60 false positives. On the other hand, τ_6 must have at least 40 true positives, and at least 55 false positives. We would therefore decide to evaluate τ_6 in preference to τ_5 at this point, because it has a higher lower-bound on the number of true positives, and a lower lower-bound on the number of false positives.

From these observations, we can then develop specific algorithms, such as the best-first algorithm we now describe.

D. A best-first algorithm

We now present a formal definition of a best-first algorithm suitable for identifying good templates, followed by an example. We start by defining two sets of templates. Set O ("open") contains templates that have been created (via initialise or generalise), but not yet evaluated. Set C ("closed") contains templates that have been evaluated, i.e. had values of TP and FP calculated. Note that $O \cap C \equiv \emptyset$.

Figure 2 gives the algorithm. After initialisation, we take the best template that has not yet been evaluated, and evaluate it, i.e. calculate its true positive and false positive scores. We then generalise it in every way possible to create child templates, and update the lower bounds on the true and false positive scores for these children. Because there are several ways that each template could be created, some children will have more than one parent. In these cases, the lower bounds are the *maximum* of the lower bounds of all the parents.

1) **begin**
2) $C \leftarrow \emptyset$, $O \leftarrow \emptyset$
3) initialise$(f) = \tau_{root} \rightarrow O$
4) **while not finished do**
   a) find estimated best template τ in O
   b) evaluate τ
   c) delete τ from O
   d) add τ to C
   e) expand τ by adding to O all templates that can be created by superset generalisation of τ
   f) update lower bounds on TP and FP for all templates in O
5) return best template from C.
6) **end**

Fig. 2. A best-first algorithm. C is the "closed" set of evaluated templates and O is the "open" set of unevaluated templates. See Section VI-D for further details.

By "best template" (Figure 2, Steps 4a and 5), we mean select the template with the highest lower-bound on the number of true positives, as inherited from each template's ancestors. If more than one template has the same maximum true positive lower-bound, then we choose between them by selecting the template with the smallest lower-bound on false positives. If this still selects more than one template, we can either pick one randomly; use them all successively; or use all the selected templates together in the subsequent steps (i.e. evaluate and generalise more than one template in one pass through the main loop).

In Step 4f we could choose to either update the bounds of just the descendents of τ in O, or else update the bounds of the descendents of every template in C. The latter would be more computationally expensive, but would lead to better estimates of the lower bounds. For each new template created, it would require a search through C to find all the other possible ancestors, besides τ, in order to calculate the new lower bounds.

Finally, we terminate the search when some pre-specified criterion is satisfied. Possible criteria include terminating: when O is empty (in which case every possible template has been evaluated); or when a limit on the number of evaluations is reached (e.g. stop after 1000 templates have been evaluated); or when a certain number of true positives (or false positives) are matched by the current best template. It would be quite possible for the user to stop the search and consider the current best template, before re-starting the search if necessary.

As a more concrete example, Figure 3 shows part of a search graph containing various templates created from the seed fragment "the cat sat", and evaluated with respect to the two small corpora shown.

Consider the right-hand portion of the graph. Near the top are two templates: <the, *, sat> and <the, cat, *>, both created using the generalise(τ, κ_Ω, i) function. The first matches one true positive and no false positives (shown as 1_+0_N in the figure). The second matches one true positive and one false positive. By the definition of superset generalisation, we know that every template derived from this second template will have at least one true positive and one false positive. So if, at one stage of a search, we only want to consider templates with no false positives, then we need not consider *any* descendent of this template. The two descendents shown (<the, *, *> and <*, *, *>) need not be evaluated explicitly therefore; they can be annotated as having at least one false positive, although in the figure, the fully evaluated scores are shown.

Now consider the third row of templates in Figure 3, i.e. those created after two generalisation functions. From the left of the graph, three of these have two true positives (<the FELINE VBD>, <the FELINE *> and <the ANIMAL VBD>) and one has only one true positive, so we focus the search on the first three. These have zero, one and one false positives respectively, making the first one look most promising: <the, FELINE, VBD>. This is the best unevaluated template so far. Several further generalisations are possible from this template, including applying generalise$(\tau, \kappa_\Omega, 3)$ to form <the, FELINE, *>. But this has already been evaluated, so need not be considered again. Applying generalise$(\tau, \kappa_\Pi, 1)$ forms <DT, FELINE, VBD> which has three true positives, and still no false positives. Given the two very small document sets, this is an optimal template, so we stop the search here. In a more realistic application, the search would continue until a stopping criterion was reached, but would not find any superior template.

One modification to the algorithm would require more memory but should lead to a faster convergence to a (possibly) superior solution. This is to expand the unevaluated template set O by repeated applications of the generalise function until it contains every template that could possibly be derived from the seed phrase, or as many as is practically possible. We would then proceed with a best-first search, but with the advantage that after each evaluation, a large number of unevaluated templates will have the lower bounds of their true- and false-positive estimates updated, because their ancestry would be explicitly represented on the search graph. This should improve the selection of the best template at each stage.

The algorithm above does not prune any part of the search graph, but merely tends to search promising areas first. In practice, this is likely to lead to very large memory requirements, which can be avoided through pruning. Pruning search *graphs* is a lot more difficult than pruning *trees* because each node can have multiple parents, and so after a node is pruned, it may reappear as part of a different path. Although algorithms such as A* are inappropriate here[1], recent variations may provide useful pruning methods, such as SMA* [10, p. 104] and Sweep A* [16].

E. Multi-objective search methods

A best-first search may miss out on good templates because of its greedy decision making. This would be true even if the estimates of the numbers of true and false positives were perfect, owing to the structure of the graph: we can't guarantee that the best parents will have the best children. One likely improvement therefore would be a population based search, such as a simple beam search or an evolutionary algorithm.

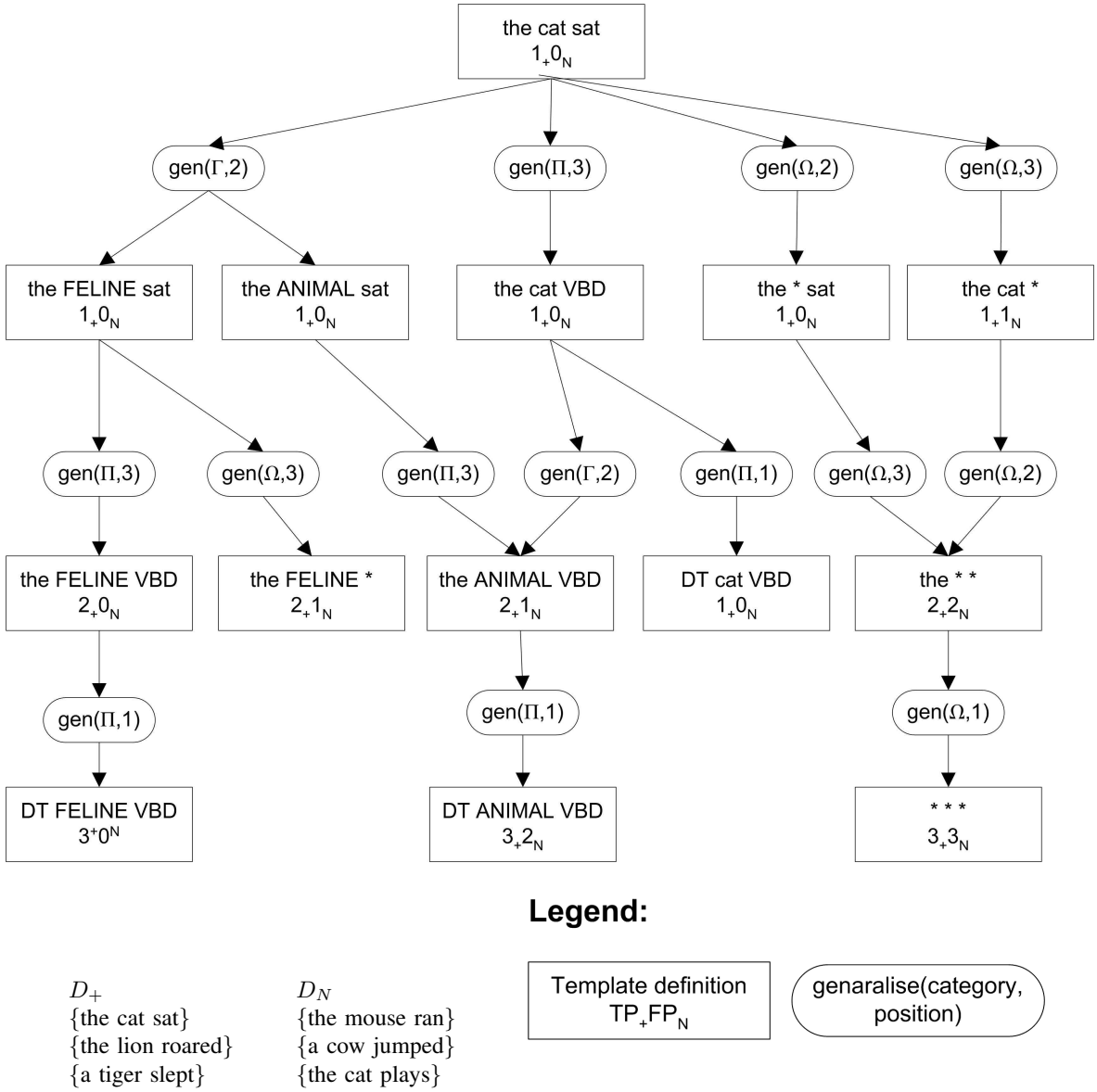
Evolutionary algorithms have been successfully used to solve a wide range of multi-objective optimisation problems [14]. Extracting information from a large corpus takes considerable computing effort, so this is an aspect worth considering. Multi-objective evolutionary algorithms can efficiently generate a range of Pareto-optimal solutions, and so explore the trade-off between the different objectives. In our case, this means that for each number of true positives, we find the template with the fewest false positives, and for each number of false positives, we find the template with the most true positives. This produces a range of solutions from which the user can then select whichever template or templates are most suitable for their particular problem.

VII. Discussion and extensions

We now briefly discuss a few of the possible extensions to the framework.

We could introduce other wildcards, such as a wildcard which matches an entire phrase, which could itself be defined as a series of terms, much like a template. This would allow optional subclauses, such as subordinate clauses, to be matched. Let $?^{\tau_1}$ designate an optional wildcard that matches a sequence of literals defined by template τ_1 or nothing at all. Then if $\tau_1 =<\{\text{which}\}, \{*\}, \{*\}>$, the template $\tau_2 =<$

[1]We have no notion of a path cost, and are only interested in the final solution rather than the path to it.



D_+
{the cat sat}
{the lion roared}
{a tiger slept}

D_N
{the mouse ran}
{a cow jumped}
{the cat plays}

Fig. 3. Part of a search graph for a three-term template. Each rectangle shows a template, starting with three sets of literals at the top initialised from the fragment "the cat sat". Various generalisation functions are then applied to it. For example, gen$(\Gamma, 2)$ means apply the generalise$(\tau, \kappa_\Gamma, 2)$ function to the upper template, τ, to produce the lower template(s). The graph includes part-of-speech labels "VBD" meaning past-tense verb and "DT" meaning determiner. The numbers in each box below the template represent the estimated numbers of true-positive and false-positive matches for each template, with respect to the positive and neutral document sets D_+ and D_N shown. Note that not every node or edge is shown.

{DT}, {ANIMAL}, $\{?^{\tau_1}\}$, {sat} > would match the fragment <the, cat, which, was, black, sat> as well as <the, cat, sat>.

An additional approach to finding good templates is to repeatedly *merge* useful templates to produce more general templates [17]. Our framework could easily be extended to allow this, by ensuring that the product of merging two templates matches every fragment that either template matches. This could be achieved by considering each pair of template elements in turn, and either performing a simple set union if they belong to the same category, or else generalising them both to the same category before such a union. In either case, the new template would match the union of the true-positives matched by the two parents, and the union of the false-positives, allowing the lower-bounds on each to be calculated.

So far, we have considered template that exist in isolation, whereas in practical systems, it is more common to apply a set of templates together. Our framework can be extended to include this. Suppose we have a template τ that matches some true positives and some false positives. We could reduce the number of false positives by creating a second template τ' that is optimised to match just the false positive fragments matched by τ. This could be achieved by defining two new

versions of D_+ and D_N based on the fragments matched by τ, and using these to guide the search for τ'. We could then apply τ and τ' together, predicting interesting fragments as $\mu(\tau, D) \setminus \mu(\tau', D)$ (i.e. fragments matched by τ but not by τ'.). In many practical applications, more than one template will be applied to a set of documents, each designed to match a different piece of information, or a different way of expressing that information.

We have assumed that we do not have a set of annotated examples, i.e. fragments known in advance to be positive or negative. Creating and annotating large sets of examples is extremely time consuming for a user, although giving a yes/no response to automatic annotations is simpler [18]. One enhancement to our system therefore would be to start with the estimates of true positive and false positive as outlined above, and search for a good template, and then use this template to annotate a number of fragments and to present these to the user. The user then marks each fragment as interesting or not interesting, and this could then be used to improve the quality of the function used to estimate the numbers of true and false positives. This improved function could be used to guide a new template search.

Finally, rather than starting with a seed fragment and a template consisting solely of literals, we could start the search using a hand-written template. This would *not* have to be optimised in advance, and in some cases, would be easy to create. The search could then start from a point chosen to be useful and optimised further through similar search processes to those outlined above.

VIII. Conclusion

We have presented a formal framework to describe information extraction, focusing on the definition of the template patterns used to convert free text into a structured database. The framework has allowed us to explicitly identify some of the fundamental issues underlying information extraction and to formulate possible solutions. We have shown that the framework allows computationally feasible heuristic search methods to be developed for automatic template creation. We believe that a practical implementation of this framework is feasible which will allow automatic template creation. We also hope that the framework will allow other researchers to gain further insights into the theory and practice of information extraction and text mining.

Acknowledgments

This work is partly funded by BBSRC grant BB/C507253/1, "Biological Information Extraction for Genome and Superfamily Annotation."

References

[1] J. Cowie and W. Lehnert, "Information extraction," *Communications of the ACM*, vol. 39, no. 1, pp. 80–91, 1996.

[2] J. Cowie and Y. Wilks, "Information extraction," in *Handbook of Natural Language Processing*, R. Dale, H. Moisl, and H. Somers, Eds. New York: Marcel Dekker, 2000.

[3] C. Blaschke and A. Valencia, "The frame-based module of the SUISEKI information extraction system," *IEEE Intelligent Systems*, vol. 17, no. 2, pp. 14–20, Mar. 2002.

[4] L. Hirschman, A. Yeh, C. Blaschke, and A. Valencia, "Overview of BioCreAtIvE: critical assessment of information extraction for biology," *BMC Bioinformatics*, vol. 6, no. Suppl 1, 2005.

[5] D. P. A. Corney, B. F. Buxton, W. B. Langdon, and D. T. Jones, "BioRAT: Extracting biological information from full-length papers," *Bioinformatics*, vol. 20, no. 17, pp. 3206–13, 2004.

[6] R. Collier, "Automatic template creation for information extraction," Ph.D. dissertation, Department of Computer Science, University of Sheffield, 1998.

[7] D. Pierce and C. Cardie, "Limitations of co-training for natural language learning from large datasets," in *2001 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics Research, 2001.

[8] E. Riloff, "Automatically constructing a dictionary for information extraction tasks," in *National Conference on Artificial Intelligence*, 1993, pp. 811–816.

[9] A. Koike, Y. Niwa, and T. Takagi, "Automatic extraction of gene/protein biological functions from biomedical text," *Bioinformatics*, vol. 21, no. 7, pp. 1227–1236, April 2005.

[10] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall, 2003.

[11] D. P. A. Corney, E. L. Byrne, B. F. Buxton, and D. T. Jones, "A logical framework for template creation and information extraction: A technical report," Dept. of Computer Science, University College London, Technical report RN/05/23, Oct. 2005.

[12] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[13] M. Marcus, B. Santorini, and M. A. Marcinkiewicz, "Building a large annotated corpus in English: the Penn Treebank," *Computational Linguistics*, vol. 19, pp. 313–330, 1993.

[14] C. M. Fonseca and P. J. Fleming, "An overview of evolutionary algorithms in multiobjective optimization," *Evolutionary Computation*, vol. 3, no. 1, pp. 1–16, 1995.

[15] J. Kim, T. Ohta, Y. Tateisi, and J. Tsujii, "GENIA corpus–semantically annotated corpus for bio-textmining," *Bioinformatics*, vol. 19 Suppl 1, pp. 180–182, 2003.

[16] R. Zhou and E. Hansen, "Sweep A*: Space-efficient heuristic search in partially-ordered graphs," in *Fifteenth IEEE International Conference on Tools with Artificial Intelligence*, Sacremento, CA, Nov. 2003.

[17] C. Nobata and S. Sekine, "Towards automatic acquisition of patterns for information extraction," in *International Conference of Computer Processing of Oriental Languages*, 1999.

[18] D. Pierce and C. Cardie, "User-oriented machine learning strategies for information extraction: Putting the human back in the loop," in *Working Notes of the IJCAI-2001 Workshop on Adaptive Text Extraction and Mining*, 2001, pp. 80–81.